Optimización por metaheurísticas aplicada a la ingeniería estructural

Luis Ivan Velasco Enriquez
Héctor Guerrero Bobadilla
Benito Velasco Hernández

INSTITUTO DE INGENIERÍA UNAM

Series
Instituto de Ingeniería UNAM
Publicación arbitrada

SD 62
Junio, 2025

Imagen generada con ChatGPT

Esta página ha sido intencionalmente dejada en blanco

Publicación arbitrada
ISBN: 978-607-587-536-1
DOI: 10.22201/iingen.9786075875361e.2025

Optimización por metaheurísticas aplicada a la ingeniería estructural

Luis Ivan Velasco Enriquez[1]
IIUNAM
LVelascoE@iingen.unam.mx

Héctor Guerrero Bobadilla[2]
IIUNAM
HGuerreroB@iingen.unam.mx

Benito Velasco Hernández[3]
SEDENA
fable_1024@hotmail.com

Serie Docencia
SD 62
Junio 2025

[1] Candidato a Doctor en Ingeniería Civil, Instituto de Ingeniería, UNAM
[2] Investigador Asociado, Instituto de Ingeniería, UNAM
[3] Coronel Ingeniero Constructor, SEDENA

Optimización por metaheurísticas aplicada a la ingeniería estructural

Primera edición, 25 junio de 2025
SD 62

Instituto de Ingeniería, UNAM
Ciudad Universitaria, CP 04510, Ciudad de México

ISBN: 978-607-587-536-1
doi: https://doi.org/10.22201/iingen.9786075875361e.2025

La obra fue editada por el Instituto de Ingeniería, de la Universidad Autónoma de México (IIUNAM). El cuidado de la edición estuvo a cargo de la Unidad de Promoción y Comunicación del IIUNAM. Esta obra está gratuitamente disponible para consulta e impresión, en archivo PDF de 13.2 MB, en la sección de Publicaciones del portal electrónico del IIUNAM, http://www.iingen.unam.mx, desde que se terminó de editar.

Dedicatoria

Para Vero y para Diana
Luis Ivan Velasco Enriquez

A mi familia y amigos
Héctor Guerrero Bobadilla

Esta página ha sido intencionalmente dejada en blanco

Prefacio

El objetivo de este libro es brindar a estudiantes, académicos y profesionales de la Ingeniería Estructural una introducción amigable y clara sobre el uso de los algoritmos metaheurísticos en el campo de la optimización. A lo largo del libro, se encontrarán diferentes ejemplos de problemas de optimización relacionados con la Ingeniería Estructural, resueltos por medio de un amplio abanico de algoritmos como Recocido Simulado, Algoritmos Genéticos, Estrategias Evolutivas, entre otros. Cada ejemplo cuenta con una descripción sistemática del problema de optimización en cuestión y un código desarrollado en Matlab, junto con una explicación detallada de su funcionamiento, para aplicar los algoritmos metaheurísticos.

Al finalizar la lectura de este documento, se conocerán los temas más relevantes sobre las metaheurísticas, su campo de aplicación y posibles usos en la Ingeniería Estructural, los principios sobre los que estos algoritmos trabajan, las controversias que existen sobre el creciente número de metaheurísticas y la imposibilidad demostrada para afirmar que una metaheurística específica siempre presentará un mayor desempeño que las demás existentes. Adicionalmente, los ejemplos aquí presentados brindarán las bases para permitir a las y los profesionistas el comenzar a aplicar las metaheurísticas en problemas prácticos y relacionados con la investigación.

Ciudad Universitaria, Cd. Mx.
Junio 2025
Luis Ivan Velasco Enriquez
Héctor Guerrero Bobadilla
Benito Velasco Hernández

Esta página ha sido intencionalmente dejada en blanco

Agradecimientos

Los autores agradecen el apoyo del Instituto de Ingeniería de la Universidad Nacional Autónoma de México por brindar las facilidades y los recursos necesarios para la publicación de este libro.

Esta página ha sido intencionalmente dejada en blanco

Resumen

Las metaheurísticas son algoritmos de inteligencia artificial que permiten resolver una gran variedad de problemas de optimización. Gracias a su versatilidad, las metaheurísticas pueden ser utilizadas en la generación automática de diseños de estructuras con el objetivo de reducir sus costos, tiempos de ejecución, huella de carbono, entre otros. Para dar a conocer de una manera más amplia estos algoritmos a académicos, estudiantes y practicantes, este libro aborda el uso de metaheurísticas en la Ingeniería Estructural por medio de un enfoque académico, riguroso y transparente, respaldado por una amplia consulta de material publicado en revistas de investigación altamente renombradas. Para conseguir su objetivo, el libro abre con una introducción amplia sobre las metaheurísticas, lo cual permite a las y los lectores familiarizarse con los conceptos fundamentales del tema y comprender la amplia aplicabilidad de estos algoritmos. El resto de los capítulos comprende diferentes ejemplos de optimización por medio de metaheurísticas. Para cada ejemplo, se brinda su código en Matlab lo que permite a las y los estudiantes no solo comprender la formulación de los algoritmos, sino adaptarlos y modificarlos para realizar sus propios trabajos académicos o de investigación.

Palabras clave*: Metaheurísticas, optimización, inteligencia artificial, ingeniería estructural*

Esta página ha sido intencionalmente dejada en blanco

Abstract

Metaheuristics are artificial intelligence algorithms that can solve various optimization problems. Due to their versatility, metaheuristics can be used in the automatic generation of structural designs with the objective of reducing their costs, execution times, carbon footprint, among others. To make these algorithms more widely known to academics, students, and practitioners, this book addresses the use of metaheuristics in Structural Engineering through an academic, rigorous, and transparent approach supported by extensive consultation of published material in highly renowned research journals. To achieve its goal, the book opens with a comprehensive introduction to metaheuristics, which allows readers to become familiar with the fundamental concepts of the subject and to understand the broad applicability of these algorithms. The remaining chapters cover different examples of optimization using metaheuristics. For each example, Matlab code is provided, allowing students not only to understand the formulation of the algorithms but also to adapt and modify them for their own academic or research work.

Keywords: *Metaheuristics, optimization, artificial intelligence, structural engineering*

Esta página ha sido intencionalmente dejada en blanco

Índice

Esta página ha sido intencionalmente dejada en blanco

1. Introducción

La inteligencia artificial (IA) es la rama de las ciencias de la computación que se encarga de estudiar la imitación de comportamientos inteligentes. Gracias a su alto nivel de desarrollo, hoy en día la IA engloba a los algoritmos predictivos, de clasificación, generativos, entre otros (Abioye *et al*., 2021). Estos algoritmos de IA son de gran utilidad dentro de la ingeniería estructural debido a que permiten apoyar en diferentes tareas como lo son el estudio de las propiedades de materiales constructivos, la predicción de comportamientos estructurales, el monitoreo de salud estructural o la creación de diseños óptimos (Tapeh & Naser, 2023). Si bien todas estas áreas presentan un diferente nivel de desarrollo, una que ha generado un gran interés dentro de la comunidad científica es la de la optimización de estructuras, esto debido a que dicho campo permite la obtención de diseños estructurales que cumplan con mayores estándares de resiliencia y sostenibilidad (Liao *et al*., 2024).

A pesar de que existen métodos muy conocidos para resolver problemas de optimización (p. ej. la programación lineal, entera, mixta, etc.), la gran mayoría de los problemas que se afrontan en la práctica profesional son más complejos de lo que estos métodos pueden manejar. Problemas como el minimizar el material empleado en la elaboración de una armadura o el diseñar un elemento estructural con la mínima huella de carbono posible son ejemplos de problemas para los cuales los métodos de optimización convencionales no ofrecen una solución adecuada. Sin embargo, esto no significa que no existan maneras de resolverlos. A lo largo de este libro se presentan y discuten los algoritmos denominados como metaheurísticos, los cuales han mostrado ser sorprendentemente eficaces en la resolución de una gran variedad de problemas de optimización complejos de muy diferentes áreas. Para ayudar a ejemplificar la implementación de esta clase de algoritmos, en los subsecuentes capítulos se resolverán diferentes problemas de optimización de tipo combinatorio, esto significa que las características de cada posible solución estarán representadas por medio de valores discretos, lo cual está en sintonía con los requerimientos de medidas estandarizadas exigidas por la práctica ingenieril (Osaba *et al*., 2021; Velasco, Hospitaler, *et al*., 2022b). Adicionalmente, todos los ejemplos brindan los códigos utilizados, tanto en forma de texto como en su versión electrónica descargable desde el siguiente repositorio público: https://github.com/LuisVelasc/Libro-Optimizacion-por-metaheuristicas-aplicada-a-la-ingenieria-estructural.

Un aspecto fundamental a señalar es que, si bien en este libro se resuelve cada problema por medio de un único algoritmo por capítulo, esta relación entre problemas y técnicas de resolución no es restrictiva. Esto quiere decir que resulta posible emplear diferentes metaheurísticas para resolver un mismo problema. Tal enfoque fue omitido en el desarrollo de este libro con el objetivo de focalizar la atención de los lectores a las características particulares de cada algoritmo, evitando a su vez una saturación de conceptos nuevos en las personas menos familiarizadas con este tema.

El documento inicia con una introducción a los algoritmos metaheurísticos, también llamados simplemente metaheurísticas. Aspectos como la complejidad de un problema y su relación con el tiempo de cómputo requerido para resolverlo son abordados en el capítulo 2. Adicionalmente se habla sobre los orígenes de las metaheurísticas y la manera en que estas se clasifican, tocando la controversia que actualmente existe respecto a su creciente número y la poca diferenciación o sustento que algunas de ellas presentan. Uno de los teoremas más importantes en este campo, denominado como Teorema *No Free Lunch*, será comentado en este capítulo, finalizando con una breve revisión de trabajos de investigación que han sido publicados sobre la optimización de problemas relacionados con la Ingeniería Estructural por medio de metaheurísticas.

En el capítulo 3 se presenta el primer ejemplo de aplicación de los algoritmos metaheurísticos por medio de la resolución del bien conocido Problema del Viajero aplicado a los Pueblos Mágicos de México. Si bien este problema no guarda relación con la Ingeniería Estructural, se decidió utilizarlo como ejemplo inicial del libro debido a que la Investigación de Operaciones es el campo de estudio que dio origen a los algoritmos de optimización, por lo tanto, la introducción más natural a estas herramientas pasa por los problemas clásicos de toma de decisiones. Por tratarse de un problema introductorio, en este ejemplo se presentan conceptos centrales de las metaheurísticas como su naturaleza estocástica, la necesidad de calibrar sus parámetros de comportamiento y la imposibilidad para conocer la cercanía de las soluciones obtenidas con el óptimo global del problema. Por último, se señala que el algoritmo metaheurístico utilizado en este capítulo es Optimización por Colonia de Hormigas el cual toma inspiración del comportamiento colectivo de las hormigas para encontrar las rutas más cortas entre sus fuentes de alimentación y sus nidos.

En el capítulo 4 se retoma un problema presentado en el *Congress on Evolutionary Computation* (CEC) de 2005, el cual consistía en hallar el mínimo de una función multimodal escalable a cualquier número de dimensiones. Este capítulo tiene como finalidad el mostrar cómo la cantidad de soluciones de un problema de tipo NP puede incrementarse rápidamente y de manera exponencial, volviéndose extremadamente complejo de resolver incluso para un número pequeño de dimensiones. En este ejemplo se utiliza la metaheurística conocida como Búsqueda Tabú.

En el capítulo 5 se realiza el primer ejemplo de aplicación de las metaheurísticas en un problema de la Ingeniería Estructural: minimizar la cantidad de acero dispuesto en una armadura. Este problema es resuelto por medio de un algoritmo conocido como Recocido Simulado, el cual fue una de las primeras metaheurísticas en ser desarrolladas. A pesar de que las armaduras son de las estructuras más simples que existen, el aplicarles un procedimiento de optimización resulta laborioso ya que es necesario el programar

un algoritmo que evalúe las fuerzas en cada una de sus barras. Por otra parte, se demostrará lo extremadamente complejo que es realizar una optimización estructural de tipo topológica.

En los capítulos 6 y 7 se resuelve el problema de optimizar el diseño de una viga de concreto reforzado simplemente apoyada. Este ejemplo es resuelto por dos metaheurísticas muy similares: Algoritmos Genéticos y Estrategias Evolutivas. A pesar de ser algoritmos semejantes, se observa que pequeñas diferencias en la forma de crear los diseños candidatos a solución permite que una de las dos metaheurísticas presente un desempeño superior en este problema en particular.

Por su parte, en el capítulo 8 se muestra una estrategia para evadir el inconveniente de programar procedimientos propios del análisis estructural en los procesos de optimización, esto es: utilizar de manera conjunta programas de elementos finitos con algoritmos metaheurísticos. El problema resuelto en este capítulo consiste en minimizar el error entre la respuesta brindada por el modelo de una columna de suelo sometida a una excitación sísmica y una respuesta de referencia que se considera como medida en campo. El algoritmo utilizado es Búsqueda por Vecindario Variable, que utiliza diferentes estructuras de vecindario para crear nuevos candidatos a solución. Como se muestra en el ejemplo, el principio que le permite a este algoritmo el resolver problemas de optimización se encuentra relacionado con las diferencias observadas entre Algoritmos Genéticos y Estrategias Evolutivas, según se mostró en los capítulos 6 y 7. En su último capítulo el libro brinda una serie de conclusiones generales sobre el uso de las metaheurísticas.

Esta página ha sido intencionalmente dejada en blanco

2. Optimización por algoritmos metaheurísticos

En este capítulo se presenta una noción general de lo que es un algoritmo metaheurístico, de tal manera que, después de leerlo, el o la lectora se habrá familiarizado con los tópicos más relevantes de este campo, como la definición misma de una metaheurística, los problemas en los que se pueden aplicar o la forma de clasificarlas. Es importante mencionar que algunas de las ideas presentadas en este capítulo fueron retomadas de la tesis de maestría del primer autor del libro (Velasco & Hospitaler, 2021); sin embargo, se realizaron modificaciones y actualizaciones para mantener la originalidad del documento. Adicionalmente, se abordará uno de los temas más polémicos de este campo de investigación, que es la validez y relevancia de las numerosas metaheurísticas que son descritas por sus propios creadores como "innovadoras". Debido al extenso número de metaheurísticas existentes, este documento se limitará a realizar una descripción básica de las más utilizadas, mismas que serán aplicadas a problemas de optimización en los subsecuentes capítulos. Por último, se realizará una breve revisión sobre la aplicación de estos algoritmos a problemas relacionados con la Ingeniería Estructural. Pero antes, resulta necesario hablar un poco sobre la resolución de problemas y el origen de los algoritmos metaheurísticos.

2.1. Sobre los problemas de optimización y su complejidad

En la rutina diaria, las personas se encuentran con una variedad de desafíos que requieren enfoques diversos para su resolución. Estos pueden abarcar desde cuestiones académicas, como resolver un examen de matemáticas o derivar funciones, hasta aspectos más comunes como gestionar los ingresos familiares, establecer el orden de las actividades diarias o elegir la ruta más eficiente para llegar al trabajo o la escuela. Aunque algunos de estos problemas son lo bastante simples como para ser resueltos rápidamente, muchos otros resultan demasiado complejos como para encontrarles una respuesta en un corto periodo de tiempo.

Para comprender con mayor claridad esta idea, consideremos el conocido Problema del Viajero, el cual se define de la siguiente manera: dado un conjunto de n ciudades que deben ser visitadas exactamente una vez, ¿en qué secuencia deben ser recorridas para minimizar la distancia total y regresar al punto de origen? A pesar de la aparente simplicidad en la definición de este problema, su resolución puede volverse fácilmente inalcanzable en muchos casos. Conforme aumenta el valor de n, el número de posibles soluciones también crece, incrementando así la complejidad del problema. En la Figura 2.1 se ilustra el esquema del problema para $n = 3$ y $n = 15$ ciudades. Cuando el problema considera tres ciudades, la distancia es constante sin importar el orden de visita, lo que implica que cualquier ruta proporciona la solución óptima. No obstante, al aumentar el número de ciudades a 15, resulta menos evidente cómo el orden de visita afecta la calidad de la solución propuesta.

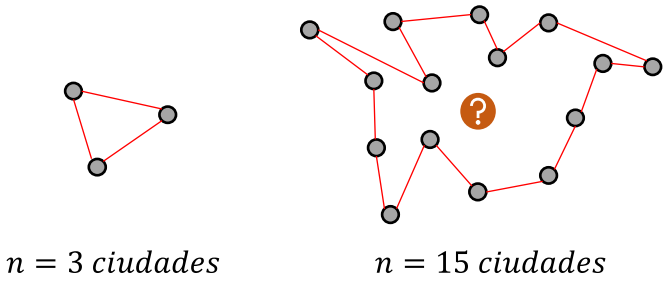


Figura 2.1 Problema del Viajero con $\boldsymbol{n = 3}$ y $\boldsymbol{n = 15}$. Tomada de Velasco y Hospitaler (2021)

Dado que no hay un método determinístico que nos conduzca de manera directa a la solución de nuestro problema, abordarlo implica que una persona debería primero enumerar y definir todas las posibles soluciones y, posteriormente, calcular la distancia recorrida por cada una. Únicamente después de examinar exhaustivamente todas las alternativas, sería posible identificar la ruta óptima al problema. El inconveniente con esta estrategia de resolución es que, al ser un método enumerativo, resulta ser excesivamente ineficiente, requiriendo incluso para una computadora desde segundos hasta años para encontrar la solución al problema, ello dependiendo del número de ciudades a visitar o, lo que es lo mismo, de la complejidad del problema.

Al igual que el Problema del Viajero, tanto en nuestras vidas personales como en la práctica profesional nos encontraremos con numerosos problemas de optimización que requieren ser resueltos en un tiempo limitado. Algunos de estos problemas pueden abordarse mediante métodos exactos, es decir, enfoques determinísticos que exploran todas las posibles soluciones y son capaces de identificar la mejor de entre ellas, por ejemplo, el método simplex (Velasco & Hospitaler, 2021). En cambio, otros problemas solo pueden resolverse mediante métodos heurísticos o metaheurísticos, los cuales generan soluciones de alta calidad en un tiempo razonable, aunque no necesariamente óptimas (Feo & Resende, 1995).

Todo problema puede ser clasificado como P o NP, basándose esta clasificación en la complejidad del problema. Cuando un problema es de tipo P significa que su resolución requiere de un tiempo

razonable, mientras que los problemas NP son aquellos cuya solución puede ser verificada en un tiempo razonable, pero no existen algoritmos polinómicos que puedan hallar dicha solución con la misma rapidez (Blum & Roli, 2003). En esta última clasificación es donde entra el Problema del Viajero. Para comprender mejor este concepto, en la Tabla 2.1 se muestran los tiempos de ejecución que requieren diferentes tipos de algoritmos en función de las n operaciones que deben realizar para resolver un problema. Para el cálculo de los tiempos se considera que cualquier operación que realiza una computadora requiere de $1x10^{-8}$ segundos en ejecutarse. Se observa que en aquellos algoritmos que presentan un tiempo de ejecución de tipo polinómico (es decir, $\log(n)$, n, n^4), la complejidad del problema, o el tiempo de cómputo requerido para resolverlo, crece de manera monótona mientras que en el resto de los algoritmos que presentan tiempos no polinómicos (es decir, 2^n y $n!$), la complejidad del problema se vuelve inmanejable para un número reducido de operaciones.

Por ejemplo, un algoritmo con tiempo de ejecución definido por la expresión n^4 requerirá de 1 segundo para realizar 100 operaciones, mientras que el tiempo de ejecución crecerá hasta $1x10^{12}$ segundos para 100,000 operaciones. A pesar de que este crecimiento del tiempo de ejecución pueda parecer considerable, lo cierto es que existen problemas que presenten crecimientos mucho más pronunciados. Considérese ahora un algoritmo que tenga un tiempo de ejecución definido por la expresión 2^n, para este se requerirá de un tiempo de ejecución de $1.02x10^{-5}$ segundos para realizar 10 operaciones. Sin embargo, si el número de operaciones crece apenas a 100, el tiempo de ejecución se dispara hasta $1.27x10^{22}$ segundos. La verdadera magnitud de este valor cobra más sentido si se considera que el universo tiene aproximadamente $4.3x10^{17}$ segundos de existencia, esto es, el resolver 100 operaciones de un problema así requeriría de un tiempo de cómputo mayor a la edad actual del universo. Por estos tiempos de cómputo excesivamente largos es que no es factible buscar la resolución exacta de los problemas de tipo NP; sin embargo, es posible encontrar soluciones de alta calidad en tiempos razonables por medio de métodos heurísticos y metaheurísticos.

