Optimización en cadenas de desmontaje

Trabajo de Fin de Grado
Grado en Matemáticas
Curso 2021-2022

Universidad Complutense de Madrid
Facultad de Ciencias Matemáticas
Departamento de Sistemas Informáticos y Computación

Realizado por: Ángela Cisneros Pascual
Tutelado por: Fernando Rubio Diez
Madrid, 2022

Índice general

Resumen

En la presente memoria se pretende estudiar el Problema de Equilibrado en Líneas de Desmontaje. Se presentará una explicación y el planteamiento de dicho problema junto con un estudio de su complejidad, resultando pertenecer a la clase $\mathcal{NP}$-completo. Debido a esto último se determina que el problema ha de ser tratado mediante métodos metaheurísticos, se elegirá para ello los algoritmos genéticos. Estos algoritmos serán estudiados presentando distintos operadores de selección, cruce y mutación para diferentes codificaciones. Tras ello se realiza un estudio sobre distintos casos de prueba para determinar qué configuración es la más adecuada para la resolución del problema a estudiar.

Abstract

This work aims to study the Disassembly Line Balancing Problem. An explanation and approach to this problem will be presented with the study of its complexity, resulting in belonging to the $\mathcal{NP}$-complete class. Due to the latter, it is determined that the problem has to be treated by metaheuristic methods, for which genetic algorithms will be chosen. These algorithms will be studied by presenting different selection, cross and mutation operators for different encodings. Subsequently, a study of different test cases is done to determine which configuration is the most appropriate for solving the case study.

Capítulo 1

Introducción

El reciclaje ha cobrado una gran importancia en las últimas décadas debido a la grave situación medioambiental en la que nos encontramos. La gran sobreproducción que ejerce nuestra sociedad contribuye a la destrucción masiva de bosques, al deterioro progresivo de la capa de ozono, a la contaminación de las masas de agua y a un gran etcétera de situaciones nocivas para nuestro planeta. Por ello, encontrar formas más responsables para la obtención de materiales se ha convertido en una indispensable estrategia para solventar este problema.

En un segundo plano, el reciclaje también posee beneficios económicos, la reutilización de materiales y objetos supone un coste mucho menor que la obtención de nuevos, ya que el gasto de energía se reduce considerablemente.

Por todo esto, el reciclaje es entendido como un acto proambiental en el que un objeto ya usado, mediante un proceso de renovación, cobra una segunda vida. Se considera que casi todos los elementos de los que hacemos uso pueden ser reciclados o reutilizados, a excepción de aquellos que poseen algún factor tóxico por el que no puedan ser manipulados. Así pues, encontrar una forma efectiva de acceder a estos materiales o piezas que componen un producto cobra importancia. Esta memoria trabaja sobre este último hecho, la optimización del tiempo del desmontaje de un producto, teniendo en cuenta la peligrosidad y la demanda de las partes de dicho objeto.

1.1. Objetivos y plan de trabajo

El Problema de Equilibrado en Líneas de Desmontaje de un producto consiste en encontrar la forma más eficiente de llevar esto a cabo, hallando el mejor orden de retirada de las piezas bajo una serie de criterios.

Para lograr este objetivo se realiza en primer lugar un profundo estudio del problema (capítulo 2), desarrollando cuáles son sus supuestos y cuáles sus restricciones. Se expondrán los objetivos a optimizar y se formalizarán las funciones objetivo. Algo indispensable para nuestro estudio será el análisis de la complejidad del problema, se concluirá que este pertenece a la clase $\mathcal{NP}$-completo, lo que nos llevará a emplear técnicas metaheurísticas.

Como hemos dicho, la complejidad de nuestro problema hace que este precise del uso de métodos metaheurísticos, en nuestro caso, recurriremos a los algoritmos genéticos. Por tanto, se hace necesario explicar qué son, cómo funcionan y las técnicas que se llevan a cabo, esto se realizará en el capítulo 3.

El siguiente paso consistirá en la aplicación de estos algoritmos al Problema de Equilibrado en Líneas de Desmontaje, se llevará a cabo en el capítulo 4. Se adecuarán todos los elementos y mecanismos de estos algoritmos a las necesidades de nuestro problema y mediante el análisis de tres instancias se determinará qué configuración del algoritmo es la más adecuada. Con este fin, se ha programado un algoritmo genético en PYTHON, se pueden ver los detalles de este en los apéndices B y C. Una vez esto se ha realizado, se aplicará a otros casos de estudio y se compararán los resultados obtenidos con los presentados por las fuentes de donde se han obtenido las instancias.

Capítulo 2

El Problema de Equilibrado en Líneas de Desmontaje

En este capítulo se realiza un profundo estudio del Problema de Equilibrado en Líneas de Desmontaje, formalizando sus funciones objetivo y desarrollando cuáles son sus supuestos y cuáles sus restricciones. También se estudiará su complejidad con el fin de determinar una buena estrategia para abordarlo. Finalmente, se expondrá una versión del mismo la cual también formará parte del estudio.

2.1. Planteamiento del problema

Según Seamus M. McGovern y Surendra M. Gupta el desmontaje se define como *"una metódica extracción de piezas o subconjuntos y materiales valiosos de productos desechados mediante una serie de procedimientos, para usar en la remanufactura o reciclaje después de las operaciones apropiadas de limpieza y prueba"* (Gupta & McGovern [18]). El problema de la secuenciación de desmontaje que identificaremos como DLBP (Disassembly Line Balancing Problem) se ocupa de determinar el mejor orden en el que se deben desmontar las piezas para su aprovechamiento.

Vamos a describir brevemente en qué consiste nuestro problema. Se nos presenta un producto a desmontar, el cual estará compuesto por una serie de piezas. Nuestro objetivo se centra en retirar dichas piezas en el orden más eficiente y beneficioso posible. Es evidente que en el proceso de desmontaje de un producto las piezas no se pueden retirar en cualquier orden, es necesario respetar ciertas relaciones de precedencia. Como ya hemos dicho, aparte de un desmontaje eficiente, nos interesará que este sea beneficioso y que, en el caso de existir piezas con algún componente que pueda suponer un riesgo, estas se retiren lo antes posible. Por esto, en nuestro estudio también valoraremos la demanda y la peligrosidad de las piezas.

¿A qué nos referimos exactamente con un desmontaje eficiente? En nuestro problema contamos con que podemos utilizar más de una máquina para efectuar el proceso de desmontaje de nuestro producto. Es obvio que si para retirar cada pieza le asignamos una estación de trabajo a cada una, el tiempo de desmontaje sería mínimo, pero el coste a asumir respecto al número de máquinas sería muy elevado. Por ello, se beneficiará aquellas soluciones con un número de máquinas menor. Por otro lado, el hecho de que todo el proceso lo asuma una sola máquina minimizaría el coste empleado en puestos de trabajo, pero no nos proporcionaría una solución muy eficiente respecto al tiempo. Por ello, se añade una restricción en cuanto al tiempo de utilización de una estación de trabajo, el ciclo de tiempo. El tiempo de funcionamiento de una máquina en el desmontaje de un ejemplar de nuestro producto no podrá

exceder dicha cantidad. Para evitar que una máquina asuma un gran gasto de tiempo y otra uno mínimo se recompensarán aquellas soluciones en las que el tiempo de inactividad no sea muy dispar entre las máquinas.

En resumen, nuestro problema tendrá como objetivos minimizar el número de estaciones de trabajo, que el tiempo de inactividad de las máquinas sea lo más equilibrado posible, eliminar las piezas que supongan un riesgo cuanto antes y obtener el mayor beneficio retirando lo antes posible las piezas con más demanda.

2.2. Modelización del problema

En esta sección vamos a profundizar en los supuestos y la formalización de los objetivos, para ello, nos basamos en la modelización expuesta por Seamus McGovern y Surendra Gupta [18]. Como ya hemos dicho, el desarrollo de este problema tiene como objetivos disminuir el número de puestos de trabajo, que el tiempo de inactividad entre las máquinas sea similar, retirar lo más rápido posible las piezas con un factor de peligrosidad y aquellas que tengan una alta demanda.

El primer objetivo que nos atañe es disminuir el número de puestos de trabajo, se propone la siguiente función objetivo,

$$min\ f_1 = m,$$

siendo m el número de máquinas.

Para cumplir el segundo objetivo, uniformar los tiempos de inactividad de las máquinas, necesitamos penalizar aquellas soluciones en las que uno o más puestos de trabajo tengan una gran inactividad respecto a los demás, para ello se presenta la siguiente función:

$$\begin{aligned} min\ f_2 &= \sum_{k=1}^{m} (CT - T_k)^2, \quad \text{donde} \\ T_k &= \sum_{i=1}^{n} x_{ik} t_i, \quad k \in \{1..m\}, \quad t_i \in \mathbb{N}\ \forall i \in \{1..n\} \\ x_{ik} &= \begin{cases} 1 & \text{si la pieza } i \text{ se ha asignado a la máquina } k \\ 0 & \text{en otro caso} \end{cases} \end{aligned}$$

siendo CT el ciclo de tiempo, m el número de máquinas, n el número de piezas del producto, T_k, con $k \in \{1..m\}$, el tiempo empleado en cada máquina y t_i, con $i \in \{1..n\}$, el tiempo empleado en la tarea de retirar la pieza i.

Para el tercer objetivo necesitamos introducir una medida de peligrosidad, la cual querremos minimizar. Esta medida está basada en una variable binaria la cual indica si una pieza posee materiales considerados peligrosos o no y su posición en la secuencia.

$$min\ f_3 = \sum_{j=1}^{n} (j \times h_{P_j}), \quad h_{P_j} = \begin{cases} 1 & \text{peligroso} \\ 0 & \text{en otro caso} \end{cases}$$

siendo P_j la j-ésima pieza retirada.

Para el cuarto objetivo, se incluye una medida de demanda para cuantificar el desempeño de cada solución respecto a este aspecto. Esta medida se fundamenta en valores enteros positivos que indican la cantidad requerida después de su retirada en relación a su posición en la secuencia. Se construye de manera que a menor valor la solución es más deseable. Luego, son favorecidas las soluciones en las cuales las piezas con más demanda se retiran con mayor antelación.

$$min\ f_4 = \sum_{j=1}^{n}(j \times d_{P_j}), \quad d_{P_j} \in \mathbb{N},\ \forall P_j$$

Las restricciones del problema son:

- Cada tarea es asignada a exactamente un puesto de trabajo.
- Los tiempos de extracción de piezas son deterministas, constantes y enteros (o que se pueda transformar en entero).
- La suma de los tiempos de extracción de piezas de todas las piezas asignadas a una estación de trabajo no debe exceder el ciclo de tiempo.
- Las relaciones de precedencia de extracción entre las piezas deben ser respetadas.

2.3. Estudio de la complejidad

El estudio de un problema y la búsqueda de un algoritmo eficiente que lo resuelva hace necesario el estudio de su complejidad computacional. Nos referimos con complejidad computacional de un problema a la cantidad de recursos necesarios para resolverlo. Comúnmente los recursos estudiados son el tiempo, número de pasos de ejecución de un algoritmo para resolver el problema; y el espacio, cantidad de memoria utilizada para resolverlo. En esta sección nos centraremos en el coste temporal. Vamos a introducir una serie de conceptos básicos para abordar esta cuestión, recurriremos al libro de Hopcroft, Motwani y Ullman [7, pp. 351–360] para su explicación.

Definición 2.3.1 *Una máquina de Turing M tiene complejidad temporal $T(n)$ si siempre que recibe una entrada w de longitud n, M se para después de realizar como máximo $T(n)$ movimientos, independientemente de si la acepta o no.*

Definición 2.3.2 *Decimos que un lenguaje pertenece a la clase $\mathcal{P}$ si existe una máquina de Turing determinista para dicho problema cuya complejidad temporal es polinómica.*

Definición 2.3.3 *Una máquina de Turing no determinista M tiene complejidad temporal $T(n)$ si ninguna ejecución sobre una cadena de longitud n requiere más de $T(n)$ pasos.*

Definición 2.3.4 *Decimos que un lenguaje pertenece a la clase $\mathcal{NP}$ si existe una máquina de Turing no determinista para dicho problema cuya complejidad temporal es polinómica.*

Para poder aplicar esta teoría en el estudio de la complejidad de un problema de optimización será necesario hacer una pequeña transformación, deberemos reformularlo como un lenguaje, es decir, transformarlo en un problema de decisión. Estos problemas solo admiten como solución dos respuestas “sí” o “no”.

La complejidad computacional considera tratables los problemas/lenguajes pertenecientes a la clase $\mathcal{P}$. En los problemas pertenecientes a la clase $\mathcal{NP}$ se puede verificar si una instancia es solución de manera eficiente.

Definición 2.3.5 *Un lenguaje L se reduce polinómicamente a L', $L \leqslant_{\boldsymbol{P}} L'$, si existe f tal que:*

1. $x \in L \Leftrightarrow f(x) \in L'$
2. *f es computable en tiempo polinómico.*

Definición 2.3.6 *Un lenguaje L es $\mathcal{NP}$-completo si:*

1. $L \in \mathcal{NP}$
2. *Si $L' \in \mathcal{NP}$ entonces $L' \leqslant_{\boldsymbol{P}} L$*

Teorema 2.3.1 *Sean L un lenguaje $\mathcal{NP}$-completo, L' perteneciente a $\mathcal{NP}$, si $L \leqslant_{\boldsymbol{P}} L'$ entonces L' es $\mathcal{NP}$-completo.*

Planteemos nuestro problema como un problema de decisión, DLBP-decisión, de la siguiente forma:

Sea P *un conjunto finito de piezas numeradas con una relación de orden parcial $\prec$ (corresponde con la precedencia de las tareas). Sea $t_i \in \mathbb{N}$ el tiempo empleado para realizar cada tarea, $h_i \in \{0, 1\}$ el indicativo de peligrosidad y $d_i \in \mathbb{N}$ el valor de demanda para cada $i \in$* P*. Sea* CT $\in \mathbb{N}$ *la capacidad de tiempo de cada estación de trabajo,* M $\in \mathbb{N}$ *el número de máquinas,* B $\in \mathbb{N}$ *la suma del tiempo de inactividad al cuadrado de cada máquina,* H $\in \mathbb{N}$ *la medida de peligrosidad y* D $\in \mathbb{N}$ *la medida de demanda.*

¿Existe una partición de P *en conjuntos disjuntos* P_1, P_2,..., P_m *ordenados tal que $m \leq$* M*, la suma de los t_i en cada P_x es a lo sumo* CT*, la suma del tiempo de inactividad al cuadrado de cada máquina sea menor o igual que* B*, la suma del valor binario de peligrosidad multiplicado por su posición en la secuencia global es como mucho* H*, la suma del valor de demanda de la pieza multiplicada por su posición en la secuencia es a lo sumo* D *y que cumpla las restricciones de precedencia?*

Primero, comprobemos que nuestro problema pertenece a la clase $\mathcal{NP}$, es decir, que cualquier instancia del problema se puede verificar en tiempo polinómico. Dada una secuencia de n partes a retirar junto con las relaciones de precedencia, el número de máquinas, la suma del tiempo de inactividad al cuadrado de cada máquina, la medida de peligrosidad y la medida de demanda (la partición se realiza según el orden de la secuencia); dadas las cantidades CT, M, B, H y D. La comprobación de que la secuencia respete las relaciones de precedencia y de que no se superen las cantidades dadas se puede realizar en tiempo polinómico, luego DLBP-decisión $\in \mathcal{NP}$.

Dado que según aumente el tamaño de nuestro problema el espacio de soluciones a explorar crece de forma considerablemente más rápida que de forma polinómica, se espera entonces que DLBP-decisión no pertenezca a la clase $\mathcal{P}$. Basándonos en la prueba presentada por Seamus McGovern y Surendra Gupta [18], estudiaremos entonces si nuestro problema pertenece a la clase $\mathcal{NP}$-completo, se procede mediante la reducción del problema de Partición a DLBP-decisión.

El problema de Partición es $\mathcal{NP}$-completo y se enuncia como: *Sea un conjunto finito S de enteros. ¿Existe una partición* $S' \subseteq S$ *en dos conjuntos disjuntos tal que* $\sum_{a \in S'} a = \sum_{a \in S\text{-}S'} a$*?*

Para que el problema de Partición se reduzca a DLBP-decisión, debemos encontrar una transformación computable en tiempo polinómico de manera que una instancia será válida para el problema de Partición si y solo si su transformación también lo es para DLBP-decisión.

Dado un conjunto S de enteros, la instancia obtenida mediante la transformación para DLBP será el conjunto S, la relación de orden parcial será vacía, el tiempo de cada elemento será su propio valor y los índices de peligrosidad y los valores de demanda serán 0 para todos ellos; junto con $CT = \frac{\sum_{a \in S} a}{2}$, $M = 2$ y $B = H = D = 0$. Es trivial que la transformación es computable en tiempo polinómico y que la instancia será válida para el problema de partición si y solamente si su transformación es válida para DLBP-decisión. Por lo que, Partición $\leqslant_{\boldsymbol{P}}$ DLBP-decisión.

Con este último resultado y junto con DLBP-decisión $\in \mathcal{NP}$, por el teorema (2.3.1) concluímos que DLBP-decisión $\in \mathcal{NP}$-completo.

Muchos problemas de optimización matemática, como es nuestro caso, resultan pertenecer a la clase $\mathcal{NP}$-completo y, por lo tanto, son potencialmente difíciles de resolver. Es por ello que se dió a lugar la investigación de métodos metaheurísticos, los cuales son procedimientos capaces de encontrar, de forma eficiente, buenas soluciones pero no aseguran su optimalidad. En esta memoria se hará uso de ellos, concretamente de los algorítmos genéticos.

2.4. Problema de Equilibrado en Líneas de Desmontaje con Relaciones de Dependencia

En esta sección vamos a presentar una variación del problema DLBP que también vamos a estudiar, el Problema de Equilibrado en Líneas de Desmontaje con Relaciones de Dependencia, SDDLBP (Sequence-Dependent Disassembly Line Balancing Problem). Esta versión del problema la desarrollan Gupta, Kalayci y Polat [16].

La particularidad de este problema es que incorpora la posibilidad de interacciones entre las piezas, las cuales podrían afectar al tiempo empleado para retirarlas, se denominan relaciones de dependencia. Estas interacciones podrían ser a causa de que una pieza bloquee parcialmente a otra y haga requerir movimientos adicionales para retirarla. Esta dependencia se genera entre piezas que no tienen ninguna relación de precedencia.

Vamos a explicar estas relaciones de dependencia con un ejemplo. El diagrama (2.1) presenta las relaciones de precedencia (líneas continuas) y las relaciones de dependencia (líneas discontinuas) de un producto a desmontar. Se recuerda que las relaciones de precedencia muestran qué piezas bloquean de forma total la posible retirada de otras.

En este ejemplo se observa una relación de dependencia entre las piezas 2 y 3 y las piezas 5 y 6. Esto quiere decir que el orden en el que quitemos estas piezas influirá en su tiempo de retirada. Por ejemplo, denominemos sd_{ij} al tiempo añadido al tiempo de retirada de la

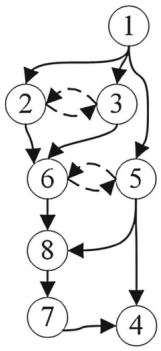

Figura 2.1: Ejemplo de relaciones de precedencia y dependencia 8 partes. Fuente: (Gupta, Kalayci & Polat [16])

pieza i si se retira antes que la pieza j, es decir, si retiramos antes la pieza 2 que la pieza 3 entonces $t'_2 = t_2 + sd_{23}$ será el tiempo que se tarde en retirar la pieza 2.

