Capítulo 6 Ingeniería de Software Basada en Búsqueda en Líneas de Productos de Software

Chapter 6 Search-Based Software Engineering in Software Product Lines

TRUJILLO-TZANAHUA, Guadalupe Isaura†*, CORTÉS-VERDÍN, Karen y JUÁREZ-MARTÍNEZ, Ulises

Tecnológico Nacional de México/I. T. Orizaba, Oriente 9, Emiliano Zapata Sur, C.P. 94320 Orizaba, Veracruz, México. Facultad de Estadística e Informática. Universidad Veracruzana, Av. Xalapa esq. Ávila Camacho S/N, Xalapa, Ver. C,P, 91020 México

ID 1er Autor: *Guadalupe Isaura, Trujillo-Tzanahua* / **ORC ID:** 0000-0002-6957-4251, **CVU CONACYT ID:** 412364

ID 1er Coautor: *Karen, Cortés-Verdín* / **ORC ID:** 0000-0002-6453-180X

ID 2do Coautor: *Ulises, Juárez-Martínez* / **ORC ID:** 0000-0002-5911-3136 , **CVU CONACYT ID:** 85999

DOI: 10.35429/H.2020.5.5.92.116

Resumen

Actualmente, la construcción del software avanza hacia la industrialización sustituyendo la forma de desarrollo a la medida por el empleo de enfoques como Líneas de Productos de Software (LPS) y Multilíneas de Productos de Software (MPL). Estos paradigmas establecen un medio de producción común para generar una variedad de productos a través de la reutilización de insumos y automatización de procesos y de este modo satisfacer las necesidades y requisitos del mercado en lugar de enfocarse a clientes específicos. Sin embargo, la gestión de múltiples LPS o MPL es un desafío porque la cantidad variante de productos de software posibles a obtener expresada en los modelos de características suele ser grande por las combinaciones de características. Por esta razón no es factible configurar, implementar o probar todas las posibles variantes de productos. Para apoyar este proceso de decisión, en el presente capítulo se investiga y aplica una variante del problema de la mochila. Específicamente, la configuración de productos en una MPL se formula como un problema de la mochila multidimensional de opción múltiple (MMKP, Multiple-Choice Multi-dimensional Knapsack Problem) y se resuelve con una técnica de Ingeniería de Software Basada en Búsqueda (SBSE). En primer lugar, se proporciona una revisión de técnicas de búsqueda y optimización con el propósito de ofrecer elementos de aplicación práctica para el desarrollo de productos de software apoyándose de LPS e Ingeniería de Software Basada en Búsqueda. Posteriormente, se implementa en Python un algoritmo genético para resolver el problema de la configuración de productos en MPL.

Ingeniería de Software, Líneas de Productos de Software, SBSE, Algoritmos genéticos

Abstract

Currently, software construction is moving towards industrialization by replacing the custom form of development with the use of approaches such as Software Product Lines (SPL) and Multiple Software Product Lines (MPL). These paradigms establish a common means of production to generate a variety of products through the reuse of inputs and automation of processes and thus meet the needs and requirements of the market instead of targeting specific customers. However, the management of multiple SPLs or MPLs is a challenge because the variant number of possible software products to be obtain expressed in the feature models is often large due to feature combinations. For this reason, it is not feasible to configure, deploy, or test all possible product variants. To support this decision-making process, this chapter investigates and applies a variant of the knapsack problem. Specifically, product configuration in an MPL is formulated as a Multiple-Choice Multi-dimensional Knapsack Problem (MMKP) and is solved with a Search-Based Software Engineering (SBSE) technique. First, a review of search and optimization techniques is provided to offer practical application elements for the development of software products based on SPL and Search-Based Software Engineering (SBSE). A genetic algorithm is then implemented in Python to solve the problem of configuring products in MPL.

Software Engineering, Software Product Lines, SBSE, Genetic algorithms

6. Introducción

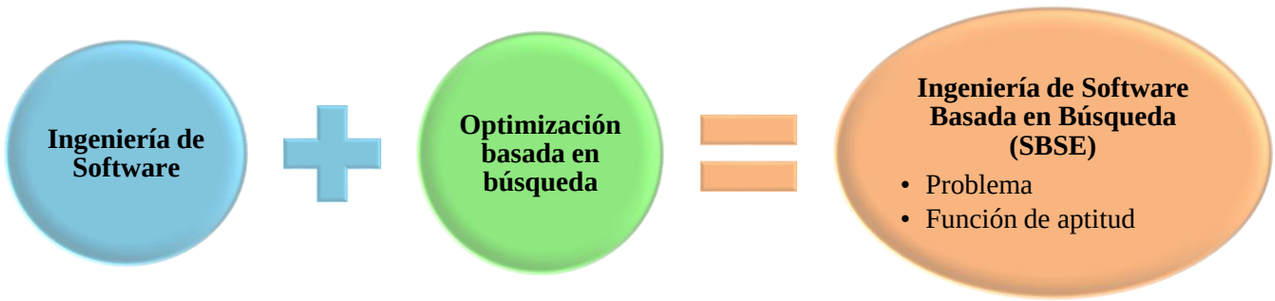
Actualmente, las empresas necesitan innovar continuamente sus productos o introducir nuevos en el mercado para mantener o mejorar el nivel de rentabilidad. Por otro lado, los clientes demandan productos de mejor calidad, menor precio y entrega más corta. Para mantener la participación en el mercado, las empresas requieren cambiar la forma en que los productos son diseñados, fabricados y entregados. En este sentido, la industria del software evoluciona constantemente utilizando enfoques para la generación automática y personalización en masa de software conocidos como Líneas de Productos de Software (LPS) y Multilíneas de Productos de Software (MPL). Estos enfoques tienen su origen en un contexto diferente al software, específicamente en las líneas de productos de industrias como la automotriz, química, metalúrgica, y electrónica, entre otras. En estas industrias se fabrica una variedad de productos a través de la reutilización de insumos y automatización de procesos con el fin de incrementar la productividad y al mismo tiempo reducir los costos de desarrollo y comercialización.

En los últimos años, la industria del software se ha encargado de automatizar las fases del desarrollo de un proyecto de software a través de diversas herramientas y enfoques como las LPS y MPL.

Asimismo, los desarrolladores de software han explorado sinergias con otras áreas y técnicas que contribuyen a la resolución de problemas y toma de decisiones complejas mediante la modelación de problemas de búsqueda y resolución a través de estrategias de optimización como los algoritmos genéticos, enjambre de partículas, recocido simulado, colonia de hormigas, entre otros.

Un problema de búsqueda es aquel en el que se buscan soluciones óptimas o casi óptimas en un espacio de búsqueda de soluciones candidatas, guiado por una función de aptitud que distingue entre soluciones mejores y peores. La formulación de problemas de búsqueda y aplicación de técnicas de optimización para su solución en el contexto de software se denomina Ingeniería de Software Basada en Búsqueda (SBSE, search-based software engineering). La **Figura 6.1** ilustra los elementos que integran a la Ingeniería de Software Basada en Búsqueda.

Figura 6.1. Elementos de la Ingeniería de Software Basada en Búsqueda



Fuente de Consulta: Elaboración propia

Según Harman, un enfoque de SBSE requiere de dos elementos para formular un problema de software como problema de búsqueda (Harman & Jones, 2001): 1) la representación del problema, para permitir su manipulación por el algoritmo de búsqueda y 2) una función de aptitud para evaluar la calidad u optimalidad de las soluciones que generalmente se basan en métricas de software.

Generalmente, los problemas de software se caracterizan porque no existe una única solución exacta por la existencia de un gran espacio de búsqueda (por ejemplo, cantidad de diseños posibles, casos de prueba o configuraciones), en el que es necesario equilibrar y resolver posibles conflictos entre múltiples objetivos y restricciones.

El objetivo de la búsqueda es identificar, entre todas las posibles soluciones, una que sea lo suficientemente buena según las métricas apropiadas (Harman, 2012; Harman & Clark, 2004; Harman & Jones, 2001). Por consiguiente, es natural pensar que la mayoría de los problemas en la Ingeniería de Líneas de Productos de Software (SPLE, Software Product Line Engineering) son problemas de optimización. Esto debido a que la variabilidad de las LPS expresada en los modelos de características convierte el análisis manual en una tarea costosa y propensa a errores porque crea un gran espacio de búsqueda en el que es posible buscar opciones de productos óptimas (o casi óptimas).

En la literatura, existen diversas publicaciones que exploran la aplicación de técnicas de SBSE en diversas actividades del ciclo de vida de las LPS como la selección de características óptimas, la generación de pruebas, la arquitectura, entre otros. Estas técnicas y aplicaciones resultan interesantes para enfrentar algunos desafíos que es posible surjan al implementar o gestionar LPS ya que la cantidad de variantes de los productos de software suele grande, por lo que no es factible configurar, implementar o probar todas las posibles configuraciones (variantes de productos) y por lo tanto es más complejo visualizar o probar todas las configuraciones.

A pesar de la experiencia obtenida en la industria, los ingenieros de LPS reconocen que es necesario que estas LPS evolucionen o se reutilicen en una nueva LPS denominada MPL, la cual permite modificar los productos de las LPS originales sin poner en riesgo su funcionamiento original. Asimismo, se detectan diversas áreas de oportunidad, especialmente en la fase de análisis de los modelos de características de las LPS, diseño de la MPL y pruebas.

El proceso de diseño de nuevos productos de software en una MPL es una actividad compleja de toma de decisiones que se encuentra influenciada por múltiples factores entre los que destacan varios tipos de relaciones de variabilidad y restricciones entre características, las configuraciones crecen exponencialmente de acuerdo con el número de características (2^n), lo que resulta en una explosión combinatoria de variantes, en las cuales las características influyen de forma positiva o negativa en el producto, por mencionar algunos.