Tabla 2.1 Tiempos de cómputo, en segundos

Operaciones, n	$\log(n)$	n	$n\log(n)$	n^2	n^4	2^n	$n!$
10	$1x10^{-8}$	$1x10^{-7}$	$1x10^{-7}$	0.000001	0.0001	$1.02x10^{-5}$	0.0363
100	$2x10^{-8}$	$1x10^{-6}$	$2x10^{-6}$	0.0001	1	$1.27x10^{22}$	$9.33x10^{149}$
1,000	$3x10^{-8}$	$1x10^{-5}$	$3x10^{-5}$	0.001	10,000	$1.07x10^{293}$	
10,000	$4x10^{-8}$	$1x10^{-4}$	$4x10^{-4}$	1	$1x10^{8}$		
100,000	$5x10^{-8}$	$1x10^{-3}$	$5x10^{-5}$	100	$1x10^{12}$		

En este momento, el lector podría preguntarse: ¿Qué implica resolver un problema de manera heurística o metaheurística? Para responder esta pregunta, es necesario comentar sobre el significado de ambas palabras. “Heurística” es una palabra que se deriva del griego "heuriskein", y hace referencia al conjunto

de conceptos y estrategias que buscan la resolución de problemas a través de enfoques creativos. Retomando el ejemplo del Problema del Viajero, abordarlo desde un enfoque heurístico implicaría emplear estrategias basadas en la observación y el ingenio que faciliten la identificación de rutas cortas. A través de estas estrategias, se evitaría revisar rutas de baja calidad y la resolución del problema requeriría un tiempo de cómputo mucho menor. A pesar de lo simple que esto suena, el principal problema reside en cómo enseñarle a una computadora a usar estrategias creativas e ingeniosas

Un aspecto notable en el estudio de la resolución de problemas es que se trata de un campo científico que no captó la atención de los investigadores hasta el siglo pasado, esto a pesar de que los seres humanos han enfrentado una amplia variedad de problemas a lo largo de la historia. Si bien la Investigación de Operaciones durante la Segunda Guerra Mundial permitió el desarrollo de métodos exactos (Sörensen *et al*., 2018), los algoritmos heurísticos no vieron la luz hasta mediados del siglo XX. Uno de los primeros en estudiar el uso de estrategias creativas para la resolución de problemas fue el matemático George Pólya, quien publicó un libro en 1945 titulado: *"How to solve it?: A new aspect of mathematical model*" (Polya, 1945). En este libro, Pólya presenta una serie de estrategias generales orientadas a ayudar a los matemáticos a resolver los diferentes problemas con los que se puedan llegar a enfrentar, algunas de estas estrategias se enlistan a continuación:

- **Aprender por analogía.** Emplear el conocimiento y la experiencia obtenida al resolver un problema previo para resolver problemas nuevos pero similares.
- **Resolución por inducción.** Generalizar las características de algunos ejemplos y utilizar este conocimiento para resolver el problema de interés.
- **Emplear problemas auxiliares.** Descomponer el problema de interés en uno o varios subproblemas de resolución más sencilla.
- **Análisis medio – fin.** Identificar los pasos necesarios para que un resultado X se asemeje o alcance el resultado deseado Y.

A pesar de que las estrategias enlistadas por Polya parecen estar más orientadas a resolver problemas matemáticos, lo cierto es que siguen siendo utilizadas hoy en día por los diseñadores de algoritmos metaheurísticos para resolver una gran variedad de problemas (Sörensen *et al*., 2018). Más aún, hay investigadores que consideran que estos enfoques son tan generales que han sido empleados por la humanidad a lo largo de su historia. Por ejemplo, es evidente que el conocimiento requerido para cazar con lanzas nació del concepto de que es posible dañar seres vivos al arrojarles objetos contundentes como piedras (Sörensen *et al*., 2018). Brindando un ejemplo más cercano, si se le preguntara a una persona cualquiera que observase la Figura 2.2 e identificara la ruta que tuviera las mayores posibilidades de recorrer la menor distancia, con total seguridad escogería la ruta de la izquierda; sin embargo, lo que sí resultaría más difícil es encontrar a alguien que definiera con claridad el motivo de su elección y, más aún, que sea capaz de transferir ese criterio a una computadora. En base a lo anterior, se podría decir que el cerebro de los seres humanos está programado para resolver problemas cotidianos de manera heurística, lo cual explicaría lo fácil que estas estrategias pueden pasar desapercibidas para nosotros.

Figura 2.2 Ejemplos comparativos de soluciones con alta y baja calidad

2.2. Origen y definición de las metaheurísticas

Aunque las estrategias enlistadas por Pólya pueden ser consideradas como enfoques heurísticos para abordar problemas matemáticos (Sörensen *et al*., 2018), por sí solas no pueden resolver los problemas de alta complejidad mencionados anteriormente. Para resolver problemas NP, como el Problema del Viajero, actualmente se recurre a algoritmos cuya formulación se inspira en procesos físicos, biológicos, naturales o antropológicos. Estos algoritmos son las denominadas metaheurísticas y presentan una versatilidad lo suficientemente general como para ser empleadas en la industria, las finanzas o la ingeniería (Boussaïd *et al*., 2013). Cabe destacar que, en un inicio, a las metaheurísticas se les conocía como "heurísticas modernas" (Blum & Roli, 2003). Sin embargo, hoy en día estos dos tipos de algoritmos se distinguen por su generalidad. Esto es, las heurísticas son algoritmos creados específicamente para resolver un único tipo de problema, mientras que las metaheurísticas son aplicables a un abanico de problemas más amplio sin que se requieran cambios importantes en la estructura del algoritmo. Es por tal motivo que las metaheurísticas llevan en su nombre la palabra griega de "meta", que en conjunto significaría "más allá de las heurísticas", en alusión a que van más allá de algoritmos creados para problemas específicos.

El origen de las metaheurísticas se puede remontar a los años 60, cuando se diseñó un algoritmo computacional para estudiar la selección natural y los procesos evolutivos. Este algoritmo, extremadamente sencillo en su planteamiento, simulaba la evolución por medio de un único individuo, el cual, a su vez, daba origen a otro por medio de una mutación. A pesar de su simpleza, este algoritmo contenía los principios fundamentales de lo que hoy en día es la familia de metaheurísticas conocidas como Algoritmos Evolutivos (AE). La primer metaheurística desarrollada fue *Evolution Strategies* (ES), la cual retomaba la idea de simular los procesos de selección natural y la aplicó al campo de la optimización. Para la resolución de problemas de optimización, ES consideró a las posibles soluciones al problema como "individuos", los cuales creaban una "descendencia" mejor o peor "adaptada" al problema que se buscaba resolver. Nótese que aquí se está aplicando el concepto de aprender por analogía ya que se sabe que, por medio de la selección natural, los individuos mejor adaptados a su ambiente tienen más posibilidades de transferir sus características a la siguiente generación. Si este

proceso se repite por un tiempo extremadamente largo, las características que le permiten a un individuo ser exitoso se impondrán en toda la especie.

Para comprender mejor el funcionamiento del primer algoritmo de ES en problemas de optimización, en la Figura 2.3 se presenta un esquema de su funcionamiento. Considérese que cada solución está representada por un conjunto de características, siendo, en este caso, representadas por diferentes figuras. En todo problema de optimización existen características que brindan una alta calidad a las soluciones. Con fines didácticos, estas características de alta calidad están representadas en nuestro ejemplo por medio de una estrella. Al iniciar el proceso de optimización, la solución A, correspondiente a la generación 1, es creada de manera aleatoria, teniendo esta una característica de alta calidad. Para la creación de la generación 2, es necesario crear una nueva solución a partir de A y conservar aquella de las dos que presente la mayor calidad. Para este proceso, la solución B es creada a través de la modificación aleatoria de la solución A por medio del operador mutación. Debido al proceso de mutación, la solución B no cuenta con ninguna característica de alta calidad, con lo cual la solución A es conservada dada su mayor calidad. Posteriormente, la solución A da origen a una nueva solución C. En este caso, apareció una nueva característica de alta calidad, por lo que ahora la solución A es descartada mientras que la solución C es conservada. Si este proceso se repite por múltiples generaciones, se obtendrá una solución con múltiples características de alta calidad, esto es, una solución altamente "adaptada" al problema de optimización que se busca resolver (Sörensen *et al*., 2018).

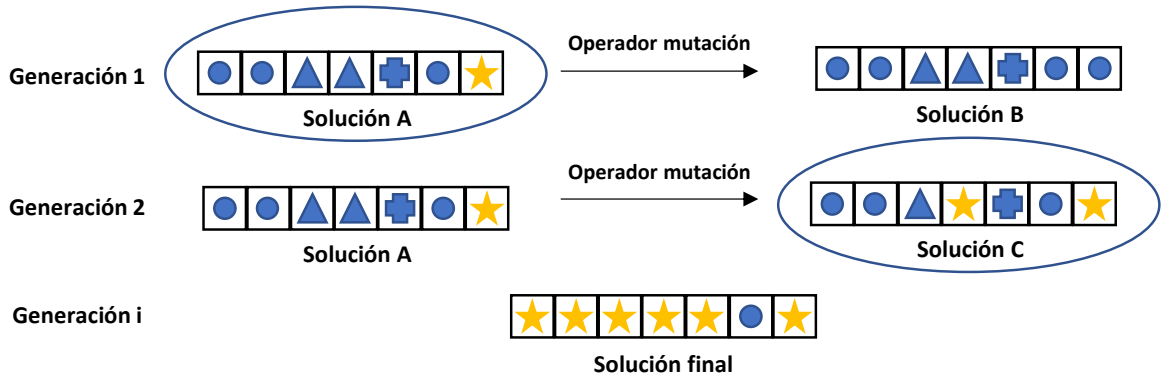


Figura 2.3 Esquema de la aplicación del primer algoritmo de Estrategias Evolutivas en la optimización de problemas**.** Tomada de (Velasco & Hospitaler, 2021)

Como el o la lectora podrán darse cuenta, la versión inicial de ES era un algoritmo de una formulación sencilla, pero que contenía la esencia de conceptos bastante poderosos. Gracias a su forma tan peculiar de explorar nuevas soluciones, resultaba posible resolver una gran variedad de problemas de optimización sin la necesidad de que estos existieran en un espacio continuo o diferenciable. Asimismo, los movimientos aleatorios con los que se generaban las nuevas soluciones resultaban ser una estrategia efectiva para salir de óptimos locales. Años más tarde, el algoritmo de ES se desarrolló

con la adición de nuevos conceptos como el de población, lo que a su vez permitió la creación del operador cruce, es decir, la transferencia de características de alta calidad entre dos o más soluciones. Los dos siguientes puntos relevantes para el desarrollo de la familia de metaheurísticas conocidas como Algoritmos Evolutivos fueron la introducción de conceptos teóricos a través del curso que John Holland impartía en la Universidad de Michigan en Ann Arbor (Holland, 1975), y la publicación del libro *'Genetic algorithms in optimization, search and machine learning'*, escrito por David Goldberg (1989), quien era alumno de Holland.

Cabe señalar que el primero en utilizar la palabra "Metaheurística" para denominar esta clase de nuevos algoritmos fue el propio David Goldberg en los años 80 (Blum & Roli, 2003). A pesar de que el concepto existe desde hace varias décadas, múltiples definiciones de la palabra metaheurística han sido brindadas por diferentes autores, algunas de las cuales se presentan a continuación:

- Las metaheurísticas dirigen y modifican otras heurísticas subordinadas para explorar soluciones más allá de óptimos locales (Glover, 1986).
- Las metaheurísticas son algoritmos independientes del problema que pueden ser empleados para crear el marco de trabajo de una nueva heurística (Sörensen, 2013).
- Las metaheurísticas son algoritmos independientes del problema, de alto nivel y que brindan un conjunto de estrategias para desarrollar algoritmos de optimización heurística (Sörensen & Glover, 2013).
- Las metaheurísticas son algoritmos diseñados para resolver, de manera aproximada, un amplio abanico de problemas de difícil optimización sin necesidad de adaptarse profundamente a cada problema (Boussaïd *et al*., 2013).
- El concepto de metaheurística hace referencia a los conceptos e ideas que permiten desarrollar algoritmos de optimización, abarcando inclusive su aplicación en problemas específicos (Sörensen *et al*., 2018).
- Las metaheurísticas son marcos de trabajo que pueden ser utilizados para diseñar heurísticas aplicables a problemas de optimización (Sevaux *et al*., 2018).

Aunque una definición precisa de lo que constituye una metaheurística puede parecer esquiva, se pueden identificar algunas características generales basadas en la literatura:

- Son algoritmos independientes del problema a resolver.
- Se basan en procesos estocásticos, por lo que siempre regresan soluciones diferentes.
- No garantizan regresar la solución óptima del problema; sin embargo, sí son capaces de regresar soluciones de alta calidad en un tiempo razonable.
- No es posible conocer de manera exacta la cercanía entre las soluciones encontradas y la solución óptima del problema.
- Requieren tiempos de ejecución relativamente cortos, por lo que son utilizadas cuando la complejidad del problema no permite encontrar su solución óptima.
- Su formulación puede tomar inspiración de procesos físicos, biológicos o antropológicos.
- Es posible combinar diferentes metaheurísticas entre sí para crear una nueva.

2.3. Clasificación de los algoritmos metaheurísticos

En la actualidad, además de los Algoritmos Evolutivos, se han desarrollado otras metaheurísticas como *Simulated Annealing* (SA), *Variable Neighborhood Search* (VNS), *Threshold Accepting* (TA), *Tabu Search* (TS), *Greedy Randomized Adaptative Search Procedure* (GRASP), *Ant Colony Optimization* (ACO), entre muchas otras. Debido al gran número de metaheurísticas existentes, se han propuesto diversas maneras de clasificar estos algoritmos, algunas de las cuales se mencionan a continuación (Birattari *et al*., 2003):

- En función a cómo se desplazan por el espacio de configuraciones (siguiendo una trayectoria o moviéndose entre soluciones por medio de "saltos").
- Si se basan en poblaciones (trabajan con varias soluciones simultáneamente) o en una única solución (se genera una solución por iteración).
- Si son capaces de almacenar o no la experiencia adquirida durante la búsqueda (algoritmos con y sin "memoria").
- Si emplean una o más vecindades (es decir, cuantas "reglas" u operadores ocupan para moverse por el espacio de configuraciones).
- Por las características de sus funciones objetivo (pueden ser dinámicas o estáticas).
- Por su fuente de inspiración (basadas en la naturaleza o en otros procesos).

De entre las diferentes formas que existen para clasificar a las metaheurísticas, la más ampliamente utilizada es aquella asociada al número de soluciones con las cuales trabaja el algoritmo en cada iteración. Cuando una metaheurística emplea dos o más soluciones, se dice que esta es un algoritmo basado en poblaciones, mientras que si solo trabaja con una solución por iteración se clasifica como un algoritmo basado en una solución. La utilidad de esta clasificación no se limita a un simple ejercicio de agrupación sino que además permite tomar en cuenta los dos enfoques principales que definen el desempeño de las metaheurísticas: **la diversificación** y **la intensificación**.

La diversificación hace referencia a la capacidad que presentan las metaheurísticas para explorar regiones distantes del espacio de soluciones. Debido a que trabajan con múltiples soluciones por iteración, las metaheurísticas basadas en poblaciones presentan una estrategia de exploración más orientada hacia la diversificación. Por otra parte, la intensificación se define como la capacidad del algoritmo para extraer las mejores soluciones posibles de la región donde se encuentra. En este caso, los algoritmos basados en una solución presentan estrategias más orientadas a la intensificación (Boussaïd *et al*., 2013), esto debido a que su búsqueda se concentra en zonas reducidas del espacio de soluciones. Cabe señalar que un adecuado desempeño de las metaheurística depende del balance entre estos enfoques de exploración (Duarte *et al*., 2018). Para ampliar más en este tema, a continuación se ofrece una breve descripción de las principales metaheurísticas basadas en una solución y en poblaciones.

2.4. Metaheurísticas basadas en poblaciones

Dado el gran número de algoritmos basados en poblaciones que actualmente existe, en las siguientes secciones solo se discutirán los pertenecientes a la familia de Algoritmos Evolutivos, además de la metaheurística conocida como Optimización por Colonia de Hormigas, las cuales fueron de las primeras de su tipo.

2.4.1. Algoritmos evolutivos

Algoritmos Evolutivos es el término que se utiliza para agrupar las metaheurísticas que toman inspiración de la selección natural. Dentro de este grupo se pueden encontrar algoritmos como Estrategias Evolutivas, Algoritmos Genéticos, Programación Genética y Programación Evolutiva. Sus características generales son las siguientes (Blum & Roli, 2003; Boussaïd *et al*., 2013; C Coello, 2018; D. Goldberg, 1989):

- Una solución al problema de optimización se denomina como "individuo".
- Al conjunto de individuos se le conoce como "población".
- A una iteración del algoritmo se le conoce como "generación".
- Para explorar el espacio de soluciones se utilizan tres operadores básicos: selección, cruce y mutación.
- El desempeño de cada individuo es evaluado de manera cuantitativa y este define su probabilidad de transferir sus características a la siguiente generación.
- Estos algoritmos convergen a soluciones únicas si se realiza un número elevado de iteraciones.

Por su parte, las características particulares de cada algoritmo evolutivo se presentan a continuación:

Estrategias Evolutivas (EE): La primer metaheurística desarrollada, cuyo origen se remonta a los años 60. En sus primeras versiones, el algoritmo dependía únicamente del operador mutación para explorar el espacio de soluciones. Posteriormente, se introdujo el concepto de población por medio de la variante llamada EE – $(\mu + 1)$, donde $\mu > 1$ era el número de progenitores disponibles para dar origen a la siguiente generación. En esta versión se mantiene el tamaño de la población fijo. Cuando se crea una nueva solución, si esta es mejor que la peor solución de la población actual, la reemplaza. El uso de la población permite utilizar otros operadores como el cruce (Boussaïd *et al*., 2013). Este algoritmo es muy usado en problemas continuos (Blum & Roli, 2003).

Algoritmos Genéticos (AG): Esta es la metaheurística más utilizada en la actualidad (Velasco & Hospitaler, 2021). AG fue presentada originalmente en 1962 (Holland, 1962) y se diferencia de EE por las características de los operadores que emplea y por el uso de una codificación binaria. Por ejemplo, para seleccionar aquellas soluciones que transferirán sus características a la siguiente generación, AG puede hacer uso de la selección por ruleta, por torneo o por ranking. Para más información sobre estos algoritmos de selección, se recomienda leer el capítulo 6 de este libro.

Asimismo, los operadores cruce y mutación empleados por AG presentan enfoques distintos a aquellos utilizados en EE. AG es un tipo de algoritmo muy utilizado en problemas de optimización combinatoria (Blum & Roli, 2003).

Programación Evolutiva (PE): PE es un algoritmo creado en la década de 1960 que experimentó una disminución en su uso debido a su notoria similitud con EE. Esta metaheurística se basa en el concepto de población, aunque se limita exclusivamente al uso de mutaciones para explorar el espacio de soluciones (Boussaïd *et al*., 2013), por lo que se puede afirmar que una de sus principales características es la codificación (máquinas de estado) de la evolución a nivel de especie.

Programación Genética (PG): PG es una metaheurística propuesta por John Koza en el año de 1989 como una técnica para la evolución automática de programas informáticos (Koza, 1989). A diferencia del resto de Algoritmos Evolutivos, PG representa a las soluciones potenciales como árboles que codifican programas informáticos y pueden ser ejecutadas directamente (Koza, 1994). Esta metaheurística es ampliamente utilizada en Aprendizaje de Máquina (*Machine Learning*) y Minería de Datos (*Data Mining*) (Boussaïd *et al*., 2013).

2.4.2. Inteligencia de colmena

Este tipo de algoritmos busca imitar el comportamiento colectivo de poblaciones de insectos y otros animales sociales (Boussaïd *et al*., 2013). La premisa fundamental de este enfoque radica en el conocimiento de que modelos probabilísticos simples son capaces de explicar patrones colectivos complejos (Colorni *et al*., 1991). Un principio análogo al utilizado por estos algoritmos se puede encontrar en la formulación de las redes neuronales artificiales (Hopfield, 1982). Aunque se han propuesto diversas metaheurísticas basadas en esta idea, solo se describirá la conocida como Optimización por Colonia de Hormigas, ya que fue el primero de su clase y sigue siendo una de las más ampliamente utilizadas.

Optimización por Colonia de Hormigas (OCH): OCH es un algoritmo de optimización inspirado en el comportamiento de las colonias de hormigas en la naturaleza. Este algoritmo fue propuesto por Marco Dorigo en la década de 1990 y se basa en los principios de cooperación y comunicación observados en las hormigas reales (Colorni *et al*., 1991). La idea central de OCH es simular el comportamiento de búsqueda de alimentos de las hormigas para encontrar soluciones de alta calidad en problemas de optimización combinatoria y paramétrica. Esta metaheurística ha demostrado ser efectiva en una variedad de problemas, incluyendo el Problema del Viajero.

2.5. Metaheurísticas basadas en una solución

Las metaheurísticas basadas en una solución son algoritmos diseñados para la intensificación, lo que significa que se centran en identificar las mejores soluciones dentro de una región específica del

espacio de soluciones. Su forma de trabajar es bastante sencilla ya que se inician con una única solución y luego se alejan progresivamente de ella, trazando una trayectoria en el espacio de soluciones (Boussaïd *et al*., 2013). Dentro de esta clasificación encuentran algoritmos como Recocido Simulado (en inglés *Simulated Annealing*), Búsqueda Tabú (en inglés *Tabu Search*), GRASP, Búsqueda en Vecindario Variable (en inglés *Variable Neighborhood Search*), entre otros. Estos algoritmos son descritos brevemente a continuación.

Recocido Simulado (RS): RS es un algoritmo de optimización metaheurística que se inspira en el proceso físico de recocido de un material. Fue propuesto por Kirkpatrick, Gelatt y Vecchi en 1983 y toma inspiración de un proceso metalúrgico en el cual un material se calienta y luego se enfría lentamente para reducir su dureza y mejorar su estructura cristalina (Kirkpatrick *et al*., 1983). La idea fundamental del Recocido Simulado es explorar el espacio de soluciones en busca de la solución óptima o cercana a la óptima. El algoritmo maneja la exploración mediante la aceptación probabilística de soluciones peores en las primeras etapas de la búsqueda y la disminución gradual de esta probabilidad a medida que avanza el proceso de optimización (Fleischer, 1995). Cabe señalar que existe otro algoritmo conocido como *Threshold Accepting* (TA), el cual emplea una idea similar, pero utiliza un criterio determinístico para tomar las soluciones degradadas, en lugar de uno probabilístico como lo hace el Recocido Simulado (Winker & Maringer, 2007). Hoy en día, RS es más utilizado como un componente de otras metaheurísticas en vez de un algoritmo de búsqueda independiente (Blum & Roli, 2003).