Capítulo 3

Algoritmos genéticos

En este capítulo estudiaremos los algorítmos genéticos, ya que, como hemos citado anteriormente, serán el método metaheurístico escogido para resolver el problema estudiado en esta memoria. Se presentarán diferentes versiones del algoritmo las cuales se adecúan a distintos tipos de problemas. Nos basaremos mayoritariamente en los trabajos de Berzal [4], Eiben y Smith [9] y Araujo y Cervigón [3].

3.1. Introducción

Los algoritmos genéticos (AG) son métodos metaheurísticos basados en los esquemas propuestos por Darwin sobre la selección natural, uno de los tipos de algoritmos evolutivos más populares debido a su eficiencia y sencillez de implementación, fueron propuestos por Holland en 1975 (Holland [19]).

Siguiendo la definición dada por Goldberg *"Los Algoritmos Genéticos son algoritmos de búsqueda basados en la mecánica de selección natural y de la genética natural. Combinan la supervivencia del más apto entre estructuras de secuencias con un intercambio de información estructurado, aunque aleatorizado, para constituir así un algoritmo de búsqueda que tenga algo de las genialidades de las búsquedas humanas."* (Goldberg [12]).

Para alcanzar la solución a un problema se genera de forma aleatoria un conjunto de individuos, llamado población. Cada uno de los cuales es candidato a ser una posible solución. A continuación, la población se somete a un proceso de evolución que modifica la composición de la población, eliminando a ciertos individuos y reforzando la presencia de otros. Un proceso de reproducción introduce nuevos individuos y una nueva evaluación actualiza los datos de la evolución.

3.2. Representación de los individuos

En un AG los individuos se codifican mediante una cadena de valores denominada cromosoma. Estas cadenas representan un punto x en el espacio de soluciones. Tomando la nomenclatura de la biología, el conjunto de parámetros representados en el cromosoma se denominará genotipo y a la información que estos representan, fenotipo.

Existen muchas formas de representar los individuos, se debe escoger la más adecuada para el problema en cuestión. A continuación, se muestran algunas de las representaciones más usadas.

3.2.1. Codificación binaria

Desde los primeros trabajos de John Holland la codificación más común es la codificación binaria debido a que su sencillez aporta características de eficiencia muy importantes para los AGs. La contrapartida es que es necesario disponer de un método de codificación y descodificación que permita pasar de la representación binaria al espacio de búsqueda natural del problema y viceversa.

Se asigna un número determinado de bits a cada parámetro y se realiza una discretización de la variable representada por cada gen. Cada uno de los bits pertenecientes a un gen suele recibir el nombre de alelo.

3.2.2. Codificación real

Una forma natural de codificar una solución asociada a un espacio de búsqueda continuo es utilizando valores reales como genes. Los individuos son representados como vectores de valores reales $X = (x_1, x_2, ..., x_k)$ con $x_i \in \mathbb{R}\ \forall i \in 1, ..., k$.

Este método confiere una mayor precisión y complejidad que la codificación binaria y a menudo está intuitivamente más cerca del espacio de soluciones.

3.2.3. Codificación permutacional

La codificación permutacional esta basada en un conjunto de genes con valores fijos y posiciones cambiantes en el cromosoma. Esta codificación es muy útil para problemas cuyo objetivo se basa en decidir en qué orden se deben realizar unas determinadas acciones, como en el caso de DLBP.

Los operadores de cruce y mutación habituales dan lugar a soluciones inadmisibles, por lo que los operadores de mutación deben cambiar al menos dos valores y los de cruce han de diseñarse específicamente para problemas de este tipo.

3.3. Inicialización y terminación

La inicialización de la población determina el proceso de creación de los individuos para el primer ciclo del algoritmo. Normalmente, la población inicial se forma a partir de individuos creados aleatoriamente. En problemas donde se disponga de información sobre qué soluciones son más prometedoras, podemos favorecer su generación al crear la población inicial. Sin embargo, es imprescindible dotar a la población inicial de gran variedad para poder explorar todas las zonas del espacio de búsqueda evitando así la convergencia prematura a óptimos locales.

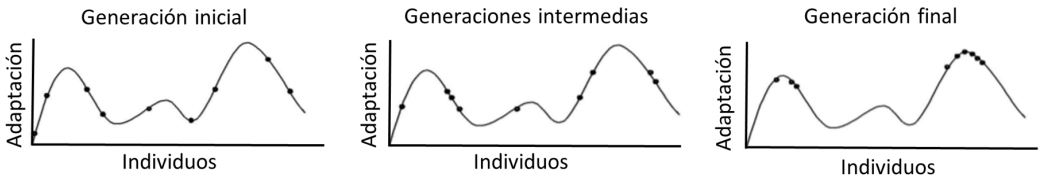

Figura 3.1: Evolución de un GA inicializándolo aleatoriamente. Fuente: Berzal [4]

Utilizar métodos alternativos para encontrar una posible inicialización mejor podría ser útil si existieran técnicas adecuadas para ello. Pero, en general, los beneficios que se obtienen no suelen ser lo suficientemente buenos como para que merezca la pena implementar dichos métodos. Esto es debido al típico comportamiento de los AG, veáse Fig.(3.2), los avances en la adaptación obtenidos en las primeras generaciones son muy grandes, luego, no se hace especialmente necesario inicializar dando peso a soluciones más prometedoras; sin embargo en las etapas finales su crecimiento es prácticamente nulo. Este comportamiento también nos da indicaciones sobre como tratar las condiciones de terminación para los AG. En la Fig.(3.2) se divide el tiempo de ejecución en dos mitades, se observa que en la primera mitad el crecimiento es significativamente mayor que en la segunda, esto sugiere que quizás no es útil permitir un número muy grande de iteraciones ya que el esfuerzo requerido no garantiza una gran mejora en la solución

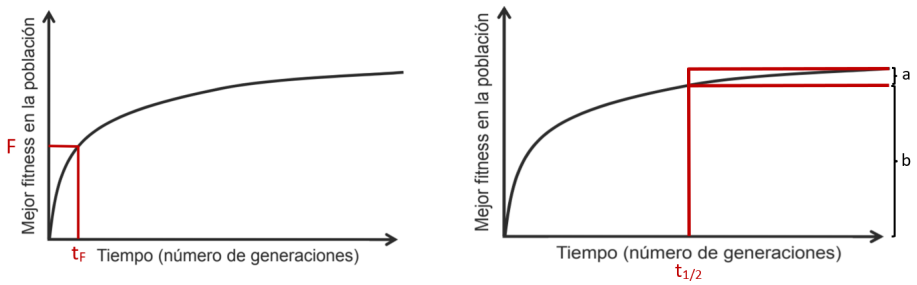

Figura 3.2: **(Izquierda)** Iniciando un AG con un método alternativo se obtiene F como valor de la función fitness. El tiempo necesario para alcanzar el nivel F usando una inicialización aleatoria es t_F. **(Derecha)** (a) Progreso en la segunda mitad. (b) Progreso en la primera mitad. Fuente: (Berzal [4])

Los algoritmos genéticos son estocásticos y la mayoría de ellos no garantizan alcanzar la solución óptima, por lo que fijar cierta precisión como condición de terminación podría conllevar que el algoritmo nunca terminase. La condición de terminación más sencilla es alcanzar un número determinado de generaciones de evolución. Otras condiciones son la detección de carencia de diversidad en las soluciones, es decir, la población ha convergido de forma similar; y que el incremento de la función fitness no supere cierto umbral en un periodo de tiempo.

3.4. Operador de selección

La población del AG se somete a un proceso de selección encargado de escoger qué individuos van a disponer de oportunidad para reproducirse y cuáles no, debe tender a favorecer la cantidad de copias de los individuos más adaptados. Sin embargo, no se debe eliminar por completo las opciones de reproducción de los individuos menos aptos, ya que en pocas generaciones la población se volvería homogénea. A continuación se exponen algunos de los métodos utilizados.

3.4.1. Selección proporcional o por ruleta

A cada individuo i de la población se le asigna una probabilidad de selección p_i, la cual es proporcional a su adaptación relativa:

$$p_i = \frac{f(i)}{f_t}$$

siendo f la función de adaptación y f_t la suma de todas las adaptaciones de la población. Los mejores individuos recibirán una proporción mayor que la recibida por los peores.

Para seleccionar un individuo seguiremos el siguiente procedimiento [22]:

- Definimos las puntuaciones acumuladas de la siguiente forma:

$$q_0 := 0$$
$$q_i := p_1 + ... + p_i \ \forall i = 1, ..., n$$

- Se genera un número aleatorio $a \in [0, 1]$
- Se selecciona al individuo i que cumpla:

$$q_{i-1} < a \leq q_i \quad \text{si } a \neq 0$$

  si $a = 0$ se selecciona al individuo $i = 1$

Este proceso se repite tantas veces como individuos se quieran seleccionar.

3.4.2. Muestreo estocástico universal

El procedimiento es similar a la selección por ruleta, se diferencia en que solo se genera un número de forma aleatoria, y a partir de él se generan los k números que se necesitan para generar k individuos espaciados de igual forma. Los números se calculan de la siguiente forma:

$$a_j := \frac{a + j - 1}{k} \ \forall j = 1, ..., k$$

Una vez generados estos números el método funciona de la misma manera que la selección por ruleta.

3.4.3. Selección por torneo

La idea principal de este método consiste en realizar la selección en base a comparaciones directas entre individuos. El proceso se repite hasta completar el número de individuos que se desee seleccionar. Existen dos versiones de selección mediante torneo:

Selección por torneo determinista

En la selección por torneo determinista se selecciona aleatoriamente un número p de individuos, generalmente se escoge $p = 2$. De entre los individuos seleccionados se elige el más apto, es decir, el de mayor valor de adaptación.

Selección por torneo probabilístico

La selección por torneo probabilístico se diferencia del anterior únicamente en el paso de selección. En vez de escoger siempre el mejor se genera un número aleatorio del intervalo [0..1], si es mayor que un parámetro l prefijado, se escoge el individuo más apto, en caso contrario, al menos apto. Generalmente l toma valores entre $0,5 < l \leq 1$.

3.4.4. Selección por ranking

Se ordena la población en base a la adaptación de peor a mejor y se asigna la probabilidad de selección según su posición en el ranking. Dicha asignación se puede realizar de muchas formas, las más comunes son lineal o exponencialmente decreciente de mejor a peor.

Se considera la función lineal

$$P_{lin}(i) = \frac{1-s}{N} + \frac{2is}{N(N-1)} \tag{3.1}$$

donde $P_{lin}(i)$ devuelve la probabilidad de que el individuo i sea seleccionado, i toma valores $0 \leq i < N$, con N el tamaño de la población y s es un parámetro $0 < s \leq 1$ que indica la presión selectiva, siendo esta el grado con el que se favorece a las mejores soluciones.

Si s=0 no existe presión selectiva y todos los individuos tendrán la misma probabilidad de ser seleccionados y si s=1 la presión selectiva es máxima.

Si queremos una presión selectiva superior, una de las alternativas es asignar las probabilidades de forma exponencial. Consideramos la función

$$P_{exp}(i) = \frac{1-e^{-i}}{c} \tag{3.2}$$

siendo la constante $c = \sum_{i=0}^{N-1} 1 - e^{-i}$ y que así la suma de las probabilidades sea 1.

3.5. Operador de cruce

Los operadores de cruce darán lugar a nuevos individuos a partir de la información proporcionada por las soluciones seleccionadas de la generación anterior. Es considerado uno de los rasgos más importantes de los algoritmos genéticos ya que es la mayor fuente de diversidad entre generaciones. Los operadores de cruce suelen ser aplicados de acuerdo a una tasa o frecuencia de cruce p_c, la cual determina la posibilidad de que un par de padres se sometan a este operador.

3.5.1. Operador de cruce para representaciones binarias

Operador de Cruce Monopunto

Consiste en la selección al azar de una única posición en la cadena de ambos padres e intercambiar las partes de los padres divididas por dicha posición. Este operador genera dos hijos combinando propiedades de ambos padres, lo que puede llevar a una mejora de la adaptación de los hijos.

Figura 3.3: Cruce Monopunto. Fuente: (Eiben & Smith [6])

Operador de Cruce n-punto

Se trata de una generalización del operador de cruce monopunto. El cromosoma se divide en más de dos segmentos, alternando la selección de los segmentos de los padres se generan los descendientes.

Figura 3.4: Cruce n-punto para $n = 2$. Fuente: (Eiben & Smith [6])

Operador de Cruce Uniforme

Este operador trabaja de manera independiente con cada gen del cromosoma. El cromosoma descendiente se genera seleccionando, de forma aleatoria, de qué progenitor heredará cada gen. El segundo descendiente heredará los genes no seleccionados.

Figura 3.5: Cruce Uniforme. Fuente: (Eiben & Smith [6])

3.5.2. Operador de cruce para representaciones reales

Recombinación discreta

Es análogo a los operadores de cruce para representaciones binarias. Cada alelo proviene de uno de sus padres y se puede hacer análogo al cruce uniforme o al de n-puntos. Tiene como inconveniente la falta de variedad respecto a los valores tomados por los padres.

Recombinaciones aritméticas

Se selecciona un parámetro al azar $\alpha \in [0, 1]$, en muchas ocasiones se selecciona $\alpha = 0{,}5$, y se generan combinaciones lineales entre los valores del gen tomados por los padres. Vamos a describir tres tipos de recombinación aritmética. Siendo los padres $[x_1, ..., x_n]$ y $[y_1, ..., y_n]$.

- **Cruce aritmético único**: Se selecciona un gen k al azar y se generan los hijos combinando los genes en la posición k de los padres:

  Primer hijo: $[x_1, ..., x_{k-1}, \alpha y_k + (1 - \alpha)x_k, ..., x_n]$
  Segundo hijo: $[y_1, ..., y_{k-1}, \alpha x_k + (1 - \alpha)y_k, ..., y_n]$

- **Cruce aritmético simple**: Se selecciona un gen k al azar y se generan los hijos combinando los genes de los progenitores a partir de dicho gen:

  Primer hijo: $[x_1, ..., x_k, \alpha y_{k+1} + (1 - \alpha)x_{k+1}, ..., \alpha y_n + (1 - \alpha)x_n]$
  Segundo hijo: $[y_1, ..., y_k, \alpha x_{k+1} + (1 - \alpha)y_{k+1}, ..., \alpha x_n + (1 - \alpha)y_n]$

- **Cruce aritmético completo**: Se realiza la combinación lineal para todos los genes a partir de los cromosomas padres:

  Primer hijo: $[\alpha y_1 + (1-\alpha)x_1, ..., \alpha y_n + (1-\alpha)x_n]$
  Segundo hijo: $[\alpha x_1 + (1-\alpha)y_1, ..., \alpha x_n + (1-\alpha)y_n]$

3.5.3. Operador de cruce para representaciones permutacionales

Este tipo de representación presenta un problema muy claro, si solo nos centramos en intercambiar subcadenas de ambos padres, es muy seguro que la solución no sea válida ya que probablemente no mantendrá la propiedad de permutación. Se han estudiado diversas soluciones para este problema. Se presentan los siguientes métodos:

Cruce por emparejamiento parcial (PMX)

El cruce por emparejamiento parcial fue propuesto por primera vez por Goldberg y Lingle como un operador de cruce para el problema del viajante. A partir de dos padres P1 y P2:

1. Se elige un segmento aleatorio y se copia de P1.
2. Desde el primer punto de cruce se buscan elementos de ese segmento en P2 que no se hayan copiado, para cada elemento i, se busca qué elemento j se ha copiado en su lugar desde P1.
3. Se coloca i en la posición ocupada por j en P2. Si el lugar ocupado por j en P2 se ha rellenado en el hijo por k, se sitúa i en la posición ocupada por k en P2.
4. El resto de elementos se rellenan de P2.

El segundo hijo se genera de forma análoga, intercambiando los papeles entre el primer padre y el segundo.

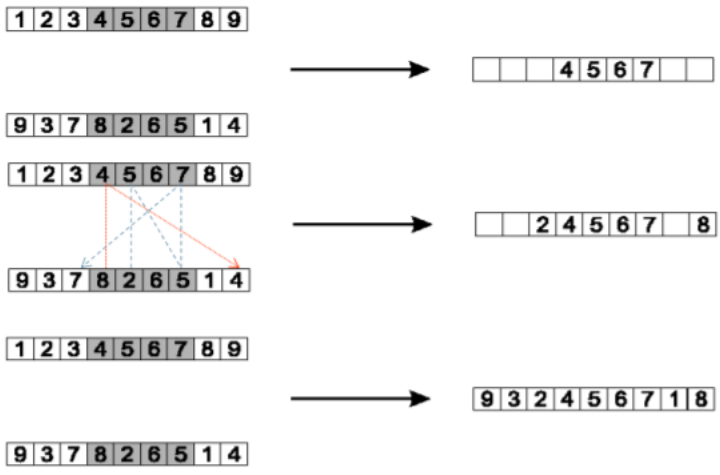

Figura 3.6: Cruce por emparejamiento parcial (PMX). Fuente: (Berzal [4])

Cruce de orden (OX)

Este método fue diseñado por Davis [8] para problemas de permutación basados en el orden. Comienza de forma similar a PMX, copiando de manera aleatoria un segmento del

primer padre. Se diferencia en que la intención es transmitir la información sobre el orden relativo del padre.

1. Se selecciona un segmento arbitrario del primer padre y se copia en el hijo.
2. Se copian los números que no se han escogido empezando desde el elemento en el segundo punto de cruce del segundo padre. Se copian desde el segundo punto de cruce del hijo volviendo al principio del cromosoma cuando hayamos llegado al final.

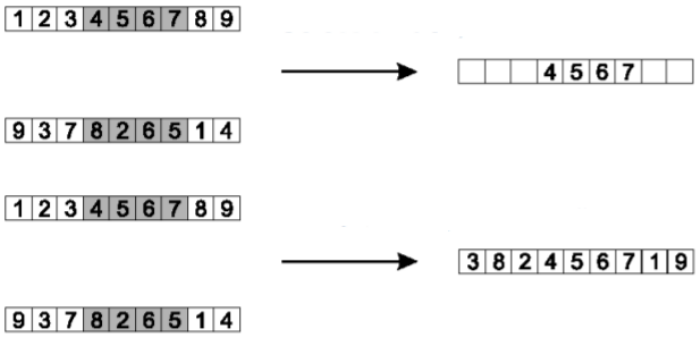

Figura 3.7: Cruce de orden (OX). Fuente: (Berzal [4])

Sin embargo, estos métodos no serían adecuados para la resolución de DLBP, ya que no consideran las restricciones de precedencia que caracterizan a nuestro problema. Por ello, introducimos los siguientes métodos de cruce.

Cruce por preservación de la precedencia (PPX)

Este método fue desarrollado por Bierwirth et al. [2]. Se basa en la creación de una máscara, una para cada hijo, formada por unos y doses indicando de qué padre debería extraerse la información. Por ejemplo, si la máscara fuese 21221 el primer gen del hijo sería el primer gen del segundo padre y este dejaría de estar disponible para su selección en ambos padres, el segundo gen sería el primero disponible del primer padre, los 2 primeros genes disponibles del segundo padre serán el tercer y cuarto gen del hijo y, el quinto, el primero disponible del primer padre.