El presente capítulo proporciona una revisión de técnicas de búsqueda y optimización que apoyan a la Ingeniería de Líneas de Productos de Software. La contribución principal de este trabajo consiste en ofrecer elementos de aplicación práctica para configurar características y obtener variantes de productos de software utilizando LPS y técnicas de SBSE. Además, esta investigación presenta la modelación de un problema de SPLE (configuración de productos en una MPL) como un problema de búsqueda (SBSE), utilizando el problema de la mochila y la implementación en Python de un algoritmo genético como estrategia de solución.

El capítulo está organizado de la siguiente manera: la sección 1 proporciona los antecedentes y conceptos básicos sobre las LPS, MPL y el problema de la mochila. La sección 2 presenta las técnicas de SBSE que se utilizan en las LPS. La sección 3 presenta la metodología empleada para la aplicación de técnicas de SBSE en el desarrollo de una MPL. La sección 3 presenta la aplicación de técnicas de SBSE al problema de configuración de productos de SBSE. Finalmente, en la sección 4 se presentan las conclusiones y el trabajo a futuro.

6.1 Antecedentes

Los antecedentes básicos de los campos de investigación que abarcan el capítulo son: Líneas de Productos de Software, Multilíneas de Productos de Software y el problema de la mochila.

Líneas de Productos de Software

El concepto de Línea de Productos de Software se utiliza tanto en la industria como en la academia para referirse al desarrollo de un conjunto de productos de software similares utilizando una plataforma común y la personalización en masa.

Según el Instituto de Ingeniería de Software de la Universidad Carnegie Mellon (SEI), se le denomina **Línea de Productos de Software** al "conjunto de sistemas de software intensivo que comparten una serie de características administradas que satisfacen las necesidades específicas de un segmento de mercado particular o misión, y que se desarrollan de forma prescrita a partir de un conjunto común de activos base" (Clements & Northrop, 2001). De esta definición se destaca que las LPS capitalizan la reutilización de los activos base (artefactos y recursos) y el rápido desarrollo de nuevas aplicaciones para un dominio objetivo. Dentro de los **activos base (core assets)** que forman las bases para las LPS destacan la arquitectura, componentes de software reutilizables, modelos de dominio, requisitos, documentación, especificaciones, calendario, presupuesto, planes de prueba, casos de prueba, por mencionar algunos.

Otro aspecto que diferencia a la Ingeniería de Líneas de Productos de Software (SPLE, Software Product Line Engineering) de la Ingeniería de Software tradicional es la gestión de la variabilidad. La **variabilidad** se refiere a la capacidad de un artefacto de software o sistema para configurarse, personalizarse, extenderse o cambiarse para su uso en un contexto específico. El propósito general de la variabilidad de una LPS es maximizar el retorno de la inversión (ROI, Return On Investment) para construir y mantener productos de software durante un período específico de tiempo.

Entre los beneficios que las LPS ofrecen a las organizaciones destacan el aumento de la productividad de los ingenieros de software, la reducción de costos de desarrollo y riesgos, la calidad de los productos se mejora porque cada producto se ajusta a las necesidades de cada cliente y se facilita el mantenimiento de los productos de software. Sin embargo, en algunas ocasiones estos beneficios resultan limitados porque las LPS no pueden evolucionar o mantenerse indefinidamente para satisfacer o cumplir con los requerimientos y necesidades del mercado ya sea por funcionalidad, enfoque o tecnología.

Por esta razón, los ingenieros reconocen que es imposible extender o adaptar la LPS (Holl, Grünbacher, & Rabiser, 2012). Una alternativa de solución es emplear un esquema de reutilización similar a las LPS denominado MPL (Multilínea de Productos de Software).

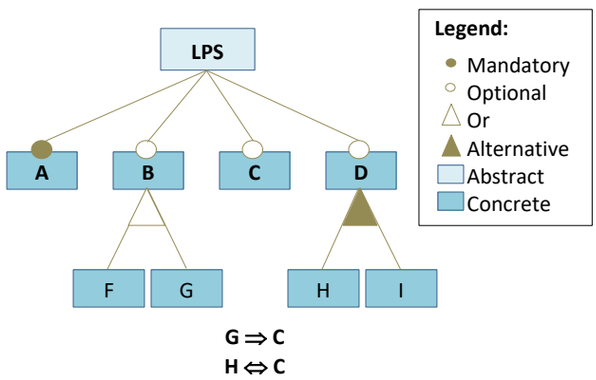
Una **MPL (Multi Product Line)** también denominada MSPL (Multiple Software Product Line) es una Línea de Productos de Software que extiende la reutilización de software de varias Líneas de Productos de Software heterogéneas (Holl et al., 2012; Lienhardt, Damiani, Donetti, & Paolini, 2018; Savolainen, Mannion, & Kuusela, 2012). El enfoque se refiere al desarrollo de software en el que los productos de software son el resultado de combinar componentes o productos desarrollados en Líneas de Productos de Software independientes, las cuales provienen de diversas organizaciones o equipos, utilizan diversos enfoques, tecnologías y proveedores. Los términos MPL, MSPL, Múltiples LPS, Líneas de Productos Anidadas, Líneas de Productos de Líneas de Productos se utilizan para denotar el mismo concepto.

Modelo de características

Una **característica (feature)** o rasgo es un elemento distintivo que representa aspectos relevantes del software y se utiliza para describir o distinguir un producto de la LPS. Las características son importantes para representar la variabilidad debido a que es posible generar diferentes productos seleccionando diferentes características (Benavides, Trinidad, & Ruiz-Cortés, 2005) .

Un **modelo de características (feature model)** es una técnica gráfica o textual para expresar la variabilidad y representar los productos de una LPS en términos de características (obligatorias, opcionales y alternativas), las cuales describen un incremento en la funcionalidad de los productos (Figura 6.2). Asimismo, los modelos de características delimitan el alcance de las LPS y documentan formalmente las configuraciones que son soportadas. También se utilizan para el modelo de verificación de una LPS, pruebas en las LPS, automatización de la configuración del producto o para el cálculo de la información pertinente.

Figura 6.2 Modelo de características



Fuente de Consulta: Elaboración propia

El conjunto de características que integran un modelo de características se organiza de la siguiente forma:

Relaciones jerárquicas

Se refieren a las relaciones entre una característica padre o raíz y sus características hijas (sub-características) las cuales pueden ser:

- **Obligatoria (Mandatory):** se refiere a la selección predeterminada de características. Además, si se selecciona la característica raíz, es necesario seleccionar la sub-característica.

- **Opcional (Optional):** indica la posibilidad de seleccionar sub-características.

- **Alternativa (Alternative):** indica la posibilidad de seleccionar una de las sub-características del grupo.

- **Or:** indica la posibilidad de elegir una o más sub-características del grupo. Es posible colocar una restricción de cardinalidad para restringir el número mínimo y máximo de características seleccionables en un grupo.

Relaciones no jerárquicas

Relaciones del tipo "si la característica A aparece, entonces la característica B se incluye o excluye".

- **Require (Requires):** si una característica G requiere una característica C significa "si la característica G es incluida en el producto", es necesario incluir la característica C, pero no viceversa.

- **Excluye (Excludes):** si una característica H excluye una característica C significa que ambas características no forman parte del mismo producto.

Proceso de desarrollo de una LPS

El proceso de desarrollo de una LPS implica gestionar los puntos de variación entre los diferentes miembros de la línea. Por esta razón se identifican los aspectos comunes y variables del dominio en cuestión. El paradigma de la Ingeniería de LPS separa dos procesos (Pohl, Böckle, & Linden, 2005):

1) La Ingeniería de Dominio es el proceso responsable de establecer la plataforma reutilizable, definir lo que es común y lo que es variante entre los productos de las LPS.

2) La Ingeniería de Aplicación se refiere al proceso en el que las aplicaciones de la LPS se construyen mediante la reutilización de artefactos del dominio y la explotación de la variabilidad de la LPS. Este proceso también se conoce como **derivación del producto**, cuya actividad principal es seleccionar o configurar un conjunto adecuado de características que satisfaga ciertos requisitos especificados. Estos requisitos son funcionales o no funcionales, con algunos requisitos compitiendo e incluso en conflicto entre sí.

Para la derivación de productos en una LPS, un ingeniero de aplicaciones recibe un modelo de características y los requisitos de la aplicación e intenta seleccionar un subconjunto de características que genere el producto de software requerido y personalizado a las necesidades y preferencias de los usuarios. Esta cuestión se conoce como el **problema de selección óptima de características** (optimal feature selection) (dos Santos Neto, Britto, Rabêlo, Cruz, & Lira, 2016; Guo, White, Wang, Li, & Wang, 2011; Y. Wang & Pang, 2014; Xue et al., 2015, 2016) o **problema de la configuración** (Asadi, Soltani, Gasevic, Hatala, & Bagheri, 2014) y se convierte en un problema de optimización que plantea desafíos para el razonamiento y la configuración de las características.

Principalmente, los problemas a enfrentar durante la configuración de las Líneas de Productos de Software se enumeran a continuación (Afzal, Mahmood, & Shaikh, 2016):

- Modelo de características vacío o nulo debido a que no hay algún producto válido en la derivación.

- Modelo de características inválido debido a que sólo produce un producto.

- Producto inconsistente porque infringe las reglas y restricciones del modelo de características.

- Modelo de características inconsistente debido a que sus características incluidas no son consistentes entre sí.

- Problema de la derivación o configuración del producto debido a que es posible derivar múltiples productos de un único dominio en el punto de variación.