Búsqueda Tabú (BT): BT es otra técnica de optimización utilizada para encontrar soluciones de alta calidad en problemas de optimización. Fue propuesta por Fred Glover en la década de 1980 (Glover, 1989a). Esta metaheurística se basa en la idea de mantener una memoria de las soluciones visitadas recientemente y evitar revisitar esas soluciones durante la búsqueda. Este mecanismo se implementa mediante una "lista tabú" que registra ciertas características o movimientos que no deben repetirse en las soluciones candidatas. En este algoritmo, la longitud de la lista tabú tiene una influencia sobre su desempeño (Boussaïd *et al*., 2013). Las listas pequeñas causan que la búsqueda se concentre en áreas concretas del espacio de búsqueda, mientras que las listas largas permiten al algoritmo visitar amplias regiones de búsqueda que no han sido exploradas.

GRASP: Del inglés *Greedy Randomized Adaptive Search Procedure*, GRASP es una metaheurística propuesta por Feo y Resende en 1989. La idea central de GRASP es combinar elementos de enfoques voraces (*greedy*) y aleatorios para explorar de manera eficiente el espacio de soluciones (Feo & Resende, 1995). A diferencia de un enfoque completamente voraz, que siempre elige la mejor opción en cada paso, y un enfoque completamente aleatorio, que elige soluciones de manera completamente aleatoria, GRASP busca un equilibrio. Este algoritmo emplea dos fases de búsqueda: la creación de soluciones seguida por un proceso de búsqueda local. El algoritmo inicia construyendo soluciones de tal manera que se elijan elementos de acuerdo con algún criterio de preferencia. Sin embargo, en lugar de seleccionar siempre la mejor opción, se introduce aleatoriedad para permitir cierta exploración del espacio de soluciones. Posteriormente, se aplica una búsqueda

local a la solución construida para mejorarla. La búsqueda local se centra en realizar cambios pequeños y locales en la solución para buscar vecindarios más prometedores.

Búsqueda de Vecindario Variable (BVV): BVV es una metaheurística propuesta por Mladenović y Hansen en 1997. La idea fundamental de BVV es explorar el espacio de soluciones por medio de diferentes vecindarios para buscar soluciones de alta calidad (Pierre Hansen *et al*., 1997). Esta metaheurística se considera como el algoritmo de búsqueda local más sencillo. El concepto fundamental de este algoritmo es el de "vecindario" o "entorno", el cual define los diferentes candidatos a solución que el algoritmo puede alcanzar en su siguiente iteración. Por medio de la variación de los vecindarios empleados durante el proceso de búsqueda, BVV puede escapar de óptimos locales y alcanzar nuevas regiones del espacio de soluciones (Velasco & Hospitaler, 2021).

2.6. Sobre el uso de las metáforas

Como se ha podido ver, los conceptos detrás de las metaheurísticas han evolucionado enormemente desde los primeros algoritmos evolutivos propuestos. Gran parte de este avance tiene su origen en factores como el desarrollo de la teoría de la programación, la gran potencia y desempeño que muestran las computadoras actuales y la facilidad con la que las ideas son compartidas e implementadas entre los diseñadores de estos algoritmos (Boussaïd *et al*., 2013; Pierre Hansen *et al*., 1997). A pesar de su aparente desarrollo, aún persiste uno de los grandes problemas que el campo de las metaheurísticas ha enfrentado desde sus inicios: la falta de fundamentos teóricos. En un principio, se consideraba suficiente el uso de analogías para el desarrollo de nuevos algoritmos, al final de cuentas, se imitaban procesos relacionados con la optimización, como el incrementar el nivel de adaptación de una especie, el buscar estados ordenados de mínima energía o el identificar la ruta más corta entre un nido y la fuente de alimento. Sin embargo, no todos los investigadores se sintieron satisfechos con una explicación de las metaheurísticas que se basaba principalmente en el uso de metáforas, lo que llevó a que en un principio las metaheurísticas fueran vistas como un campo carente de seriedad científica (Sörensen, 2013), tal como lo señala Glover (1977): "*Los algoritmos son concebidos en la pureza de las altas esferas de la investigación académica, las heurísticas nacen de la conveniencia en los rincones oscuros de las guaridas de quienes se dedican a las aplicaciones, teniendo, por lo tanto, estándares más bajos*".

A pesar de que la ausencia de fundamentos teóricos impide que el campo de optimización metaheurística sea visto como un área científica madura, no hay que perder de vista que estos algoritmos suelen ser la única opción viable para la resolución de los complejos problemas de optimización de diferentes disciplinas (Sörensen *et al*., 2018). Por esta razón, la mayor parte de los trabajos publicados sobre las metaheurísticas tratan de su aplicación en diversos ejemplos. De manera adicional a estos artículos, otro tipo de publicación relativamente común es aquella en donde se presentan nuevas metaheurísticas a las que sus creadores suelen denominar como "innovadoras". Sin embargo, existe una polémica alrededor de estos algoritmos debido a que se pone en tela de juicio la validez de su aportación al campo de investigación, habiéndose incluso presentado casos donde las

nuevas metaheurísticas no eran más que plagios de otras presentadas anteriormente. Para dar un ejemplo de la cantidad de trabajos publicados donde se presentan metaheurísticas innovadoras, en la revisión bibliográfica realizada por Dokeroglu *et al*. (2019) se reporta la existencia de casi 100 algoritmos metaheurísticos diferentes para el año 2019. Adicionalmente, Velasco *et al*. (2023) reportaron que, solo en el 2022, 111 estudios fueron publicados donde se presentaban metaheurísticas denominadas por sus propios creadores como "nuevas", "innovadoras" o "mejoradas".

Debido a que muchos de estos algoritmos parecen ser "aportaciones" innecesarias al campo de investigación, parte de la comunidad científica se siente inconforme con esta tendencia al alza del número de algoritmos existentes. Este problema ha alcanzado una gravedad tal que inclusive algunas revistas científicas han elaborado lineamientos editoriales enfocados a reducir el número de artículos publicados donde se presenten nuevas metaheurísticas. Las revistas Journal of Heuristics (2015) y ACM Transactions on Evolutionary Learning and Optimization (2022) han dejado de aceptar artículos donde se presenten algoritmos con ideas ya conocidas o que no expliquen de manera adecuada la razón por la que el algoritmo propuesto mostraría un desempeño superior. Adicionalmente, Swarm Intelligence (Dorigo, 2016) ha solicitado a su cuerpo editorial que reduzca el número de artículos publicados que presenten nuevas metaheurísticas, indicándoles aceptar para revisión únicamente aquellos que presenten los más altos estándares de calidad.

A pesar de estos intentos por evitar la aparición de más algoritmos de relevancia dudosa, aún son presentados decenas de nuevas metaheurísticas cada año. Evidentemente, esta enorme cantidad de algoritmos no solo puede abrumar a los académicos que trabajan en este campo de investigación sino también a aquellas personas que se están iniciando en este tema. Para brindar un poco de luz en este aspecto tan particular de las metaheurísticas, en las siguientes secciones se abordará la manera en que las metáforas pueden esconder algoritmos poco innovadores y si es posible desarrollar una metaheurística que siempre se desempeñe mejor al resto.

2.7. La controversia sobre las metaheurísticas "innovadoras"

Ya se ha señalado que muchos procesos naturales o artificiales han sido tomados como fuente de inspiración en la creación de nuevas metaheurísticas, esto fue así en los años 60 y lo sigue siendo en la actualidad. En un principio, la idea de usar analogías como la selección natural o el comportamiento colectivo de insectos no parecía tan descabellada. Al final de cuentas, la naturaleza ha sido muy ingeniosa en la creación de estrategias que maximizan la probabilidad de supervivencia de una especie y ha tenido mucho tiempo para perfeccionar todos estos procesos. Sin embargo, algunos de los nuevos algoritmos novedosos mencionados antes parecen tomar inspiración en procesos poco relacionados con la optimización, haciendo ver a la analogía como un elemento decorativo que apuntala su propio método. Ejemplos de la sobre explotación de analogías hay muchos, por mencionar algunos, se han publicado algoritmos inspirados en la improvisación de jazz (Geem *et al*., 2001), gateríos (Chu *et al*., 2006), ranas (Eusuff *et al*., 2007), el funcionamiento de imperios (Atashpaz-Gargari & Lucas, 2007), enjambres de

abejas (Karaboga & Basturk, 2007), gotas de agua inteligentes (Shah, 2008), pájaros cucús (Yang & Deb, 2009), la gravedad (Rashedi *et al*., 2009), abejorros (Yannis *et al*., 2010), murciélagos (Yang, 2010), cuerpos en colisión (Kaveh & Mahdavi, 2014), duelos (T. Biyanto & at al., 2016), espirales (Tamura & Yasuda, 2016), ballenas jorobadas (Mirjalili & Lewis, 2016), ballenas asesinas (T. R. Biyanto *et al*., 2017), chapulines (Saremi *et al*., 2017), el ciclo hidrológico (Wedyan *et al*., 2017), pingüinos emperador (Harifi *et al*., 2019), halcones (Heidari *et al*., 2019), el pastoreo de rebaños (Kaveh & Zaerreza, 2020), efímeras (Zervoudakis & Tsafarakis, 2020), equipos de investigación forense (Chou & Nguyen, 2020), viudas negras (Hayyolalam & Pourhaji Kazem, 2020), águilas doradas (Mohammadi-Balani *et al*., 2021), grupos de nómadas (N. Lin *et al*., 2022), serpientes (Hashim & Hussien, 2022), entre muchos otros.

Otro aspecto importante por destacar es que la comunidad científica no parece ser reacia a utilizar los nuevos algoritmos propuestos, esto sin importar lo dudosa de su procedencia. De manera similar a Dokeroglu *et al*. (2019), durante la escritura de este libro se realizó una búsqueda del número de publicaciones relacionadas con algunas de las metaheurísticas que se han mencionado. En la Figura 2.4 se muestran el número de artículos que mencionan las metaheurísticas presentadas en las secciones 2.5 y 2.6. La información empleada para elaborar la gráfica fue obtenida de Google Scholar el 10 de enero del 2023. En la nomenclatura empleada se tomó del nombre de los algoritmos en inglés y es la siguiente: *Genetic Algorithm* (GA), *Simulated Annealing* (SA), *Genetic Programming* (GP), *Ant Colony Optimization* (ACO), *Tabu Search* (TS), *Evolution Strategies* (ES), *Greedy Randomized Adaptive Search Procedures* (GRASP) y *Variable Neighborhood Search* (VNS).

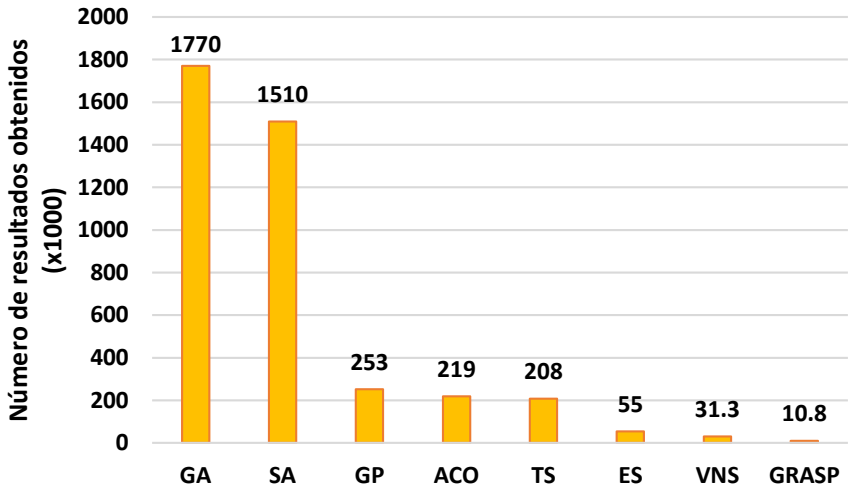


Figura 2.4 Número de resultados obtenidos por Google Scholar sobre metaheurísticas presentadas en las secciones 2.5 y 2.6. Elaboración propia con datos obtenidos en enero de 2023

Para contrastar, en la Figura 2.5 se muestran el número de artículos que mencionan algunas de las metaheurísticas "novedosas" presentadas en esta sección. La nomenclatura empleada es la siguiente: *Artificial Bee Colony* (ABC), *Harmony Search* (HS), *Cuckoo Search* (CS), *Bat Algorithm* (BA),

Gravitational Search Algorithm (GSA), *Whale Optimization Algorithm* (WOA), *Shuffled Frog-Leaping Algorithm* (SFLA), *Grasshopper Optimization Algorithm* (GOA), *Cat Swarm Optimization* (CSO), *Intelligent Water Drops* (IWD) y *Colliding Bodies Optimization* (CBO).

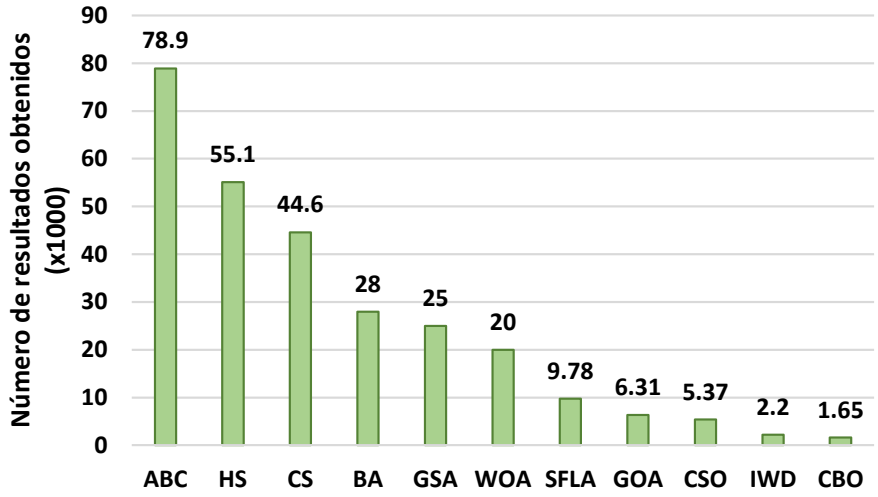


Figura 2.5 Número de resultados obtenidos por Google Scholar sobre metaheurísticas "novedosas". Elaboración propia con datos obtenidos en enero de 2023

De las Figuras 2.4 y 2.5 se observa que actualmente la metaheurística más utilizada es GA, cuyo nombre aparece en más de 1.7 millones de artículos. Con 1.5 millones de artículos relacionados, SA es la segunda metaheurística que más interés ha generado y cuya popularidad se puede explicar por su uso en combinación con otros algoritmos. Por otra parte, de las metaheurísticas "novedosas" revisadas, las más utilizadas son ABC, HS y CS, las cuales son mencionadas en decenas de miles de artículos. A pesar de que su popularidad es relativamente menor al de las primeras metaheurísticas, su número de menciones demuestra muchos consideran que estos algoritmos presentan una alta aplicabilidad en la investigación científica.

Un tema de vital importancia que surge al existir tal variedad y número de algoritmos es el identificar y contrastar los aportes específicos de cada uno. Actualmente, el realizar esto se ha vuelto extremadamente difícil debido a las decenas de "nuevos" algoritmos que se proponen cada año y, más aún, esta tarea se ve usualmente obstruida por el lenguaje empleado por los diseñadores de algoritmos.

El uso de metáforas para justificar la elaboración de metaheurísticas ha provocado usos abusivos del lenguaje propio del campo del que se toma la inspiración (Aranha *et al*., 2021; Del Ser *et al*., 2019; Sörensen, 2013; Sörensen *et al*., 2018; Tzanetos & Dounias, 2021). Sobre este punto anterior, Sörensen brinda un excelente ejemplo al discutir sobre el algoritmo denominado como "*Imperialist Competitive Algorithm"* (Sörensen, 2013). Con fines didácticos, el diagrama de flujo de la metaheurística mencionada se presenta en la Figura 2.6. Como se puede apreciar, la metaheurística

está relacionada con conceptos como imperios y colonias, siendo estos presentados de una manera ambigua y poco científica. Debido a las palabras empleadas para describir al algoritmo, el lector no puede conocer con seguridad qué concepto es una solución, qué proceso se utiliza para moverse por el espacio de soluciones, o inclusive por qué el concepto mismo de imperialismo es aplicable a la optimización.

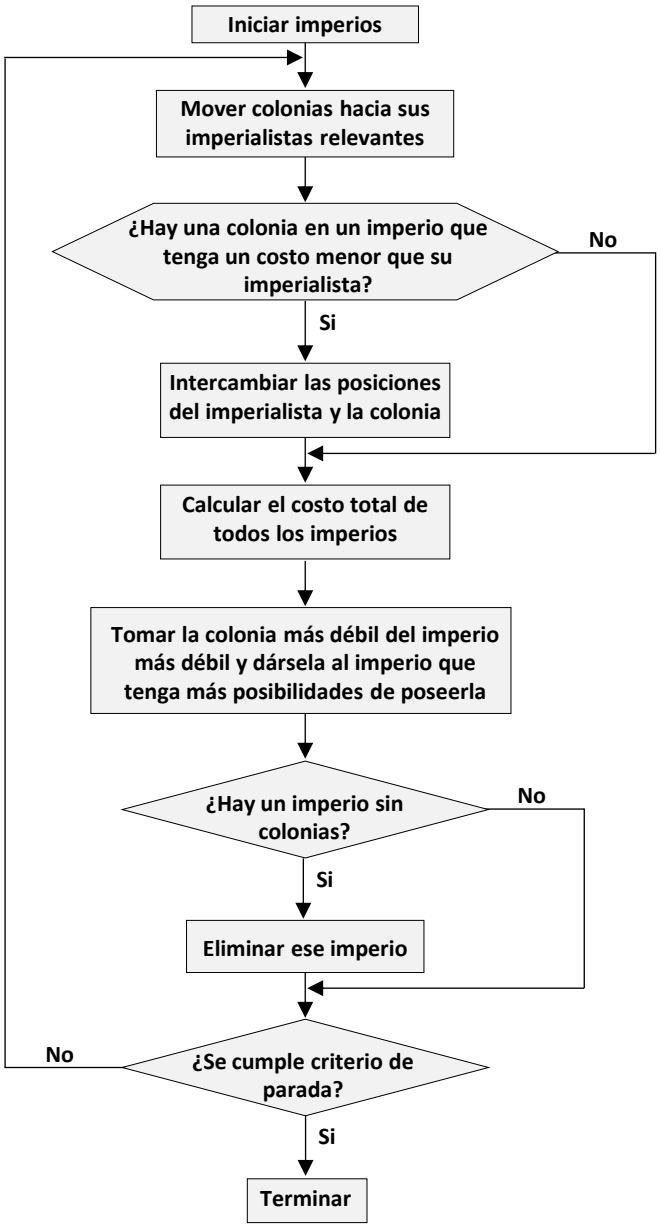


Figura 2.6 Diagrama del *Imperialist Competitive Algorithm* propuesto por Atashpaz-Gargari y Lucas (Atashpaz-Gargari & Lucas, 2007).

Además de hacer parecer al uso de las metaheurísticas más con un juego que como un trabajo científico, se han presentado casos en los que el abuso del lenguaje relacionado con una analogía termina por esconder similitudes con trabajos presentados anteriormente. Dos casos llamativos son los de *Intelligent Water Drops* (IWD), creado por Shah (2008), y *Harmony Search* (HS), presentado en Geem *et al*. (2001). Ambos algoritmos han sido acusados de ser idénticos a otros más conocidos. En el caso de IWD, Camacho *et al*. (2019) indican que el método es un caso especial de Optimización por Colonia de Hormigas (OCH), mientras que, para el caso de HS, Weyland (2010) señala que es un caso especial de *Evolution Strategies* (ES).

El caso de HS es especialmente interesante debido a que parece ser que la acusación de plagio no tuvo gran impacto en la comunidad científica, llegándose incluso al caso de que existen más trabajos publicados sobre HS que sobre ES (como se puede deducir de las Figuras 2.4 y 2.5). En un trabajo posterior Weyland (2015) demostró de manera rigurosa que HS es un caso especial de ES y reporta que, al buscar "*harmony search*" (con comillas) en Google Scholar, se obtienen 9,000 resultados. Como se observa en la Figura 2.5, a la fecha de realizar este trabajo, la misma búsqueda brinda 55,100 resultados, lo cual hace evidente la creciente popularidad del algoritmo. El incremento de artículos relacionados con HS, aún a pesar de las declaraciones de Weyland (2010, 2015), se puede explicar tanto por los buenos resultados que se reportan como por la difusión que su creador le da por medio de una página web especialmente elaborada para el algoritmo. En dicho sitio web se brinda toda una variedad de trabajos relacionados con HS, desde artículos publicados, hasta libros y presentaciones.

A pesar de lo extravagante que esto pueda parecer, hay varios diseñadores de algoritmos que utilizan la misma estrategia de difusión. Algunas capturas de pantalla que ejemplifican esto se muestran en las Figuras 2.7 – 2.9. Como los investigadores más serios podrán observar, esta es una forma inapropiada de realizar la labor de divulgación científica, ya que cada creador busca la difusión de su algoritmo más por su popularidad que por las auténticas ventajas que este pudiera proporcionar.