Padre 1:	2	3	1	5	4
Padre 2:	1	5	3	4	2
Máscara:	2	1	2	2	1
Hijo:					

Padre 1:	2	3	X	5	4
Padre 2:	X	5	3	4	2
Máscara:	2	1	2	2	1
Hijo	1				

Padre 1:	X	3	X	5	4
Padre 2:	X	5	3	4	X
Máscara:	2	1	2	2	1
Hijo:	1	2			

Padre 1:	X	3	X	X	4
Padre 2:	X	X	3	4	X
Máscara:	2	1	2	2	1
Hijo	1	2	5		

Padre 1:	X	X	X	X	4
Padre 2:	X	X	X	4	X
Máscara:	2	1	2	2	1
Hijo:	1	2	5	3	

Padre 1:	X	X	X	X	X
Padre 2:	X	X	X	X	X
Máscara:	2	1	2	2	1
Hijo:	1	2	5	3	4

Cuadro 3.1: Ejemplo PPX

Cruce por reordenación de fragmentos (FRC)

El cruce por reordenación de fragmentos (Akpınar and Bayhan [1], 2011, Leu *et al.*[21], 1994) funciona de la siguiente manera, primero dos puntos se escogen de forma aleatoria, estos puntos determinarán los tres fragmentos en los que se dividirán los padres (cabeza, medio y cola). Como se muestra en la figura (3.8), el primer hijo mantiene la cabeza y la cola del primer padre. La parte del medio se rellena añadiendo todos los genes que faltan en el orden que marca el segundo padre. Para el segundo hijo los padres intercambiarán los papeles.

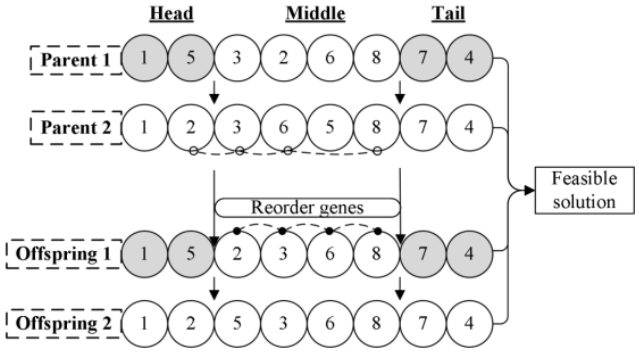

Figura 3.8: Cruce por reodernación de fragmentos (FRC). Fuente: (Kalayci [20])

3.6. Operador de mutación

El operador de mutación crea un nuevo individuo realizando algún tipo de alteración en un individuo de la población. Los operadores de mutación suelen ser aplicados de acuerdo a una tasa o frecuencia de mutación p_m, la cual determina la posibilidad de que se aplique el operador, suele tener un valor bastante pequeño.

En muchas ocasiones la mutación produce individuos con peor adaptación que los individuos originales, ya que puede romper las posibles correlaciones entre genes. Contribuyen a mantener la diversidad de la población, la cual es fundamental para el buen desarrollo del algoritmo.

3.6.1. Operador de mutación para representaciones binarias

Dado un cromosoma se altera el valor de cada una de las posiciones de la cadena, si es cero pasa a uno y viceversa. Para cada posición comprobamos la tasa de mutación, se genera un número aleatorio y si es menor que p_m aplicamos el operador.

3.6.2. Operador de mutación para representaciones reales

La mutación más utilizada para este tipo de representación es la mutación uniforme. Funciona de forma similar a la mutación para representaciones binarias. Se genera un número aleatorio y si es menor que p_m se aplica el operador, la diferencia reside en cómo varía el valor del gen. Para la representación binaria este puede tomar solo dos valores, sin embargo, aquí nuestro espacio de búsqueda es continuo. Cambiaremos el valor del gen de forma aleatoria siguiendo una distribución uniforme dentro de su dominio dado un valor L_i el ínfimo y U_i el supremo para el gen i, resultando la siguiente transformación.

$$[x_1, ..., x_n] \rightarrow [x'_1, ..., x'_n], \quad \text{donde } x_i, x'_i \in [L_i, U_i]$$

3.6.3. Operador de mutación para representaciones permutacionales

Existen diferentes métodos de mutación para este tipo de representación.

- Inserción: Se eligen dos alelos aleatoriamente y se coloca el segundo justo después del primero.
- Intercambio: Se seleccionan dos alelos aleatoriamente y se intercambian.
- Inversión: Se elige una subcadena y se invierte su orden.
- Revuelto: Se seleccionan dos alelos aleatoriamente y se reordena de forma aleatoria la subcadena delimitada por ellos.

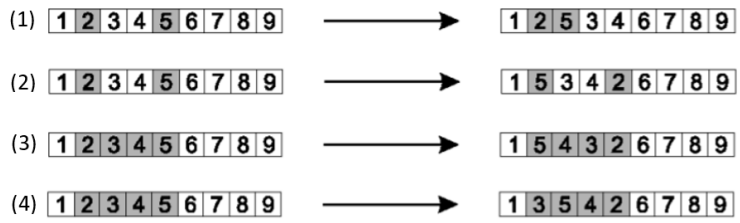

Figura 3.9: (1) Inserción. (2) Intercambio. (3) Inversión. (4) Revuelto. Fuente: (Berzal [4])

3.7. Reemplazamiento

Habitualmente los algoritmos genéticos mantienen un tamaño de población fijo. Por lo que, para mantener el tamaño de la población, hay que definir un mecanismo para crear la siguiente generación a partir de la anterior y la nueva. En función de la cantidad de individuos reemplazados en la población anterior se consideran distintas formas de realizar esta selección.

3.7.1. Algoritmos genéticos generacionales

En estos algoritmos la población se renueva por completo de una generación a otra, no se tiene en cuenta la adaptación de los individuos. Este procedimiento no excluye la permanencia de las soluciones con una alta adaptación en la población, pero esto depende de ser elegidos en la fase de selección y de sobrevivir en las etapas de cruce y mutación.

3.7.2. Algoritmos genéticos con estado estacionario

La descendencia de los individuos seleccionados en cada generación se incluye en la población anterior conservándose parte de esta última. Existen varias formas de establecer los criterios de reemplazo.

- Reemplazo de los padres: Los descendientes sustituyen a los padres.
- Reemplazo aleatorio: Los individuos a eliminar se escogen aleatoriamente. El número de eliminados queda determinado por el tamaño de la descendencia que a su vez está definido por las tasas de cruce y mutación y el tamaño de la población.
- Reemplazo de los individuos peor adaptados (elitismo): Los individuos a eliminar se eligen aleatoriamente, pero solo entre los que tienen una mala adaptación. Pretende asegurar que aquel o aquellos individuos más adaptados de anteriores generaciones y de la actual sobrevivan y continúen participando en el proceso evolutivo, pasando a la siguiente generación de manera intacta.
- Reemplazo de los individuos de adaptación similar: Los nuevos individuos reemplazan a aquel que tiene un valor de adaptación similar al suyo.

3.7.3. Elitismo

El mecanismo elitista garantiza que no se pierdan las mejores aptitudes e individuos encontrados hasta el momento. Se ha comprobado que este procedimiento es una herramienta muy útil para la convergencia global del algoritmo. Existen dos clases de elitismo, elitismo parcial y elitismo global. El elitismo parcial mantiene una cantidad M de los mejores individuos de una población de tamaño N; el elitismo global conserva los N mejores individuos.

En principio podría parecer que conservar en cada generación los mejores individuos mejoraría la convergencia al óptimo global, pero esto no es cierto. La población se acumularía alrededor de un individuo con buenas cualidades, en comparación con el resto de individuos ya generados, pero no significa que dicho individuo sea el mejor posible, esto se denomina convergencia prematura. Podemos procurar que esto no ocurra centrándonos no solo en la explotación si no también en la exploración. Con lo cual se deduce que una buena estrategia a seguir es el elitismo parcial, conservando a algunos de los mejores individuos ya encontrados para no perder sus buenas cualidades y preservando la diversidad de la población para continuar con la exploración.

Capítulo 4

Algoritmos genéticos aplicados a DLBP

Como se ha explicado en el anterior capítulo el paso principal para resolver un problema con algoritmos genéticos es elegir una buena representación de las soluciones. En esencia, nuestro problema se basa en permutar el orden en el que se retiran las piezas buscando una serie de objetivos y respetando ciertas restricciones. Por ello, resulta sensato elegir una codificación permutacional para nuestros cromosomas. En concreto, nuestros cromosomas serán una lista de números de 1 a n, siendo n el número de piezas, sin repeticiones y ordenados de forma que se respeten las relaciones de precedencia, cada número estará asociado a una pieza en particular.

Nuestra función fitness evaluará los objetivos a minimizar priorizándolos de la siguiente manera, primero el número de máquinas, segundo el balance, seguido de la peligrosidad y, por último, tendrá en cuenta la demanda.

Para la selección es importante elegir un operador que premie aquellos cromosomas con un buen valor de la función fitness pero asegure diversidad en la población. Los métodos basados en la selección proporcional al fitness (selección por ruleta y muestreo estocástico universal) presentan una tasa de crecimiento temprano muy alta y de crecimiento tardío muy baja, lo que podría derivar en la convergencia a un óptimo local. Tanto en los métodos de selección proporcionales como por ranking se requiere información global sobre la población lo que dificulta su implementación y supone un aumento de su complejidad en tiempo. Los metodos proporcionales requiren la suma de todos los valores de la función fitness y los métodos de selección por ranking requieren el acceso a la adaptación de cada individuo perteneciente a la población para realizar el ranking. Sin embargo, la selección por torneo no cuenta con ninguno de estos inconvenientes, además de tener una fácil implementación, se ha comprobado que favorece la diversidad al mismo tiempo que beneficia aquellas soluciones con un mejor fitness. Son estas las razones por las que el método de selección por torneo será el elegido.

Como operadores de cruce se han seleccionado PPX y FRC ya que ambos respetan las restricciones de precedencia. Como operadores de mutación se han escogido los métodos de inserción y de intercambio, estos se han modificado para que no se de a lugar individuos no factibles.

Para el reemplazamiento entre generaciones haremos uso de un elitismo parcial, es decir, se forzará la supervivencia de los mejores individuos en la anterior generación. Se llevará a cabo sustituyendo los m peores individuos, siendo m el número de hijos generados.

En la siguiente sección haremos un estudio sobre qué configuración de parámetros y de operadores es el más adecuado. Los detalles de este se encuentran en el apéndice A.

4.1. Resultados numéricos

Para estudiar las distintas configuraciones se ha ejecutado nuestro algoritmo genético sobre tres conjuntos de datos. Se han realizado 20 ejecuciones por configuración. En nuestro estudio contemplaremos distintas configuraciones en cuanto a parámetros y en cuanto a operadores. Se puede consultar la implementación del algoritmo genético en el apéndice B.

Con respecto a los parámetros estudiaremos distintos valores de la probabilidad de cruce (p_c), de la probabilidad de mutación (p_m), el número de generaciones y el tamaño de la población (N). Para los operadores, como hemos dicho anteriormente, se estudiarán distintos operadores de cruce y de mutación.

La probabilidad de cruce se estudiará para los valores 0,6, 0,7, 0,8 y 0,9, y para la probabilidad de mutación lo estudiaremos para 0,1 y 0,2. Para el número de generaciones se ha realizado un estudio previo con cada conjunto de datos para ver aproximadamente cuándo el algoritmo entra en estado asintótico. El tamaño de población se ha ido ajustando con cada caso de prueba para cada set de datos.

En este análisis no solo se tendrá en cuenta la calidad de los resultados obtenidos por cada configuración, si no que también se valorará el tiempo de ejecución. Se recuerda que la implementación de métodos metaheurísticos tiene como objetivo, aparte de una aproximación al óptimo, rebajar el tiempo de búsqueda.

4.1.1. Instancia de 10 partes

Este conjunto de datos consiste en el desmontaje de un hipotético producto con 10 componentes. En la tabla (4.1) se muestra la información sobre cada tarea. El ciclo de tiempo para cada máquina será $CT = 40s$.

Task	Time	Hazardous	Demand
1	14	No	No
2	10	No	500
3	12	No	No
4	17	No	No
5	23	No	No
6	14	No	750
7	19	Yes	295
8	36	No	No
9	14	No	360
10	10	No	No

Figura 4.1: Instancia de 10 partes. Fuente: (Gupta & McGovern [18])

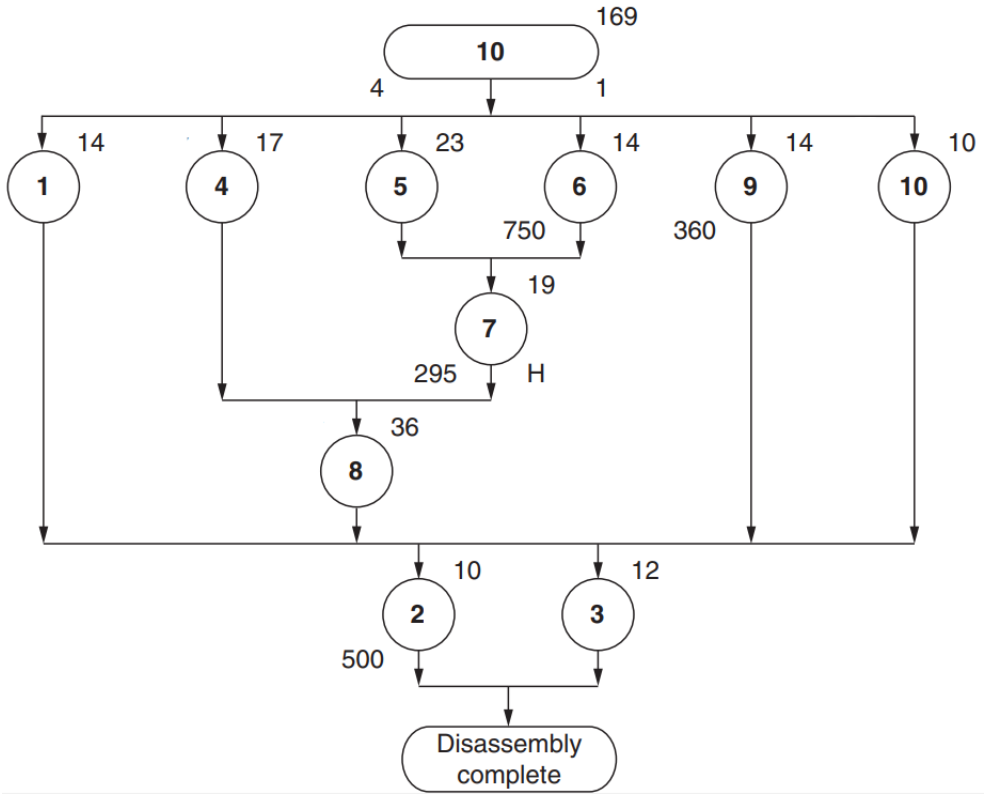

Figura 4.2: Relaciones de precedencia 10 partes. Fuente: (Gupta & McGovern [18])

En la figura (4.2) se representan las relaciones de precedencia entre las piezas, además, en cada pieza, incluye en la esquina superior derecha su tiempo de retirada, en la inferior izquierda su valor de demanda, si tiene, y si tiene algún componente peligroso o no en la esquina inferior derecha. Incluyendo en la parte superior los datos globales, es decir, el tiempo total de retirada, el número de piezas que tienen demanda y las que poseen un factor de peligrosidad.

Al ser de un tamaño pequeño, por medio del método exhaustivo (apéndice D), podemos obtener la solución óptima, en la cual el valor del número de máquinas es 5, $B^* = 211$, $H^* = 4$ y $D^* = 9730$. Estos valores se obtienen a partir de 2 soluciones distintas, es decir, hay 2 óptimos. Todas las configuraciones de nuestro algoritmo proporcionan como el número óptimo de máquinas a 5 en todas las ejecuciones. Sin embargo, no pasa lo mismo con los demás valores a optimizar.

Tras un profundo estudio de las posibles configuraciones, se ha determinado que los mejores valores para los parámetros son $p_c = 0{,}8$, $p_m = 0{,}2$, $N = 75$ y 500 generaciones. Para estos operadores los resultados son muy buenos, en las 20 ejecuciones se ha alcanzado la solución óptima en todas ellas, excepto para la combinación PPX e inserción que se ha alcanzado en 19 ejecuciones. Sin embargo, sí hay una distinción notable en relación al tiempo de ejecución entre dichas configuraciones.

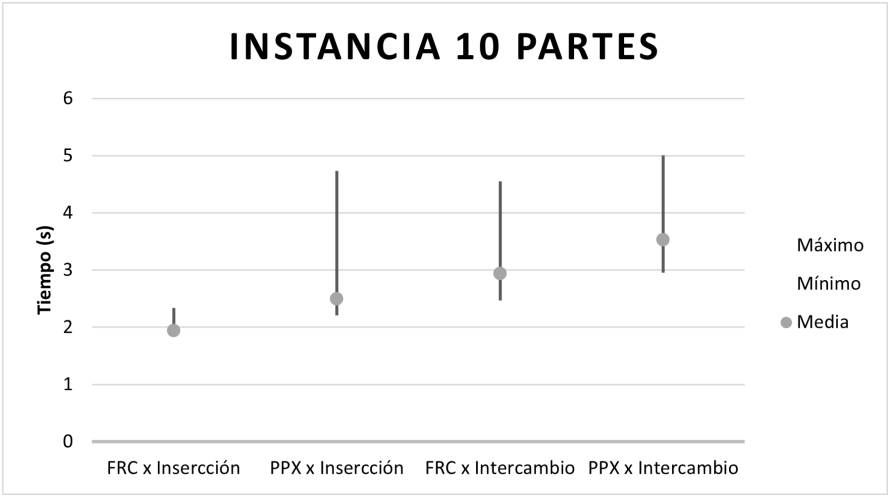

Figura 4.3: Variación del tiempo en las ejecuciones frente a los operadores.

En el gráfico (4.3) se pueden observar dichas diferencias. Determinamos que la mejor configuración para este problema tiene como operador de cruce a FRC y como operador de mutación a la inserción.

Las 2 soluciones óptimas para este problema son:

Pieza	5	10	6	7	9	4	8	1	2	3
Tiempo	23	10	14	19	14	17	36	14	10	12
Puesto de trabajo	1	1	2	2	3	3	4	5	5	5
Peligrosidad	0	0	0	1	0	0	0	0	0	0
Demanda	0	0	750	295	360	0	0	0	500	0

Pieza	10	5	6	7	9	4	8	1	2	3
Tiempo	10	23	14	19	14	17	36	14	10	12
Puesto de trabajo	1	1	2	2	3	3	4	5	5	5
Peligrosidad	0	0	0	1	0	0	0	0	0	0
Demanda	0	0	750	295	360	0	0	0	500	0

Cuadro 4.1: Soluciones óptimas para la instancia de 10 piezas.

Se contrasta las soluciones con las proporcionadas en el libro de Gupta y McGovern [18] del cual hemos obtenido este conjunto de datos. Su algoritmo alcanza en todas las ejecuciones el número óptimo de máquinas y de balance pero, a diferencia del nuestro, obtiene una media para el índice de peligrosidad de $\overline{H} = 4{,}8$ y de demanda de $\overline{D} = 9054$ (obtiene mejores resultados para la demanda que el óptimo, a costa de un peor índice de peligrosidad).

4.1.2. Instancia Teléfono Móvil (25 partes)

En este conjunto de datos el producto a desmontar es un teléfono móvil, consta de 25 componentes. El tamaño del espacio de búsqueda, en esta instancia, asciende a $25! = 1{,}55112 \cdot 10^{25}$, haciendo al método exhaustivo totalmente ineficaz. En la siguiente tabla (4.4) se muestra la información sobre cada tarea y en el diagrama de (4.4) las relaciones de precedencia. El ciclo de tiempo para cada máquina será $CT = 18s$.