- Problema de la gestión de la variabilidad

- Errores relacionados con los modelos de características como características redundantes, cardinalidades inadecuadas y características variables muertas y falsas.

    - Un modelo de características es redundante si la información semántica (al menos una) es modelada de una manera múltiple.

    - Una cardinalidad es inadecuada si adopta más características clonadas que las permitidas por el modelo de características.

    - Las características muertas se definen en los modelos de características, pero nunca son parte de una configuración de producto válida.

    - Las características de las variables falsas son aquellas que se etiquetan como opcionales en los modelos de características, pero tienen una restricción de selección obligatoria en el caso de la selección de características de los padres.

Problema de la mochila

El problema de la mochila (Knapsack problem) es un problema clásico de la optimización combinatoria con diversas aplicaciones en la industria, finanzas, ciencias aplicadas o en la vida real. Este problema permite modelar situaciones como presupuestos de capital, reducción de inventarios, asignación de procesos en sistemas distribuidos, selección de proyectos, distribuciones de carga física o eléctrica, por mencionar algunos.

El problema de la mochila modela la situación de colocar ***n*** objetos con diferentes pesos o valores **(P_i)** en una mochila, la cual tiene una determinada capacidad de peso (***C***). El objetivo del problema de la mochila es encontrar un subconjunto de objetos con el cual se maximice el beneficio que proporcionan los objetos mientras se satisface la restricción de no sobrepasar la capacidad de la mochila (contenedor) donde serán colocados los objetos (Connolly, Martello, & Toth, 1991). Formalmente, el problema de la mochila se define de la siguiente forma:

$$Maximizar \quad Z = \sum_{i=1}^{n} P_i X_i \tag{1}$$

$$Sujeto\ a \quad \sum_{i=1}^{n} W_i X_i \leq C \tag{2}$$

$$X_i = 0 \text{ ó } 1, \qquad i = 1, \dots n \tag{3}$$

El problema de la mochila tiene diversas variantes entre las que destacan:

- El problema de la mochila multidimensional (MDKP, Multi-Dimensional Knapsack Problem) el cual se refiere a un problema de la mochila con restricciones.

- El problema de la mochila múltiple (MKP, Multiple Knapsack problem) que considera *m* mochilas.

- El problema de la mochila de opción múltiple (MCKP, Multiple-Choice Knapsack Problem) en el cual los artículos son divididos en clases y exactamente un elemento de cada clase se utiliza.

- El problema de la mochila multidimensional de opción múltiple (MMKP, Multiple-Choice Multi-dimensional Knapsack Problems) supone n conjuntos compuestos por elementos mutuamente excluyentes. El objetivo es seleccionar exactamente un elemento por conjunto, maximizando la utilidad general, sin violar una familia de restricciones de mochila.

Técnicas de SBSE utilizadas en las LPS

En los últimos años, la Ingeniería de Software Basada en Búsqueda ha permitido a los ingenieros de software aplicar técnicas de optimización para automatizar tareas y resolver problemas o asuntos en las diferentes etapas del ciclo de vida de las LPS. A continuación, se describen las técnicas de SBSE principalmente utilizadas en el contexto de LPS.

Recocido Simulado

Originalmente, el recocido simulado se utilizó como un medio para encontrar la configuración de equilibrio de una colección de átomos a una temperatura dada.

El recocido simulado (SA, Simulated Annealing) es un método de búsqueda aplicado a problemas de optimización combinatoria. Este método se inspira en el proceso físico de recocido de sólidos, el cual utiliza un procedimiento que va disminuyendo la temperatura, con lo cual se modifica la estructura del material. El enfriamiento se realiza de forma lenta para obtener configuraciones moleculares resistentes. Cada etapa del enfriamiento tiene asociada una energía y una configuración del material determinadas.

En el contexto de las LPS, el recocido simulado se utiliza para optimizar el alcance de una plataforma de producto de software (Alsawalqah, Kang, & Lee, 2014) y para mejorar la eficiencia de la configuración del producto (Tan, Lin, Ye, & Zhang, 2013). Para mejorar la eficiencia de la configuración del producto en la LPS, la situación de identificar el mínimo conjunto de puntos de variación en un modelo de características es modelada mediante el problema de la cobertura de vértices y resuelta mediante un algoritmo de aproximación (Tan et al., 2013).

Algoritmos genéticos

Los **algoritmos genéticos (GA, Genetic algorithm)** son técnicas basadas en la Teoría de Darwin utilizadas para resolver problemas de búsqueda y optimización simulando el proceso de evolución natural. El procedimiento de búsqueda tiene el propósito de mantener una población de soluciones potenciales (cromosomas) mientras se realiza una investigación paralela de soluciones con una función de aptitud alta.

Los algoritmos genéticos tienen diferentes variaciones, específicamente para los operadores genéticos (cruce, mutación), la selección y cómo se reemplaza a los individuos para formar la nueva población. Existen tres pasos principales en un algoritmo genético: cruce, mutación y selección. En la literatura, se proponen diversas variantes para el operador de cruce, pero el principio común es combinar dos cromosomas para generar cromosomas de próxima generación, mediante un intercambio de genes simple o no, con pequeñas variaciones. La mutación cambia aleatoriamente los valores del gen para generar una nueva combinación de genes para la próxima generación. Matemáticamente, el interés principal de la mutación consiste en saltar soluciones óptimas locales. La selección es el último paso donde se copian las mejores soluciones cromosómicas en la próxima generación.

El enfoque denominado GA-FL (Genetic Algorithm to Feature Location) utiliza algoritmos genéticos para ubicar características en LPS basadas en modelos (Font, Arcega, Haugen, & Cetina, 2016).

La ubicación de características se refiere al proceso de encontrar el conjunto de artefactos de software que implementan una característica en particular. Por otro lado, los algoritmos genéticos son empleados para la minimización de pruebas en las LPS, específicamente en la identificación y eliminación de casos de prueba redundantes con el fin de reducir la cantidad total de casos de prueba a ejecutar, mejorando así la eficiencia de la prueba. Para lograr esto, se formula el problema de minimización como un problema de búsqueda y se define una función de aptitud considerando varios objetivos de optimización. Los algoritmos genéticos se utilizan para evaluar el desempeño de la función de aptitud (S. Wang, Ali, & Gotlieb, 2015).

Colonia de hormigas

La **optimización por colonia de hormigas (ACO, Ant Colony Optimization)** es una técnica de optimización aproximada inspirada en las colonias de hormigas ya que cada hormiga deposita un rastro de feromonas en la ruta o camino de su hormiguero a una fuente de alimento para que las demás integrantes de la colonia lo sigan y encuentren el camino más corto. Por esta razón, esta técnica se utiliza para resolver problemas computacionales que buscan los mejores caminos o rutas en un grafo.

En relación a la selección de características con restricciones de recursos (FSRC) en las LPS, el enfoque ACOFES transforma el modelo de características en un grafo dirigido y el problema a resolver es encontrar el mejor camino o ruta desde el origen hasta el destino, independientemente de las restricciones entre árboles. Posteriormente, el algoritmo ACO es modificado para encontrar la ruta óptima en el grafo dirigido (Y. Wang & Pang, 2014).

Murciélago

El **algoritmo de murciélago (BA, Bat algorithm)** es un algoritmo heurístico que imita el comportamiento de ecolocación de los murciélagos, la cual les permite localizar y cazar a su presa en la oscuridad. Este algoritmo permite realizar una optimización global y resulta de la combinación de un algoritmo de enjambre y un algoritmo de rutas. En (Alsariera, Majid, & Zamli, 2015), se presenta un enfoque llamado SPLBA para la reducción de casos de pruebas para LPS utilizando un algoritmo inspirado en los murciélagos.

Multiobjetivo

La **optimización multi-objetivo (MOO, Multi-objective optimization) o multi-criterio** considera problemas que requieren optimizar simultáneamente más de una función objetivo. En este caso, la noción de óptimo se redefine porque en lugar de buscar una única mejor solución (solución exacta), se busca un conjunto de buenas soluciones. El desafío principal de la optimización multi-objetivo es encontrar un conjunto de soluciones para ofrecer al tomador de decisiones las mejores alternativas entre las disponibles, para que seleccione una de ellas.

El diseño de arquitecturas de Líneas de Productos (PLA) se identifica como un problema de optimización que se ha resuelto mediante algoritmos multi-objetivo para encontrar soluciones óptimas que satisfagan los objetivos definidos. MOA4PLA (Multi-Objective Optimisation Approach for PLA design) es un enfoque basado en la búsqueda para respaldar las arquitecturas de LPS (Colanzi, Vergilio, Gimenes, & Oizumi, 2014). MOA4PLA utiliza una PLA modelada en un diagrama de clase UML y la optimiza. Se genera un conjunto de soluciones con la mejor compensación entre objetivos y el arquitecto selecciona la arquitectura a utilizar. En (Henard, Papadakis, Perrouin, Klein, & Traon, 2013), se presenta un enfoque para manejar múltiples objetivos en conflicto en la generación de pruebas para LPS. Este enfoque combina algoritmos genéticos y técnicas de resolución de restricciones para tratar los siguientes objetivos: 1) maximizar la cobertura por pares, 2) minimizar la cantidad de productos seleccionados y 3) minimizar el costo total del conjunto de pruebas.