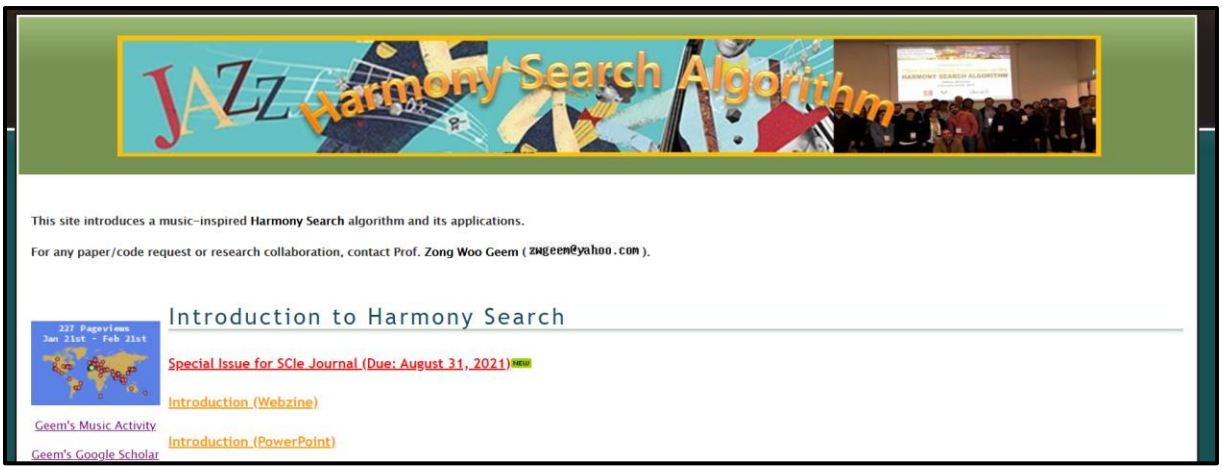


Figura 2.7 Sitio web de *Harmony Search* (HS), algoritmo propuesto por Geem *et al*. (2001). Captura tomada de https://sites.google.com/a/hydroteq.com/www/.

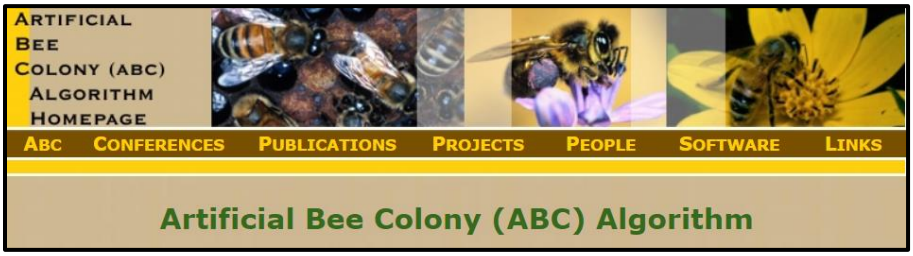


Figura 2.8 Sitio web de *Artificial Bee Colony* (ABC), algoritmo propuesto por Karaboga *et al*. (2007). Captura tomada de https://abc.erciyes.edu.tr.

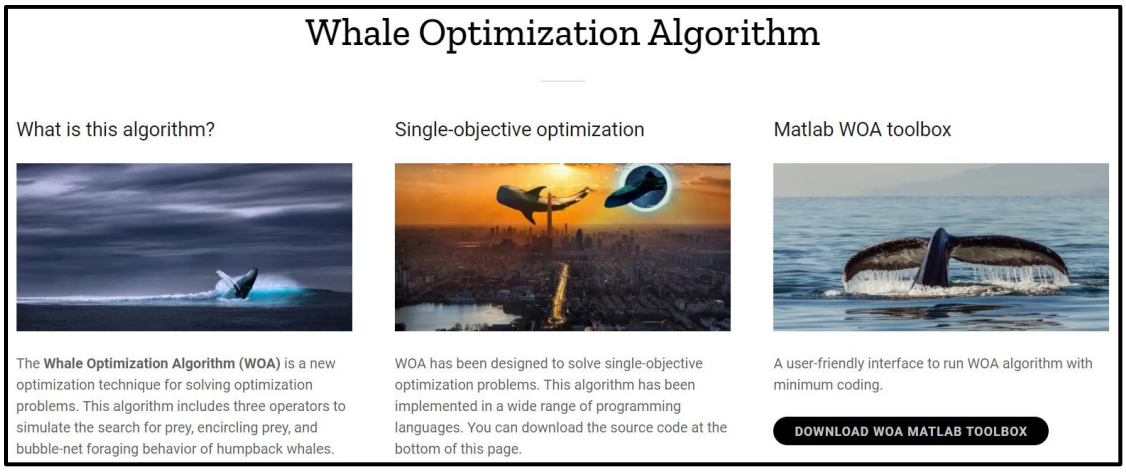


Figura 2.9 Sitio web de *Whale Optimization* Algorithm (WOA), algoritmo propuesto por Mirjalili *et al*. (2016). Captura tomada de https://seyedalimirjalili.com/woa.

Resulta evidente que una parte importante del pronunciado incremento del número de algoritmos metaheurísticos publicados se origina por la falta de un lenguaje común que facilite la comparación de los nuevos métodos con otros más antiguos. Para subsanar esta deficiencia, Sörensen (2013) propone las siguientes prácticas:

- Indicar claramente los componentes empleados por el algoritmo.
- Señalar en qué otros algoritmos aparecen los componentes empleados en la nueva metaheurística.
- Indicar cómo los componentes empleados brindan un buen desempeño al algoritmo.

Para finalizar la discusión sobre la polémica sobre el uso de metáforas, se mencionan algunos puntos de vista que presentan las y los académicos sobre este tema. Del Ser *et al*. (2019) señalan de manera acertada que el valor científico de las metaheurísticas no recae ni en la variedad ni en la cantidad de algoritmos existentes. Sin embargo, también mencionan que, como en el pasado, aún es posible obtener elementos novedosos por medio de analogías, con lo cual, consideran no debería de excluirse su uso en el desarrollo de nuevos algoritmos. De manera contraria, Sörensen *et al*. (2013, 2018), Tzanetos *et al*. (2021) y Aranha *et al*. (2021)

afirman que la gran mayoría de los "nuevos" algoritmos no presentan innovación alguna y que afectan el rigor científico del campo. Debido a que los autores de este libro conocen varios algoritmos con planteamientos y aportaciones dudosas, estos no pueden más que sentir sintonía por las opiniones expresadas por Sörensen *et al*. (2013, 2018), Tzanetos *et al*. (2021) y Aranha *et al*. (2021).

2.8. Teoremas de No Free Lunch (NFL)

A pesar de lo que muchos diseñadores de algoritmos pudieran afirmar, lo cierto es que se ha demostrado matemáticamente que ninguna metaheurística es capaz de siempre mostrar un desempeño mejor al resto de algoritmos. Este importante resultado del campo de la optimización está plasmado en los teoremas de "*No Free Lunch"* (NFL), desarrollados por Wolpert y Macready (1997). Debido a su gran relevancia, a continuación se brinda un resumen de la derivación de NFL.

Considérese un espacio de búsqueda finito X asociado a un espacio finito Y mediante una función objetivo $f: X \to Y$ donde $Y \subset \mathbb{R}$. Sea además F el espacio de todos los posibles problemas de optimización. Los problemas de optimización (también denominados "funciones objetivo") se representan utilizando la teoría de la probabilidad y una distribución uniforme $P(f)$, definida sobre F, que da la probabilidad de que cada $f \in F$ sea el problema de optimización considerado, esto es:

$$P(f) = P(f(x_1), f(x_2), \dots, f(x_{|X|})) \tag{2.1}$$

Wolpert y Macready indican que emplear una distribución de probabilidad también permite realizar análisis que consideren una única función objetivo cuyas incertidumbres se codifican en $P(f)$. Asimismo, se considera que los algoritmos de búsqueda comprueban un total de m distintas soluciones $x \in X$ con sus respectivas evaluaciones de la función objetivo $y = f(x) \in Y$. Esta muestra de soluciones x y sus respectivas evaluaciones y puede definirse como un conjunto de configuraciones distintas y ordenadas cronológicamente $\bar{d}_m$, es decir:

$$\bar{d}_m \equiv \left\{ \left(d_m^x(1), d_m^y(1) \right), \dots, \left(d_m^x(m), d_m^y(m) \right) \right\} \tag{2.2}$$

Asimismo, la probabilidad que tiene cualquier algoritmo de búsqueda α de encontrar una muestra particular $\bar{d}_m$ (denotada como $P(d_m^y | f, m, \alpha)$), depende de la función objetivo f, de la iteración m y del propio algoritmo empleado. Basándose en lo anterior, y denominando cualquier par de algoritmos como a y b, Wolpert y Macready demostraron que:

$$\sum_f P(d_m^y \mid f, m, a) = \sum_f P(d_m^y \mid f, m, b) \tag{2.3}$$

El resultado anterior se conoce como el primer teorema de NFL. Por su parte, el segundo teorema de NFL es muy similar al primero pero se aplica para funciones objetivo que son dependientes del tiempo, esto es, que sean dinámicas. Los resultados antes mencionados tienen dos implicaciones muy importantes para los algoritmos de búsqueda, las cuales son:

- Los teoremas de NFL implican que el desempeño medio de dos algoritmos de búsqueda cualesquiera, aplicados a la resolución de todos los posibles problemas de optimización, será idéntico. Este resultado se mantiene aún si uno de los algoritmos es una búsqueda aleatoria y el otro es una metaheurística.
- Para que dos algoritmos cualesquiera presenten un desempeño medio igual es necesario que desempeños altos en un grupo especifico de problemas sean pagados con desempeños bajos en el resto de problemas posibles.

En la Figura 2.10 se presenta un esquema que ayuda a ejemplificar las ideas principales de NFL, esto por medio del desempeño comparado de una búsqueda aleatoria y dos algoritmos cualesquiera, denominados A y B. Al ser la búsqueda aleatoria un simple proceso de adivinanza, su desempeño se mantiene inalterado para todos los posibles problemas de optimización. Por otra parte, el algoritmo A presenta un desempeño superior para un grupo reducido de problemas, el cual se ve compensado por un desempeño pobre en el resto de posibles problemas. En un punto intermedio se encuentra al algoritmo B, el cual presenta un desempeño moderado para un rango más amplio de problemas de optimización. De acuerdo con NFL, el área debajo de las tres curvas presentadas es idéntica debido a que los tres algoritmos presentan el mismo desempeño medio y a que desempeños altos para un grupo particular de problemas de optimización deben ser pagados con desempeños bajos en el resto de los problemas posibles.

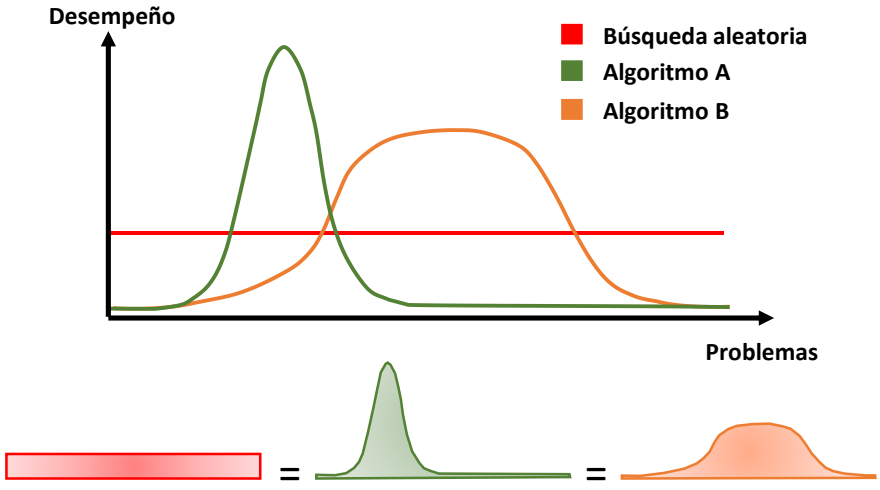


Figura 2.10 Comparación del desempeño de una búsqueda aleatoria y dos algoritmos de búsqueda cualesquiera A y B. Tomada de (Velasco & Hospitaler, 2021)

Finalmente, es importante señalar que estos teoremas tienen importantes implicaciones conceptuales, como que es esperable que los algoritmos altamente especializados en un problema en específico se desempeñen mal en el resto de los problemas. Sin embargo, su aplicabilidad práctica es algo limitada debido principalmente a que NFL no brinda una manera de conocer a priori los conjuntos de problemas para los cuales un algoritmo mostrará un buen o mal desempeño.

2.9. Problemas de optimización

Habiendo explicado de manera cualitativa los aspectos más relevantes de las metaheurísticas, en esta sección se procederá a definir de manera formal los problemas de optimización. Esto con la intención de finalizar el presente capítulo haciendo una breve revisión bibliográfica de las formas y ejemplos en los que estos algoritmos se han aplicado en el área de la Ingeniería Estructural. Pero antes, es necesario hablar conceptualmente de los componentes básicos que conforman un problema de optimización, los cuales se presentan en el cuadro de la Figura 2.11. De manera general, los problemas de optimización se componen de: 1) el objeto a optimizar y el algoritmo de búsqueda empleado en la resolución; 2) las variables de decisión y parámetros del problema; 3) las restricciones, que son criterios que establecen la factibilidad o no factibilidad de las soluciones propuestas y 4) la función objetivo, la cual brinda una medida cuantitativa de la calidad de las soluciones generadas por el algoritmo y que es usualmente denominada como $f(x)$.

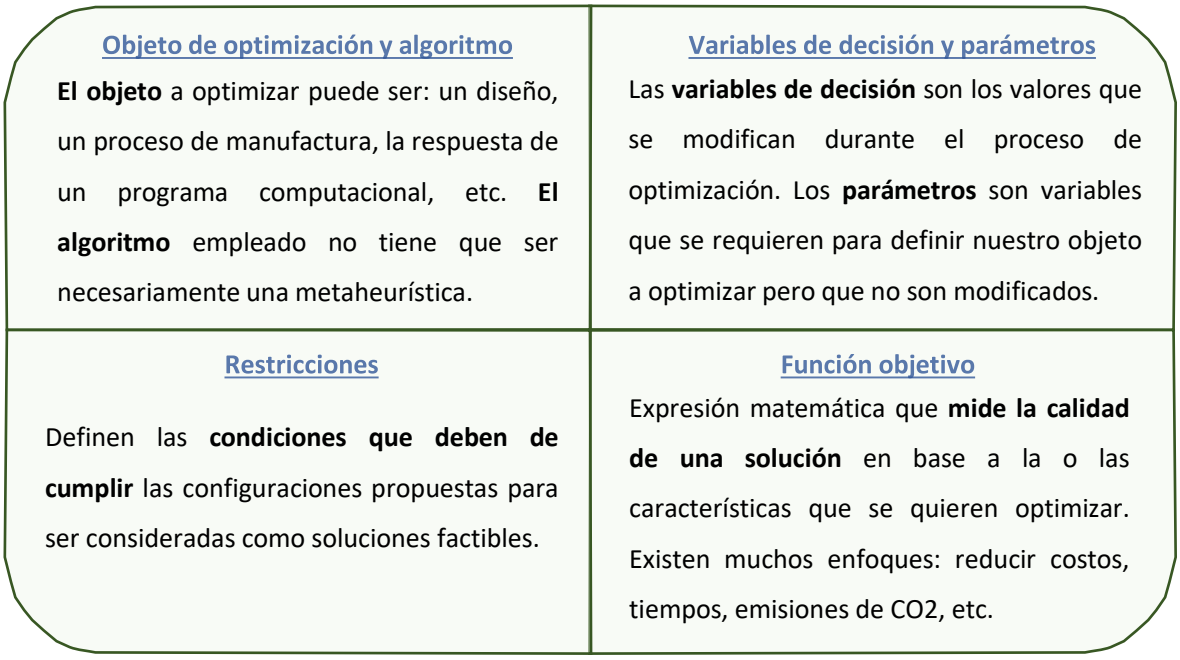


Figura 2.11 Componentes de un problema de optimización

De estos componentes básicos se generan otros conceptos más complejos. Por ejemplo, al conjunto de combinaciones que se pueden formar con los parámetros y variables de decisión consideradas en nuestro problema se le denomina **espacio de configuraciones**, espacio de soluciones o espacio de búsqueda, el cual no debe ser ni suave ni diferenciable cuando se realiza el proceso de optimización por medio de algoritmos metaheurísticos (Winker & Maringer, 2007). Dentro del espacio de configuraciones existe un subconjunto denominado **espacio de soluciones factibles**, el cual engloba todas las configuraciones que cumplen con las restricciones impuestas al problema. Usualmente, el sentido común nos hace pensar en el espacio de configuraciones como un espacio n-dimensional con picos y valles, los cuales representan respectivamente los máximos y mínimos del problema. Sin embargo, tal como lo señala Jones (1995), el operador utilizado en el proceso de optimización y, por lo tanto, el algoritmo empleado, es el que define en última instancia la existencia de estas zonas. Este tema se verá con más profundidad en el Capítulo 7.

De manera formal, y considerando un problema de minimización, los problemas de optimización se pueden definir de la siguiente manera:

$$min\,\{f(\boldsymbol{x}) \mid \boldsymbol{x} \in \overline{\boldsymbol{X}}, \overline{\boldsymbol{X}} \subseteq \boldsymbol{S}\} \tag{2.4}$$

Sujeto a:

$$\begin{aligned} g_i(\boldsymbol{x}) &\leq 0 && \forall\, i \in \{1,2,\dots,m\} \\ h_i(\boldsymbol{x}) &= 0 && \forall\, i \in \{1,2,\dots,n\} \\ a_j \leq x_j &\leq b_j && \forall\, j \in \{1,2,\dots,p\} \end{aligned}$$

donde g_i y h_i son las restricciones del problema, a_j y b_j son los límites de las variables de decisión, $\boldsymbol{S}$ es el espacio de configuraciones, $\overline{\boldsymbol{X}}$ es el conjunto de soluciones factibles, $f(\boldsymbol{x})$ es la función objetivo y $\boldsymbol{x}$ es una solución formada por el vector de variables de decisión:

$$\boldsymbol{x}^T = \{x_1, x_2, \dots, x_q\} \tag{2.5}$$

donde q es el tamaño del problema. La solución $\boldsymbol{x}^*$ es óptima si cumple que:

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \qquad \forall\, \boldsymbol{x} \in \overline{\boldsymbol{X}} \tag{2.6}$$

Existen problemas para los cuales se requiere optimizar más de una característica de las soluciones, estos se conocen como **problemas de optimización multi-objetivo** y se definen de la siguiente manera:

$$min\,\{f_1(\boldsymbol{x}), \dots, f_k(\boldsymbol{x}) \mid \boldsymbol{x} \in \overline{\boldsymbol{X}},\ \overline{\boldsymbol{X}} \subseteq \boldsymbol{S}\} \tag{2.7}$$

donde k es el número de funciones objetivo del problema. Coello (2018) indicó que cuando existe un único punto que optimice de manera simultánea todas las funciones objetivo consideradas es porque no existe

conflicto entre ellas. Al ser esto poco común, en estos problemas se suelen buscar puntos de equilibrio entre los objetivos a optimizar. Una forma de hacerlo es por medio del **Óptimo de Pareto** (Pareto, 1896). Una solución $\boldsymbol{x}^* \in \bar{\boldsymbol{X}}$ es un óptimo de Pareto si no existe otro vector $\boldsymbol{x} \in \bar{\boldsymbol{X}}$ de tal manera que:

$$f_i(\boldsymbol{x}) \leq f_i(\boldsymbol{x}^*) \; \forall \; i = 1,2,\ldots,k$$
$$f_j(\boldsymbol{x}) < f_j(\boldsymbol{x}^*) \; para \; al \; menos \; un \; valor \; de \; j \qquad (2.8)$$

Esto significa que no existe un vector posible de las variables de decisión $\boldsymbol{x} \in \bar{\boldsymbol{X}}$ que podría producir alguna mejora en uno de los objetivos sin provocar de manera simultánea la degradación o empeoramiento de otro objetivo. Es usual que existan múltiples óptimos de Pareto, conociéndose a la imagen formada por el conjunto de estos como **Frontera de Pareto**. Un ejemplo de la frontera de Pareto se muestra en la Figura 2.12. Se puede observar que la Frontera de Pareto está conformada por la unión de aquellas soluciones representadas por puntos verdes. En estas existe un punto de equilibrio entre los objetivos representados por las funciones objetivo f_1 y f_2, por tal razón son consideradas como óptimos de Pareto. Por otra parte, los puntos de color naranja no pueden ser considerados como óptimos de Pareto debido a que es posible mejorar la situación de f_1 o f_2 sin producir una afectación en la otra función objetivo.

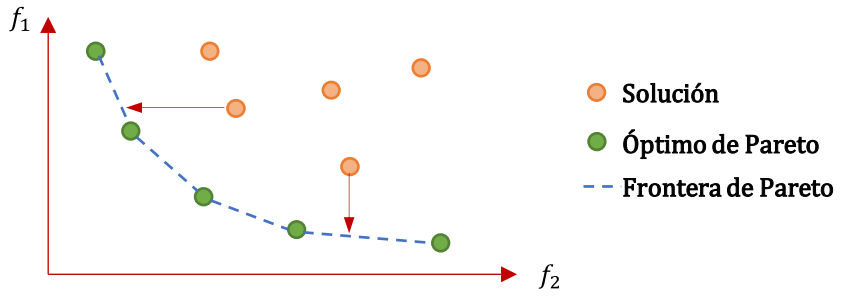


Figura 2.12 Frontera de Pareto. Tomada de (Velasco & Hospitaler, 2021)

2.10. Optimización aplicada a la Ingeniería Estructural

Actualmente existen una gran cantidad de trabajos publicados sobre la aplicación de los algoritmos metaheurísticos en problemas de optimización de muy diversas áreas como la química, la electrónica, la toma de decisiones, etc. Al estar este libro enfocado a la Ingeniería Estructural se hablará principalmente sobre artículos publicados donde se optimizan estructuras sujetas tanto a cargas

gravitatorias como dinámicas. En un principio, la optimización de estructuras puede parecer solo un ejercicio académico; sin embargo, su uso en la práctica puede brindar grandes beneficios tanto económicos como sociales.