Task	Part Removal Description	Time	Hazardous	Demand
1	Antenna	3	Yes	4
2	Battery	2	Yes	7
3	Antenna guide	3	No	1
4	Bolt (type 1) a	10	No	1
5	Bolt (type 1) b	10	No	1
6	Bolt (type 2) 1	15	No	1
7	Bolt (type 2) 2	15	No	1
8	Bolt (type 2) 3	15	No	1
9	Bolt (type 2) 4	15	No	1
10	Clip	2	No	2
11	Rubber seal	2	No	1
12	Speaker	2	Yes	4
13	White cable	2	No	1
14	Red/blue cable	2	No	1
15	Orange cable	2	No	1
16	Metal top	2	No	1
17	Front cover	2	No	2
18	Back cover	3	No	2
19	Circuit board	18	Yes	8
20	Plastic screen	5	No	1
21	Keyboard	1	No	4
22	Liquid crystal display	5	No	6
23	Sub-keyboard	15	Yes	7
24	Internal circuit	2	No	1
25	Microphone	2	Yes	4

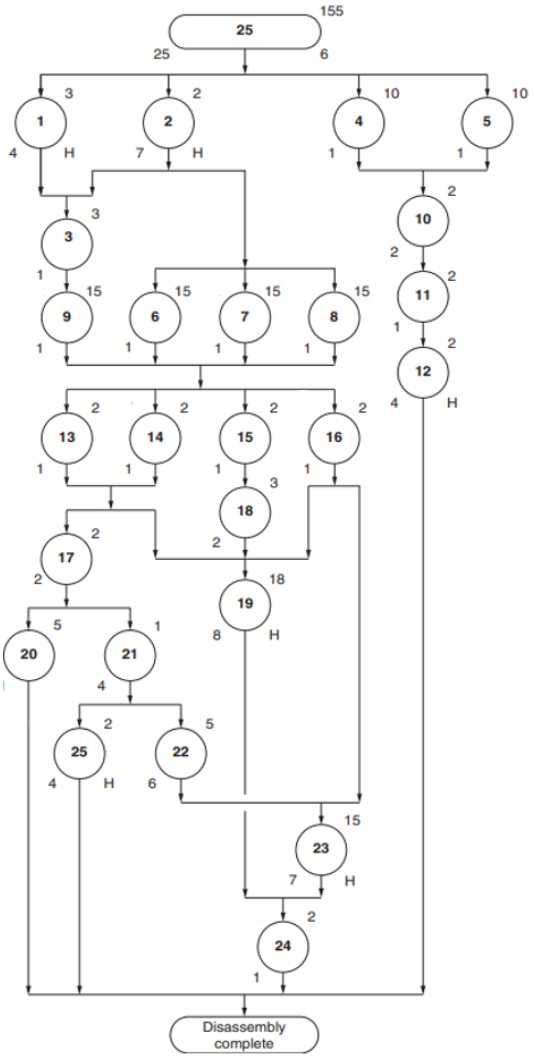

Figura 4.4: Instancia Teléfono Móvil. Fuente: (Gupta & McGovern [18])

Para hacer el análisis de este conjunto de datos primero se estudia qué configuración de parámetros da mejores resultados, observándose que las probabilidades de cruce y mutación 0,8 y 0,1 ofrecen buenos resultados. Y en cuanto al número de generaciones y el tamaño de población se observa que para 1000 iteraciones y entre 50 y 75 individuos se alcanzan buenas soluciones. Como el factor tiempo para las ejecuciones es también importante finalmente el tamaño de la población se fijará en 50.

Para determinar qué operadores son los más adecuados comparamos las diferentes configuraciones con los parámetros fijados anteriormente. Obteniendo la tabla de la figura (4.5):

	Número de máquinas	Balance	Índice de peligrosidad	Índice de demanda
FRC				
Inserción	9,95	77	79,45	915,1
Intercambio	10	80,7	78,55	913,75
PPX				
Inserción	9,95	77,4	77,9	909,55
Intercambio	10	81	78,1	915,5

Figura 4.5: Comparación de los operadores para la instancia Teléfono Móvil.

Se observa que las diferencias entre las configuraciones no son muy destacables, sin embargo, FRC y PPX con el operador de mutación inserción obtienen mejores soluciones. Para determinar qué operador utilizar, volvemos a recurrir al tiempo de ejecución. Se observa en la figura (4.6) que la combinación de FRC e inserción es más rápida.

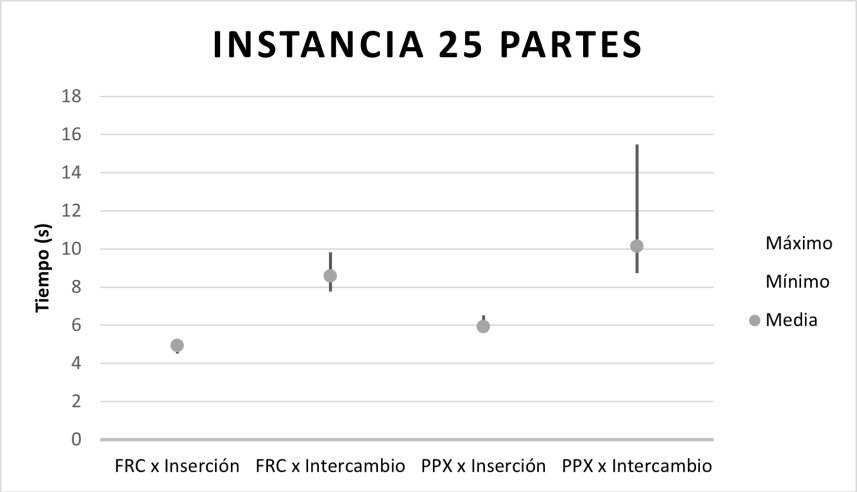

Figura 4.6: Variación del tiempo en las ejecuciones frente a los operadores.

Aunque la mayoría de las soluciones obtenidas hacen uso de 10 máquinas, con una media del balance de $\overline{B} = 80{,}68$ y del índice de peligrosidad y demanda de $\overline{H} = 78{,}78$ y $\overline{D} = 913{,}737$, respectivamente, la configuración elegida previamente ha sido capaz de proporcionar la siguiente solución (cuadro (4.2)), la cual mejora el número de máquinas a 9 y disminuye considerablemente el balance $B = 9$.

Pieza	2	7	1	6	3	8	9	15	18	14	4	16	5	10	13	11
Tiempo	2	15	3	15	3	15	15	2	3	2	10	2	10	2	2	2
Puesto de trabajo	1	1	2	2	3	3	4	4	5	5	5	5	6	6	6	6
Peligrosidad	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Demanda	7	1	4	1	1	1	1	1	2	1	1	1	1	2	1	1

Pieza	19	12	17	21	25	22	20	23	24
Tiempo	18	2	2	1	2	5	5	15	2
Puesto de trabajo	7	8	8	8	8	8	8	9	9
Peligrosidad	1	1	0	0	1	0	0	1	0
Demanda	8	4	2	4	4	6	1	7	1

Cuadro 4.2: Mejor solución encontrada por AG para la instancia Teléfono Móvil.

El libro de Gupta y McGovern [18] obtiene como resultado que todas las soluciones hacen uso de 10 máquinas con un balance $B = 81$, la media del índice de peligrosidad $\overline{H} = 78,40$ y una media del índice de demanda $\overline{D} = 916,20$. Se observa que nuestro algoritmo ha sido capaz de mejorar dichas soluciones.

4.1.3. Instancia Motor de un Automóvil (34 partes)

En este conjunto de datos el producto a desmontar es el motor de un automóvil, consta de 34 componentes. En la tabla (4.7) se muestra la información sobre cada tarea. El ciclo de tiempo para cada máquina será $CT = 726s$.

Unit	Name	Disassembly prerequisite	Disassembly time	Disassembly demand	Risky
1	Alternator brace	3	35	115	0
2	Alternator	3	21	0	0
3	Drive belt		33	392	0
4	Water pump pulley	3	56	543	0
5	Special washer	3	160	0	0
6	Crankshaft pulley	3, 5	11	0	0
7	Damper pulley	3, 5, 6	13	0	0
8	Oil level guide		12	0	0
9	Timing belt upper cover		69	0	0
10	Timing belt lower cover	7	142	0	0
11	Timing belt	9, 10	42	180	0
12	Tensioner spring	9, 10	35	890	0
13	Tensioner	12	34	0	0
14	Crankshaft sprocket, flange	7, 10, 11	71	0	0
15	Camshaft sprocket	9, 11	10	0	0
16	Water pump	3, 4	55	0	0
17	Rocker cover, & gasket	18, 19	10	0	0
18	Intake manifold		196	0	0
19	Exhaust manifold		141	467	0
20	Cylinder head, distributor, camshaft & valves	17, 18, 19	722	0	0
21	Oil filter		62	0	1
22	Oil pan		129	0	1
23	Oil screen	22	38	0	0
24	Oil seal	14	8	0	0
25	Front case	11, 14	81	1013	0
26	Oil pump	25	51	0	1
27	Piston + connecting rod	22, 22, 23	29	0	0
28	connecting rod cup	22, 22, 23	24	0	0
29	Flywheel	28	650	0	0
30	Rear plate	29	46	0	0
31	Bell housing cover	30	34	467	0
32	Oil seal case	30, 31, 34	61	0	0
33	Rear oil seal	30, 32	31	0	0
34	Crankshaft	28	520	0	0
35	Block	* full disassembly			

Figura 4.7: Instancia Motor Automóvil. Basado en los datos de Seidi [23]

Siguiendo los pasos previos, primero estudiaremos qué configuración de parámetros da mejores resultados. Se observa que para la probabilidad de cruce 0,8 y de mutación 0,2, 1500 generaciones y con un tamaño de población de 50 obtenemos buenos resultados. La configuración 0,9 y $0,2$ de tasas de probabilidad y de mutación, respectivamente, junto con 1000 iteraciones y un tamaño de población de entre 40 y 50 individuos también genera buenas soluciones. Para decidir qué configuración escogeremos, volvemos a recurrir al tiempo medio de ejecución, resultando la última configuración junto con 40 individuos la más rápida.

Para determinar qué operadores son los más adecuados comparamos las diferentes configuraciones con los parámetros fijados. Obteniendo la tabla de la figura (4.8):

	Número de máquinas	Balance	Índice de peligrosidad	Índice de demanda
FRC				
Inserción	6	110859,1	34,75	74309,6
Intercambio	6	111236	35,3	74116,65
PPX				
Inserción	6	110939,3	34,95	74538,4
Intercambio	6	112246,7	35,85	73854,75

Figura 4.8: Comparación de los operadores para la instancia Motor Automóvil.

La configuración que mejor resultados obtiene vuelve a ser la combinación FRC junto la inserción, pero al no ser muy destacables las diferencias estudiaremos los tiempos de ejecución.

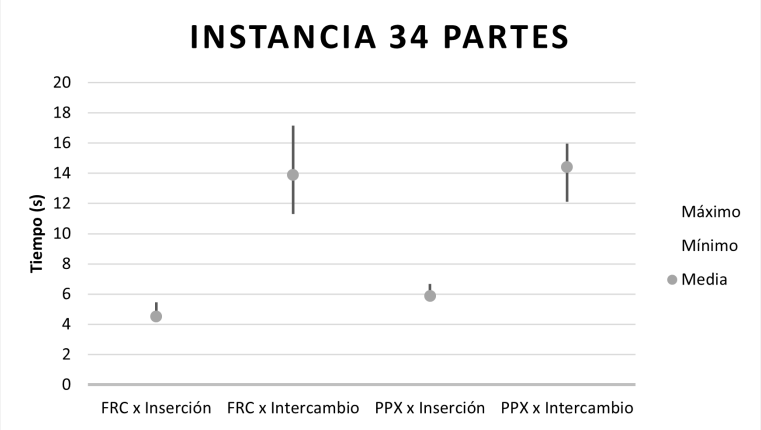

Figura 4.9: Variación del tiempo en las ejecuciones frente a los operadores.

Se confirma en la figura (4.9) que la mejor configuración tiene como operador de cruce a FRC y de mutación a la inserción. Todas las soluciones obtenidas hacen uso de 6 máquinas, con una media del balance de $\overline{B} = 110859{,}1$, del índice de peligrosidad de $\overline{H} = 34{,}75$ y demanda de $\overline{D} = 74309{,}6$. La mejor solución obtenida por la configuración elegida es:

Pieza	21	22	19	3	1	9	2	8	23	28	29	34	30	18	4	31	5
Tiempo	62	129	141	33	35	69	21	12	38	24	650	520	46	196	56	34	160
Máquina	1	1	1	1	1	1	1	1	1	1	2	3	3	4	4	4	4
Peligrosidad	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Demanda	0	0	467	392	115	0	0	0	0	0	0	0	0	0	543	467	0

Pieza	6	7	16	27	17	20	10	11	14	25	26	12	24	32	15	13	33
Tiempo	11	13	55	29	10	722	142	42	71	81	51	35	8	61	10	34	31
Máquina	4	4	4	4	4	5	6	6	6	6	6	6	6	6	6	6	6
Peligrosidad	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Demanda	0	0	0	0	0	0	0	180	0	1013	0	890	0	0	0	0	0

Cuadro 4.3: Mejor solución encontrada por AG para la instancia Motor de un Automóvil.

Para este caso de estudio no se pueden comparar los resultados directamente con la fuente de donde se han obtenido, ya que el artículo de Seidi [23] trata una versión diferente del problema, el Problema de Equilibrado en Líneas de Desmontaje en un ambiente difuso. Los datos han sido modificados para adaptarlos a DLBP. Las alteraciones se han realizado calculando el tiempo medio asociado a cada tarea y la media de los valores de demanda.

Calculando el valor de nuestras funciones objetivo con la solución obtenida en el artículo se obtienen los siguientes resultados: el número de máquinas utilizadas es 6, el valor del balance es $B = 337818$, el índice de peligrosidad es $H = 17$ y el de demanda $D = 39311$. Se observa que el primer valor a optimizar es el mismo, pero sin embargo, en el balance ya encontramos diferencias.

En la siguiente tabla presentamos un pequeño resumen con las configuraciones elegidas para cada caso:

	10 piezas	25 piezas	34 piezas
p_c	0,8	0,8	0,9
p_m	0,2	0,1	0,2
Número de generaciones	500	1000	1000
Tamaño de la población	75	50	40
Operador de cruce	FRC	FRC	FRC
Operador de mutación	Inserción	Inserción	Inserción

4.2. Aplicación de los resultados

En esta sección extrapolaremos los resultados anteriores a otros casos de prueba.

4.2.1. Instancia Ordenador Personal (8 partes)

En este conjunto de datos el producto a desmontar es un ordenador personal, consta de 8 componentes. En la tabla (4.10) se muestra la información sobre cada tarea. El ciclo de tiempo para cada máquina será $CT = 40s$.

Task	Part Removal Description	Time	Hazardous	Demand	Predecessors
1	Cover	14	No	360	n/a
2	Media drive	10	No	500	1
3	Hard drive	12	No	620	1
4	Back plane	18	No	480	7
5	Adaptor cards	23	No	540	1
6	Memory modules (2)	16	No	750	2
7	Power supply	20	Yes	295	8
8	Motherboard	36	No	720	2, 3, 5, 6

Figura 4.10: Instancia Ordenador personal. Fuente: (Gupta & McGovern [18])

Como el tamaño de la instancia es pequeño, aplicaremos la configuración obtenida en la instancia de 10 partes. Obteniéndose la siguiente solución en 1,78s:

Pieza	1	5	2	6	3	8	7	4
Tiempo	14	23	10	16	12	36	20	18
Puesto de trabajo	1	1	2	2	2	3	4	4
Peligrosidad	0	0	0	0	0	0	1	0
Demanda	360	540	500	750	620	720	295	480

Cuadro 4.4: Mejor solución encontrada por AG para la instancia Ordenador Personal.

Al igual que para la instancia de 10 partes, su pequeño tamaño nos permite comprobar si dicha solución es la óptima mediante el método exhaustivo y, efectivamente, lo es. En la solución se hace uso de 4 máquinas, $B^* = 33$, $H^* = 7$ y $D^* = 19265$.

4.2.2. Instancia Portátil (47 partes)

Este conjunto de datos consiste en el desmontaje de un portátil con 47 componentes. En la siguiente tabla (4.11) se muestra la información sobre cada tarea. El ciclo de tiempo para cada máquina será $CT = 110s$.

Task no.	Description	Time	Demand	Hazardous	Predecessor
1	2 bolt group	16	1	No	0
2	4 bolt group	32	1	No	1
3	Hard drive	5	3	Yes	2
4	Hard drive cover	4	1	No	2
5	Battery	5	5	Yes	0
6	Battery cover	6	1	No	5
7	Notebook feet	10	1	No	0
8	2 bolt group	16	1	No	0
9	ZIF connector	5	2	No	8
10	4 bolt group	32	1	No	8
11	CD driver	5	4	Yes	9, 10
12	CD driver cover	6	1	No	10
13	1 bolt group	8	1	No	0
14	RAM cover	2	2	No	13
15	3 bolt group	24	1	No	0
16	Strip cover	7	1	No	15
17	4 bolt group	32	1	No	16
18	Keyboard	6	2	No	17
19	RAM (bottom)	5	7	Yes	14
20	2 bolt group	16	1	No	18
21	Modem	5	6	Yes	20
22	PCI card	5	7	Yes	18
23	4 bolt group	32	1	No	22
24	4 bolt group	32	1	No	22
25	Monitor	16	6	Yes	23, 24
26	11 bolt group	88	1	No	19
27	Heat sink cover	8	2	No	26
28	4 bolt group	32	1	No	27
29	Heat sink	7	4	Yes	28
30	1 bolt group	8	1	No	29
31	Processor	5	7	Yes	30
32	13 bolt group	104	1	No	1, 5, 9, 18
33	Top cover	18	2	No	32
34	Cable	4	2	No	33
35	1 bolt group	8	1	No	33
36	Speaker	9	4	Yes	34, 35
37	9 bolt group	72	1	No	36
38	System board	10	1	No	37
39	9 bolt group	72	1	No	38
40	Fan	10	3	No	39
41	2 bolt group	16	1	No	40
42	Audio board	5	3	Yes	41
43	2 bolt group	16	1	No	36
44	LED board	5	4	Yes	43
45	4 bolt group	32	1	No	44
46	Touchpad	4	3	No	45
47	RAM (top)	5	7	Yes	18

Figura 4.11: Instancia Portátil. Basado en los datos de Kalayci, Hancilar *et. al* [14]

Como el tamaño de la instancia es grande, aplicaremos la configuración obtenida en el conjunto de datos con 34 partes. Obteniéndose la solución del cuadro (4.5) en 6,336s. Se hace uso de 9 máquinas, $B = 1738$, $H = 333$ y $D = 2711$.

Piezas	5	13	14	1	8	15	7	6	16	17	18	22	2	4	20	10	19
Tiempo	5	8	2	16	16	24	10	6	7	32	6	5	32	4	16	32	5
Máquina	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3
Peligrosidad	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
Demanda	5	1	2	1	1	1	1	1	1	1	2	7	1	1	1	1	7

Piezas	47	21	9	11	12	23	26	3	32	24	25	33	34	35	36	27
Tiempo	5	5	5	5	6	32	88	5	104	32	16	18	4	8	9	8
Máquina	3	3	3	3	3	3	4	4	5	6	6	6	6	6	6	6
Peligrosidad	1	1	0	1	0	0	0	1	0	0	1	0	0	0	1	0
Demanda	7	6	2	4	1	1	1	3	1	1	6	2	2	1	4	2

Piezas	43	44	37	28	29	30	31	45	46	38	39	40	41	42
Tiempo	16	5	72	32	7	8	5	32	4	10	72	10	16	5
Máquina	7	7	7	8	8	8	8	8	8	8	9	9	9	9
Peligrosidad	0	1	0	0	1	0	1	0	0	0	0	0	0	1
Demanda	1	4	1	1	4	1	7	1	3	1	1	3	1	3

Cuadro 4.5: Mejor solución encontrada por AG para la instancia Portátil.