Lógica difusa

La lógica difusa proporciona un mecanismo de inferencia que permite simular los procedimientos del razonamiento humano en sistemas basados en el conocimiento. La teoría de la lógica difusa permite modelar la incertidumbre de los procesos cognitivos humanos proporcionando herramientas formales para su tratamiento. Básicamente, cualquier problema se resuelve a partir de un conjunto de variables de entrada (espacio de entrada) y se obtiene un valor adecuado de variables de salida (espacio de salida). La lógica difusa permite establecer este mapeo de una forma adecuada, atendiendo a criterios de significado (y no de precisión). Actualmente, se utiliza en un amplio sentido, agrupando la teoría de conjunto difusos, reglas si-entonces, aritmética difusa, cuantificadores, entre otros. En (Robak & Pieczynski, 2003), los autores utilizan la lógica difusa para asignar pesos (prioridades) a las características en un modelo de características. La asignación de prioridades en un modelo de características ayuda a los desarrolladores a seleccionar las características de alta prioridad sobre las más bajas. Definen los tipos de la función de pertenencia, número y los tipos de distribución de los conjuntos difusos.

Sistemas de recomendación

Los sistemas de recomendación (RS) aprenden de las preferencias de los usuarios y predicen futuros elementos de interés para ellos. Su objetivo es aliviar el problema de la sobrecarga de información que se produce en la configuración de la LPS, donde el número de características y configuraciones posibles es alto para que un solo usuario lo manipule. En (Pereira, Matuszyk, Krieter, Spiliopoulou, & Saake, 2016) proponen un Sistema de Recomendación de características colaborativo, el cual se basa en configuraciones de usuarios anteriores para generar recomendaciones personalizadas para un usuario actual. Esto es debido a que la configuración manual de un producto es un proceso masivo y difícil. Para facilitar este proceso, adaptan tres algoritmos de recomendación personalizada con el escenario de la configuración de la LPS: vecino más cercano, similitud media y factorización de matrices. Estos algoritmos utilizan configuraciones anteriores para estimar y predecir la relevancia de las características con el fin de guiar al usuario a través del proceso de selección de características.

La **Tabla 6.1** muestra la comparación de los trabajos relacionados que identifican problemas de optimización en las LPS y las técnicas de SBSE utilizadas para resolverlos. Los criterios de comparación son: Tipo (LPS o MPL), Tipo de modelo, técnica de SBSE y el objetivo de la función objetivo.

Tabla 6.1. Trabajos relacionados

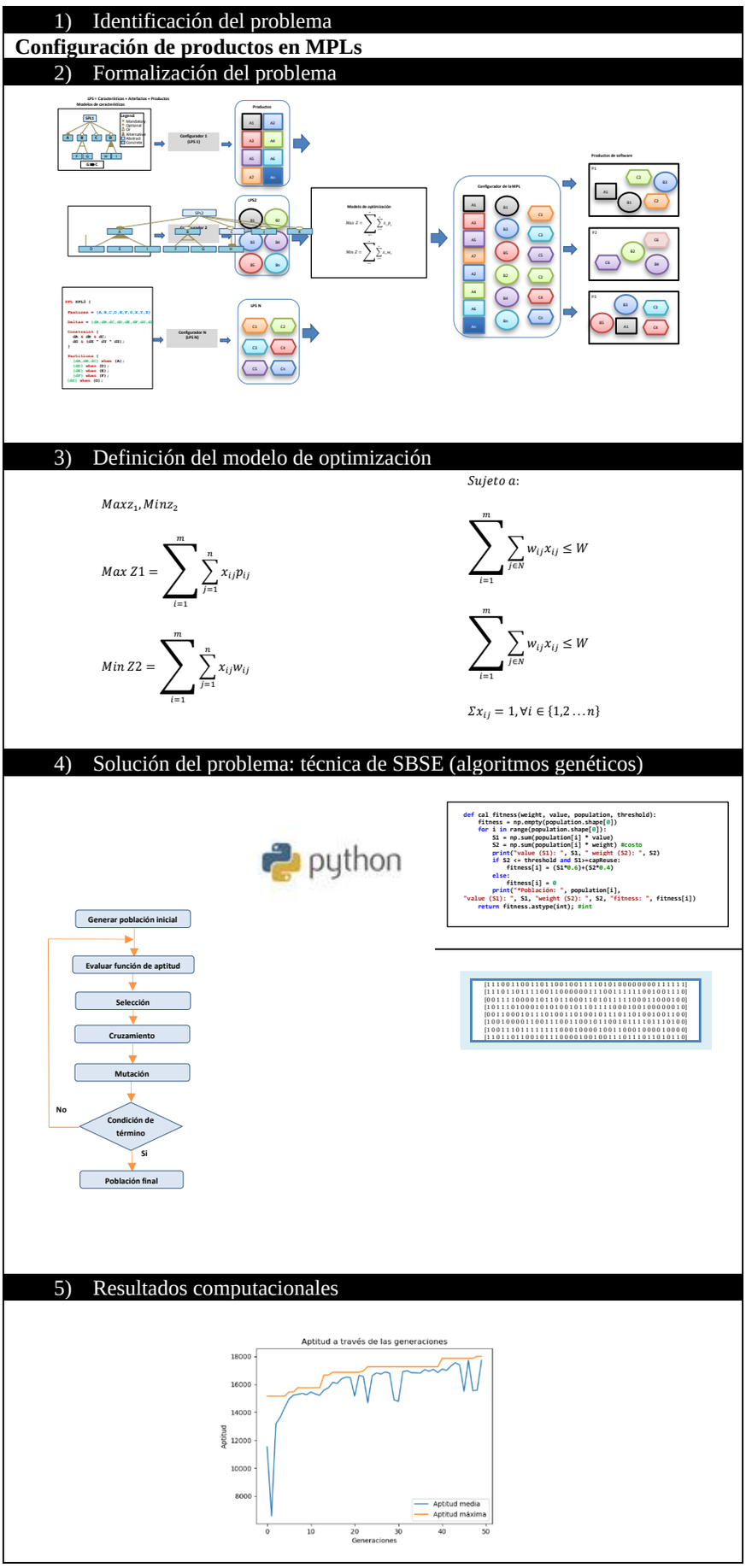
Propuesta	Tipo de LPS	Tipo de modelo	Técnica SBSE	Caso de estudio	Función objetivo
(Robak & Pieczynski, 2003)	LPS	Mono-objetivo	Lógica Difusa	Didáctico (coche)	Gestión de prioridades
(Tan et al., 2013)	LPS	No se menciona	Recocido simulado	Biblioteca	No se menciona
GAFES (Guo et al., 2011)	LPS	Mono-objetivo	Algoritmo genético	Didáctico (red de sensores)	Minimizar consumo de recursos
ACOFES (Y. Wang & Pang, 2014)	LPS	Mono-objetivo	Colonia de hormigas	Didáctico (Tienda online)	Minimizar consumo de tiempo
(Cruz et al., 2013)	LPS	Multi-objetivo	Sistemas de inferencia difusos	Didáctico (Libros)	Minimizar costo y maximizar relevancia de los productos candidatos
IVEA (Lian & Zhang, 2015)	LPS	Multi-objetivo	Algoritmo genético	Didáctico (Portal web)	Maximizar el número de características seleccionadas, minimizar los defectos, costo y el número de violaciones de reglas
(Afzal, Mahmood, Rauf, & Shaikh, 2014)	LPS	Mono-objetivo	Algoritmo genético	Didáctico (Smartphone)	Minimizar inconsistencias del modelo de características
(Trujillo-Tzanahua, Juarez-Martinez, Aguilar-Lasserre, Cortes-Verdin, & Azzaro-Pantel, 2019)	MPL	Multi-objetivo	Algoritmo genético	Inmótica	Minimizar costo, maximizar reutilización y compatibilidad de las características.

Fuente de Consulta: Elaboración propia

6.3 Metodología

La metodología propuesta para la aplicación de técnicas de SBSE en el desarrollo de una MPL consiste en 5 etapas, las cuales se describen gráficamente en la **Figura 6.3**. y se detallan en la sección 5.

Figura 6.3. Metodología propuesta



Fuente de Consulta: Elaboración propia

Aplicación de técnicas SBSE al problema de configuración de productos

Durante la última década, el alcance de la Ingeniería de Software Basada en Búsqueda se ha extendido para cubrir diferentes aspectos del ciclo de vida de las LPS, tales como: selección de características, generación de pruebas, arquitectura, mantenimiento, entre otros. Estas técnicas resultan interesantes para enfrentar algunos desafíos que es posible surjan al reutilizar varias LPS, la interoperabilidad, la derivación de productos, la arquitectura, las pruebas de los productos, entre otras.

En esta sección, se presenta la problemática elegida para probar la efectividad de utilizar técnicas de SBSE en las LPS. La problemática que se aborda es la **configuración de productos** reutilizando los modelos de características de n LPS. Es decir, se requiere configurar un conjunto de productos de software para generar nuevas aplicaciones a través de una nueva LPS denominada MPL.