Tal como indica la ONU en su noveno objetivo para alcanzar el desarrollo sostenible, resulta imperativo que los países comiencen a establecer sistemas de infraestructura que sean resilientes y sostenibles (United Nations, 2021). Existen diferentes formas de definir la sostenibilidad de una infraestructura y una de ellas hace referencia al empleo mínimo de recursos naturales durante su proceso constructivo. Debido a que los algoritmos metaheurísticos abren la posibilidad para reducir los costos y volúmenes de material empleados en la industria de la construcción, su estudio y aplicación es un campo que se encuentra en sintonía con el noveno objetivo de desarrollo sostenible y, por lo mismo, que ha generado un gran interés dentro la comunidad científica. Como se observará más adelante, existen diferentes enfoques de aplicación de las metaheurísticas en la Ingeniería Estructural. Para facilitar el estudio de la bibliografía, Zavala *et al*. (2014) realizaron una clasificación de las aplicaciones en dos grupos principales, uno enfocado al **diseño de elementos** y otro enfocado al **diseño topológico de las estructuras**. El primero de estos abarca todos los trabajos relacionados con la optimización de formas y dimensiones de los elementos de una estructura, mientras que el segundo hace referencia a los trabajos en los que se definía la forma completa de la estructura con el objetivo de encontrar la topología o geometría que permitieran un diseño óptimo.

De manera adicional a los dos grandes grupos presentados por Zavala *et al*. (2014), resulta necesario agregar una tercera clasificación que abarque los problemas enfocados a la **actualización de modelos de elementos finitos**. En esta clase de problemas, se busca minimizar el error de la respuesta del modelo computacional de una estructura en comparación con la respuesta de su homóloga real ensayada en laboratorio (Gu *et al*., 2012; Hofmeister *et al*., 2019).

Debido a la complejidad asociada a programar rutinas de análisis estructural y verificación de restricciones, lo más conveniente es utilizar de manera conjunta programas de elementos finitos con algoritmos metaheurísticos para la optimización de estructuras. Sobre esto, Ashwini y Vijaykumar (2016) señalan que hoy en día es tendencia el combinar ambos tipos de herramientas, existiendo además una gran variedad de algoritmos de optimización capaces de resolver problemas complejos de manera efectiva y eficiente.

A continuación, se describen brevemente algunos trabajos publicados relacionados con la optimización de estructuras. Es importante indicar que los siguientes estudios están ordenados de manera cronológica en lugar de por temática o por área de aplicación. Debido a la gran cantidad de trabajos existentes, sólo se mencionarán algunas características y conclusiones sobresalientes.

Uno de los primeros trabajos donde se optimizó una estructura por medio de metaheurísticas fue realizado por Richard Balling (1991), quien utilizó recocido simulado (RS) para optimizar la estructura de un edificio tridimensional irregular de seis niveles. En este trabajo se compararon los resultados obtenidos por RS con el algoritmo de programación no lineal conocido como *Branch and Bound*, el cual enumera las posibles soluciones del problema y enfoca la búsqueda en aquellas más prometedoras. En el estudio se observó una robustez mayor de algoritmo metaheurístico. Posteriormente, Hajela *et*

al. (1994) diseñaron una armadura por medio de un proceso de optimización topológica que tomaba en cuenta requisitos de rigidez y resistencia. De manera adicional, en el proceso de optimización se tomaron en cuenta las dimensiones de los elementos de la armadura, buscando minimizar el peso total de la estructura. Kocer y Singh (1996) optimizaron el diseño de postes de transmisión pretensados por medio de algoritmos genéticos y un algoritmo que combinaba *Branch and Bound*, un método enumerativo y Programación Cuadrática. Como parte de las conclusiones, los autores indicaron que los algoritmos genéticos presentaron un desempeño superior. En 1997, Coello *et al*. (1997) resolvieron el problema de optimizar una viga de concreto armado simplemente apoyada por medio de algoritmos genéticos. En el proceso de optimización se buscó minimizar el costo total de la viga y se planteó la idea de identificar las características del diseño óptimo con la finalidad de facilitar el proceso de diseño de esta clase de elementos.

Hasta este punto, la mayoría de los trabajos de optimización de estructuras habían utilizado variables de decisión de valores continuos. Como todos los practicantes saben, esto es contrario al uso de elementos con medidas estandarizadas que requiere la ingeniería. Uno de los primeros en tomar en cuenta este aspecto fueron Rajeev y Krishnamoorthy (1998) quienes optimizaron el diseño de marcos planos de concreto reforzado por medio de algoritmos genéticos y considerando variables de decisión discretas. Esta forma de definir los posibles valores de las variables de decisión permitió alcanzar diseños más realistas.

En 1999 Manoharan y Shanmuganathan (1999) optimizaron una serie de armaduras por medio de tres algoritmos metaheurísticos: búsqueda tabú, recocido simulado y algoritmos genéticos. En este trabajo también se puso a prueba el algoritmo conocido como *Branch and Bound*. El estudio concluyó que todos los algoritmos metaheurísticos eran capaces de localizar óptimos satisfactorios para los casos estudiados, lo cual no era aplicable para el algoritmo *Branch and Bound*. De manera posterior, Coello y Christiansen (2000) emplearon un algoritmo genético multi-objetivo para optimizar dos armadura de 25 y 200 elementos. En el proceso se buscó minimizar el peso de la estructura por medio de la maximización del índice de aprovechamiento de cada barra. El algoritmo demostró ser efectivo debido a que separaba los objetivos en diferentes funciones objetivos, más sencillas de resolver. Hasançebi y Erbatur (2002) realizaron el proceso de optimización de una armadura por medio de recocido simulado. En este trabajo se modificó tanto la topología como las dimensiones de los elementos de la armadura. El proceso permitió minimizar el peso de la estructura tomando en cuenta las restricciones impuestas por el código de diseño de referencia, es decir: desplazamientos, esfuerzos máximos y estabilidad de la estructura. En este estudio en particular se consideraron perfiles comerciales con sus respectivas propiedades, siendo además necesario considerar un proceso adicional que incluían o removían nodos y/o elementos de la armadura (Oguzhan Hasançebi & Erbatur, 2002).

Otro trabajo que se menciona es el realizado por Degertekin *et al*. (2008) quienes optimizaron un marco tridimensional de acero. En el proceso de optimización se minimizó el peso de la estructura, considerándose perfiles disponibles comercialmente y efectos de no linealidad geométrica. La estructura se optimizó por algoritmos genéticos y búsqueda tabú. Los resultados obtenidos por ambos algoritmos fueron comparados, encontrándose que búsqueda tabú brindó una estructura óptima con un

peso ligeramente inferior al de aquella obtenida por algoritmos genéticos. Entrando en el área del ingeniería sísmica, Shook *et al*. (2008) utilizaron algoritmos genéticos para optimizar el sistema de control de amortiguadores magneto – reológicos. En este trabajo se probó el modelo de una estructura de tres niveles y 9 m de altura en una mesa vibradora para verificar los resultados, los cuales mostraron que los sistemas de control obtenidos por algoritmos genéticos permitieron una reducción efectiva de las aceleraciones y desplazamientos presentados por la estructura. Atabay (2009) optimizó un marco tridimensional de concreto reforzado, cuyo sistema estructural se conformaba por muros de carga. Se utilizaron algoritmos genéticos para el proceso de optimización, gracias a los cuales se minimizó el costo de los materiales requeridos para la construcción de los muros de carga. Se destaca que se utilizó el programa GENOPT (Wetter, 2001) como herramienta para el análisis de la estructura, lo cual permitió simplificar de manera considerable el algoritmo de optimización.

Martínez *et al*. (2010) aplicaron la optimización por metaheurísticas a la estructura de puentes, específicamente, optimizaron el costo del diseño de pilas cortas las cuales estaban compuestas por secciones rectangulares huecas de concreto reforzado. El proceso se realizó por medio de tres metaheurísticas distintas: algoritmos genéticos, optimización por colonia de hormigas y *threshold acceptance*. El costo de la mejor solución encontrada presentó una reducción del 33% con respecto al costo de estructuras típicas, demostrándose así el gran beneficio que esta clase de algoritmos pueden proveer. Adicionalmente, se propuso eliminar el concepto de visibilidad del algoritmo de optimización por colonia de hormigas debido a que carecía de sentido práctico en la optimización de estructuras. Por su parte, De Albuquerque *et al*. (2012) utilizaron metaheurísticas para optimizar el diseño de viguetas prefabricadas. Como variables de decisión se consideraron las características de las viguetas y su disposición en planta, además de los costos de manufactura, transporte y ensamblaje. La metaheurística utilizada fue algoritmos genéticos. En este estudio se encontró que el costo de manufactura de las soluciones de mayor calidad encontradas representaba el 80% del costo total del sistema de piso, mientras que el 20% restante se repartía entre las etapas de transporte y ensamblaje.

Otro trabajo asociado a la ingeniería sísmica fue presentado por Ohsaki y Nakajima (2012) quienes optimizaron los rigidizadores empleados en marcos de acero equipados con contravientos excéntricos. Las variables de decisión fueron la posición de los rigidizadores y el espesor de placa utilizado. En el trabajo se utilizó el algoritmo metaheurístico de búsqueda tabú. Para verificar las posibles soluciones se utilizó el programa de elementos finitos ABAQUS (Dassault Systemes, 2017). El estudio concluyó que era posible maximizar la energía disipada por la estructura al definir la localización y espesor de los rigidizadores. Asimismo, se señaló que la evaluación de las propuestas de solución involucraba un importante costo computacional del proceso de optimización. Por otra parte, Kaveh y Nasrollahi (2014) realizaron el proceso de optimización del diseño basado en desempeño de marcos planos de acero. En este trabajo se utilizó el algoritmo conocido como *charged system optimization*, tomándose como variables de decisión los perfiles utilizados en la estructura. Durante el proceso de optimización, se minimizó el peso total de la estructura. Se observó que el uso de algoritmos metaheurísticos brindaba soluciones de mayor calidad a aquellas obtenidas por procesos convencionales diseño.

Además de optimizar el material empleado o el costo de las estructuras, también es posible minimizar su impacto ambiental, lo cual repercute de manera directa en su sostenibilidad. Como ejemplo de esto, Yeo y Potra (2015) optimizaron la huella de carbono de un marco plano de concreto reforzado. Una conclusión remarcable del estudio es que los diseños que brindan una menor huella de CO2 son aquellos que poseen una mayor cantidad de acero de refuerzo, esto en comparación con los diseños óptimos obtenidos al minimizar el costo de la misma estructura.

Debido a que la optimización de diseños sísmicos requiere de una gran cantidad de recursos computaciones, algunos autores han estudiado el uso de modelos sustitutos, los cuales son métodos simplificados de evaluación de la respuesta estructural. Dentro de este enfoque de optimización, se puede mencionar el estudio realizado por Gholizadeh y Mohammadi (2017) quienes utilizaron redes neuronales artificiales (RNA) para predecir la respuesta no lineal de marcos planos de acero. Al utilizar RNA fueron capaces de ejecutar los procesos de optimización en fracciones de hora, lo cual implica una importante reducción del tiempo de ejecución de estos algoritmos. De manera similar, Mokarram y Banan (2018) emplearon análisis estáticos no lineales como modelo sustituto para estimar la repuesta dinámica no lineal de marcos planos de acero de diferentes alturas. En este caso, se realizó una optimización multi-objetivo en la cual se buscó minimizar de manera conjunta el costo inicial y los costos de reparación de la estructura. Asimismo, se reportó que la estrategia empleada permitió reducir de manera considerable el tiempo requerido por los procesos de optimización.

Bekdaş *et al*. (2018) optimizaron el diseño de un amortiguador de masa sintonizada instalado en un edificio de 10 niveles. Las variables de diseño consideradas fueron la masa, el periodo y el coeficiente de amortiguamiento del amortiguador. En este trabajo se utilizó un algoritmo metaheurístico denominado como *bat algorithm*. El estudio identificó que el factor más importante del diseño sísmico es el peso añadido a la estructura por la masa sintonizada. De manera posterior, Seo *et al*. (2018) optimizaron el proceso de refuerzo sísmico de una escuela cuya estructura se componía de elementos de concreto reforzado. Las variables de decisión fueron el número y la posición de las columnas a reforzar. En este trabajo se utilizó el algoritmo metaheurístico de optimización por colonia de hormigas. Los resultados indicaron que fue posible cumplir con los requisitos de desempeño reforzando únicamente el 60% de las columnas de la estructura estudiada, lo cual también disminuyó los costos de la operación.

En otro estudio, Velasco *et al*. (2022) optimizaron el refuerzo sísmico de un marco de concreto reforzado con contraventeos restringidos al pandeo (CRP). Como variables de decisión se tomaron las características geométricas del núcleo de los disipadores, su orientación y distribución en el marco. Los resultados del estudio mostraron que los diseños optimizados ahorraron, en promedio, un 65% del material empleado en los disipadores en comparación con un diseño obtenido por medio del método propuesto por Guerrero *et al*. (2016) para el diseño de este tipo de estructuras.

De manera adicional a las RNA y métodos de evaluación simplificados, existen otras técnicas de inteligencia artificial que pueden ser empleadas como modelos sustitutos. Xiao *et al*. (2022) emplearon un proceso de regresión Gaussiano como modelo sustituto en la optimización de estructuras tridimensionales de gran altura. Los procesos de regresión Gaussianos son métodos de aprendizaje

máquina que emplean una distribución de probabilidad para realizar predicciones, en este caso, sobre la respuesta dinámica de los diseños candidatos a solución. Como en casos anteriores, se reportó que el uso de este modelo sustituto disminuyó la dificultad del proceso de optimización del diseño sísmico de estructuras de gran altura.

Adicionalmente, Shan *et al*. (2023) propusieron un método de optimización estructural apoyado por modelos sustitutos que se compone de tres fases: en la primera etapa, el proceso de optimización se realizan de manera usual a través de análisis estructurales. Posteriormente, en la segunda etapa se utiliza la información obtenida de los análisis estructurales para entrenar algún método de aprendizaje máquina. Finalmente, en la tercera etapa, el algoritmo de aprendizaje máquina se utiliza para dirigir el proceso de optimización. La efectividad de este método fue estudiada considerando tres algoritmos de inteligencia artificial: *support vector regression*, redes neuronales y *light gradient boosting machine regression*. De los resultados obtenidos se concluyó que el uso de modelos sustitutos permitió una reducción del tiempo de ejecución del 48% en comparación con estrategias de optimización convencionales. Finalmente, Velasco *et al*. (2024) realizaron un estudio comparativo de los desempeños mostrados por metaheurísticas basadas en poblaciones y en una solución al ser aplicadas para resolver un problema de optimización de diseño sísmico. En este estudio se encontró que los algoritmos basados en poblaciones emplean mayores tiempos de ejecución en comparación con los algoritmos basados en una solución para esta clase de problemas. Tal resultado es de gran relevancia debido a que la mayoría de los estudios publicados sobre la optimización de diseños sísmicos emplean algoritmos basados en poblaciones.

De la revisión bibliográfica, se observa que hay varios estudios que muestran el uso de algoritmos de optimización en problemas de Ingeniería Estructural. La opinión de estos autores es que en los próximos años habrá un uso creciente de las metaheurísticas en problemas de esta área, lo que permitirá usar menos recursos de nuestro planeta y cumplir con el noveno objetivo de la ONU para alcanzar el desarrollo sostenible.

3. El problema de la ruta más corta

En este capítulo se presenta el problema de encontrar la ruta más corta para visitar un número determinado de ciudades. Para ello, se utiliza un ejemplo sencillo que no guarda relación directa con la Ingeniería Estructural, pero que cumple un objetivo esencial para la correcta comprensión de este libro: brindar una introducción ligera al uso del programa Matlab (MathWorks Inc, 2020) y su aplicación en los algoritmos de optimización. Uno de los problemas más antiguos y sencillos relacionados con la optimización resulta ser el ya mencionado Problema del Viajero, el cual implica encontrar la ruta más corta que permita visitar un número n de lugares, volviendo al punto de partida inicial al finalizar el recorrido.

Buscando crear un ejemplo que el lector pueda identificar en la realidad, se decidió optimizar la ruta que nos permita visitar los 132 Pueblos Mágicos de México que al año 2022 se encuentran inscritos en el programa dirigido por la Secretaría de Turismo (2021). El programa de los pueblos mágicos está orientado a alentar a los turistas, tanto nacionales como extranjeros, a visitar los pueblos que presentan el mayor legado cultural en nuestro país, y ha obtenido tan buen recibimiento que otros países como España (Instituto de Desarrollo Local, 2021) han implementado programas similares en sus territorios.

El Problema del Viajero se puede afrontar por medio de diferentes algoritmos metaheurísticos. Incluso, a lo largo de este libro el lector se dará cuenta que la generalidad de esta clase de algoritmos permite su aplicación a una gran variedad de problemas. Sin embargo, la solución al problema aquí presentado será buscada por medio del algoritmo conocido como Optimización por Colonia de Hormigas (OCH), mismo que es descrito de manera detallada en la siguiente sección. Antes de iniciar con la descripción del algoritmo, el lector debe notar que las metaheurísticas hacen uso de diferentes parámetros de búsqueda, también denominados hiperparámetros. Los valores que toman los hiperparámetros afectan de manera directa el desempeño del algoritmo; sin embargo, no existen métodos para definir su valor “apropiado” e inclusive estos resultan ser dependientes del problema. Por tal motivo, al optimizar con metaheurísticas es necesario realizar un proceso de calibración, basado en la prueba y el error, en el que se modifiquen de manera controlada los valores de los

hiperparámetros. Solo después de probar varias combinaciones de valores, y apoyándose en experiencias previas sobre el uso de estos algoritmos, es que el usuario puede definir la combinación de hiperparámetros más conveniente para el problema en cuestión.

3.1. Optimización por Colonia de Hormigas

Optimización por Colonia de Hormigas (OCH, o en inglés *Ant Colony Optimization*, *ACO*) es una metaheurística presentada por Dorigo *et al*. (1991). La originalidad de este algoritmo se encuentra en la idea de simular el comportamiento de una colonia de hormigas, empleando un conjunto de objetos computacionales simples, capaces de comunicarse entre sí por medio de "feromonas". Gracias a su enorme versatilidad, OCH ha sido implementado para resolver problemas diversos, como el Problema del Viajero (De Oliveira *et al*., 2021; Wang & Han, 2021), la gestión del tráfico (Lakshmanaprabu *et al*., 2019; Nguyen & Jung, 2021), la optimización de procesos de ensamblaje (Xie *et al*., 2015), el manejo de sistemas de energía (Marzband *et al*., 2016) o la optimización de diseños estructurales (Bland, 2001; Camp *et al*., 2005; O Hasançebi & Çarbaş, 2011; Martínez *et al*., 2010, 2011). Esta última aplicación se vuelve muy relevante en el contexto de este libro enfocado en la optimización en ingeniería estructural.

Para comprender el funcionamiento de este algoritmo, es necesario describir el proceso natural al cual es análogo. En la naturaleza, las hormigas no son capaces de realizar tareas complicadas cuando se encuentran solas. Sin embargo, es sabido que, al juntar un número elevado de estas, se comienza a conformar una "inteligencia" colectiva que les permite resolver problemas complejos como el localizar alimento o el identificar la ruta más corta a su hormiguero. Buena parte de esta inteligencia colectiva se encuentra sostenida por la comunicación entre los individuos de la colonia, misma que se desarrolla a través de sustancias químicas llamadas **feromonas**. Basándose en estos conceptos, Dorigo *et al*. (1991) propusieron OCH con la siguiente formulación matemática.