Al igual que en la instancia Motor de un Automóvil, se han modificado los datos ya que la fuente también trabaja en un ambiente difuso. En este caso solo era necesario modificar los tiempos de realización de cada tarea. De nuevo no se puede realizar una comparación directa con los resultados de la fuente. La función objetivo para el balance es distinta por lo que no podemos comparar dicho valor, sin embargo, el resto de valores sí son similares. Los resultados proporcionados para las funciones objetivo son 10 máquinas, $H = 336$ y $D = 2840$.

4.3. Aplicación de los resultados al problema SDDLBP

En esta sección aplicaremos los resultados obtenidos para el problema DLBP al problema SDDLBP. Para ello se ha modificado el algoritmo añadiendo una estructura que controle las relaciones de dependencia (apéndice C). Es razonable aplicar las configuraciones obtenidas para la versión DLBP ya que los cambios no son sustanciales.

Contamos con tres conjuntos de datos, la versión SDDLBP de la instancia Ordenador Personal, la de la instancia de 10 partes y la versión de la instancia Motor de un Automóvil.

4.3.1. Instancia Ordenador Personal (8 partes) (SDDLBP)

El producto a desmontar será un Ordenador Personal, las características serán las mismas que las indicadas en la tabla (4.10). La versión SDDLBP de este problema añadirá las siguientes relaciones de dependencia a tener en cuenta (figura 4.12).

Los tiempos añadidos toman los siguientes valores $sd_{23} = 4$, $sd_{32} = 2$, $sd_{56} = 3$ y $sd_{65} = 1$. Se recuerda que sd_{ij} es el tiempo añadido al tiempo de la tarea i si la pieza i se retira antes que j, es decir, $t'_i = t_i + sd_{ij}$.

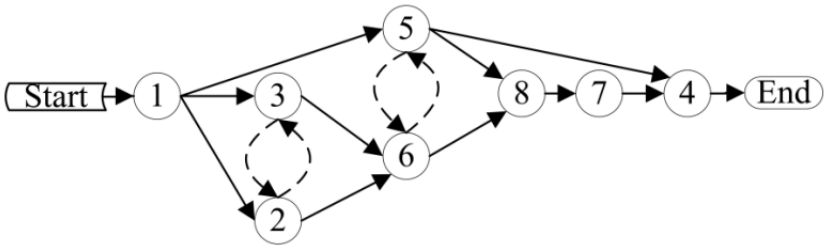

Figura 4.12: Relaciones de precedencia (líneas continuas) y de dependencia (líneas discontinuas) para la instancia Ordenador Personal (SDDLBP). Fuente: (Kalayci [20])

Volvemos a aplicar la misma configuración que para la versión DLBP. Se ha obtenido la siguiente solución (cuadro (4.6)) en 2,774s. Se hace uso de 4 máquinas, $B = 20$, $H = 7$ y $D = 19145$.

Pieza	1	2	3	6	5	8	7	4
Tiempo	14	10	12	16	23	36	20	18
Puesto de trabajo	1	1	1	2	2	3	4	4
Peligrosidad	0	0	0	0	0	0	1	0
Demanda	360	500	620	750	540	720	295	480

Cuadro 4.6: Mejor solución encontrada por AG para la instancia Ordenador Personal (SDDLBP).

Al ser ser una instancia de pequeño tamaño se puede comprobar que la solución que hemos obtenido es óptima por medio del método exhaustivo, el cual se ha modificado para introducir las relaciones de dependencia (apéndice E).

4.3.2. Instancia 10 partes (SDDLBP)

Como se ha explicado anteriormente, este conjunto de datos consiste en el desmontaje de un hipotético producto con 10 componentes. La instancia mantendrá las mismas características expuestas en la tabla (4.1), se añaden las relaciones de dependencia indicadas en el diagrama (4.13). El ciclo de tiempo para cada máquina será $CT = 40s$.

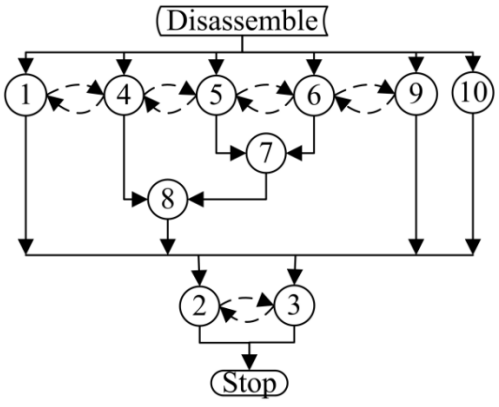

Figura 4.13: Relaciones de precedencia (líneas continuas) y de dependencia (líneas discontinuas) para la instancia 10 partes (SDDLBP). Fuente: (Kalayci [20])

Los tiempos añadidos toman los siguientes valores $sd_{14} = 4$, $sd_{23} = 3$, $sd_{32} = 2$, $sd_{41} = 1$, $sd_{45} = 2$, $sd_{54} = 4$, $sd_{56} = 4$, $sd_{65} = 2$, $sd_{69} = 1$ y $sd_{96} = 3$.

Aplicaremos la configuración que hemos determinado para la instancia de 10 partes, es decir, los parámetros son $p_c = 0{,}8$, $p_m = 0{,}2$, $N = 75$ y 500 generaciones, los operadores de cruce y de mutación serán FRC e inserción respectivamente. Al igual que para el conjunto de datos anterior, ser una instancia de pequeño tamaño nos permite, por medio del método exhaustivo, comprobar que la solución que hemos obtenido es óptima.

Se obtiene la siguiente solución en 3,3435s haciéndose uso de 5 máquinas, $B = 67$, $H = 5$ y $D = 9605$.

Pieza	6	1	5	10	7	4	8	9	2	3
Tiempo	14	14	23	10	19	17	36	14	10	12
Puesto de trabajo	1	1	2	2	3	3	4	3	5	5
Peligrosidad	0	0	0	0	1	0	0	0	0	0
Demanda	750	0	0	0	295	0	0	360	500	0

Cuadro 4.7: Mejor solución encontrada por AG para la instancia 10 partes (SDDLBP).

4.3.3. Instancia Teléfono Móvil (25 partes) (SDDLBP)

Como en las instancias anteriores, el esquema principal del conjunto de datos se mantiene, se puede consultar la información en la tabla (4.4). El ciclo de tiempo para cada máquina es $CT = 18s$. Las relaciones de precedencia y dependencia se muestran en el siguiente diagrama (4.14).

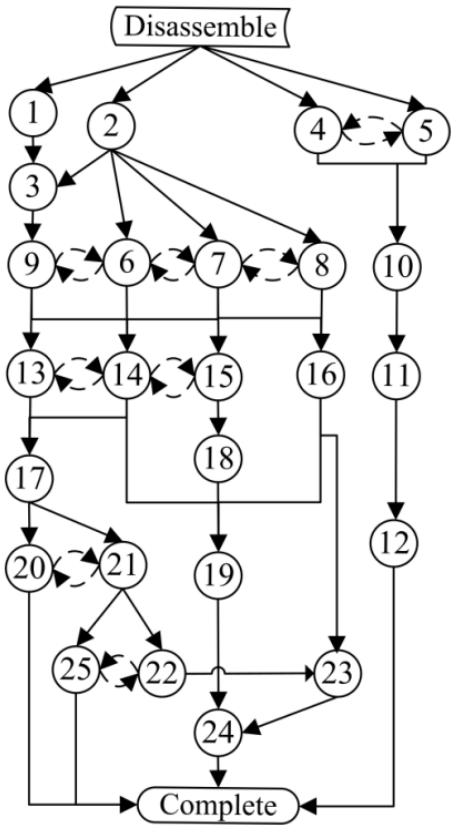

Figura 4.14: Relaciones de precedencia (líneas continuas) y de dependencia (líneas discontinuas) para la instancia Teléfono Móvil (SDDLBP). Fuente: (Kalayci [20])

Los tiempos añadidos toman los siguientes valores $sd_{45} = 1$, $sd_{54} = 2$, $sd_{67} = 2$, $sd_{69} = 1$, $sd_{76} = 1$, $sd_{78} = 2$, $sd_{87} = 1$, $sd_{96} = 2$, $sd_{13-14} = 2$, $sd_{14-13} = 1$, $sd_{14-15} = 1$, $sd_{15-14} = 2$, $sd_{20-21} = 2$, $sd_{21-20} = 1$, $sd_{22-25} = 2$ y $sd_{25-22} = 1$.

Como en los casos anteriores, se aplicará la configuración previamente escogida para la versión DLBP, es decir, los parámetros son $p_c = 0{,}8$, $p_m = 0{,}1$, $N = 50$ y 1000 generaciones, los operadores de cruce y de mutación serán FRC e inserción respectivamente. En la siguiente tabla se muestra la solución obtenida, el tiempo de ejecución ha sido 8,80s. Se hace uso de 10 máquinas, $B = 9$, $H = 80$ y $D = 925$.

Pieza	2	1	5	4	10	3	11	9	6	7	12	8	15	18	13	14
Tiempo	2	3	10	10	2	3	2	15	15	15	2	15	2	3	2	2
Puesto de trabajo	1	1	1	2	2	2	2	3	4	5	6	6	7	7	7	7
Peligrosidad	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Demanda	7	4	1	1	2	1	1	1	1	1	4	1	1	2	1	1

Pieza	17	16	19	20	21	22	25	23	24
Tiempo	2	2	18	5	1	5	2	15	2
Puesto de trabajo	7	7	8	9	9	9	9	10	10
Peligrosidad	0	0	1	0	0	0	1	1	0
Demanda	2	1	8	1	4	6	4	7	1

Cuadro 4.8: Mejor solución encontrada por AG para la instancia Teléfono Móvil (SDDLBP).

Capítulo 5

Conclusiones

En esta memoria se ha abordado un estudio teórico y práctico sobre el Problema de Equilibrado en Líneas de Desmontaje. Dicho estudio se ha llevado a cabo según sigue:

1. Se ha explicado de forma extensa en qué se basa dicho problema, exponiéndose los objetivos y las restricciones a tratar. Se ha estudiado la complejidad, obteniéndose que nuestro problema pertenece a la clase $\mathcal{NP}$-completo, requiriéndose así un método metaheurístico para tratarlo. Se ha planteado una versión del problema en la que se incluye una variante que afecta al tiempo de retirada de las piezas.

2. Para tratar nuestro problema se han escogido los algoritmos genéticos como método metaheurístico. Para poder aplicar estos algoritmos se ha introducido la teoría que les caracteriza, tipos de codificaciones y sus respectivos operadores de selección, cruce y mutación, así como los métodos de reemplazamiento, deteniéndonos en la inicialización y la terminación.

3. Se ha realizado un estudio sobre la aplicación de los algoritmos genéticos a nuestro problema. Mediante un algoritmo programado en PYTHON, se han probado distintas configuraciones sobre tres instancias diferentes. Recolectando una gran cantidad de información sobre estas ejecuciones se ha procedido a hacer un análisis para determinar qué configuración funciona mejor con cada instancia.

4. Para finalizar, una vez obtenidas las configuraciones, se han extrapolado a otros casos de estudio y a la versión antes mencionada, comparándose los resultados alcanzados con los ofrecidos por las fuentes de donde se han obtenido las instancias. Se ha conseguido llegar a la conclusión de que nuestro algoritmo es capaz de llegar a mejores soluciones que las proporcionadas sin un gran coste en tiempo.

Apéndice A

Detalles del estudio numérico

En este apéndice se explicará el procedimiento seguido para el análisis de las configuraciones, se trabajarán las siguientes para cada instancia:

p_c	0,6	0,7	0,8	0,9
p_m	0,1	0,2	—	—
Operador de cruce	FRC	PPX	—	—
Operador de mutación	Inserción	Intercambio	—	—

El número de generaciones se determinará teniendo en cuenta cuándo el algoritmo entra en estado asintótico. Para ello en cada instancia se ejecuta el algoritmo para un alto número de iteraciones y se observa aproximadamente cuándo esto ocurre, a partir de ahí se empezará a ajustar dicho número combinándolo con el tamaño de la población, observando qué valores dan mejores resultados.

Los parámetros, es decir, la probabilidad de cruce y de mutación, el tamaño de la población y el número de generaciones se determinan mediante un proceso empírico. Sobre cada instancia y sobre cada combinación de los operadores de cruce y de mutación se ejecuta veinte veces cada configuración de los parámetros citados, se tiene en cuenta la media y la desviación típica de las cuatro funciones objetivo, el mejor y peor resultado de las veinte ejecuciones, y la media y desviación típica del tiempo de ejecución.

En la tabla (A.1) se muestra un ejemplo ilustrativo de la recolección de datos. No se muestran la mejor y la peor solución por falta de espacio.

Valorando los resultados obtenidos con las diferentes configuraciones tanto de parámetros como de operadores se escoge la combinación más satisfactoria. Si no hay diferencias significativas, es decir, la adaptación media es la misma para las distintas configuraciones, se recurre al tiempo de ejecución. Para determinar si las diferencias entre las configuraciones son o no significativas se ha recurrido al estadístico de Kruskal Wallis, el cual nos permite realizar comparaciones entre distintos grupos, los cuales no tienen que distribuirse necesariamente como una normal (sí deben proceder de poblaciones con la misma distribución). Se presenta ahora un ejemplo ilustrativo de cómo se ha aplicado este estadístico.

p_c	p_m	n_gen	N	c	m	m_maquinas	sd_maquinas	m_balance	sd_balance	m_peligrosidad	sd_peligrosidad	m_demanda	sd_demanda	m_tiempo	sd_tiempo
0,7	0,1	1500	50	FRC	Inserción	10	0	80,7	0,71414284	78,85	2,219797288	914,95	3,9556921	6,62089632	0,2954695
0,7	0,1	1500	50	FRC	Intercambio	10	0	80,9	0,43588989	78,2	0,871779789	914,25	1,08972474	10,7571031	0,77290258
0,7	0,1	1500	50	PPX	Inserción	10	0	81	2,8422E-14	78	0	914	3,4106E-13	8,11061674	0,63486033
0,7	0,1	1500	50	PPX	Intercambio	10	0	81	2,8422E-14	78	0	914,45	0,80467385	12,434299	0,79057466
0,7	0,2	1500	50	PPX	Inserción	9,9	0,3	73,7	21,5710454	78,6	1,562049935	911,45	14,6883457	8,25908107	0,65120092
0,7	0,2	1500	50	FRC	Inserción	10	0	80,2	0,9797959	79,7	2,123676058	911,8	2,8213472	6,39572003	0,30238817
0,7	0,2	1500	50	FRC	Intercambio	10	0	80,9	0,43588989	78,2	0,871779789	914,25	1,08972474	15,5719334	0,99836861
0,7	0,2	1500	50	PPX	Intercambio	10	0	81	2,8422E-14	78	0	914,5	0,80622577	16,3074251	0,86577936
0,8	0,1	1000	50	FRC	Inserción	9,95	0,21794495	77	15,6204994	79,45	2,72901081	915,1	4,7738873	4,92185057	0,20738199
0,8	0,1	1000	50	PPX	Inserción	9,95	0,21794495	77,4	15,6920362	77,9	0,435889894	909,55	19,3971003	5,91432261	0,22205632
0,8	0,1	1000	50	FRC	Intercambio	10	0	80,7	0,71414284	78,55	1,321930407	913,75	1,92028644	8,57373146	0,63150807
0,8	0,1	1000	50	PPX	Intercambio	10	0	81	2,8422E-14	78,1	0,3	915,5	1,91049732	10,1422791	1,68739653
0,8	0,2	1000	50	FRC	Inserción	10	0	80,2	0,9797959	79,85	2,495495943	912,7	3,33316666	5,4350695	0,76149855
0,8	0,2	1000	50	FRC	Intercambio	10	0	80,8	0,6	78,3	0,9	913,5	1,5	12,2795677	0,81866851
0,8	0,2	1000	50	PPX	Intercambio	10	0	80,8	0,6	78,6	1,8	915,05	1,93584607	13,2137711	1,30645566
0,8	0,2	1000	50	PPX	Inserción	10	0	80,8	0,6	78,65	2,104162541	914,95	5,39884247	6,4474926	0,48289796
0,8	0,1	1000	75	FRC	Intercambio	9,95	0,21794495	77	15,6204994	78,8	1,939071943	908,35	19,285422	16,1502668	1,22726563
0,8	0,1	1000	75	FRC	Inserción	10	0	79,9	0,99498744	80,7	2,83019434	914,1	5,61159514	7,69145293	0,55861321
0,8	0,1	1000	75	PPX	Inserción	10	0	80,7	0,71414284	78,6	1,428285686	913,75	1,92028644	11,5482992	1,3165639
0,8	0,1	1000	75	PPX	Intercambio	10	0	80,9	0,43588989	78,3	1,307669683	914,65	1,10792599	17,7423031	1,14180751
0,8	0,2	1000	75	FRC	Inserción	10	0	79,8	0,9797959	80,6	2,39582971	912,7	4,01372645	9,39147828	1,32188574
0,8	0,2	1000	75	FRC	Intercambio	10	0	80,6	0,8	78,7	1,417744688	913,5	2,17944947	21,4859226	1,53071316
0,8	0,2	1000	75	PPX	Inserción	10	0	80,9	0,43588989	78,2	0,871779789	913,75	1,08972474	8,8396874	0,23349428
0,8	0,2	1000	75	PPX	Intercambio	10	0	80,9	0,43588989	78,2	0,871779789	913,8	1,12249722	24,9058522	1,60824031

Figura A.1: Distintas configuraciones de parámetros sobre la instancia Teléfono Móvil

Las hipótesis a decidir son las siguientes:

$$\begin{cases} H_0: \text{ No hay diferencias significativas. La adaptación media es la misma.} \\ H_1: \text{ La adaptación media de al menos uno de los grupos no es la misma.} \end{cases}$$

Este test estadístico se basa en la ordenación de los datos y la creación de un ranking. Luego, mediante la fórmula (A.1) se obtiene el valor del estadístico, el cual se distribuye como una Chi-Cuadrado con $g-1$ grados de libertad, siendo g el número de grupos a comparar. Fijamos el nivel de significación $\alpha = 0{,}05$.

$$K = (N-1)\frac{\sum_{i=1}^{g} n_i(\overline{r}_{i\cdot} - \overline{r})^2}{\sum_{i=1}^{g}\sum_{j=1}^{n_i}(r_{ij} - \overline{r})^2}, \quad \text{siendo} \tag{A.1}$$

- n_i: número de observaciones en el grupo i.
- r_{ij}: es el rango de la observación j del grupo i.
- N: número total de observaciones.
- $\overline{r}_{i\cdot} = \frac{\sum_{j=1}^{n_i} r_{ij}}{n_i}$
- $\overline{r} = (N+1)/2$

Como ejemplo vamos a utilizar la instancia Motor de un Automóvil. En la memoria se explica que la configuración con una probabilidad de cruce de 0,8 y de mutación de 0,2, 1500 generaciones con un tamaño de población de 50; y que la configuración 0,9 y 0, 2 de tasas de probabilidad y de mutación, respectivamente, junto con 1000 iteraciones y un tamaño de población de entre 40 y 50 individuos, generan buenos resultados. Veamos que entre estas tres configuraciones no hay diferencias significativas. Se valoraran 12 grupos (figura A.2) ya que cada configuración de parámetros consta de 4 configuraciones de operadores.