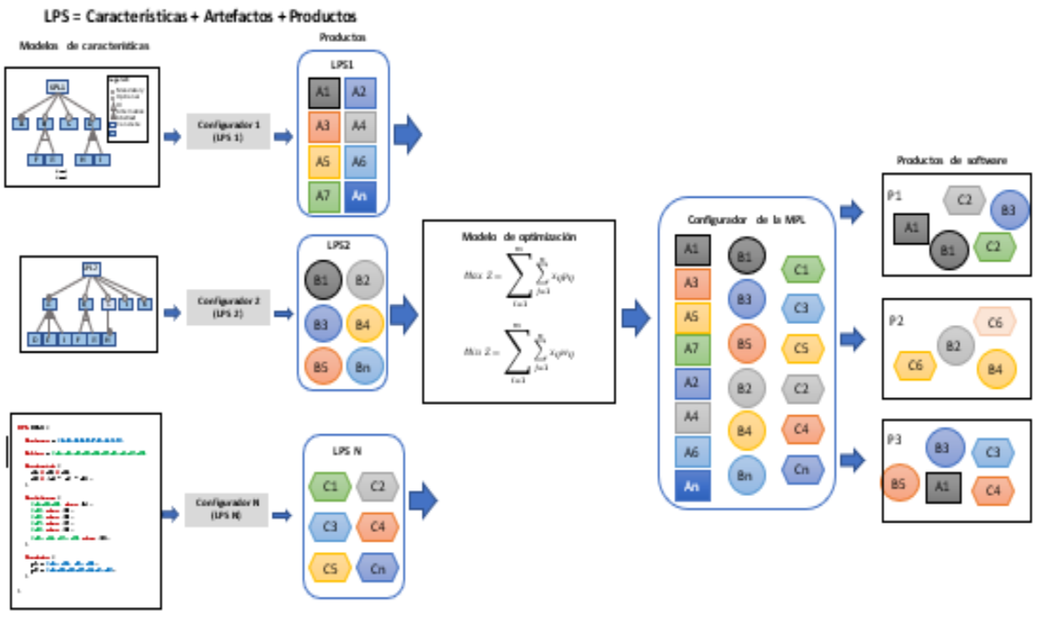
Identificación del problema

La configuración de una MPL es un problema de optimización combinatoria no lineal y multi-objetivo en el que se involucran un conjunto finito de Líneas de Productos de Software, una cierta cantidad de características y un conjunto finito de restricciones. Cada característica tiene diferentes valores y pertenece a diferentes modelos de características o Líneas de Productos de Software y cuyo número de combinaciones crece exponencialmente, dando lugar a múltiples configuraciones (soluciones óptimas) para un producto de software. Este fenómeno se denomina explosión combinatoria, debido a la gran cantidad de soluciones factibles y no factibles que aparecen. En la práctica, las técnicas exactas no son aplicables para su solución debido al gran tiempo de cómputo necesario. Estos problemas se conocen como problemas de tipo NP (nondeterministic polynomial time) completo y no es posible encontrar un algoritmo que los resuelva en un tiempo de cómputo polinomial.

El modelo de la mochila se consideró para resolver el problema de configuración de productos en una MPL como un caso especial de MMKP (Multiple Choice Knapsack Problem). Como se ilustra en la **Figura 6.4**, un configurador de MPL es análogo a una mochila en la cual es posible colocar n características proveniente de m LPS y combinarlas para producir productos. Por lo tanto, es posible utilizar el problema de la mochila como una herramienta para resolver el problema de la configuración de productos.

Formalización del problema

Figura 6.4 Configuración de productos en una MPL



Fuente de Consulta: Elaboración propia

Dado *m* modelos de características y una especificación de requisitos, una configuración de productos es la selección de características subdivididas en *m* LPS. Las características seleccionadas deben ser reutilización de los artefactos de la LPS disponibles. En consecuencia, si un producto de software en particular se define por n características, con cada característica $f_{ij} \in N_i$, la cual tiene una recompensa de selección P_{ij} por reutilización y un costo de desarrollo w_{ij}, es posible modelar el problema de configuración de productos en una MPL como un caso especial de MMKP. Ver figura 6.4.

Notación

Para formular el problema matemáticamente, la notación empleada se muestra en la Tabla 6.2.

Tabla 6.2 Notación empleada

Datos	
F	Característica
N	Número de características
M	Número de LPS = Número de modelos de características
I	Índice de una característica
J	Índice de una LPS
w_{ij}	Costo de desarrollo de la característica i si se asigna a la LPS j
p_{ij}	Beneficio o puntaje de la característica i si se asigna a la LPS j por reutilización
Variables	
x_{ij}	Variable igual a 1 si la característica i se asigna a la LPS j; 0 en caso contrario
Restricciones	
W	Presupuesto disponible
P	Valor límite mínimo de reutilización requerido
MA	Conjunto de características obligatorias
O	Conjunto de características opcionales
XOR	Conjunto de todas las características alternativas exclusivas
OR	Conjunto de todas las características alternativas no exclusivas

Fuente de Consulta: Elaboración propia

Definición del modelo de optimización

Para resolver el problema de la configuración de productos en una MPL, se formuló un modelo de optimización. Los elementos que integran el modelo son:

1) **Función objetivo o de aptitud:** se refiere a una medida cuantitativa del funcionamiento del sistema que se desea maximizar o minimizar.

En este caso, la problemática consiste en la configuración de productos de software o selección de características en una MPL. La problemática se abordó bajo un enfoque de optimización multi-objetivo considerando el antagonismo de criterios técnico y económico para la selección de características (costo y reutilización) facilitando así la toma de decisiones al momento de elegir las posibles configuraciones.

El modelo de optimización multi-objetivo (Ecuación 4) integra y optimiza simultáneamente dos funciones objetivo: reutilización del software (Z1) y costo (Z2) con el mismo conjunto de características.

La función objetivo (Ecuación 5) es una optimización por maximización, la cual tiene como objetivo seleccionar características de cada LPS con el máximo beneficio total (reutilización), mientras que el costo de desarrollo de las características elegidas no debe exceder la restricción de presupuesto W (Ecuación 7).

La función objetivo (Ecuación 6) es una optimización por minimización, la cual tiene como objetivo seleccionar características de cada LPS con el costo mínimo de desarrollo, mientras que la reutilización debe exceder el valor límite de reutilización (Ecuación 8).

$$Max\ Z_1, Min\ Z_2 \tag{4}$$

$$Max\ Z_1 = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} p_{ij} \quad (5)$$

$$Min\ Z_2 = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} w_{ij} \quad (6)$$

Sujeto a:

$$\sum_{i=1}^{m} \sum_{j \in N} w_{ij} x_{ij} \leq W \quad (7)$$

$$\sum_{i=1}^{m} \sum_{j \in N} w_{ij} p_{ij} <= P \quad (8)$$

$$\Sigma x_{ij} = 1, \forall i \in \{1,2 \dots n\} \quad (9)$$

- **Variables:** representan las decisiones que es posible tomar para afectar el valor de la función objetivo. En este caso la variable (x) representa la selección o no de la característica.

- **Restricciones:** representan el conjunto de relaciones (ecuaciones o inecuaciones) que ciertas variables están obligadas a satisfacer.

La Tabla 6.3 proporciona una muestra del conjunto de datos de características para generar una cartera de productos para la MPL.

Tabla 6.3 Datos del caso de prueba

No.	Característica	LPS	P1	P2	P3	P4	P5	P6	U	Reutilización	Costo
1	Luces	1	✓	✓	✓	✓	✓	✓	6	1	1000
2	Hardware	1	✓	✓	✓	✓	✓	✓	6	1	500
3	Arduino	1	✓					✓	2	0.333333333	400
4	RaspberryPi	1		✓	✓	✓	✓		4	0.666666667	500
5	R3MA	1		✓					1	0.166666667	725
6	R3MB	1			✓				1	0.166666667	300
7	R2MA	1				✓			1	0.166666667	500
8	Sensor	1	✓	✓	✓	✓	✓	✓	6	1	400
9	Temperatura	1	✓					✓	2	0.333333333	1000
10	Humedad	1	✓	✓	✓			✓	4	0.666666667	800
11	Gas	1	✓						1	0.166666667	500
12	MQ2	1	✓	✓				✓	3	0.5	200
13	MQ7	1	✓						1	0.166666667	300
14	Flama	1	✓						1	0.166666667	1650
15	Protocolo	1	✓						1	0.166666667	320
16	Wifi	1	✓	✓	✓		✓		4	0.666666667	298
17	Bluetooth	1		✓		✓		✓	3	0.5	2356
18	Iluminación	2	✓	✓	✓	✓	✓	✓	6	1	2250
19	Automática	2	✓	✓		✓			3	0.5	3500
20	Semiautomática	2	✓		✓		✓	✓	4	0.666666667	1500
21	Ambiente	2	✓		✓				2	0.333333333	2000
22	PC	2	✓	✓	✓		✓		4	0.666666667	1000
23	Switch	2	✓	✓	✓	✓	✓	✓	6	1	2000
24	Dispositivo	2	✓	✓	✓	✓	✓	✓	6	1	1000
25	RaspberryPi	2			✓	✓	✓	✓	4	0.666666667	900
26	Arduino	2	✓	✓					2	0.333333333	800
27	Alarma	2	✓		✓		✓	✓	4	0.666666667	1000
28	Inundación	2	✓		✓		✓		3	0.5	860
29	Fuego	2	✓		✓			✓	3	0.5	950
30	Vigilancia	2	✓	✓	✓	✓			4	0.666666667	2500
31	Sensor	2	✓	✓	✓	✓	✓	✓	6	1	1000
32	MQ7	2	✓		✓				2	0.333333333	500
33	Luminosidad	2	✓	✓	✓	✓	✓	✓	6	1	1200
34	Flama	2	✓	✓					2	0.333333333	980
35	DHT11	2	✓		✓		✓		3	0.5	250
36	DHT22	2		✓	✓	✓		✓	4	0.666666667	300
37	Movimiento	2	✓		✓				2	0.333333333	700
38	Humedad	2	✓		✓		✓		3	0.5	800
39	Bluetooth	2	✓	✓	✓			✓	4	0.666666667	1000
40	Wifi	2	✓			✓		✓	3	0.5	1200
41	ZigBee	2					✓		1	0.166666667	2000