Siendo n el número de lugares a visitar y $a_i(t)$ el número de hormigas en lugar i para el instante t, el número total de hormigas m es igual a:

$$m = \sum_{i=1}^{n} a_i(t) \tag{3.1}$$

La distancia que separa el lugar i del lugar j, midiéndose esta de manera euclidiana, es igual a:

$$d_{ij} = \left[\left(X_1^i - X_1^j \right)^2 + \left(X_2^i - X_2^j \right)^2 \right]^{\frac{1}{2}} \tag{3.2}$$

donde X_1 y X_2 son las coordenadas de cada lugar. Los factores que determinan la probabilidad de que una hormiga, en el lugar i, decida moverse al lugar j son dos: **la visibilidad** (η) y **la intensidad del camino** (τ). La visibilidad (η_{ij}) del lugar j desde el lugar i, representa la cercanía entre ambos puntos, y es inversamente proporcional a la distancia que los separa, d_{ij}, esto es:

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{3.3}$$

Obsérvese que la visibilidad no es más que un criterio de cercanía, el cual empujará a las hormigas a visitar los lugares más cercanos disponibles, lo que tiene todo sentido si se considera que se desea minimizar la distancia total recorrida. Por su parte, la intensidad del camino (τ_{ij}) que va del lugar i al lugar j representa las "feromonas" que han dejado otras hormigas que tomaron dicho camino, y se define de la siguiente manera:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t+1) \tag{3.4}$$

donde ρ es un **coeficiente de evaporación** con valor < 1, $\tau_{ij}(t)$ es la intensidad del camino en el instante t, y $\Delta\tau_{ij}$ es la intensidad añadida al pasar las hormigas por el camino entre los lugares i y j, entre los instantes t y $t+1$. Entonces $\Delta\tau_{ij}(t, t+1)$, que es el incremento de feromonas entre los instantes t y $t+1$, se obtiene como:

$$\Delta\tau_{ij}(t, t+1) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t, t+1) \tag{3.5}$$

donde $\Delta\tau_{ij}^{k}(t, t+1)$ son las feromonas por unidad de longitud liberadas por la hormiga k en el camino de i a j, para los instantes de tiempo t y $t+1$. Aquí es importante señalar que las feromonas actúan como una memoria para el algoritmo, si un número elevado de hormigas ha recorrido en el pasado el trayecto entre el lugar i y el lugar j, ya sea por motivos de visibilidad o porque permite generar rutas más optimizadas, el algoritmo recordará dicho trayecto y alentará a las hormigas a utilizarlo en el futuro. Asimismo, el coeficiente de evaporación ρ provoca que el algoritmo olvide rutas que fueron importantes en el pasado pero que ya no son útiles para incrementar la calidad de la solución actual y, por ende, resulta necesario que el algoritmo las deje de considerar. Inicialmente Dorigo *et al.* (Colorni *et al.*, 1991) presentaron tres formas de calcular $\Delta\tau_{ij}^{k}$. Aquí, por simplicidad, se empleará aquella que les dio los mejores resultados y que se calcula una vez que las hormigas han visitado todas las ciudades del problema en cuestión:

$$\Delta\tau_{ij}^{k}(t, t+1) = \frac{Q}{L^{k}} \tag{3.6}$$

siendo Q la cantidad de feromonas liberadas y L^k la distancia total recorrida por la hormiga k una vez que ha visitado las n ciudades y regresado a su punto de partida. Como se mencionó antes, la visibilidad y la intensidad definen la probabilidad de que una hormiga decida trasladarse entre el lugar i y el lugar j, esto se hace de la siguiente forma:

$$P_{ij}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}}{\sum_{j=1}^{n}\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}} \tag{3.7}$$

donde α y β son factores que le permiten al usuario definir la importancia relativa entre la intensidad (τ) y la visibilidad (η), respectivamente. Obsérvese que, cuanto más visible sea un lugar, mayor es la probabilidad de que una hormiga se dirija hacia él, lo mismo ocurre con la intensidad. Esto se muestra de manera esquemática en la Figura 3.1. Ahí se observa a una hormiga que tiene que escoger a dónde dirigirse (A, B o C). Debido a su lejanía, solo una hormiga ha decidido moverse hacia B. Por otra parte, A se encuentra más cerca, pero muchas hormigas se están moviendo hacia C, con lo cual, estos dos lugares tienen altas probabilidades de ser escogidos como destino de la hormiga en O.

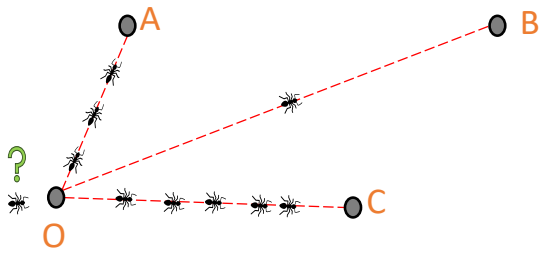


Figura 3.1 Esquema de la elección del lugar de destino en función de la visibilidad y la intensidad

3.2. Los Pueblos Mágicos de México

En la Figura 3.2 se muestra la ubicación de los 132 Pueblos Mágicos que actualmente se encuentran inscritos en el programa del Gobierno Federal (Secretaria de Turismo, 2021). Para facilitar el cálculo de las distancias que los separan, sus posiciones dentro del territorio nacional se encuentran referenciadas en coordenadas geográficas. Obsérvese que en la península de Baja California se encuentran ubicados tres Pueblos Mágicos.

Para evitar incrementar la complejidad de este problema, se realizarán dos consideraciones. La primera implica no tomar en cuenta los tres pueblos ubicados en la península de Baja California ya que, en caso contrario, se estaría forzando al algoritmo a generar una ruta que cruce por el mar. La segunda consideración es que la distancia entre dos pueblos cualesquiera (d_{ij}) se puede determinar con suficiente exactitud por medio de la fórmula de Haversine, la cual hace uso de coordenadas geográficas, esto es:

$$d_{ij} = 2R\ asen\left(\sqrt{sen^2\left(\frac{\theta_j - \theta_i}{2}\right) + \cos(\theta_i)\cos(\theta_j)\, sen^2\left(\frac{\phi_j - \phi_i}{2}\right)}\right) \qquad (3.8)$$

siendo θ y ϕ la latitud y la longitud de los pueblos *i* y *j*, respectivamente; y *R* el radio de la Tierra, considerándolo en este ejemplo igual a 6,367.5 km.

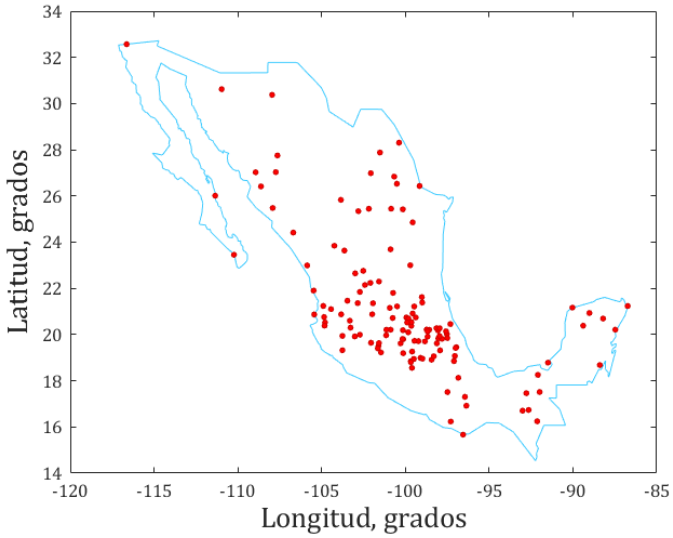


Figura 3.2 Ubicación de los 132 Pueblos Mágicos de México

3.3. Tipo de optimización

En este problema se busca minimizar la distancia total requerida para visitar todos los pueblos considerados y volver al punto de salida.

3.4. Variables de decisión y parámetros del problema

Como parámetros del problema se consideran las posiciones de los pueblos mientras que la secuencia en que estos son visitados es tomada como variable de decisión.

3.5. Codificación de las rutas

Al emplear una metaheurística es necesario establecer el tipo de codificación que tendrán las posibles soluciones ya que esto definirá los operadores que es posible utilizar durante el proceso de optimización. Por ejemplo, para el Problema del Viajero es posible utilizar codificaciones de tipo binario, de trayectoria, de adyacencia, ordinal y matricial (Larrañaga *et al*., 1999). Para ejemplificar esto, considérese un Problema del Viajero en el que se tiene que hallar la ruta más corta para visitar seis ciudades, siendo cada ciudad nombrada por los números del 1 al 6. En dicho ejemplo, si se utilizara una representación binaria, la ruta definida por las ciudades 3-2-6-1-5-4 se codificaría de la siguiente manera:

(011 010 110 001 101 100)

Por otra parte, en una codificación por trayectoria, la misma ruta sería representada por el vector:

(3, 2, 6, 1, 5, 4)

Por simplicidad, la explicación del resto de posibles codificaciones será omitida pero mayor información sobre ellas puede ser encontrada en Larrañaga *et al*. (1999). Asimismo, como el lector puede observar, la codificación por trayectoria brinda una interpretación directa de la ruta considerada. Por tal motivo, todas las rutas consideradas en este ejemplo presentarán una codificación por trayectoria.

3.6. Restricciones del problema

En este caso solo se considera una única restricción la cual es que cualquier pueblo, con excepción de aquel tomado como punto de partida, solo puede ser visitado una vez.

3.7. Descripción del código de Optimización por Colonia de Hormigas

En la Figura 3.3 se muestra un diagrama de flujo que indica cómo aplicar OCH para resolver este problema. El código completo, ejecutable en Matlab, se presenta en el Anexo 1 de este libro, y se recomienda leerlo de

manera paralela a esta sección para su mejor comprensión. Es importante mencionar que algunos aspectos del código no se ven reflejados en el diagrama de flujo de la figura 3.3, esto debido a que el diagrama de flujo busca representar las ideas generales del código y no todos y cada uno de sus detalles particulares. A pesar de lo anterior, y dado que este es el primer ejemplo del libro, su código será explicado con la mayor profundidad posible, presentándose descripciones de código más resumidas en los siguientes ejemplos.

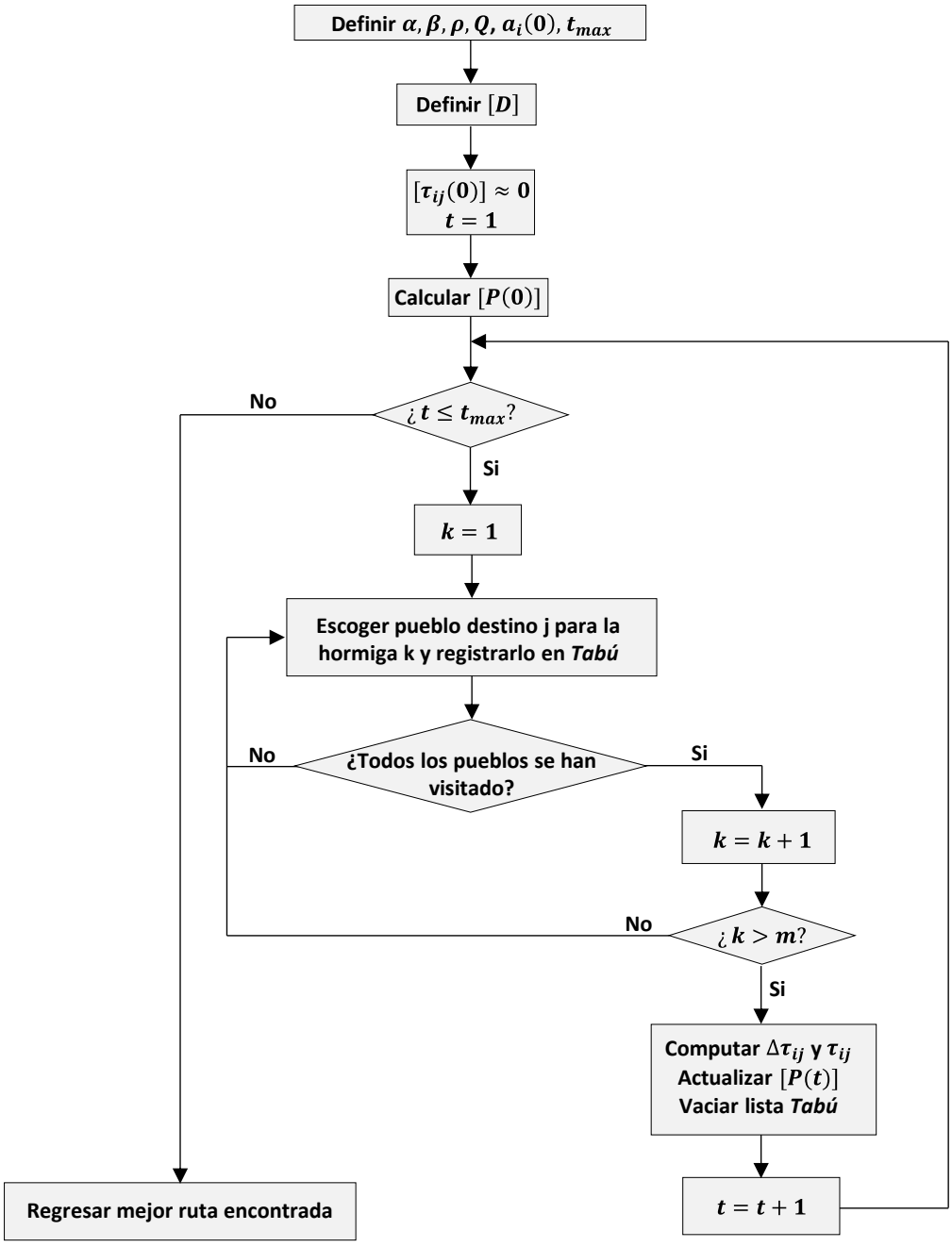


Figura 3.3 Diagrama de flujo empleado en Optimización por Colonia de Hormigas

Las dos primeras instrucciones del código son limpiar la pantalla del programa y eliminar todas las variables que pudieran existir, esto se realiza por medio de los comandos *clc* y *clear* en las líneas 2 y 3, respectivamente. Antes de iniciar el proceso de optimización, es necesario definir los valores de los hiperparámetros que controlan el funcionamiento del algoritmo, siendo estos: $\alpha, \beta, \rho, Q, a_i(0), [\tau_{ij}(0)]$ y t_{max}. Como se mencionó anteriormente, α y β son factores que controlan la influencia de la "intensidad" y la "visibilidad" en el comportamiento de OCH. Por su parte, ρ es el coeficiente de evaporación del algoritmo, el cual permite "olvidar" viejas rutas que ya no resultan útiles para el proceso de optimización. Q es la cantidad de feromonas dejadas por las hormigas durante la creación de rutas, mientras que $a_i(0)$ son las hormigas existentes en cada pueblo al inicio del proceso. Por último, $[\tau_{ij}(0)]$ y t_{max} son la matriz de intensidad para la iteración cero y el número de iteraciones que se le permitirá ejecutar al algoritmo, respectivamente. Los valores de estos parámetros se establecen entres las líneas 6 a 12 del código.

En las líneas 13 y 14 se especifica el valor considerado del radio de la Tierra y el parámetro cRestr $= 1$, el cual restringe los destinos posibles de las hormigas que se encuentran en los pueblos de la península de Yucatán con apoyo de las listas CRest y CY (línea 21 y 22). Esto se realizó debido a que existía la posibilidad de que el algoritmo genere alguna ruta en la cual se salga de la península de Yucatán por medio del mar, lo cual sería una solución poco realista. Por tal motivo, resultó necesario imponer una penalización cuando el algoritmo genere una ruta en la que, partiendo de un pueblo en Yucatán (lista CY), no se escoja como siguiente destino un pueblo en las cercanías de la península (lista $CRest$) o viceversa.

Las coordenadas geográficas de todos los Pueblos Mágicos se encuentran almacenadas en el archivo denominado "PueblosMagicos.txt", para acceder a esta información basta con utilizar el comando *load()* y brindar la ruta y del nombre del archivo. En este caso, dicha información es almacenada en la variable *X*. Para simplificar la ejecución del código, se recomienda que el archivo "PueblosMagicos.txt" se encuentre ubicado en la carpeta de trabajo de Matlab.

En las líneas 25 a 41 se generan diferentes variables que almacenan el número de pueblos del problema (*n*), el número de hormigas totales que controlará el algoritmo de manera simultánea (*m*) y diferentes matrices de ceros donde se registrarán valores como la distancia (*D*), la visibilidad (*Nu*), la probabilidad de traslación (*P*) entre los diferentes pueblos o las longitudes (*L*) y rutas (*MResultados*) recorridas por las hormigas en las iteraciones del algoritmo. Otras variables necesarias para el funcionamiento del código son la matriz de intensidad de feromonas entre los distintos pueblos (*T*), la variable *MejorL* que almacena la longitud de la mejor ruta encontrada y el vector *MejorRuta* que contiene el orden de los pueblos visitados en dicha ruta.

Posteriormente, entre las líneas 44 a 67 se generan un par de ciclos *for* anidados que son utilizados para definir la matriz de distancias. Las distancias entre pueblos se calculan por medio sus coordenadas geográficas y la ecuación de Haversine (ecuación 3.8). En estas líneas de código, los dos ciclos *for* toman de manera recursiva los diferentes pueblos considerados, de tal manera que el primero define el pueblo de partida (almacenado en la variable *i*) mientras que el segundo define el pueblo de destino (almacenado en la variable *j*). De esta sencilla manera se calculan todas las

distancias que separan a todos los pueblos del problema. Adicionalmente, en este mismo paso se impone una restricción sobre los destinos válidos que se pueden tomar desde la península de Yucatán. En este caso en particular, se observó que bastaba con multiplicar por 1,000 la distancia recorrida para desincentivar la generación de rutas que consideren traslados entre dos pueblos cruzando por el mar. Dicha penalización se codifica por medio de dos condicionales *if*, los cuales determinan si las distancias calculadas consideran pueblos alejados de Yucatán. Esto funciona de la siguiente manera: en la línea 51 se utiliza la función *find()* para determinar si el pueblo almacenado en la variable *j* se encuentra en la lista *CY*. En caso afirmativo, la función regresará un vector de dimensión distinto de cero y se procederá a verificar si el pueblo almacenado en la variable *i* no se encuentra en la lista *CRest*. Si ambas condicionantes se cumplen, se concluye que se está calculando la distancia entre un pueblo ubicado en la península de Yucatán y otro que se encuentra alejado de la península. Bajo dicha condición, la distancia entre ambos pueblos será multiplicada por 1,000. Un proceso similar ocurre en los condicionales establecidos en las líneas 58 y 59.

Por su parte, las matrices de visibilidad y de probabilidad de traslado entre los pueblos se definen en las líneas 70 a 86, siguiendo las ecuaciones 3.3 y 3.7. Es importante señalar que, en la primera iteración del algoritmo, todas las rutas posibles presentan la misma cantidad de feromonas *Q*, la cual es defina por el usuario en las líneas iniciales del código. Antes de comenzar con el proceso de optimización, es necesario definir una última matriz que registre los pueblos que pueden escoger las hormigas como siguiente destino, esta matriz se denomina aquí como $Tabu$, y en ella un valor unitario indica que un pueblo ya ha sido visitado y que ahora se encuentra prohibido como destino, mientras que un valor igual a cero indica que el movimiento está permitido. Con la finalidad de cumplir la restricción de que todos los pueblos solo serán visitados una única vez, la matriz $Tabu$ se actualiza en cada paso del proceso; asimismo, en la primera iteración el único pueblo prohibido es aquel de donde parten las hormigas.

El proceso de optimización se desarrolla en las líneas 96 a 180. Cada iteración del algoritmo inicia registrando el pueblo inicial de la ruta, y se finaliza una vez que todas las hormigas han visitado todos los pueblos considerados y han regresado a su punto de partida. Hay que recordar que una de las restricciones del problema es que solo se puede visitar cada pueblo una sola vez, por lo tanto y antes de seleccionar el siguiente pueblo a visitar, el código identifica aquellos destinos que están permitidos. Esto se realiza entre las líneas 104 a 116 donde se multiplica por cero las probabilidades de traslado hacia aquellos pueblos que ya fueron visitados con anterioridad.

Para seleccionar el pueblo destino *j* partiendo desde *i* se utiliza un número aleatorio que sigue una distribución de probabilidad uniforme entre 0 y 1, en combinación con una estructura tipo ruleta, misma que es mostrada en la Figura 3.3. La estructura tipo ruleta es una forma sencilla de escoger un objeto de un grupo de manera aleatoria y parte de la idea de que los 129 pueblos considerados en el problema presentan una probabilidad asociada de ser escogidos como destino. Asimismo, la suma de todas las probabilidades es igual a 1, de tal manera que la ruleta planteada es un intervalo entre 0 y 1 dividido en intervalos más pequeños que representan las

probabilidades P_{ij} de cada pueblo. Para comprender mejor este proceso, se brinda el siguiente ejemplo: considérese que el número aleatorio generado es igual a 0.72, teniendo este número, el algoritmo buscará en la ruleta el primer pueblo cuya sumatoria de probabilidades P_{ij} de los i pueblos anteriores más la suya propia sean mayor a 0.72, el pueblo que cumpla con tal condición será seleccionado como pueblo destino. Evidentemente, la selección por ruleta refleja el hecho de que la probabilidad de seleccionar a un pueblo como siguiente destino depende únicamente del valor de P_{ij}.

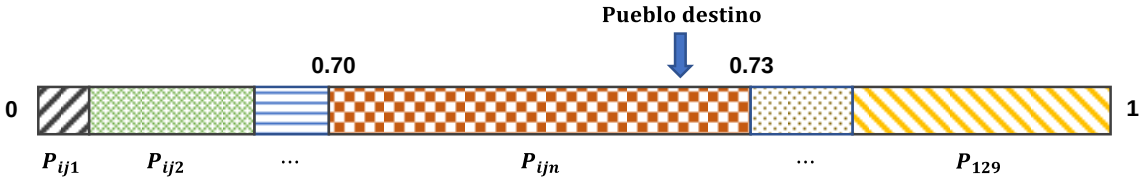


Figura 3.4 Ruleta empleada en la selección del pueblo destino

A pesar de su aparente sencillez, la selección del destino *j* por medio de una ruleta puede generar un bucle del cual al algoritmo le puede tomar un tiempo considerable salir, este proceso se describe a continuación. En primera instancia hay que considerar que todos los posibles destinos llevan asociada una probabilidad de ser seleccionados como destino. Sin embargo, no todos los movimientos son permitidos de manera simultánea, provocando esto que, para cualquier paso distinto al inicial, la probabilidad acumulada en la ruleta por los destinos permitidos sea menor a 1. A medida que se va incrementando el número de destinos prohibidos también lo hace el intervalo de la ruleta asociada a estos movimientos. Por lo tanto, para los últimos pasos de la iteración, el algoritmo requiere de muchos intentos antes de poder seleccionar un destino permitido, esto debido a que les corresponde un intervalo muy reducido de la ruleta. Claramente esto es un caso indeseable ya que afecta en última instancia el desempeño de nuestro algoritmo.