Grupo 1	0.8	0.2	1500	50	PPX	intercambio
Grupo 2						inserción
Grupo 3					FRC	intercambio
Grupo 4						inserción
Grupo 5	0.9	0.2	1000	40	PPX	intercambio
Grupo 6						inserción
Grupo 7					FRC	intercambio
Grupo 8						inserción
Grupo 9	0.9	0.2	1000	50	PPX	intercambio
Grupo 10						inserción
Grupo 11					FRC	intercambio
Grupo 12						inserción

Figura A.2: Grupos a comparar con Kruskal Wallis

Aplicando la fórmula a las 20 soluciones por cada grupo que se han obtenido, el estadístico toma el valor $K = 15,89836901$, consultando la tabla de distribución de una Chi-Cuadrado con 11 grados de libertad, se obtiene el p-valor= 0,1449. Luego, p-valor $> \alpha$, por tanto, se acepta la hipótesis nula (H_0), es decir, no hay diferencias significativas entre los grupos. Por lo que para decidir qué configuración se escoge, se recurre al tiempo medio de ejecución, resultando la configuración del grupo 8 la seleccionada.

Apéndice B

Implementación algoritmo genético en PYTHON (DLBP)

```
import random
import numpy as np
import copy

"""
  n: entero. Número de piezas.
  t: lista de enteros. Indica el tiempo de ejecución de cada tarea.
  h: lista de enteros. Indica la peligrosidad de cada tarea.
     (binaro: 1 -> peligroso   0 -> no peligroso)
  d: lista de enteros. Indica la demanda de cada tarea.
  M: diccionario.
     Clave: entero. Indica la pieza.
     Valor: lista de enteros. Indica qué piezas preceden a la pieza indicada en
            la clave.
     Si una pieza no aparece en el diccionario, a esa pieza no le precede
     ninguna otra.
  CT: entero. Ciclo de tiempo de cada máquina.
  p_c: real. Probabilidad de cruce.
  p_m: real. Probabilidad de mutación.
  n_gen: entero. Número de generaciones.
  N: entero. Tamaño de la población.
  op_cruce: cadena de caracteres. Indica el operador de cruce: 'FRC' o 'PPX'.
  op_mutacion: cadena de caracteres. Indica el operador de mutación:
               'insercion' o 'intercambio'.
  n_maquina: Entero. Número de máquinas.
  B: entero. Balance.
  P: entero. Peligrosidad.
  D: entero. Demanda.
  poblacion: lista de individuos.
  individuo: [n_maquina,B,P,D,secuencia_individuo]
  secuencia_individuo: lista de enteros. Indica el orden de retirada de las
                       piezas.
"""

"""
 #Instancia Ordenador Personal (8 partes)

  M={2:[1],3:[1],4:[7],5:[1],6:[2],7:[8],8:[2,3,5,6]}
  t=[14,10,12,18,23,16,20,36]
  h=[0,0,0,0,0,0,1,0]
  d=[360,500,620,480,540,750,295,720]
  CT=40

```

```
    #Instancia 10 partes

     M={2:[1, 8, 9, 10],3:[1, 8, 9, 10],7:[5, 6],8:[4, 7]}
     t=[14,10,12,17,23,14,19,36,14,10]
     h=[0,0,0,0,0,0,1,0,0,0]
     d=[0,500,0,0,0,750,295,0,360,0]
     CT=40
"""
    #Instancia Teléfono Móvil (25 partes)

M=({3:[1, 2],6:[2],7:[2],8:[2],9:[3],10:[4, 5],11:[10],12:[11],
        13:[9, 6, 7, 8],14:[9, 6, 7, 8],15:[9, 6, 7, 8],16:[9, 6, 7, 8],
        17:[13, 14],18:[15],19:[13, 14, 15, 16, 18], 20:[17], 21:[17],22:[21],
        23:[16, 22],24:[19, 23],25:[21]})
t=[3,2,3,10,10,15,15,15,15,2,2,2,2,2,2,2,2,3,18,5,1,5,15,2,2]
h=[1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,1]
d=[4,7,1,1,1,1,1,1,1,2,1,4,1,1,1,1,2,2,8,1,4,6,7,1,4]
CT=18
"""
    #Instancia Motor de un Automóvil (34 partes)

     M=({1:[3],2:[3],4:[3],5:[3],6:[3, 5],7:[3, 5, 6],10:[7],11:[9, 10],
          12:[9, 10],13:[12],14:[7, 10, 11], 15:[9, 11],16:[3, 4],17:[18, 19],
          20:[17, 18, 19],23:[22],24:[14],25:[11, 14],26:[25],27:[22, 23],
          28:[22, 23],29:[28],30:[29],31:[30],32:[30, 31, 34],33:[30, 32],34:[28]})
     t=[35,21,33,56,160,11,13,12,69,142,42,35,34,71,10,55,10,196,141,722,62,129,
        38,8,81,51,29,24,650,46,34,61,31,520]
     h=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0]
     d=[115,0,392,543,0,0,0,0,0,0,180,890 ,0,0,0,0,0,0,467,0,0,0,0,0,1013,0,0,0,0,
        0,467,0,0,0]
     CT=726

    #Instancia Pórtatil (47 partes)
     M=({2:[1],3:[2],4:[2],6:[5],9:[8],10:[8],11:[9, 10],12:[10],14:[13],16:[15],
          17:[16],18:[17],19:[14],20:[18],21:[20],22:[18],23:[22],24:[22],
          25:[23, 24],26:[19],27:[26],28:[27],29:[28],30:[29],31:[30],
          32:[1, 5, 9, 18],33:[32],34:[33],35:[33],36:[34, 35],37:[36],38:[37],
          39:[38],40:[39],41:[40],42:[41],43:[36],44:[43],45:[44],46:[45],47:[18]})
     t=[16,32,5,4,5,6,10,16,5,32,5,6,8,2,24,7,32,6,5,16,5,5,32,32,16,88,8,32,7,8,
        5,104,18,4,8,9,72,10,72,10,16,5,16,5,32,4,5]
     h=[0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,1,0,0,1,0,0,0,1,0,1,0,0,0,0,1,0,
        0,0,0,0,1,0,1,0,0,1]
     d=[1,1,3,1,5,1,1,1,2,1,4,1,1,2,1,1,1,2,7,1,6,7,1,1,6,1,2,1,4,1,7,1,2,2,1,4,1,
        1,1,3,1,3,1,4,1,3,7]
     CT=110
"""


def generar_pob_inic(n,N,M,t,h,d,CT): #Genera la población inicial
    """
    Genera N individuos factibles de manera aleatoria. Con la función fitness
    calcula la adaptación de cada nuevo individuo. Si algún individuo generado
    ya está en la población se fuerza su salida en la próxima generación
    asociándole el mayor número de máquinas posible, es decir, el número de
    piezas y sumándole 1.
    El diccionario precedencia representa las relaciones de precedencia según
    se retiran piezas.
    Si la lista asociada a una pieza no es vacía no se puede retirar la pieza.
    Si se retira una pieza se elimina dicha pieza de todas las listas del
    diccionario precedencia liberando así a las piezas precedidas por ella.
    """
    poblacion = []
```

```
    mejor_solucion=[n+1,0,0,0,[]]
    for i in range(N):
        #Lista de piezas que faltan por retirar.
        piezas = list(range(1,n+1))
        #Relaciones de precedencia según se retiran piezas.
        precedencia = copy.deepcopy(M)
        individuo=[]
        while piezas!=[]:
            pieza=random.choice(piezas)
            #Miramos si las piezas que preceden a 'pieza' ya se han retirado.
            if (not pieza in precedencia.keys()) or precedencia[pieza]==[]:
                individuo.append(pieza)
                piezas.remove(pieza)
                #Liberamos las piezas que estaban precedidas por 'pieza'.
                for j in precedencia.keys():
                    if pieza in precedencia[j]:
                        precedencia[j].remove(pieza)
        #Calculamos la adaptación del nuevo individuo.
        (B,P,D,m)=fitness(individuo,n,t,h,d,CT)
        #Comprobar si el individuo está duplicado.
        if [m,B,P,D,individuo] in poblacion:
            #Forzamos a que abandone la población en la siguiente generación
            #(no puede haber más máquinas que piezas).
            poblacion.append([n+1,B,P,D,individuo])
        else:
            poblacion.append([m,B,P,D,individuo])
            if [m,B,P,D]<mejor_solucion[0:4]:
                mejor_solucion=[m,B,P,D,individuo]
    return (poblacion,mejor_solucion)


def fitness(individuo,n,t,h,d,CT):
    """
    Calcula la adaptación del individuo.
    n_maquina: número de máquinas necesarias.
    B: medida sobre cuán balanceada está la solución.
    P: índice de peligrosidad.
    D: índice de demanda
    """
    tiempo=np.array([0])
    n_maquina=0
    P=0
    D=0
    for pieza in individuo:
        #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
        if tiempo[n_maquina]+t[pieza-1]<=CT:
            #Se añade la pieza.
            tiempo[n_maquina]+=t[pieza-1]
        else:
            #Como no cabe, se añade una nueva máquina.
            tiempo=np.append(tiempo,t[pieza-1])
            n_maquina+=1
    n_maquina+=1
    #Se calcula el balance.
    B=np.sum((CT-tiempo)**2)
    #Se calcula el índice de peligrosidad y de la demanda.
    for i in range(n):
        if h[individuo[i]-1]==1:
            P+=i+1
        D+=(i+1)*d[individuo[i]-1]
    return (B,P,D,n_maquina)
```

```
169 def seleccion_torneo(poblacion,N,p_c):
    """
171     Se realiza la selección de los padres mediante la selección por torneo
    determinístico.
173     """
    padres=[]
175     n_padres=round(N*p_c)
    if n_padres%2==1: #Número par de padres.
177         n_padres-=1
    for i in range(n_padres):
179         #Se generan dos números aleatorios que seleccionan que individuos se
        #enfrentan en el torneo.
181         a=random.randint(0,N-1)
        b=random.randint(0,N-1)
183         #Se selecciona el individuo con mejor adaptación.
        if poblacion[a][0:4]<=poblacion[b][0:4]:
185             padre=poblacion[a]
        else:
187             padre=poblacion[b]
        padres.append(padre[4])
189     return (padres,n_padres)

191
def cruce_PPX(padres,n_padres,n):
193     """
    Se realiza el cruce de la siguiente manera:
195       1. Se escogen dos progenitores.
      2. Se genera aleatoriamente una máscara consistente en 1s y 2s generados
197       aleatoriamente, indicando de qué padre se debe extraer la información.
      3. Se procede de la siguiente manera:
199
        padre1:  1 3 2 6 5 8 7 4
201       padre2:  1 2 3 5 6 8 7 4
        máscara: 2 2 1 2 1 1 1 2
203       ---------------------------
        hijo:
205
      Se elige el primer elemento disponible del padre indicado.
207
        padre1:  1 3 2 6 5 8 7 4
209       padre2:  1 2 3 5 6 8 7 4    Se coge el 1 2 del segundo padre.
        máscara: 2 2 1 2 1 1 1 2
211       ---------------------------
        hijo:    1 2
213
      Se tachan de ambos padres las piezas ya seleccionadas.
215
        padre1:  X 3 X 6 5 8 7 4    Se coge el 3 del primer padre.
217       padre2:  X X 3 5 6 8 7 4
        máscara: 2 2 1 2 1 1 1 2
219       ---------------------------
        hijo:    1 2 3
221
      Se tachan de ambos padres las piezas ya seleccionadas.
223
        padre1:  X X X 6 5 8 7 4
225       padre2:  X X X 5 6 8 7 4    Se coge el 5 del segundo padre.
        máscara: 2 2 1 2 1 1 1 2
227       ---------------------------
        hijo:    1 2 3 5
229
```

```
        Se tachan de ambos padres las piezas ya seleccionadas.
231
          padre1:  X X X 6 X 8 7 4    Se coge el 6, 8, y 7 del primer padre.
          padre2:  X X X X 6 8 7 4
233       máscara: 2 2 1 2 1 1 1 2
          ---------------------------
          hijo:    1 2 3 5 6 8 7
235
        Se tachan de ambos padres las piezas ya seleccionadas.
237
          padre1:  X X X X X X X 4
          padre2:  X X X X X X X 4    Se coge el 4 del segundo padre.
239       máscara: 2 2 1 2 1 1 1 2
          ---------------------------
          hijo:    1 2 3 5 6 8 7 4
241
        De esta forma se respetan las relaciones de precedencia.
        De cada par de progenitores se genera un par de hijos.
    """
243     hijos=[]
    for i in range(0,n_padres,2):
245         padre1_1=copy.deepcopy(padres[i])
            padre2_1=copy.deepcopy(padres[i+1])
247         padre1_2=copy.deepcopy(padres[i])
            padre2_2=copy.deepcopy(padres[i+1])
249         hijo1=[]
            hijo2=[]
251         for j in range(n):
                mask1=random.randint(1,2)
253             if mask1==1:
                    hijo1.append(padre1_1[0])
255                 padre2_1.remove(padre1_1[0])
                    padre1_1.pop(0)
257             else:
                    hijo1.append(padre2_1[0])
259                 padre1_1.remove(padre2_1[0])
                    padre2_1.pop(0)
261             mask2=random.randint(1,2)
                if mask2==1:
263                 hijo2.append(padre1_2[0])
                    padre2_2.remove(padre1_2[0])
265                 padre1_2.pop(0)
                else:
267                 hijo2.append(padre2_2[0])
                    padre1_2.remove(padre2_2[0])
269                 padre2_2.pop(0)
            hijos.append(hijo1)
271         hijos.append(hijo2)
    return hijos
273

275 def cruce_FRC(padres,n_padres,n):
    """
277     Se realiza el cruce de la siguiente manera:
        1. Se escogen dos progenitores.
279     2. Se generan dos números aleatorios que determinaran los tres fragmentos
        en los que se dividirán los padres.
281     3. El primer hijo mantiene la cabeza y la cola del primer padre. La parte
        del medio se rellena añadiendo todos los genes que faltan en el orden que
283     marca el segundo padre. Para el segundo hijo los padres intercambiarán
        los papeles.
285 """
```

```
    hijos=[]
293 for i in range(0,n_padres,2):
        padre1_1=copy.deepcopy(padres[i])
295     padre2_1=copy.deepcopy(padres[i+1])
        padre1_2=copy.deepcopy(padres[i])
297     padre2_2=copy.deepcopy(padres[i+1])
        a=random.randint(0, n-1)
299     b=random.randint(0, n-1)
        hijo1=[]
301     cola1=[]
        hijo2=[]
303     cola2=[]
        if a<=b:
305         for j in range(a):
                hijo1.append(padre1_1[j])
307             padre2_1.remove(padre1_1[j])
                hijo2.append(padre2_2[j])
309             padre1_2.remove(padre2_2[j])
            for j in range(b,n):
311             cola1.append(padre1_1[j])
                padre2_1.remove(padre1_1[j])
313             cola2.append(padre2_2[j])
                padre1_2.remove(padre2_2[j])
315         hijo1=hijo1+padre2_1+cola1
            hijo2=hijo2+padre1_2+cola2
317     else:
            for j in range(b):
319             hijo1.append(padre1_1[j])
                padre2_1.remove(padre1_1[j])
321             hijo2.append(padre2_2[j])
                padre1_2.remove(padre2_2[j])
323         for j in range(a,n):
                cola1.append(padre1_1[j])
325             padre2_1.remove(padre1_1[j])
                cola2.append(padre2_2[j])
327             padre1_2.remove(padre2_2[j])
            hijo1=hijo1+padre2_1+cola1
329         hijo2=hijo2+padre1_2+cola2
        hijos.append(hijo1)
331     hijos.append(hijo2)
    return hijos
333

335 def precedencia(individuo,gen1,gen2,M,n):
    """
337 Se encarga de comprobar si un individuo respeta las relaciones de
    precedencia.
339 """
    prec = copy.deepcopy(M)
341 #Relaciones de precedencia según se retiran piezas.
    ind=copy.deepcopy(individuo)
343 pieza1=individuo[gen1]
    pieza2=individuo[gen2]
345 ind[gen1]=pieza2
    ind[gen2]=pieza1
347 #Se comprueba que ind es válido.
    for i in range(n):
349     pieza=ind[i]
        #No incumple precedencia.
351     if (not (pieza in prec.keys())) or prec[pieza]==[]:
            #Liberamos las piezas que estaban precedidas por 'pieza'.
353         for j in prec.keys():
```

```
                if pieza in prec[j]:
355                 prec[j].remove(pieza)
        #Sí incumple precedencia.
357     else:
            return False
359 return True

361
def mutacion_intercambio(hijos,n,M,p_m):
363 """
    Muta a algunos individuos del conjunto hijos intercambiando dos genes.
365 Con la función precedencia se asegura que el individuo resultante de la
    mutación sea válido.
367 """
    hijos_mut=[]
369 for hijo in hijos:
        a=random.random()
371     if a<=p_m: #Se elige a 'hijo' para ser mutado.
            genes=list(range(n)) #Lista de genes a elegir.
373         gen1=random.choice(genes) #Gen a mutar.
            genes.remove(gen1)
375         b=False #Indica si se ha encontrado un intercambio válido.
            while (not b) and genes!=[]: #Se busca un intercambio válido.
377             gen2=random.choice(genes)
                genes.remove(gen2)
379             #Miramos que no se incumpla ninguna relación de precedencia.
                b=precedencia(hijo,gen1,gen2,M,n)
381         if b:
                mut=hijo[gen1]
383             hijo[gen1]=hijo[gen2]
                hijo[gen2]=mut
385     hijos_mut.append(hijo)
    return hijos_mut
387

389 def mutacion_insercion(hijos,n,M,p_m):
    """
391 La mutación se realiza eligiendo dos alelos aleatoriamente y colocando el
    segundo después del primero, si se incumple las relaciones de precedencia
393 se situará en la última posición donde estas sí se cumplian.
    """
395 hijos_mut=[]
    for hijo in hijos:
397     a=random.random()
        if a<=p_m:
399         genes=list(range(n))
            gen1=random.choice(genes) #Gen a mutar.
401         genes.remove(gen1)
            gen2=random.choice(genes)
403         pieza2=hijo[gen2]
            #Indica que la configuración del cromosoma sigue respetando las
405         #relaciones de precedencia.
            b=True
407         if gen1<gen2:
                while b and gen1!=gen2-1:
409                 pieza3=hijo[gen2-1]
                    if (pieza2 in M.keys()) and (pieza3 in M[pieza2]):
411                 #Pieza3 precede a pieza2 luego no puede retroceder más.
                        b=False
413                 else:
                        hijo[gen2-1]=pieza2
415                     hijo[gen2]=pieza3
```

```
                    gen2-=1
417             else:
                    while b and gen1!=gen2:
419                     pieza3=hijo[gen2+1]
                        if (pieza3 in M.keys()) and (pieza2 in M[pieza3]):
421                     #Pieza3 precede a pieza2 y por tanto no puede avanzar más.
                            b=False
423                     else:
                            hijo[gen2+1]=pieza2
425                         hijo[gen2]=pieza3
                        gen2+=1
427         hijos_mut.append(hijo)
        return hijos_mut
429

431 def reemplazo_elitista(poblacion,hijos,N,n,t,h,d,CT):
        """
433     Se encarga de reemplazar parte de los individuos de la anterior generación
        por los hijos.
435     Se reemplazan los peor adaptados.
        """
437     hijos_fit=[]
        mejor_hijo=[n+1,0,0,0,[]]
439     #Se calcula el fitness de los hijos.
        for hijo in hijos:
441         (B,P,D,n_maquina)=fitness(hijo,n,t,h,d,CT)
            hijos_fit.append([n_maquina,B,P,D,hijo])
443         if [n_maquina,B,P,D]<mejor_hijo[0:4]:
                mejor_hijo=[n_maquina,B,P,D,hijo]
445     n_hijos=len(hijos)
        for i in range(n_hijos):
447         #Eliminamos de la población los peor adaptados.
            poblacion.remove(max(poblacion))
449     for hijo in hijos_fit:
            #Comprobamos si el individuo está duplicado.
451         if hijo in poblacion:
                #Se fuerza a que abandone la población en la siguiente generación
453             #(no puede haber más máquinas que piezas).
                hijo[0]=n+1
455             poblacion.append(hijo)
            else:
457             poblacion.append(hijo)
        return (poblacion,mejor_hijo)
459

461 def evaluacion(mejor_solucion,mejor_hijo):
        """
463     Se encarga de encontrar la nueva mejor solución.
        """
465     if mejor_hijo[0:4]<mejor_solucion[0:4]:
            mejor_solucion=mejor_hijo
467     return mejor_solucion

469
    def informacion(individuo,t,CT):
471     """
        Proporciona la distribución de las piezas en las máquinas y el tiempo libre
473     de cada una.
        """
475     tiempo=np.array([0])
        maquinas=[[]]
477     n_maquina=0
```

```
        for pieza in individuo:
479         #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
            if tiempo[n_maquina]+t[pieza-1]<=CT:
481             #Se añade la pieza.
                tiempo[n_maquina]+=t[pieza-1]
483             #Se registra en qué máquina se retira 'pieza'.
                maquinas[n_maquina].append(pieza)
485         else:
                #Como no cabe, se añade una nueva máquina.
487             tiempo=np.append(tiempo,t[pieza-1])
                #Se registra en qué máquina se retira 'pieza'.
489             maquinas.append([pieza])
                n_maquina+=1
491     #Se calcula el tiempo libre de cada máquina.
        tiempo=list(np.array(CT-tiempo))
493     return (maquinas,tiempo)

495
    def main(op_cruce,op_mutacion,M,t,h,d,CT,N,n_gen,p_c,p_m):
497     n=len(t)
        (poblacion,mejor_solucion)=generar_pob_inic(n,N,M,t,h,d,CT)
499     for i in range(n_gen):
            (padres,n_padres)=seleccion_torneo(poblacion,N,p_c)
501         if op_cruce=='FRC':
                hijos=cruce_FRC(padres,n_padres,n)
503         else:
                hijos=cruce_PPX(padres,n_padres,n)
505         if op_mutacion=='insercion':
                hijos=mutacion_insercion(hijos,n,M,p_m)
507         else:
                hijos=mutacion_intercambio(hijos,n,M,p_m)
509         (poblacion,mejor_hijo)=reemplazo_elitista(poblacion,hijos,N,n,t,h,d,CT)
            mejor_solucion=evaluacion(mejor_solucion,mejor_hijo)
511     (distrib_maquinas,tlibre_maquina)=informacion(mejor_solucion[4],t,CT)
        return (mejor_solucion,distrib_maquinas,tlibre_maquina)
```