Fuente de Consulta: Elaboración propia

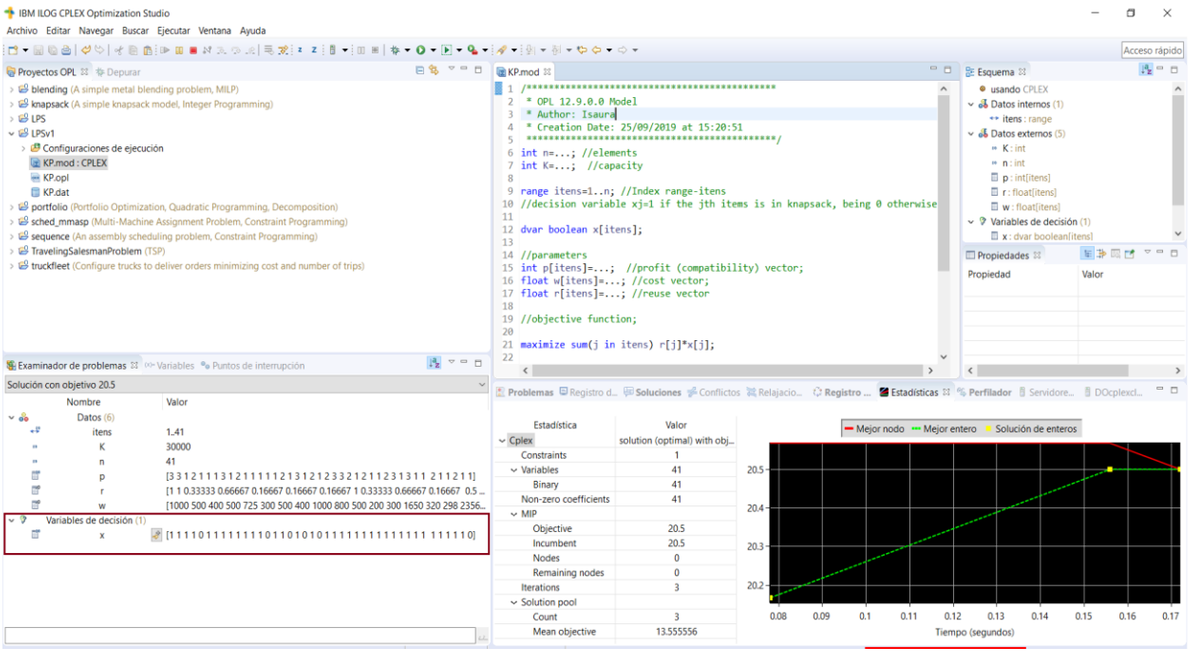
Solución del problema mediante una técnica de SBSE

Actualmente, existen herramientas computacionales para resolver problemas combinatorios y de optimización, sin embargo, estas pierden eficiencia conforme el número de variables incrementa, requiriendo mayor tiempo y recurso computacional. Por otro lado, existen herramientas comerciales como CPLEX que resuelve problemas de optimización de una forma exacta o mediante metaheurísticos como RiskOptimizer. Sin embargo, estas herramientas tienen un costo significativo para su uso y la optimización que realizan es de forma mono-objetivo. Por esta razón, se han desarrollado diversos métodos metaheurísticos para tener alternativas más rápidas y alcanzables para este tipo de problemas (NP y multi-objetivo) como por ejemplo los algoritmos genéticos.

Por otro lado, se identificó que no existe una solución exacta para el problema de la selección de características en una MPL porque es un problema multi-objetivo y es posible obtener un conjunto de soluciones factibles.

El problema se resolvió de forma mono-objetivo mediante IBM ILOG CPLEX Optimization Studio Community Edition obteniendo una solución exacta (**Figura 6.5**).

Figura 6.5. Resultados de la resolución del modelo de optimización utilizando CPLEX

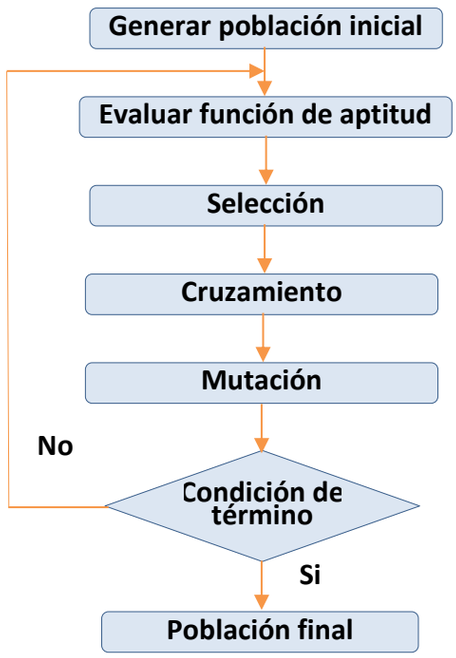


Fuente de Consulta: Elaboración propia

Implementación del algoritmo genético en Python

El problema se resolvió de forma multiobjetivo mediante la implementación de un algoritmo genético en Python. El funcionamiento general del algoritmo genético está descrito en la Figura 6.6. Previo a la generación de la población inicial se inicializan los elementos de la LPS (costo, reutilización y características). Se procede a generar la población inicial, es decir las soluciones (selección de características). Se evalúa la función de aptitud que selecciona a los más aptos, en función de costo y reutilización. A continuación, se realiza el cruzamiento seguido de la mutación. Se eligen a los más aptos para determinar la población final.

Figura 6.6 Proceso del algoritmo genético



Fuente de Consulta: Elaboración propia

1) **Generar una población inicial.** El proceso del algoritmo genético comienza con un conjunto de individuos que se denomina población (**Tabla 6.4**). Cada individuo representa una solución al problema que se desea resolver. A cada solución se le denomina individuo, el cual se codifica como un cromosoma (cadena) que a su vez está representado por un conjunto de parámetros (variables que representan las características de las LPS) conocidos como genes. Por lo general, para representar la cadena se utilizan valores binarios (0,1) que representan la selección o no de la característica.

En este problema, el algoritmo genético comienza con la generación de la población inicial de tamaño *n* (8 individuos) en la que sus soluciones son generadas de forma aleatoria. Cada solución consiste en *m* genes (características), de modo que cada índice de gen corresponde al índice de característica en la lista. Cada gen tiene un valor 1 o 0 que indica si la característica correspondiente está presente o no en la configuración de productos. La función que permite calcular la aptitud de cada configuración corresponde a la función del código en Python denominada **cal_fitness**.

Tabla 6.4 Población inicial del AG

No	Cromosomas (soluciones)	Z_1	Z_2	Aptitud $Max\, Z_1, Min\, Z_2$
1	[1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1]	34	15048	9042.4
2	[1 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0]	29	15023	9025.4
3	[0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 0 0]	31	15884	9542.8
4	[0 1 1 0 0 0 1 0 1 0 0 0 1 1 0 1 0 0 1 0 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0]	22	15428	9265.5999999999
5	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	40	28691	17230.6
6	[1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 1 1]	45	30189	0.0
7	[1 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 1 0 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0 0 1 0 0]	31	16000	9612.4
8	[1 1 0 1 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 0 1 0 0 1 0 1 0 0 0 0 1]	32	16090	9666.8

Fuente de Consulta: Elaboración propia

También en esta etapa, se definen los parámetros que influyen en la diversidad de la población como el tamaño de la población, número de generaciones, la tasa de cruce y la tasa de mutación. La **Tabla 6.5** muestra los parámetros de entrada utilizados para la ejecución del algoritmo genético.

Tabla 6.5. Parámetros

Parámetros	Descripción	Valor
Población	Número de individuos que conforman la población	8
Mutación	Índice de mutación (0.1 – 1.0)	0.5
Cruzamiento	Índice de cruzamiento (0.1 – 1.0)	0.8
Generaciones	Numero de generaciones	50

Fuente de Consulta: Elaboración propia

El tamaño de la población es el factor más distintivo que influye en la diversidad de la población.

Por ejemplo, una tasa de cruce excesivamente alta hará que la solución converja rápidamente antes de que se encuentre el óptimo. Por otro lado, una tasa de cruce baja disminuye la diversidad de la población y da como resultado un largo tiempo de cálculo.

La tasa de mutación también influye en el rendimiento del algoritmo genético ya que determina la frecuencia de búsqueda aleatoria. En general, se recomienda una tasa de mutación muy baja para evitar que el proceso del algoritmo genético se convierta en una búsqueda aleatoria pura, lo que perjudica la propiedad del AG.

2) **Evaluar la función de aptitud.** A cada uno de los cromosomas de la población se le aplica la función de aptitud o función objetivo para saber qué tan "buena" es la solución que se está codificando.

3) **Condición de término**. El algoritmo genético se detendrá cuando se alcance la solución óptima, pero ésta generalmente se desconoce, por lo que se utilizan otros criterios de detención. Normalmente se utilizan dos criterios: 1) ejecutar el algoritmo genético un número máximo de iteraciones (generaciones) o 2) detenerlo cuando no haya cambios en la población.

El proceso del algoritmo genético finaliza cuando se alcanza alguno de los criterios de termino fijados. Los más usuales suelen ser:

- **Los mejores individuos de la población representan soluciones suficientemente buenas para el problema que se desea resolver.**

- **La población ha convergido. Cuando esto ocurre la media de aptitud de la población se aproxima a la aptitud del mejor individuo.**

- Se ha alcanzado el número de generaciones máximo especificado

4) Mientras no se cumpla la condición de término se repiten los siguientes pasos:

- **Selección.** Se eligen los mejores individuos de la población.

- **Cruzamiento.** En la literatura, se proponen muchas variantes para el operador de crossover, pero el principio común es combinar dos cromosomas para generar cromosomas de próxima generación, mediante un intercambio de genes simple o no, con pequeñas variaciones.

- **Mutación.** La mutación consiste en cambiar aleatoriamente los valores del gen para generar una nueva combinación de genes para la próxima generación. Matemáticamente, el interés principal de la mutación consiste en saltar las soluciones óptimas locales.