Para evitar este inconveniente, se normaliza la probabilidad acumulada en la ruleta por los destinos permitidos de tal manera que su sumatoria siempre sea igual a 1. Lo descrito anteriormente se muestra de manera esquemática en la Figura 3.5 y se realizan entre las líneas 120 a 131 del código de la siguiente manera. Entre las líneas 120 a 122 se utiliza un ciclo *for* para obtener la probabilidad acumulada de los primeros *k* pueblos y se almacena en el vector *W*. En la línea 125 se normaliza cada componente del vector *W* con respecto a la probabilidad acumulada de todos los pueblos, la cual debe ser menor a 1 si existen pueblos que están prohibidos como destino. Posteriormente, en las líneas 128 y 131 se definen los pueblos de salida *i* y destino *j*, este último por medio de la generación, con la función *rand()*, de un número aleatorio y siguiendo el proceso de ruleta anteriormente descrito.

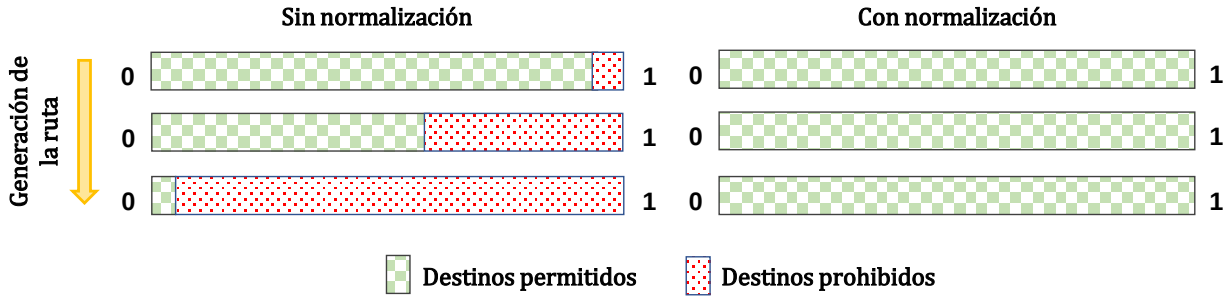


Figura 3.5 Ruleta con y sin normalización de probabilidades

Una vez se ha seleccionado el nuevo destino, se registra el pueblo destino a la matriz $Tabu$ para que la hormiga no vuelva a visitarlo de nuevo (línea 134) y se registran la distancia y ruta recorrida (línea 138 y 139, respectivamente). Cuando todas las hormigas han visitado todos los pueblos considerados y han regresado a su punto de salida, se actualiza la matriz de feromonas empleando las ecuaciones 3.4 a 3.6. Esto se realiza en las líneas 157 a 165 del código. Por último, se actualiza la matriz de probabilidad de transición (líneas 168 a 173) y se reinicia la matriz $Tabu$ (líneas 175 a 179) para iniciar la siguiente iteración del algoritmo. Las líneas finales del código sirven para identificar la mejor ruta encontrada al finalizar el proceso de optimización (líneas 183 a 186), almacenar los resultados en una variable denominada "CiudadesOrdenadas" (líneas 191 a 195) y visualizar la ruta (líneas 198 a 209).

Antes de proceder a discutir los resultados que se pueden obtener por medio de OCH, resulta conveniente detenerse un poco a analizar las características del problema. Esto con la finalidad de garantizar que la implementación del algoritmo está justificada y que además es capaz de encontrar soluciones de alta calidad. Para hacer esto es importante enfocarse en dos aspectos: el tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) del problema y la calidad de las soluciones que se pueden obtener por medio de una Búsqueda Aleatoria.

3.8. Tamaño del espacio de configuraciones

Como se recordará del primer capítulo, la principal ventaja de los algoritmos metaheurísticos es que permiten encontrar soluciones de alta calidad a problemas complejos en un tiempo razonable. Si se deseara resolver el Problema del Viajero de manera determinística, se tendría que utilizar un método enumerativo que revisase cada una de las posibles rutas del problema, es decir, que explore la totalidad del espacio de configuraciones o de soluciones.

Para este problema en particular, el número de posibles configuraciones o rutas que se pueden formar depende del número n de pueblos a visitar. Todas las rutas presentan el mismo conjunto de elementos, sin embargo, estas son diferenciables debido al orden en el que aparecen los elementos. Adicionalmente, no está permitida la repetición de elementos en las rutas de lo contrario esto

significaría que un pueblo es visitado más de una vez. Estas características implican que el tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) está definido por un número de permutaciones circulares. Hay dos condiciones del problema que se tienen que tomar en cuenta al momento de calcular las permutaciones: primero es posible descontar el paso del regreso al inicio de la ruta debido a que ese está representado por el primer elemento del conjunto. Por otra parte, la mitad de las secuencias se repiten ya que cada ruta se puede tomar en dos sentidos, esto es, la ruta se puede iniciar desde el elemento 1 o desde el elemento n. Con base en lo anterior, el problema de visitar todos los Pueblos Mágicos considerados, con $n = 129$, tiene una dimensión de:

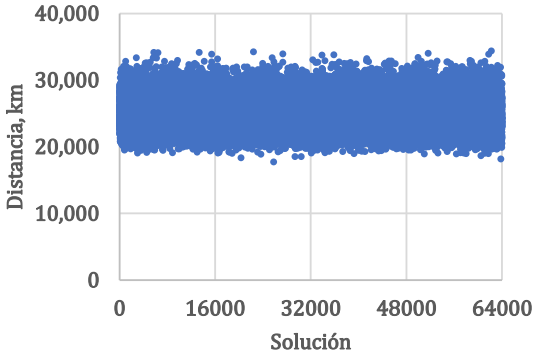
$$|\boldsymbol{S}| = \frac{P_{(n-1)}}{2} = \frac{(n-1)!}{2} = \frac{(128)!}{2} = 1.928x10^{215} \text{ rutas posibles} \qquad (3.9)$$

Este es claramente un número grande, pero para comprender mejor la magnitud del problema, considérese que, si alguien fuera capaz de revisar una ruta cada segundo, le tomaría $6.1x10^{207}$ años en revisarlas todas ¡lo cual es un tiempo mayor que la edad actual del universo! Evidentemente, al tenerse un tiempo de cómputo tan alto requerido para encontrar la solución exacta al problema, queda justificado el uso de algoritmos metaheurísticos.

3.9. Muestra del espacio de configuraciones

Una manera de afrontar este problema es considerando que la ruta más corta está conformada por la unión de los caminos más cortos entre los nodos. Siguiendo esta idea, se generaron 64,510 rutas con la probabilidad de transición entre pueblos (P_{ij}) definida únicamente en función de la visibilidad (η_{ij}). Esto se consigue haciendo el factor $\alpha = 0$, de tal manera que las hormigas tenderán a visitar el pueblo más cercano a su posición actual. Adicionalmente, se eliminó la penalización en la longitud de las rutas que salían de Yucatán por el mar, esto con el fin de evitar una distorsión de los resultados. Para la creación de este grupo de soluciones sólo se realizó una iteración del algoritmo, lo cual permitió considerar la mayor aleatoriedad posible. Las distancias recorridas por las rutas obtenidas se muestran en la Figura 3.6 por medio de un diagrama de dispersión.

De los estadísticos descriptivos de la muestra generada se observa un valor medio de la distancia recorrida cercano a 25,000 km. Sin embargo, algunas soluciones presentan valores de longitud considerablemente menores. El análisis estadístico indica que la ruta más corta presentaba una longitud de 17,754 km mientras que las rutas pertenecientes al primer cuartil recorrían una distancia menor a 23,803 km. Al presentar estas soluciones una calidad relativamente mayor a la del resto de la muestra, se considera apropiado que OCH encuentre rutas con distancias cercanas o menores a 17,000 km.



Estadísticos descriptivos		
Media =	25,260	km
Desviación estándar =	2,088	km
Mínimo =	17,754	km
Primer cuartil =	23,803	km
Mediana =	25,175	km
Tercer cuartil =	26,616	km
Máximo =	34,447	km

Figura 3.6 Muestra de rutas generadas basándose en el criterio de visibilidad

3.10. Resultados y discusión

Dada su naturaleza estocástica, es común que los algoritmos metaheurísticos regresen soluciones diferentes en cada ejecución, provocando que la calidad de las soluciones encontradas varíe de ejecución en ejecución. Para tener una mayor seguridad de que la respuesta encontrada a nuestro problema de optimización presenta la mayor calidad posible es necesario ejecutar el algoritmo en múltiples ocasiones, siendo imposible definir de manera directa el número de ejecuciones adecuado para nuestro problema. Para más información sobre la variación de la calidad de los resultados en función del número de ejecuciones se recomienda revisar el artículo de Brandeau y Chiu (1993).

El desempeño de todos los algoritmos metaheurísticos no sólo depende de la generación de números aleatorios sino también de los valores de los factores que lo controlan. Para el caso particular de OCH, los hiperparámetros que controlan su desempeño son: $\alpha, \beta, \rho, Q, a_i(0), [\tau_{ij}(0)]$ y t_{max}. De manera similar a lo que ocurre con el número de ejecuciones apropiadas por algoritmo, no existe algún método que nos permita determinar los valores de aquellos factores que nos brindarán las soluciones de mayor calidad, siendo, por lo tanto, necesario recurrir a un procedimiento de prueba y error. Para más información sobre la manera en que el desempeño de una metaheurística se puede ver afectado por el valor de sus hiperparámetros se recomienda consultar Velasco *et al.* (2022). En este caso particular se encontró que con los valores $\alpha = 2.0$, $\beta = 2.5$, $\rho = 0.95$ y $Q = 10^5$, $a_i(0) = 4$, $[\tau_{ij}(0)] = 0.1$ y $t_{max} = 25$ se obtenían resultados aceptables. Al tratarse este del primer ejemplo del libro, el algoritmo se ejecutó un total de 15 veces con los valores de los factores antes mencionados. Los resultados obtenidos de las 15 ejecuciones del algoritmo se muestran en la Tabla 2.1.

Las 15 rutas obtenidas por OCH recorren una distancia media de 12,581 km, con una desviación estándar de 168 km. El valor relativamente pequeño de la desviación estándar en comparación con la media nos permite concluir que la calidad de nuestras soluciones presenta poca variabilidad entre usos del algoritmo. Por otra parte, al considerar únicamente los 5 primeros resultados, se obtiene una

distancia media de 12,592 km con una desviación estándar de 128 km. Se puede observar que no se presentan grandes diferencias entre los valores medios y las desviaciones estándar obtenidas al considerar 5 o 15 ejecuciones del algoritmo. No obstante este resultado, se recomienda al lector el estudiar el número de ejecuciones apropiadas para cada problema de manera particular.

Tabla 2.1 Resultados obtenidos por Optimización por Colonia de Hormigas

Ejecución	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**	**E7**	**E8**	**E9**
Distancia, km	12,510	12,680	12,589	12,751	12,431	12,541	12,881	12,643	12,487
Ejecución	**E10**	**E11**	**E12**	**E13**	**E14**	**E15**	**Media**	**Desviación estándar**	
Distancia, km	12,468	12,769	12,388	12,816	12,373	12,381	**12,581**	**168**	

La ruta de mayor calidad obtenida por OCH mostró una longitud de 12,373 km, es decir, fue un 30% más corta que la distancia recorrida por la mejor ruta obtenida a través del criterio de visibilidad únicamente (17,754 km). En la Figura 3.7 se presenta la trayectoria de la ruta. Cabe aclarar que todas las rutas entre dos pueblos se consideraron rectas, ello definitivamente no es real pero se consideró así por simplicidad.

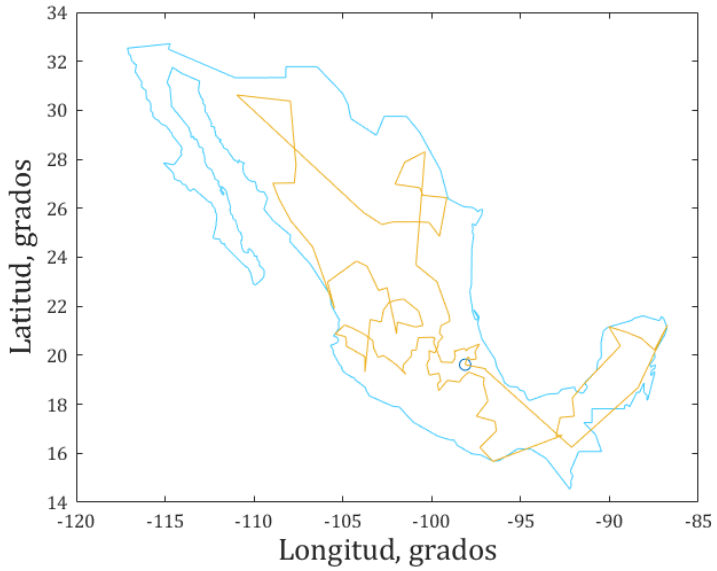


Figura 3.7 Mejor ruta encontrada por OCH, longitud = 12,373 km

3.11. Conclusiones y comentarios del ejemplo

En este capítulo se desarrolló un ejemplo de optimización para minimizar la distancia total requerida para visitar los Pueblos Mágicos de México. Se observó que las rutas obtenidas por Optimización por Colonia de Hormigas (OCH) fueron significativamente menores en distancia en comparación cuando se utilizó una Búsqueda Aleatoria. Por otra parte, se pudo comprender que, gracias a su versatilidad, las metaheurísticas tienen un gran potencial para su uso en la optimización de problemas complejos y diversos. Más adelante en el libro se demostrará que estos algoritmos también pueden ser utilizados para el diseño y mejoramiento del desempeño de sistemas estructurales ante diversos tipos de acciones.

Cabe destacar que el ejemplo estudiado sólo es un ejercicio académico que muestra los resultados que se pueden obtener cuando se aplican metaheurísticas a problemas de tipo No Polinomial. En los subsecuentes capítulos se aplican otros algoritmos para resolver diferentes problemas de optimización, aunque siguiendo una estructura similar a la mostrada en este primer ejemplo. Para finalizar, es importante recalcar que la ruta encontrada por OCH está lejos de ser una respuesta definitiva, esto debido a que las metaheurísticas no son algoritmos exactos. Seguramente, existe alguna otra alternativa con una longitud menor a las aquí encontradas. El inconveniente recae en cómo encontrarla en un tiempo relativamente corto. Se invita a los lectores a probar suerte ejecutando el código de Matlab, variando los valores de los hiperparámetros considerados con la esperanza de encontrar una solución de mayor calidad.

Esta página ha sido intencionalmente dejada en blanco

4. Minimización de una función por Búsqueda Tabú

En este capítulo se realiza un ejemplo sobre el uso del algoritmo metaheurístico conocido como Búsqueda Tabú (BT) en la minimización de una función (la función de Rastrigin, que se describe en la sección 4.3) que presenta múltiples valores mínimos y máximos. Este ejemplo, relativamente sencillo desde la perspectiva de la optimización, tiene como objetivo el brindarle al lector una noción sobre las afectaciones que sufre el desempeño de una metaheurística debido a la dimensión o complejidad de los problemas.

4.1. Búsqueda Tabú

Búsqueda Tabú (en inglés *Tabu Search, TS*) es una metaheurística desarrollada por Fred Glover en el año de 1977 (Laguna, 2018). Este es un algoritmo de búsqueda basado en una solución única que presenta la capacidad de aprovechar el conocimiento recabado durante la exploración gracias al empleo de estructuras de memoria. Su uso se enfoca principalmente a problemas de optimización combinatoria aunque también es posible aplicarlo a problemas cuyo espacio de soluciones sea continuo (B. Lin & Miller, 2004).

De manera general, BT se compone de una solución inicial (la cual puede ser generada de manera aleatoria o por medio de otra heurística), de un algoritmo de **Búsqueda Local** (BL) y de una estructura de memoria que se denomina como **Lista Tabú**. Este algoritmo recibe su nombre debido a que, durante el proceso de búsqueda, define una serie de tabúes o movimientos no permitidos que controlan la generación de soluciones nuevas. En la Figura 4.1 se muestra el diagrama de flujo básico de BT. Como se puede observar, BT se mueve a través del espacio de configuraciones por medio de la modificación de una solución única $\boldsymbol{x}$. Para aceptar la nueva solución propuesta, denominada como $\boldsymbol{x}'$, esta no solo tiene que presentar una calidad mayor a $\boldsymbol{x}$ sino que además debe ser construida sin efectuar un movimiento prohibido o tabú. En caso de que $\boldsymbol{x}'$ sea aceptada y reemplace a la solución $\boldsymbol{x}$, la Lista Tabú del algoritmo es actualizada y movimientos nuevos son prohibidos.

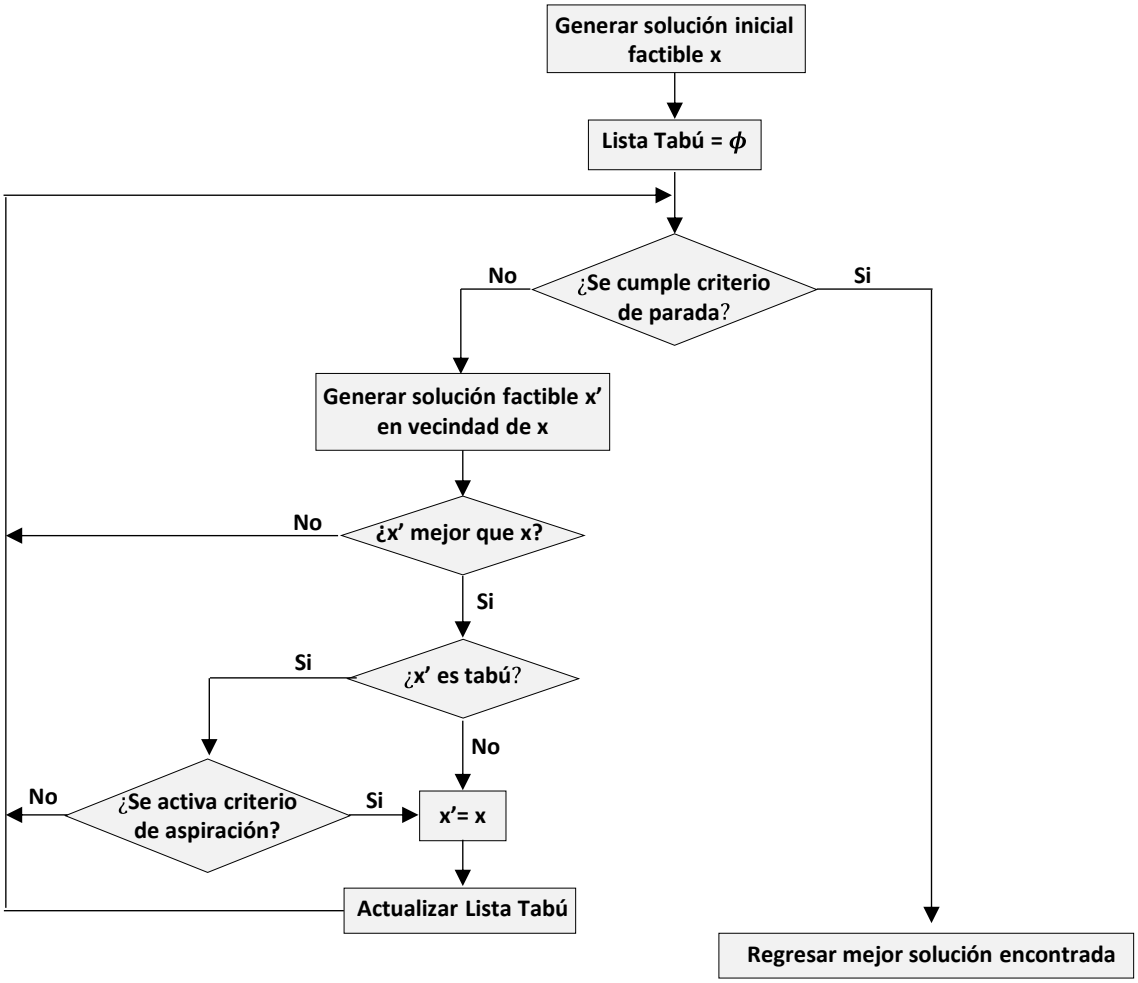


Figura 4.8 Diagrama de flujo de Búsqueda Tabú

A diferencia del fuerte sustrato matemático que dirige a Optimización por Colonia de Hormigas, el funcionamiento de BT está más bien controlado por una serie de reglas de fácil programación y que están asociadas a la generación de soluciones nuevas, basándose en la información registrada en la Lista Tabú. Como ocurre con muchas metaheurísticas, hay versiones de BT que incluyen componentes más complejos en el algoritmo, los cuales pueden desde modificar su enfoque de búsqueda hasta aceptar movimientos establecidos como prohibidos debido a su estatus de tabú. Para más información sobre estos elementos adicionales se recomienda consultar las referencias correspondientes (Glover, 1989a, 1989b). A continuación, se discuten brevemente los componentes básicos mencionados de BT.