alg_genetico.py

Apéndice C

Implementación algoritmo genético en PYTHON (SDDLBP)

```
import random
import numpy as np
import copy

"""
  n: entero. Número de piezas.
  t: lista de enteros. Indica el tiempo de ejecución de cada tarea.
  dependencia: diccionario.
               Clave: entero i. Indica la pieza.
               Valor: diccionario.
                      Clave: entero j. Indica que hay una relación de
                             dependencia entre i y j, señalando que si se
                             retira i antes que j se deberá añadir un tiempo
                             adicional a la tarea de retirar i.
                      Valor: entero. Cantidad añadida.
               {i:{j:n}} si se retira i antes que j entonces t[i]=t[i]+n.
  h: lista de enteros. Indica la peligrosidad de cada tarea.
     (binaro: 1 -> peligroso   0 -> no peligroso)
  d: lista de enteros. Indica la demanda de cada tarea.
  M: diccionario.
     Clave: entero. Indica la pieza.
     Valor: lista de enteros. Indica qué piezas preceden a la pieza indicada en
            la clave.
     Si una pieza no aparece en el diccionario, a esa pieza no le precede
     ninguna otra.
  CT: entero. Ciclo de tiempo de cada máquina.
  p_c: real. Probabilidad de cruce.
  p_m: real. Probabilidad de mutación.
  n_gen: entero. Número de generaciones.
  N: entero. Tamaño de la población.
  op_cruce: cadena de caracteres. Indica el operador de cruce: 'FRC' o 'PPX'.
  op_mutacion: cadena de caracteres. Indica el operador de mutación:
               'insercion' o 'intercambio'.
  n_maquina: Entero. Número de máquinas.
  B: entero. Balance.
  P: entero. Peligrosidad.
  D: entero. Demanda.
  poblacion: lista de individuos.
  individuo: [n_maquina,B,P,D,secuencia_individuo]
  secuencia_individuo: lista de enteros. Indica el orden de retirada de las
                       piezas.
"""

```

```
"""
#Instancia Ordenador Personal (8 partes)

 M={2:[1],3:[1],4:[7],5:[1],6:[2],7:[8],8:[2,3,5,6]}
 t=[14,10,12,18,23,16,20,36]
 dependencia={2:{3:4},3:{2:2},5:{6:3},6:{5:1}}
 h=[0,0,0,0,0,0,1,0]
 d=[360,500,620,480,540,750,295,720]
 CT=40

#Instancia 10 partes

 M={2:[1, 8, 9, 10],3:[1, 8, 9, 10],7:[5, 6],8:[4, 7]}
 t=[14,10,12,17,23,14,19,36,14,10]
 dependencia=({1:{4:4},2:{3:3},3:{2:2},4:{1:1,5:2},5:{4:4,6:4},6:{5:2,9:1},
              9:{6:3}})
 h=[0,0,0,0,0,0,1,0,0,0]
 d=[0,500,0,0,0,750,295,0,360,0]
 CT=40

#Instancia Teléfono Móvil (25 partes)

 M=({3:[1, 2],6:[2],7:[2],8:[2],9:[3],10:[4, 5],11:[10],12:[11],
     13:[9, 6, 7, 8],14:[9, 6, 7, 8],15:[9, 6, 7, 8],16:[9, 6, 7, 8],
     17:[13, 14],18:[15],19:[13, 14, 15, 16, 18], 20:[17], 21:[17],22:[21],
     23:[16, 22],24:[19, 23],25:[21]})
 t=[3,2,3,10,10,15,15,15,15,2,2,2,2,2,2,2,2,3,18,5,1,5,15,2,2]
 dependencia=({4:{5:1},5:{4:2},6:{7:2,9:1},7:{6:1,8:2},8:{7:1},9:{6:2},
              13:{14:2},14:{13:1,15:1},15:{14:2},20:{21:2},21:{20:1},
              22:{25:2},25:{22:1}})
 h=[1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,1]
 d=[4,7,1,1,1,1,1,1,1,2,1,4,1,1,1,1,2,2,8,1,4,6,7,1,4]
 CT=18
"""


def generar_pob_inic(n,N,M,t,dependencia,h,d,CT): #generar poblacion inicial
    """
    Genera N individuos factibles de manera aleatoria. Con la función fitness
    calcula la adaptación de cada nuevo individuo. Si algún individuo generado
    ya está en la población se fuerza su salida en la próxima generación
    asociándole el mayor número de máquinas posible, es decir, el número de
    piezas y sumándole 1.
    El diccionario precedencia representa las relaciones de precedencia según
    se retiran piezas.
    Si la lista asociada a una pieza no es vacía no se puede retirar la pieza.
    Si se retira una pieza se elimina dicha pieza de todas las listas del
    diccionario precedencia liberando así a las piezas precedidas por ella.
    """
    poblacion = []
    mejor_solucion=[n+1,0,0,0,[]]
    for i in range(N):
        #Lista de piezas que faltan por retirar.
        piezas = list(range(1,n+1))
        #Relaciones de precedencia según se retiran piezas.
        precedencia = copy.deepcopy(M)
        individuo=[]
        while piezas!=[]:
            pieza=random.choice(piezas)
            #Miramos si las piezas que preceden a 'pieza' ya se han retirado.
            if (not pieza in precedencia.keys()) or precedencia[pieza]==[]:
```

```
                    individuo.append(pieza)
107                 piezas.remove(pieza)
                    #Liberamos las piezas que estaban precedidas por 'pieza'.
109                 for j in precedencia.keys():
                        if pieza in precedencia[j]:
111                         precedencia[j].remove(pieza)
            #Calculamos la adaptación del nuevo individuo.
113         (B,P,D,m)=fitness(individuo,n,t,dependencia,h,d,CT)
            #Comprobar si el individuo está duplicado.
115         if [m,B,P,D,individuo] in poblacion:
                #Forzamos a que abandone la población en la siguiente generación
117             #(no puede haber más máquinas que piezas).
                poblacion.append([n+1,B,P,D,individuo])
119         else:
                poblacion.append([m,B,P,D,individuo])
121             if [m,B,P,D]<mejor_solucion[0:4]:
                    mejor_solucion=[m,B,P,D,individuo]
123     return (poblacion,mejor_solucion)

125 def fitness(individuo,n,t,dependencia,h,d,CT):
        """
127     Calcula la adaptación del individuo.
        n_maquina: número de máquinas necesarias.
129     B: medida sobre cuán balanceada está la solución.
        P: índice de peligrosidad.
131     D: índice de demanda
        """
133     tiempo=np.array([0])
        dep=copy.deepcopy(dependencia)
135     n_maquina=0
        P=0
137     D=0
        for pieza in individuo:
139         temp=t[pieza-1]
            #Se comprueba si alguna pieza depende de 'pieza'.
141         for p in dep:
                if pieza in dep[p]:
143                 #Se elimina porque al retirarse primero ya no bloquea a 'p'.
                    del dep[p][pieza]
145             #Se comprueba si alguna pieza bloquea a 'pieza'.
                if p==pieza:
147                 #Se añaden los tiempos adicionales.
                    for clave,valor in dep[p].items():
149                     temp+=valor
            #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
151         if tiempo[n_maquina]+temp<=CT:
                #Se añade la pieza.
153             tiempo[n_maquina]+=temp
            else:
155             #Como no cabe, se añade una nueva máquina.
                tiempo=np.append(tiempo,temp)
157             n_maquina+=1
        n_maquina+=1
159     #Se calcula el balance.
        B=np.sum((CT-tiempo)**2)
161     #Se calcula el índice de peligrosidad y de la demanda.
        for i in range(n):
163         if h[individuo[i]-1]==1:
                P+=i+1
165         D+=(i+1)*d[individuo[i]-1]
        return (B,P,D,n_maquina)
167
```

```
def seleccion_torneo(poblacion,N,p_c):
    """
    Se realiza la selección de los padres mediante la selección por torneo
    determinístico.
    """
    padres=[]
    n_padres=round(N*p_c)
    if n_padres%2==1: #Número par de padres
        n_padres-=1
    for i in range(n_padres):
        #Se generan dos números aleatorios que seleccionan que individuos se
        #enfrentan en el torneo.
        a=random.randint(0,N-1)
        b=random.randint(0,N-1)
        #Se selecciona el individuo con mejor adaptación.
        if poblacion[a][0:4]<=poblacion[b][0:4]:
            padre=poblacion[a]
        else:
            padre=poblacion[b]
        padres.append(padre[4])
    return (padres,n_padres)

def cruce_PPX(padres,n_padres,n):
    """
    Se realiza el cruce de la siguiente manera:
      1. Se escogen dos progenitores.
      2. Se genera aleatoriamente una máscara consistente en 1s y 2s generados
      aleatoriamente, indicando de qué padre se debe extraer la información.
      3. Se procede de la siguiente manera:

        padre1:  1 3 2 6 5 8 7 4
        padre2:  1 2 3 5 6 8 7 4
        máscara: 2 2 1 2 1 1 1 2
       ---------------------------
        hijo:

      Se elige el primer elemento disponible del padre indicado.

        padre1:  1 3 2 6 5 8 7 4
        padre2:  1 2 3 5 6 8 7 4    Se coge el 1 2 del segundo padre.
        máscara: 2 2 1 2 1 1 1 2
       ---------------------------
        hijo:    1 2

      Se tachan de ambos padres las piezas ya seleccionadas.

        padre1:  X 3 X 6 5 8 7 4    Se coge el 3 del primer padre.
        padre2:  X X 3 5 6 8 7 4
        máscara: 2 2 1 2 1 1 1 2
       ---------------------------
        hijo:    1 2 3

      Se tachan de ambos padres las piezas ya seleccionadas.

        padre1:  X X X 6 5 8 7 4
        padre2:  X X X 5 6 8 7 4    Se coge el 5 del segundo padre.
        máscara: 2 2 1 2 1 1 1 2
       ---------------------------
        hijo:    1 2 3 5

      Se tachan de ambos padres las piezas ya seleccionadas.

```

```
          padre1:  X X X 6 X 8 7 4    Se coge el 6, 8, y 7 del primer padre.
          padre2:  X X X X 6 8 7 4
         máscara: 2 2 1 2 1 1 1 2
        ---------------------------
          hijo:    1 2 3 5 6 8 7

        Se tachan de ambos padres las piezas ya seleccionadas.

          padre1:  X X X X X X X 4
          padre2:  X X X X X X X 4    Se coge el 4 del segundo padre.
         máscara: 2 2 1 2 1 1 1 2
        ---------------------------
          hijo:    1 2 3 5 6 8 7 4

        De esta forma se respetan las relaciones de precedencia.
        De cada par de progenitores se genera un par de hijos.
    """
    hijos=[]
    for i in range(0,n_padres,2):
        padre1_1=copy.deepcopy(padres[i])
        padre2_1=copy.deepcopy(padres[i+1])
        padre1_2=copy.deepcopy(padres[i])
        padre2_2=copy.deepcopy(padres[i+1])
        hijo1=[]
        hijo2=[]
        for j in range(n):
            mask1=random.randint(1,2)
            if mask1==1:
                hijo1.append(padre1_1[0])
                padre2_1.remove(padre1_1[0])
                padre1_1.pop(0)
            else:
                hijo1.append(padre2_1[0])
                padre1_1.remove(padre2_1[0])
                padre2_1.pop(0)
            mask2=random.randint(1,2)
            if mask2==1:
                hijo2.append(padre1_2[0])
                padre2_2.remove(padre1_2[0])
                padre1_2.pop(0)
            else:
                hijo2.append(padre2_2[0])
                padre1_2.remove(padre2_2[0])
                padre2_2.pop(0)
        hijos.append(hijo1)
        hijos.append(hijo2)
    return hijos

def cruce_FRC(padres,n_padres,n):
    """
    Se realiza el cruce de la siguiente manera:
      1. Se escogen dos progenitores.
      2. Se generan dos números aleatorios que determinaran los tres fragmentos
      en los que se dividirán los padres.
      3. El primer hijo mantiene la cabeza y la cola del primer padre. La parte
      del medio se rellena añadiendo todos los genes que faltan en el orden que
      marca el segundo padre. Para el segundo hijo los padres intercambiarán
      los papeles.
    """
    hijos=[]
    for i in range(0,n_padres,2):
        padre1_1=copy.deepcopy(padres[i])
```

```
            padre2_1=copy.deepcopy(padres[i+1])
293         padre1_2=copy.deepcopy(padres[i])
            padre2_2=copy.deepcopy(padres[i+1])
295         a=random.randint(0, n-1)
            b=random.randint(0, n-1)
297         hijo1=[]
            cola1=[]
299         hijo2=[]
            cola2=[]
301         if a<=b:
                for j in range(a):
303                 hijo1.append(padre1_1[j])
                    padre2_1.remove(padre1_1[j])
305                 hijo2.append(padre2_2[j])
                    padre1_2.remove(padre2_2[j])
307             for j in range(b,n):
                    cola1.append(padre1_1[j])
309                 padre2_1.remove(padre1_1[j])
                    cola2.append(padre2_2[j])
311                 padre1_2.remove(padre2_2[j])
                hijo1=hijo1+padre2_1+cola1
313             hijo2=hijo2+padre1_2+cola2
            else:
315             for j in range(b):
                    hijo1.append(padre1_1[j])
317                 padre2_1.remove(padre1_1[j])
                    hijo2.append(padre2_2[j])
319                 padre1_2.remove(padre2_2[j])
                for j in range(a,n):
321                 cola1.append(padre1_1[j])
                    padre2_1.remove(padre1_1[j])
323                 cola2.append(padre2_2[j])
                    padre1_2.remove(padre2_2[j])
325             hijo1=hijo1+padre2_1+cola1
                hijo2=hijo2+padre1_2+cola2
327         hijos.append(hijo1)
            hijos.append(hijo2)
329     return hijos

331 def precedencia(individuo,gen1,gen2,M,n):
        """
333     Se encarga de comprobar si un individuo respeta las relaciones de
        precedencia.
335     """
        prec = copy.deepcopy(M)
337     #Relaciones de precedencia según se retiran piezas.
        ind=copy.deepcopy(individuo)
339     pieza1=individuo[gen1]
        pieza2=individuo[gen2]
341     ind[gen1]=pieza2
        ind[gen2]=pieza1
343     #Se comprueba que ind es válido.
        for i in range(n):
345         pieza=ind[i]
            #No incumple precedencia.
347         if (not (pieza in prec.keys())) or prec[pieza]==[]:
                #Liberamos las piezas que estaban precedidas por 'pieza'.
349             for j in prec.keys():
                    if pieza in prec[j]:
351                     prec[j].remove(pieza)
            #Sí incumple precedencia.
353         else:
```

```
                return False
355     return True

357
    def mutacion_intercambio(hijos,n,M,p_m):
359     """
        Muta a algunos individuos del conjunto hijos intercambiando dos genes.
361     Con la función precedencia se asegura que el individuo resultante de la
        mutación sea válido.
363     """
        hijos_mut=[]
365     for hijo in hijos:
            a=random.random()
367         if a<=p_m: #Se elige a 'hijo' para ser mutado.
                genes=list(range(n)) #Lista de genes a elegir.
369             gen1=random.choice(genes) #Gen a mutar.
                genes.remove(gen1)
371             b=False #Indica si se ha encontrado un intercambio válido.
                while (not b) and genes!=[]: #Se busca un intercambio válido.
373                 gen2=random.choice(genes)
                    genes.remove(gen2)
375                 #Miramos que no se incumpla ninguna relación de precedencia.
                    b=precedencia(hijo,gen1,gen2,M,n)
377             if b:
                    mut=hijo[gen1]
379                 hijo[gen1]=hijo[gen2]
                    hijo[gen2]=mut
381         hijos_mut.append(hijo)
        return hijos_mut
383

385 def mutacion_insercion(hijos,n,M,p_m):
        """
387     La mutación se realiza eligiendo dos alelos aleatoriamente y colocando el
        segundo después del primero, si se incumple las relaciones de precedencia
389     se situará en la última posición donde estas sí se cumplian.
        """
391     hijos_mut=[]
        for hijo in hijos:
393         a=random.random()
            if a<=p_m:
395             genes=list(range(n))
                gen1=random.choice(genes) #Gen a mutar.
397             genes.remove(gen1)
                gen2=random.choice(genes)
399             pieza2=hijo[gen2]
                #Indica que la configuración del cromosoma sigue respetando las
401             #relaciones de precedencia.
                b=True
403             if gen1<gen2:
                    while b and gen1!=gen2-1:
405                     pieza3=hijo[gen2-1]
                        if (pieza2 in M.keys()) and (pieza3 in M[pieza2]):
407                     #Pieza3 precede a pieza2 luego no puede retroceder más.
                            b=False
409                     else:
                            hijo[gen2-1]=pieza2
411                         hijo[gen2]=pieza3
                        gen2-=1
413             else:
                    while b and gen1!=gen2:
415                     pieza3=hijo[gen2+1]
```

```
                    if (pieza3 in M.keys()) and (pieza2 in M[pieza3]):
417                 #Pieza3 precede a pieza2 y por tanto no puede avanzar más.
                        b=False
419                 else:
                        hijo[gen2+1]=pieza2
421                     hijo[gen2]=pieza3
                    gen2+=1
423         hijos_mut.append(hijo)
    return hijos_mut
425

427 def reemplazo_elitista(poblacion,hijos,N,n,t,dependencia,h,d,CT):
    """
429 Se encarga de reemplazar parte de los individuos de la anterior generación
    por los hijos.
431 Se reemplazan los peor adaptados.
    """
433 hijos_fit=[]
    mejor_hijo=[n+1,0,0,0,[]]
435 #Se calcula el fitness de los hijos.
    for hijo in hijos:
437     (B,P,D,n_maquina)=fitness(hijo,n,t,dependencia,h,d,CT)
        hijos_fit.append([n_maquina,B,P,D,hijo])
439     if [n_maquina,B,P,D]<mejor_hijo[0:4]:
            mejor_hijo=[n_maquina,B,P,D,hijo]
441 n_hijos=len(hijos)
    for i in range(n_hijos):
443     #Eliminamos de la población los peor adaptados.
        poblacion.remove(max(poblacion))
445 for hijo in hijos_fit:
        #Comprobamos si el individuo está duplicado.
447     if hijo in poblacion:
            #Se fuerza a que abandone la población en la siguiente generación
449         #(no puede haber más máquinas que piezas).
            hijo[0]=n+1
451         poblacion.append(hijo)
        else:
453         poblacion.append(hijo)
    return (poblacion,mejor_hijo)
455

def evaluacion(mejor_solucion,mejor_hijo):
457 """
    Se encarga de encontrar la nueva mejor solución.
459 """
    if mejor_hijo[0:4]<mejor_solucion[0:4]:
461     mejor_solucion=mejor_hijo
    return mejor_solucion
463

def informacion(individuo,t,dependencia,CT):
465 """
    Proporciona la distribución de las piezas en las máquinas y el tiempo libre
467 de cada una.
    """
469 tiempo=np.array([0])
    maquinas=[[]]
471 dep=copy.deepcopy(dependencia)
    n_maquina=0
473 for pieza in individuo:
        temp=t[pieza-1]
475     #Se comprueba si alguna pieza depende de 'pieza'.
        for p in dep:
477         if pieza in dep[p]:
```

```
                #Se elimina porque al retirarse primero ya no bloquea a 'p'.
479                 del dep[p][pieza]
            #Se comprueba si alguna pieza bloquea a 'pieza'.
481             if p==pieza:
                    for clave,valor in dep[p].items():
483                     temp+=valor
        #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
485         if tiempo[n_maquina]+temp<=CT:
                #Se añade la pieza.
487             tiempo[n_maquina]+=temp
                #Se registra en qué máquina se retira 'pieza'.
489             maquinas[n_maquina].append(pieza)
            else:
491             #Como no cabe, se añade una nueva máquina.
                tiempo=np.append(tiempo,temp)
493             #Se registra en qué máquina se retira 'pieza'.
                maquinas.append([pieza])
495             n_maquina+=1
    #Se calcula el tiempo libre de cada máquina.
497     tiempo=list(np.array(CT-tiempo))
        return (maquinas,tiempo)
499

501 def main(op_cruce,op_mutacion,M,t,dependencia,h,d,CT,N,n_gen,p_c,p_m):
        n=len(t)
503     (poblacion,mejor_solucion)=generar_pob_inic(n,N,M,t,dependencia,h,d,CT)
        for i in range(n_gen):
505         (padres,n_padres)=seleccion_torneo(poblacion,N,p_c)
            if op_cruce=='FRC':
507             hijos=cruce_FRC(padres,n_padres,n)
            else:
509             hijos=cruce_PPX(padres,n_padres,n)
            if op_mutacion=='insercion':
511             hijos=mutacion_insercion(hijos,n,M,p_m)
            else:
513             hijos=mutacion_intercambio(hijos,n,M,p_m)
            (poblacion,mejor_hijo)=reemplazo_elitista(poblacion,hijos,N,n,t,
    dependencia,h,d,CT)
515         mejor_solucion=evaluacion(mejor_solucion, mejor_hijo)
        (distrib_maquinas,tlibre_maquina)=informacion(mejor_solucion[4],t,
    dependencia,CT)
517     return (mejor_solucion,distrib_maquinas,tlibre_maquina)
```