5) **Población final.** Los nuevos individuos, producto de los cruces y mutaciones se insertan en la población para dar lugar a la nueva generación (**Tabla 6.6**).

Tabla 6.6. Población final del AG

No	Cromosomas (soluciones)	Z_1	Z_2	Aptitud $Max\,Z_1, Min\,Z_2$
1	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	45	29989	18011.4
2	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	45	29989	18011.4
3	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	45	29989	18011.4
4	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	45	29989	18011.4
5	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	44	27989	16810.9999996
6	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	42	27739	16660.1999997
7	[1 0 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	42	29489	17710.1999997
8	[1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]	45	29989	18011.4

Fuente de Consulta: Elaboración propia

Resultados computacionales

A continuación, se presenta los resultados empíricos de los experimentos que se realizaron para evaluar el algoritmo genético.

El algoritmo se implementó en Python y se probaron en una PC con un procesador Intel(R) Core (TM) i7 a 2,8 GHz, 16 GB de RAM y sistema operativo Microsoft Windows 10.

Para evaluar el rendimiento del modelo de optimización multi-objetivo, es necesario seleccionar el mejor enfoque para identificar soluciones candidatas. Se utilizaron las siguientes métricas: valor de conveniencia para la reutilización y costo.

El modelo de optimización multi-objetivo propuesto se ejecutó un total de diez veces bajo los mismos parámetros. Para cada ejecución, los valores iniciales de las variables de optimización se cambiaron para iniciar la búsqueda desde diferentes puntos y observar si la evolución se comporta de la misma manera. Las diez ejecuciones del modelo dieron resultados mayores que los requeridos, por lo cual es posible concluir que el modelo se desempeña adecuadamente en el escenario de optimización propuesto. Se propuso encontrar la configuración de las características que permitirían generar un producto de software con un costo límite de $30,000.00 (presupuesto) y una reutilización mínima de 15.

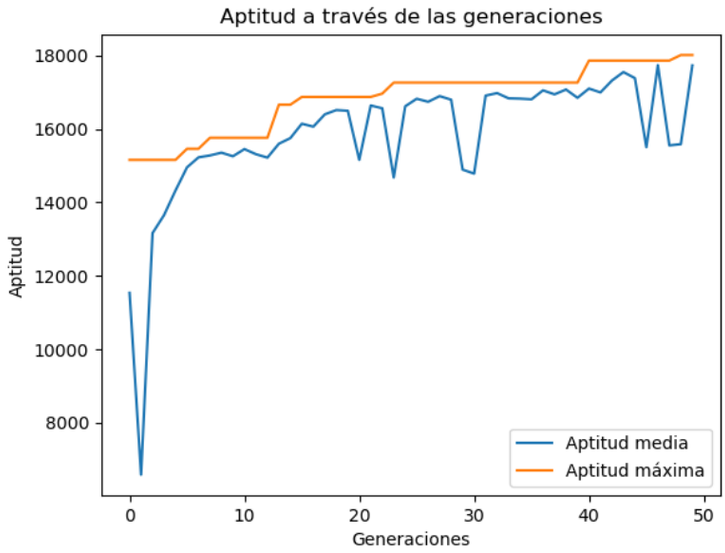
La Figura 6.7 muestra un ejemplo de los Frentes de Pareto obtenidos con las pruebas realizadas en el algoritmo genético propuesto y se visualiza cómo cambia el estado físico (aptitud) con cada generación. Los resultados indican con un 1 si la característica se selecciona y con un 0 si no se selecciona. La **Tabla 6.7** muestra la selección óptima del conjunto de soluciones óptimas (población final)

Tabla 6.7. Características seleccionadas

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1

Fuente de Consulta: Elaboración propia

Figura 6.7 Frentes de Pareto



Fuente de Consulta: Elaboración propia

6.4 Conclusiones

El mercado demanda la búsqueda de enfoques novedosos para la configuración y generación de sistemas más complejos, evolución y mantenimiento como las Líneas de Productos de Software, Multilíneas de Productos de Software o técnicas de búsqueda y optimización.

La Ingeniería de Software Basada en Búsqueda es de gran importancia no solo para la Ingeniería de Software tradicional sino para la Ingeniería de Líneas de Productos de Software porque es un medio viable para guiar y ayudar a los ingenieros de aplicaciones en la toma de decisiones en el ciclo de vida desarrollo de software principalmente durante las fases de configuración, depuración o pruebas.

La implementación del algoritmo genético en Python permitió estimar la factibilidad para desarrollar nuevos productos de software en un configurador MPL reutilizando los insumos de LPS previamente desarrolladas. Esto debido a que el algoritmo facilita la selección y combinación de características proporcionadas por múltiples LPS (modelos de características) porque reduce la carga mental y complejidad cognitiva asociadas con el proceso de configuración de productos. Asimismo, mediante la aplicación de técnicas de SBSE es posible diseñar productos de software que mejor se adapten a los requisitos de las partes interesadas en base a dos objetivos de optimización: costo y reutilización.

Los resultados obtenidos comprueban que la optimización de la selección de características en la fase de configuración otorga grandes beneficios tanto a la calidad y desempeño final del producto de software, así como un mayor beneficio económico al ajustar a medida del cliente el costo de producción de un producto de software determinado.

Como trabajo a futuro se pretende actualizar e introducir nuevas funcionalidades a la MPL, por lo que se vislumbra refinar y complementar el modelo de optimización e implementación en Python con variables (características) adicionales para personalizar otros productos de software y abarcar otros segmentos de mercado relacionados a la automatización inteligente de edificios (interiores y exteriores) y así tener una mayor variabilidad de productos de software.

Como trabajo adicional, se pretende utilizar el caso de estudio y resolver el modelo de optimización con otras técnicas de Ingeniería de Software Basada en Búsqueda (por ejemplo, colonia de hormigas o recocido simulado). Asimismo, el modelo podría utilizarse como base para aplicarse en otras etapas del ciclo de vida (pruebas, ubicación de características, mantenimiento) de las MPL.

6.5 Apéndice

Se presenta abajo el código en Python del algoritmo genético, mismo que se describe en la Figura 5.6.

```
import numpy as np
import pandas as pd
import random as rd
from random import randint
import matplotlib.pyplot as plt

# --------  Inicialización de la lista de elementos de la LPS ----------------------
reusability=[1, 1, 0.333333333, 0.666666667, 0.166666667, 0.166666667, 0.166666667, 1, 0.3333333
33, 0.666666667, 0.166666667, 0.5, 0.166666667, 0.166666667, 0.166666667, 0.666666667, 0.5, 1, 0.
5, 0.666666667, 0.333333333, 0.666666667, 1, 1, 0.666666667, 0.333333333, 0.666666667, 0.5, 0.5, 0
.666666667, 1, 0.333333333, 1, 0.333333333, 0.5, 0.666666667, 0.333333333, 0.5, 0.666666667, 0.5,
0.166666667]
reusability=np.asarray(reusability)

value=[3, 3, 1, 2, 1, 1, 1, 3, 1, 2, 1, 1, 1, 1, 1, 2, 1, 3, 1, 2, 1, 2, 3, 3, 2, 1, 2, 1, 1, 2, 3, 1, 3, 1, 1, 2, 1, 1,
2, 1, 1]
value=np.asarray(value)

weight=[1000, 500, 400, 500, 725, 300, 500, 400, 1000, 800, 500, 200, 300, 1650, 320, 298, 2356, 225
0, 3500, 1500, 2000, 1000, 2000, 1000, 900, 800, 1000, 860, 950, 2500, 1000, 500, 1200, 80, 250, 300,
700, 800, 1000, 1200, 2000]
weight=np.asarray(weight)
y = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,
36,37,38,39,40,41]

noFeatures=np.asarray(y)
limite = 30000
capReuse=15
print('Feature.  Weight  Value  Reusability')
for i in range(noFeatures.shape[0]):
    print('{0}{1}{2}{3}\n'.format(noFeatures[i], weight[i], value[i], reusability[i]))

# Generación de la población inicial
solutions_per_pop = 8
pop_size = (solutions_per_pop, noFeatures.shape[0])
print('Tamaño de la población= {}'.format(pop_size))
initial_population = np.random.randint(2, size = pop_size)
initial_population = initial_population.astype(int)
num_generations = 50
print('Población inicial: \n{}'.format(initial_population))

# Definición de la función de aptitud
def cal_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight) #costo
        if S2 <= threshold and S1>=capReuse:
            fitness[i] = (S1*0.6)+(S2*0.4)
        else :
            fitness[i] = 0
        print ("*Población: ",population[i],"   value (S1): ", S1, "    weight (S2): ", S2, "     fitness: ", fitnes
s[i])
```

```
return fitness.astype(int);

fitness_initial_gen=cal_fitness(weight, value, initial_population, limite)
print('Fitness de la primera generación: \n{}\n'.format(fitness_initial_gen))

# Selección de los individuos más aptos para que puedan someterse a un cruzamiento
def selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return parents

# Para el cruzamiento, se establece una tasa de cruce
def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    crossover_point = int(parents.shape[1]/2)
    crossover_rate = 0.8
    i=0
    while (parents.shape[0] < num_offsprings):
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        x = rd.random()
        if x > crossover_rate:
            continue
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        offsprings[i,0:crossover_point] = parents[parent1_index,0:crossover_point]
        offsprings[i,crossover_point:] = parents[parent2_index,crossover_point:]
        i=+1
    return offsprings

# En la mutación, se utiliza la técnica de cambio de bits, es decir, si el gen
seleccionado que va a sufrir mutación es 1, cámbielo a 0 y viceversa.
def mutation(offsprings):
    mutants = np.empty((offsprings.shape))
    mutation_rate = 0.5
    for i in range(mutants.shape[0]):
        random_value = rd.random()
        mutants[i,:] = offsprings[i,:]
        if random_value > mutation_rate:
            continue
        int_random_value = randint(0,offsprings.shape[1]-1)
        if mutants[i,int_random_value] == 0 :
            mutants[i,int_random_value] = 1
        else :
            mutants[i,int_random_value] = 0
    return mutants

def optimize(weight, value, population, pop_size, num_generations, threshold):
    parameters, fitness_history = [], []
    num_parents = int(pop_size[0]/2)
    num_offsprings = pop_size[0] - num_parents
    for i in range(num_generations):
        fitness = cal_fitness(weight, value, population, threshold)
        fitness_history.append(fitness)
        parents = selection(fitness, num_parents, population)
```

```
        offsprings = crossover(parents, num_offsprings)
        mutants = mutation(offsprings)
        population[0:parents.shape[0], :] = parents
        population[parents.shape[0]:, :] = mutants

    print('Última generación: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(weight, value, population, limite)
    print('Fitness de la última generación: \n{}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen == np.max(fitness_last_gen))
    parameters.append(population[max_fitness[0][0],:])
    return parameters, fitness_history

#Los elementos correspondientes de los parámetros en la matriz noFeatures serán los que se elegirán
parameters, fitness_history = optimize(weight, value, initial_population, pop_size, num_generations, li
mite)
print('Los paramétros optimizados son \n{}'.format(parameters))
selected_items = noFeatures * parameters
print('\n Elementos seleccionados')
for i in range(selected_items.shape[1]):
  if selected_items[0][i] != 0:
     print('{}\n'.format(selected_items[0][i]))

# Generación de la gráfica para visualizar cómo cambia el estado físico con cada generación.
fitness_history_mean = [np.mean(fitness) for fitness in fitness_history]
fitness_history_max = [np.max(fitness) for fitness in fitness_history]
plt.plot(list(range(num_generations)), fitness_history_mean, label = 'Aptitud media')
plt.plot(list(range(num_generations)), fitness_history_max, label = 'Aptitud máxima')
plt.legend()
plt.title('Aptitud a través de las generaciones')
plt.xlabel('Generaciones')
plt.ylabel('Aptitud')
plt.show()
print(np.asarray(fitness_history).shape)
```