alg_genetico_dependencia.py

Apéndice D

Implementación algoritmo exhaustivo en PYTHON

```
import numpy as np
import copy

"""
  n: entero. Número de piezas.
  t: lista de enteros. Indica el tiempo de ejecución de cada tarea.
  h: lista de enteros. Indica la peligrosidad de cada tarea.
     (binaro: 1 -> peligroso   0 -> no peligroso)
  d: lista de enteros. Indica la demanda de cada tarea.
  M: diccionario.
     Clave: entero. Indica la pieza.
     Valor: lista de enteros. Indica qué piezas preceden a la pieza indicada en
            la clave.
     Si una pieza no aparece en el diccionario, a esa pieza no le precede
     ninguna otra.
  CT: entero. Ciclo de tiempo de cada máquina.
  n_maquina: Entero. Número de máquinas.
  B: entero. Balance.
  P: entero. Peligrosidad.
  D: entero. Demanda.
  conjunto: lista. Conjunto de soluciones óptimas
  solución: [n_maquina,B,P,D,secuencia_individuo]
  secuencia_individuo: lista de enteros. Indica el orden de retirada de las
                       piezas.
"""

"""
 #Instancia Ordenador Personal (8 partes)

  M={2:[1],3:[1],4:[7],5:[1],6:[2],7:[8],8:[2,3,5,6]}
  t=[14,10,12,18,23,16,20,36]
  h=[0,0,0,0,0,0,1,0]
  d=[360,500,620,480,540,750,295,720]
  CT=40

 #Instancia 10 partes

  M={2:[1, 8, 9, 10],3:[1, 8, 9, 10],7:[5, 6],8:[4, 7]}
  t=[14,10,12,17,23,14,19,36,14,10]
  h=[0,0,0,0,0,0,1,0,0,0]
  d=[0,500,0,0,0,750,295,0,360,0]
  CT=40
"""
```

```
45
def exhaustivo(M,t,h,d,CT):
47    """
      Proporciona todas las soluciones óptimas y su adaptación.
49    """
      n=len(t)
51    solucion=[0]*n
      mejores_sol=[]
53    mejor_adap=(n+1,0,0,0)
      piezas=list(range(1,n+1))
55    (conjunto,adaptacion)=exhaustivo_rec(M,t,h,d,CT,n,mejores_sol,mejor_adap,
      solucion,piezas,0)
      return (conjunto,adaptacion)
57

59 def exhaustivo_rec(M,t,h,d,CT,n,mejores_sol,mejor_adap,solucion,piezas,k):
      """
61    Se realiza el método de vuelta atrás para encontrar todas las soluciones.
      """
63    for i in piezas:
          #Se escoge la pieza número i para añadirla en la posición k.
65        precedencia=copy.deepcopy(M)
          #Comprobamos si es posible retirar la pieza.
67        if (not (i in precedencia.keys())) or precedencia[i]==[]:
              piezas2=copy.deepcopy(piezas)
69            piezas2.remove(i)
              solucion[k]=i
71            #Liberamos las piezas que estaban precedidas por la pieza i.
              for j in precedencia.keys():
73                if i in precedencia[j]:
                      precedencia[j].remove(i)
75            if piezas2==[]: #Hemos obtenido una solución factible.
                  (B,H,D,m)=fitness(solucion, n, t, h, d, CT)
77                if (m,B,H,D)<mejor_adap:
                      mejor_adap=(m,B,H,D)
79                    mejor_solucion=copy.deepcopy(solucion)
                      mejores_sol=[mejor_solucion]
81                elif (m,B,H,D)==mejor_adap:
                      mejor_solucion=copy.deepcopy(solucion)
83                    mejores_sol.append(mejor_solucion)
              else:
85                (mejores_sol,mejor_adap)=exhaustivo_rec(precedencia, t, h, d,
      CT, n, mejores_sol, mejor_adap, solucion, piezas2, k+1)
      return (mejores_sol,mejor_adap)
87

89 def fitness(individuo,n,t,h,d,CT):
      """
91    Calcula la adaptación del individuo.
      n_maquina: número de máquinas necesarias.
93    B: medida sobre cuán balanceada está la solución.
      P: índice de peligrosidad.
95    D: índice de demanda
      """
97    tiempo=np.array([0])
      n_maquina=0
99    P=0
      D=0
101   for pieza in individuo:
          #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
103       if tiempo[n_maquina]+t[pieza-1]<=CT:
```

```
                #Se añade la pieza.
105             tiempo[n_maquina]+=t[pieza-1]
            else:
107             #Como no cabe, se añade una nueva máquina.
                tiempo=np.append(tiempo,t[pieza-1])
109             n_maquina+=1
        n_maquina+=1
111     #Se calcula el balance.
        B=np.sum((CT-tiempo)**2)
113     #Se calcula el índice de peligrosidad y de la demanda.
        for i in range(n):
115         if h[individuo[i]-1]==1:
                P+=i+1
117         D+=(i+1)*d[individuo[i]-1]
        return (B,P,D,n_maquina)
```

exhaustivo.py

Apéndice E

Implementación algoritmo exhaustivo en PYTHON (SDDLBP)

```
import numpy as np
import copy

"""
  n: entero. Número de piezas.
  t: lista de enteros. Indica el tiempo de ejecución de cada tarea.
  dependencia: diccionario.
               Clave: entero i. Indica la pieza.
               Valor: diccionario.
                      Clave: entero j. Indica que hay una relación de
                             dependencia entre i y j, señalando que si se
                             retira i antes que j se deberá añadir un tiempo
                             adicional a la tarea de retirar i.
                      Valor: entero. Cantidad añadida.
               {i:{j:n}} si se retira i antes que j entonces t[i]=t[i]+n.
  h: lista de enteros. Indica la peligrosidad de cada tarea.
     (binaro: 1 -> peligroso   0 -> no peligroso)
  d: lista de enteros. Indica la demanda de cada tarea.
  M: diccionario.
     Clave: entero. Indica la pieza.
     Valor: lista de enteros. Indica qué piezas preceden a la pieza indicada en
            la clave.
     Si una pieza no aparece en el diccionario, a esa pieza no le precede
     ninguna otra.
  CT: entero. Ciclo de tiempo de cada máquina.
  n_maquina: Entero. Número de máquinas.
  B: entero. Balance.
  P: entero. Peligrosidad.
  D: entero. Demanda.
  conjunto: lista. Conjunto de soluciones óptimas
  solución: [n_maquina,B,P,D,secuencia_individuo]
  secuencia_individuo: lista de enteros. Indica el orden de retirada de las
                       piezas.
"""

"""
 #Instancia Ordenador Personal (8 partes)

 M={2:[1],3:[1],4:[7],5:[1],6:[2],7:[8],8:[2,3,5,6]}
 t=[14,10,12,18,23,16,20,36]
```

```
    dependencia={2:{3:4},3:{2:2},5:{6:3},6:{5:1}}
42  h=[0,0,0,0,0,0,1,0]
    d=[360,500,620,480,540,750,295,720]
44  CT=40

46 #Instancia 10 partes

48  M={2:[1, 8, 9, 10],3:[1, 8, 9, 10],7:[5, 6],8:[4, 7]}
    t=[14,10,12,17,23,14,19,36,14,10]
50  dependencia=({1:{4:4},2:{3:3},3:{2:2},4:{1:1,5:2},5:{4:4,6:4},6:{5:2,9:1},
                  9:{6:3}})
52  h=[0,0,0,0,0,0,1,0,0,0]
    d=[0,500,0,0,0,750,295,0,360,0]
54  CT=40
"""
56


58 def exhaustivo(M,t,dependencia,h,d,CT):
    """
60     Proporciona todas las soluciones óptimas y su adaptación.
    """
62     n=len(t)
    solucion=[0]*n
64     mejores_sol=[]
    mejor_adap=(n+1,0,0,0)
66     piezas=list(range(1,n+1))
    (conjunto,adaptacion)=exhaustivo_rec(M,t,dependencia,h,d,CT,n,mejores_sol,
    mejor_adap,solucion,piezas,0)
68     return (conjunto,adaptacion)

70
def exhaustivo_rec(M,t,dependencia,h,d,CT,n,mejores_sol,mejor_adap,solucion,
    piezas,k):
72     """
    Se realiza el método de vuelta atrás para encontrar todas las soluciones.
74     """
    for i in piezas:
76         #Se escoge la pieza número i para añadirla en la posición k.
        precedencia=copy.deepcopy(M)
78         #Comprobamos si es posible retirar la pieza.
        if (not (i in precedencia.keys())) or precedencia[i]==[]:
80             piezas2=copy.deepcopy(piezas)
            piezas2.remove(i)
82             solucion[k]=i
            #Liberamos las piezas que estaban precedidas por la pieza i.
84             for j in precedencia.keys():
                if i in precedencia[j]:
86                     precedencia[j].remove(i)
            if piezas2==[]: #Hemos obtenido una solución factible.
88                 (B,H,D,m)=fitness(solucion, n, t, dependencia, h, d, CT)
                if (m,B,H,D)<mejor_adap:
90                     mejor_adap=(m,B,H,D)
                    mejor_solucion=copy.deepcopy(solucion)
92                     mejores_sol=[mejor_solucion]
                elif (m,B,H,D)==mejor_adap:
94                     mejor_solucion=copy.deepcopy(solucion)
                    mejores_sol.append(mejor_solucion)
96             else:
                (mejores_sol,mejor_adap)=exhaustivo_rec(precedencia, t,
    dependencia, h, d, CT, n, mejores_sol, mejor_adap, solucion, piezas2, k+1)
98     return (mejores_sol,mejor_adap)
```

```
100
def fitness(individuo,n,t,dependencia,h,d,CT):
102     """
        Calcula la adaptación del individuo.
104     n_maquina: número de máquinas necesarias.
        B: medida sobre cuán balanceada está la solución.
106     P: índice de peligrosidad.
        D: índice de demanda
108     """
        tiempo=np.array([0])
110     dep=copy.deepcopy(dependencia)
        n_maquina=0
112     P=0
        D=0
114     for pieza in individuo:
            temp=t[pieza-1]
116         #Se comprueba si alguna pieza depende de 'pieza'.
            for p in dep:
118             if pieza in dep[p]:
                    #Se elimina porque al retirarse primero ya no bloquea a 'p'.
120                 del dep[p][pieza]
                #Se comprueba si alguna pieza bloquea a 'pieza'.
122             if p==pieza:
                    for clave,valor in dep[p].items():
124                     temp+=valor
            #Comprobamos si la máquina puede realizar esta tarea sin exceder CT.
126         if tiempo[n_maquina]+temp<=CT:
                #Se añade la pieza.
128             tiempo[n_maquina]+=temp
            else:
130             #Como no cabe, se añade una nueva máquina.
                tiempo=np.append(tiempo,temp)
132             n_maquina+=1
        n_maquina+=1
134     #Se calcula el balance.
        B=np.sum((CT-tiempo)**2)
136     #Se calcula el índice de peligrosidad y de la demanda.
        for i in range(n):
138         if h[individuo[i]-1]==1:
                P+=i+1
140         D+=(i+1)*d[individuo[i]-1]
        return (B,P,D,n_maquina)
```

exhaustivo_dependencia.py

Bibliografía

[1] Sener Akpınar and G. Mirac Bayhan, *A hybrid genetic algorithm for mixed model assembly line balancing problem with parallel workstations and zoning constraints*, Vol. 24, 2011.

[2] Christian Bierwirth, Dirk C. Mattfeld, and Herbert Kopfer, *On permutation representations for scheduling problems* (Hans-Michael and Ebeling Voigt Werner and Rechenberg, ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[3] Lourdes Araujo and Carlos Cervigón, *Algoritmos Evolutivos: Un enfoque práctico*, Ra-Ma, 2009.

[4] Fernando Berzal, *Algoritmos Genéticos*, Departamento de Ciencia de la Computación e I.A. Universidad de Granada.

[5] Augusto Cortez, *Teoría de la complejidad computacional y teoría de la computabilidad*, RISI. Revista de Investigación de Sistemas e Informática **1** (2004), no. 1, 102-105.

[6] Agoston E. Eiben and James E. Smith, *Introduction to evolutionary computing*, Springer, 2003.

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman D., *Introducción a la teoría de autómatas, lenguajes y computación*, Pearson, 2007.

[8] Lawrence Davis, *Handbook of genetic algorithms*, CumInCAD, 1991.

[9] Agoston E. Eiben and James E. Smith, *Introduction to evolutionary computing*, Springer, 2015.

[10] Pablo Estévez Valencia, *Optimización Mediante Algoritmos Genéticos*, Anales del Instituto de Ingenieros de Chile (1997).

[11] Marcos Gestal Pose, *Introducción a los Algoritmos Genéticos*, Departamento de Tecnologías de la Información y las Comunicaciones Universidade da Coruña (2013).

[12] David E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[13] David E. Goldberg and Robert Lingle, *Alleles, loci, and the traveling salesman problem*, Proceedings of international conference on genetic algorithms and their applications, 1985, pp. 154–159.

[14] Askiner Gungor, Surendra M. Gupta, Arif Hancilar, and Can B. Kalayci, *Multi-objective fuzzy disassembly line balancing using a hybrid discrete artificial bee colony algorithm*, Journal of Manufacturing Systems **37** (2015), 672-682. Reverse Supply Chains.

[15] Surendra M. Gupta and Can B. Kalayci, *Artificial bee colony algorithm for solving sequence-dependent disassembly line balancing problem*, Expert Systems with Applications **40** (2013), 7231–7241.

[16] Surendra M. Gupta, Can B. Kalayci, and Olcay Polat, *A hybrid genetic algorithm for sequence-dependent disassembly line balancing problem*, Annals of Operations Research **242** (2014), 321-354.

[17] Surendra M. Gupta and Seamus M. McGovern, *A balancing method and genetic algorithm for disassembly line balancing*, European Journal of Operational Research **179** (2007), 692–708.

[18] ———, *The Disassembly Line. Balancing and Modeling*, Mc Graw Hill, 2011.

[19] John Henry Holland, *Adaptation in natural and Artificial Systems*, MI: University of Michigan Press, 1975.

[20] Can Berk Kalayci, *Algorithms for Sequence-Dependent Disassembly Line Balancing Problem* (2011).

[21] Yow-Yuh Leu, Lance A. Matheson, and Loren Paul Rees, *Assembly Line Balancing Using Genetic Algorithms with Heuristic-Generated Initial Populations and Multiple Evaluation Criteria*, Vol. 25, 1994.

[22] Anselmo Pérez Serrada, *A balancing method and genetic algorithm for disassembly line balancing* (1996).

[23] Masoud Seidi, *The Balancing of Disassembly Line of Automobile Engine Using Genetic Algorithm (GA) in Fuzzy Environment*, Industrial Engineering and Management Systems **15** (2016), 364-373.