6.6Agradecimientos

Agradecemos al Tecnológico Nacional de México campus Orizaba y Universidad Veracruzana por el apoyo brindado para llevar a cabo esta investigación.

6.7 Referencias

Afzal, U., Mahmood, T., Rauf, I., & Shaikh, Z. A. (2014). Minimizing feature model inconsistencies in software product lines. *17th IEEE International Multi Topic Conference: Collaborative and Sustainable Development of Technologies, IEEE INMIC 2014 - Proceedings*, 137–142. https://doi.org/10.1109/INMIC.2014.7097326

Afzal, U., Mahmood, T., & Shaikh, Z. (2016). Intelligent software product line configurations: A literature review. *Computer Standards and Interfaces*, *48*, 30–48. https://doi.org/10.1016/j.csi.2016.03.003

Alsariera, Y. A., Majid, M. A., & Zamli, K. Z. (2015). SPLBA: An interaction strategy for testing software product lines using the Bat-inspired algorithm. *2015 4th International Conference on Software Engineering and Computer Systems, ICSECS 2015: Virtuous Software Solutions for Big Data*, 148–153. https://doi.org/10.1109/ICSECS.2015.7333100

Alsawalqah, H. I., Kang, S., & Lee, J. (2014). A method to optimize the scope of a software product platform based on end-user features. *Journal of Systems and Software*, *98*, 79–106. https://doi.org/10.1016/j.jss.2014.08.034

Asadi, M., Soltani, S., Gasevic, D., Hatala, M., & Bagheri, E. (2014). Toward automated feature model configuration with optimizing non-functional requirements. *Information and Software Technology*, *56*(9), 1144–1165. https://doi.org/10.1016/j.infsof.2014.03.005

Benavides, D., Trinidad, P., & Ruiz-Cortés, A. (2005). Automated Reasoning on Feature Models. In *Proceedings of the 17th international conference on Advanced Information Systems Engineering* (pp. 491–503). https://doi.org/10.1007/11431855_34

Clements, P. C., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Retrieved from http://dl.acm.org/citation.cfm?id=501065

Colanzi, T. E., Vergilio, S. R., Gimenes, I. M. S., & Oizumi, W. N. (2014). A search-based approach for software product line design. *Proceedings of the 18th International Software Product Line Conference on - SPLC '14*, 237–241. https://doi.org/10.1145/2648511.2648537

Connolly, D., Martello, S., & Toth, P. (1991). Knapsack Problems: Algorithms and Computer Implementations. *The Journal of the Operational Research Society*, *42*(6), 513. https://doi.org/10.2307/2583458

Cruz, J., Neto, P. S., Britto, R., Rabelo, R., Ayala, W., Soares, T., & Mota, M. (2013). Toward a hybrid approach to generate Software Product Line portfolios. *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, 2229–2236. https://doi.org/10.1109/CEC.2013.6557834

dos Santos Neto, P. de A., Britto, R., Rabêlo, R. de A. L., Cruz, J. J. de A., & Lira, W. A. L. (2016). A hybrid approach to suggest software product line portfolios. *Applied Soft Computing Journal*, *49*, 1243–1255. https://doi.org/10.1016/j.asoc.2016.08.024

Font, J., Arcega, L., Haugen, Ø., & Cetina, C. (2016). Feature location in model-based software product lines through a genetic algorithm. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9679*, 39–54. https://doi.org/10.1007/978-3-319-35122-3_3

Guo, J., White, J., Wang, G., Li, J., & Wang, Y. (2011). A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, *84*(12), 2208–2221. https://doi.org/10.1016/j.jss.2011.06.026

Harman, M. (2012). The role of artificial intelligence in software engineering. *2012 1st International Workshop on Realizing AI Synergies in Software Engineering, RAISE 2012 - Proceedings*, 1–6. https://doi.org/10.1109/RAISE.2012.6227961

Harman, M., & Clark, J. (2004). Metrics are fitness functions too. *Proceedings - International Software Metrics Symposium*, 58–69. https://doi.org/10.1109/METRIC.2004.1357891

Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, *43*(14), 833–839. https://doi.org/10.1016/S0950-5849(01)00189-6

Henard, C., Papadakis, M., Perrouin, G., Klein, J., & Traon, Y. Le. (2013). Multi-objective test generation for software product lines. *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*, 62. https://doi.org/10.1145/2491627.2491635

Holl, G., Grünbacher, P., & Rabiser, R. (2012). A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology*, *54*(8), 828–852. https://doi.org/10.1016/j.infsof.2012.02.002

Lian, X., & Zhang, L. (2015). Optimized feature selection towards functional and non-functional requirements in Software Product Lines. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, 191–200. https://doi.org/10.1109/SANER.2015.7081829

Lienhardt, M., Damiani, F., Donetti, S., & Paolini, L. (2018). Multi Software Product Lines in the Wild. *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems - VAMOS 2018*, 89–96. https://doi.org/10.1145/3168365.3170425

Pereira, J. A., Matuszyk, P., Krieter, S., Spiliopoulou, M., & Saake, G. (2016). A feature-based personalized recommender system for product-line configuration. *ACM SIGPLAN Notices*, *52*(3), 120–131. https://doi.org/10.1145/3093335.2993249

Pohl, K., Böckle, G., & Linden, F. J. van der. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques* (Vol. 10). Retrieved from https://dl.acm.org/citation.cfm?id=1095605

Robak, S., & Pieczynski, A. (2003). Employing fuzzy logic in feature diagrams to model variability in software product-lines. *Proceedings - 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, ECBS 2003*, 305–311. https://doi.org/10.1109/ECBS.2003.1194812

Savolainen, J., Mannion, M., & Kuusela, J. (2012). Developing platforms for multiple software product lines. *ACM International Conference Proceeding Series*, *1*, 220–228. https://doi.org/10.1145/2362536.2362567

Tan, L., Lin, Y., Ye, H., & Zhang, G. (2013). Improving product configuration in software product line engineering. *Conferences in Research and Practice in Information Technology Series*, *135*, 125–134. Retrieved from http://crpit.com/confpapers/CRPITV135Tan.pdf

Trujillo-Tzanahua, G.-I., Juarez-Martinez, U., Aguilar-Lasserre, A.-A., Cortes-Verdin, M.-K., & Azzaro-Pantel, C. (2019). Multiple Software Product Lines to configure applications of Internet of Things. *IET Software*. https://doi.org/10.1049/iet-sen.2019.0032

Wang, S., Ali, S., & Gotlieb, A. (2015). Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software*, *103*, 370–391. https://doi.org/10.1016/j.jss.2014.08.024

Wang, Y., & Pang, J. (2014). Ant colony optimization for feature selection in software product lines. *Journal of Shanghai Jiaotong University (Science)*, *19*(1), 50–58. https://doi.org/10.1007/s12204-013-1468-0

Xue, Y., Chen, M., Tan, T. H., Liu, Y., Dong, J. S., & Sun, J. (2015). Optimizing selection of competing features via feedback-directed evolutionary algorithms. *2015 International Symposium on Software Testing and Analysis, ISSTA 2015 - Proceedings*, 246–256. https://doi.org/10.1145/2771783.2771808

Xue, Y., Zhong, J., Tan, T. H., Liu, Y., Cai, W., Chen, M., & Sun, J. (2016). IBED: Combining IBEA and DE for optimal feature selection in software product line engineering. *Applied Soft Computing Journal*, *49*, 1215–1231. https://doi.org/10.1016/j.asoc.2016.07.040