El núcleo de BT es una estructura de memoria que se conoce como **Lista Tabú (LT)** y su objetivo es dirigir el proceso de búsqueda estableciendo o prohibiendo algunos movimientos en función de los atributos que almacena. Durante el proceso de optimización, BT visita múltiples soluciones factibles, siendo los atributos de estas almacenados de manera íntegra o parcial en la LT. Cuando un atributo se encuentra en la LT, se dice que este es un tabú activo y permanecerá en dicho estado un número dado

de iteraciones, llamándose a ese número como “tenure” o **Tenencia**. La información almacenada en la LT se utiliza de dos maneras: ya sea impidiendo que se visiten soluciones con atributos similares a los tabús activos o prohibiendo que los atributos considerados como tabú activos, presentes en la solución actual, sean alterados en las siguientes soluciones creadas.

El funcionamiento de la LT se comprende mejor si se conoce el concepto de vecindario. Por ahora se nombrará como vecindario $N(\boldsymbol{x})$ a la región del espacio de soluciones inmediatamente próxima a la solución actual $\boldsymbol{x}$, de tal manera que las soluciones que conforman el vecindario pueden ser alcanzadas en la próxima iteración del algoritmo. Debido a que parte del vecindario actual se encuentra restringido por los tabús activos, denominándose a la región prohibida como $\boldsymbol{T}$, el grupo de soluciones que pueden ser visitadas en la próxima iteración resulta ser $N(\boldsymbol{x}) - \boldsymbol{T}$. Esto se ejemplifica en la Figura 4.2. Ciertamente, la reducción del vecindario $N(x)$ resulta ser un aspecto con mayor interés teórico que práctico, pero es debido a esto que BT puede ser visto como una metaheurística de vecindario variable (Glover *et al*., 1993).

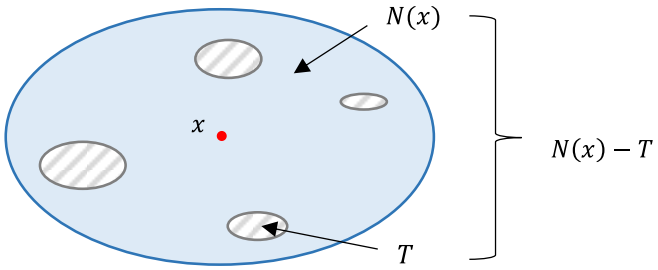


Figura 4.2 Conjunto de soluciones $\boldsymbol{N(x) - T}$

Para que un atributo sea considerado como tabú, este debe ser seleccionado o desechado en la creación de una nueva solución factible. Las implicaciones del tabú activo dependen del caso que elija el usuario del algoritmo. Si se indica que se consideren como tabús activos aquellos atributos que sean añadidos a la nueva solución, el algoritmo no podrá retirarlo de las nuevas soluciones factibles hasta que finalice su tenencia (Laguna, 2018). Por otra parte, si el atributo es desechado en la nueva solución y se considera como tabú activo significa que este no podrá ser añadido nuevamente en la creación de las siguientes soluciones hasta que finalice su tenencia (Glover & Laguna, 1999).

Resulta claro que cada alternativa presenta sus ventajas y sus inconvenientes. Considerando el caso en que se tome como tabú los atributos recién añadidos, la imposibilidad de modificarlos provocará que el algoritmo intensifique la búsqueda en la vecindad del valor del atributo considerado como tabú, consumiendo a la vez tiempo de cómputo en una zona que se desconoce si es prometedora o no. En el caso de considerar como tabú los atributos recién desechados, el evitar que estos sean aceptados de nuevo imposibilita al algoritmo el revisitar regiones de baja calidad. Es necesario indicar que no existe una regla general que defina cómo, cuándo ni qué implicaciones tiene que un atributo sea considerado como tabú activo, siendo necesario definir esto en función de lo que se sabe del problema en cuestión.

Existen dos maneras en que las Listas Tabú registran los datos durante el proceso de búsqueda: la primera implica registrar de manera íntegra todos los atributos de las soluciones visitadas, mientras que en la segunda solo se registran algunos atributos. Las listas de memoria que almacenan todos los atributos se denominan como **Memorias Explícitas**, mientras que aquellas que solo registran una fracción de los atributos que conforman una solución se denominan como **Memorias Atributivas**.

En lo que respecta a la **Tenencia**, lo más común es considerarla como constante, no existiendo una regla específica para definir su valor. Sin embargo, parecer ser que el valor óptimo crece a medida que aumenta el número de variables de decisión (Martínez-Gavara *et al*., 2021). La Tenencia, además de definir la longitud de la Lista Tabú, modifica el enfoque que presenta el algoritmo de búsqueda siendo posible utilizar más de una lista para obtener un algoritmo con enfoques complementarios de diversificación y de intensificación (Glover & Laguna, 1999). Para realizar esto es necesario emplear dos Listas Tabú: una denominada como **Lista de Memoria Corta (LMC)**, la cual se encarga de dirigir la trayectoria del algoritmo, y otra denominada como **Lista de Memoria Larga (LML)**, misma que es utilizada para escapar de óptimos locales.

Como su nombre lo indica, la LMC emplea una Tenencia de valor pequeño y registra únicamente los atributos de las soluciones encontradas más recientemente. Esta lista es utilizada por el algoritmo para controlar la creación de nuevas soluciones $\boldsymbol{x'}$ en la vecindad de la solución actual $\boldsymbol{x}$ haciendo uso de los criterios arriba mencionados sobre prohibir o fijar algunos valores de las variables de decisión. Debido a su baja Tenencia, lo mejor es que estas listas hagan uso de Memorias Atributivas. Por su parte, la LML presenta una Tenencia mayor que se utiliza cuando el algoritmo no es capaz de encontrar una solución de mayor calidad en el entorno de la solución actual. Cuando esto ocurre se dice que el algoritmo quedó atrapado en un óptimo local y provoca que se vea imposibilitado para continuar explorando el espacio de configuraciones. Uno de los métodos más utilizados para salir de los óptimos locales es reiniciar el proceso de búsqueda, es decir, generar una nueva solución en algún otro punto del espacio de configuraciones. Este método presenta el claro inconveniente de que la nueva solución no cuenta con valores optimizados en sus variables de decisión. Sin embargo, empleando la LML, es posible evitar perder toda la información obtenida hasta ese punto por el algoritmo.

Para escapar de un óptimo local, BT puede recurrir a la LML para construir una nueva solución, ya sea por medio del agrupamiento de bloques de atributos desarrollados, es decir, grupos de valores que brinden una alta calidad a la solución, o reiniciando la búsqueda en una zona alejada del espacio de configuraciones. Esto por medio de valores de las variables de decisión que fueron utilizados con menor frecuencia durante el proceso de búsqueda hasta ese punto. Una característica usual de las LML es registrar la frecuencia de los movimientos realizados durante el proceso de optimización. A las que presentan esta característica se les conoce como **Listas de Frecuencia** (Glover *et al*., 1993; Laguna, 2018). Las Listas de Frecuencia emplean fracciones que cuentan en sus numeradores los cambios y permanencias de los atributos de las soluciones actuales, siendo posible utilizar tres valores en los denominadores: la suma total de eventos o iteraciones, el promedio de los numeradores o el valor máximo de los numeradores (Laguna, 2018).

Dos elementos adicionales que permiten refinar el algoritmo de BT son los denominados como **Criterio de Aspiración** y **Oscilación Estratégica**. El primero indica cuándo un tabú activo puede ser ignorado al momento de realizar un movimiento (Martínez-Gavara *et al*., 2021). Lo más común de este criterio es activarlo cuando la nueva solución permite una mejora de la calidad en comparación con la mejor solución encontrada hasta ese momento. Por su parte, la **Oscilación Estratégica** se aplica a las listas de largo plazo, permitiéndoles variar entre una estrategia de intensificación y otra de diversificación al momento de reiniciar la búsqueda. Como el lector se habrá dado cuenta, más allá de haber enunciado las reglas sobre prohibir o fijar algunos atributos considerados como tabúes, no se ha brindado una descripción detallada sobre el proceso de generación de nuevas soluciones. Esto se debe a que BT es usualmente combinado con otro algoritmo denominado BL.

4.2. Algoritmo de Búsqueda Local

BL es uno método de búsqueda eficiente que se combina frecuentemente con otros algoritmos como BT o Recocido Simulado (Michiels *et al*., 2018). Su funcionamiento se basa en la idea de que es posible alcanzar óptimos locales a partir de soluciones iniciales por medio de mejoras discretas y consecutivas. En la Figura 4.3 se muestra el diagrama de flujo de un algoritmo de BL considerando que se desea minimizar la función objetivo $f(x)$.

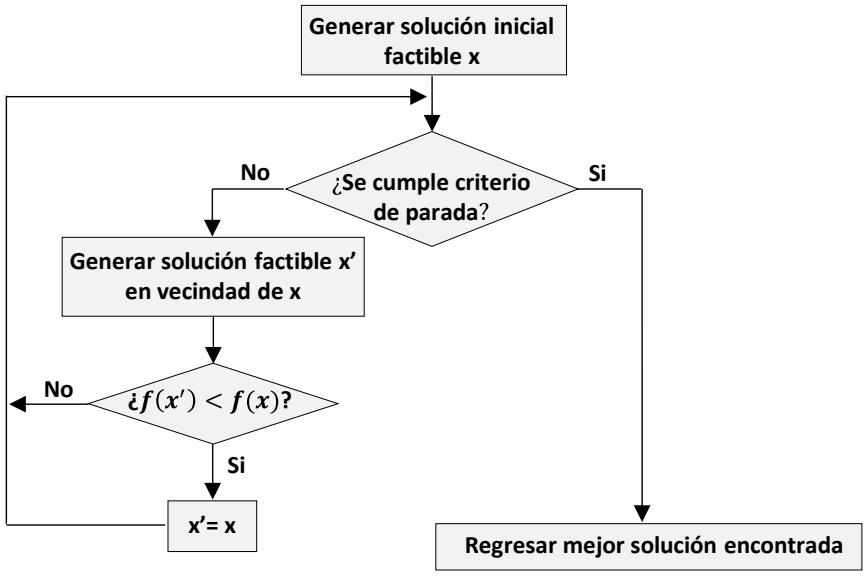


Figura 4.3 Diagrama de flujo del algoritmo de Búsqueda Local

El proceso de búsqueda de estos algoritmos parte de una solución $\boldsymbol{x}$ cualquiera y, como su nombre lo indica, explora su vecindad inmediata por medio de cambios discretos en los valores de sus variables de decisión con la esperanza de encontrar una solución $\boldsymbol{x}'$ de mayor calidad. El número de variables de decisión a modificar es definido por el usuario siendo usual seleccionarlas de manera aleatoria. En lo que respecta a los valores de las variables de decisión, lo más recomendable es que estos cambien a aquellos valores en su vecindad inmediata, escogiendo la dirección del cambio también de manera aleatoria. En caso de encontrar dicha solución $\boldsymbol{x}'$, esta reemplaza a $\boldsymbol{x}$ como la solución actual y el proceso se repite. El algoritmo se detiene cuando queda atrapado en un óptimo local. En la Figura 4.4 se muestra de manera esquemática la trayectoria seguida por BL en un proceso de optimización, siendo los vecindarios de la solución actual de cada iteración representados por medio de elipses. Como se puede observar, BL genera una trayectoria en el espacio de configuraciones, la cual está definida en este ejemplo por las soluciones 1 a 4. Este proceso de saltar de solución en solución se repite hasta que no es posible encontrar una nueva de mayor calidad en la vecindad inmediata de la solución actual.

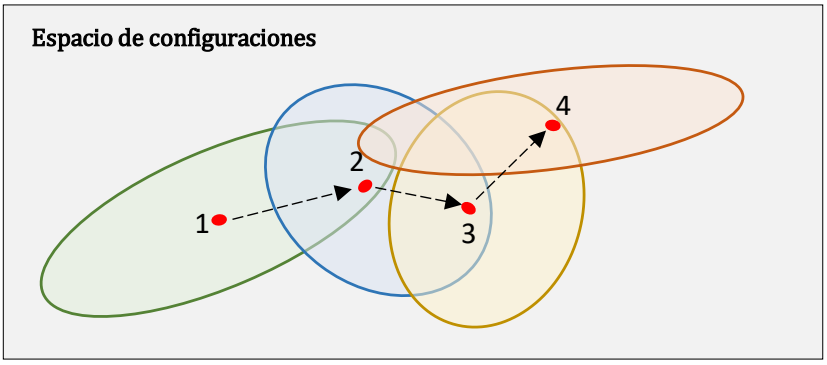


Figura 4.4 Esquema de la trayectoria de búsqueda seguida por Búsqueda Local

Los algoritmos de BL tienen dos enfoques por medio de los cuales se determina la procedencia del reemplazo de la solución actual $\boldsymbol{x}$ por la nueva solución $\boldsymbol{x}'$ encontrada. Estos enfoques se denominan como **Mayor Mejora** y **Primera Mejora**. En el primer caso, la solución actual $\boldsymbol{x}$ solo puede ser reemplaza por aquella solución $\boldsymbol{x}'$ que presente la mayor calidad en su vecindad, si es que esta existe. Claramente, la hipotética solución $\boldsymbol{x}'$ solo puede ser nombrada después de haber revisado todas y cada una de las soluciones existentes en la vecindad de $\boldsymbol{x}$, proceso que puede requerir de un tiempo de cómputo excesivo. Por otra parte, el enfoque de Primera Mejora permite el reemplazo de la solución actual $\boldsymbol{x}$ por la primera solución $\boldsymbol{x}'$ que encuentre BL y que además presente una mejora en la calidad.

Debido a su sencillez, los algoritmos de BL presentan el inconveniente de ser incapaces de escapar de óptimos locales. Sin embargo, este problema se puede resolver por medio de múltiples reinicios o al combinarlos con otras metaheurísticas.

4.3. Función de Rastrigin

En este ejemplo se realizará un ejercicio similar al del Congress on Evolutionary Computation (CEC) de 2005, donde se pusieron a prueba diferentes algoritmos de optimización por medio de una serie de problemas de referencia (Suganthan *et al*., 2005). Por simplificación, BT solo será probado en la búsqueda del mínimo de una de las 25 funciones empleadas en el CEC – 2005, específicamente se minimizará la función conocida como Función de Rastrigin, misma que se define de la siguiente manera:

$$F(\boldsymbol{x}) = \sum_{i=1}^{D} (z_i^2 - A\,cos(2\pi z_i)) + A_n \tag{4.1}$$

pudiendo A y A_n tomar cualquier valor real. Cabe aclarar que en el CEC – 2005, la Función de Rastrigin empleada originalmente era la siguiente:

$$F(\boldsymbol{x}) = \sum_{i=1}^{D} (z_i^2 - 10\,cos(2\pi z_i) + 10) + f_{bias} \tag{4.2}$$

donde D es el número de dimensiones de la función, $f_{bias} = -330$ es el valor del mínimo global, mismo que se encuentra ubicado en el punto $\boldsymbol{o} = \{o_1, o_2, \dots, o_D\}$. El vector $\boldsymbol{z} = \{z_1, z_2, \dots, z_D\}$ se define como la diferencia entre el punto $\boldsymbol{x} = \{x_1, x_2, \dots, x_D\}$ donde se evalúa la función, y las coordenadas del mínimo global.

$$\boldsymbol{z} = \boldsymbol{x} - \boldsymbol{o} \tag{4.3}$$

En la Figura 4.5 se muestra la Función de Rastrigin considerando $D = 2$ y para $\boldsymbol{x} \in [-5{,}5]^2$. Como se puede observar, se trata de una función multimodal, es decir, posee múltiples máximos y mínimos locales. Tal como se realizó en el CEC – 2005, los valores del vector $\boldsymbol{o}$ se definen de manera aleatoria cada vez que la función es generada.

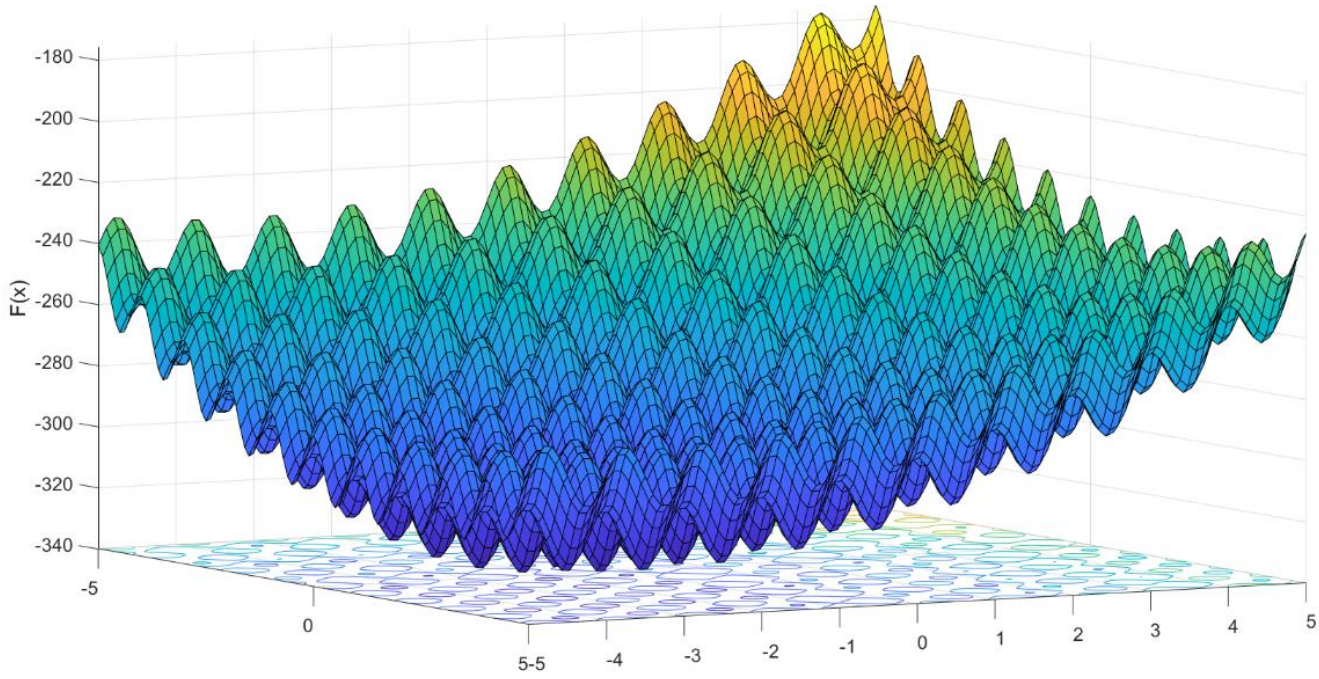


Figura 4.5 Función de Rastrigin

4.4. Tipo de optimización

El tipo de optimización a realizar será determinar el valor mínimo de la Función de Rastrigin. A pesar de que es posible utilizar BT en espacios de soluciones de tipo continuo, por simplificación, la función se dividirá en intervalos $\Delta x = 0.1$, de tal manera que el problema se considerará como de tipo combinatorio.

4.5. Variables de decisión y parámetros del problema

La Función de Rastrigin es escalable a cualquier número de dimensiones *D*. Aprovechándose de esta característica, se demostrarán las ventajas que representa el utilizar las estructuras de memoria de BT en comparación con un simple algoritmo de BL multi-reinicio, esto a través de la minimización de la Función de Rastrigin para $D = 10, 20, 30\ y\ 50$ dimensiones. Resulta claro que, a mayor valor de D, resultará más complicado para el algoritmo el encontrar no solo el mínimo global de la función sino también soluciones de alta calidad.

4.6. Codificación de las soluciones

Las propuestas a solución de este problema fueron representadas por valores discretos, almacenándose sus características en un arreglo vectorial de longitud D.

$$\boldsymbol{x} = (x_1, x_2, x_3, \dots, x_{\mathrm{i}}, \dots, x_D) \tag{4.4}$$

En esta representación, cada componente del vector puede presentar valores entre -5 y 5, ambos inclusive, con un paso Δx entre valores de 0.1. Cabe señalar que este problema podría resolverse considerando variables de decisión de valores continuos; sin embargo, se abordó a través de un dominio discreto debido a que estos son más apropiados para el diseño de estructuras dado el uso de elementos con dimensiones estandarizadas que exige la industria de la construcción.

4.7. Restricciones del problema

En este ejemplo no existen restricciones más que las impuestas por el intervalo en el cual existe la función, es decir, todas las soluciones posibles son factibles.

4.8. Descripción del código de Búsqueda Tabú

En la Figura 4.6 se muestra el diagrama de flujo del algoritmo de BT empleado en este ejemplo. El código fue ejecutado en el programa Matlab y se puede revisar de manera completa en el Anexo 2.1. Se recomienda leer de manera paralela el código junto con esta sección para una mejor comprensión de su funcionamiento.

Para evitar problemas de interpretación en el código de Matlab, se denominó a la solución actual como $\boldsymbol{X}$ y su valor en la Función de Rastrigin como F (ambas en mayúsculas). Por su parte, la candidata a solución y su respectivo valor en la función fueron denominados como $\boldsymbol{x}$ y f, respectivamente. Como en el ejemplo anterior, se utilizan los comandos *clear* y *clc* para eliminar las variables existentes y limpiar la pantalla del programa, respectivamente. En las líneas 6 a 9 del código se definen las características del espacio de soluciones y de la Función de Rastrigin, siendo estas las siguientes:

- Número de dimensiones del problema (D). Para modificar la complejidad del problema, este valor se variará entre 10 y 50 dimensiones.
- Valor del mínimo global de la función (f_{bias}).
- Vector que contiene los posibles valores de las variables de decisión del problema ($\boldsymbol{V}$). A través de este vector, también se establece el intervalo en el cual existe la Función de Rastrigin.

- Posición del mínimo global (***o***). Este vector se define en base al intervalo definido para la función y se genera de manera aleatoria, con la función randi(), cada vez que se inicia el proceso de búsqueda.

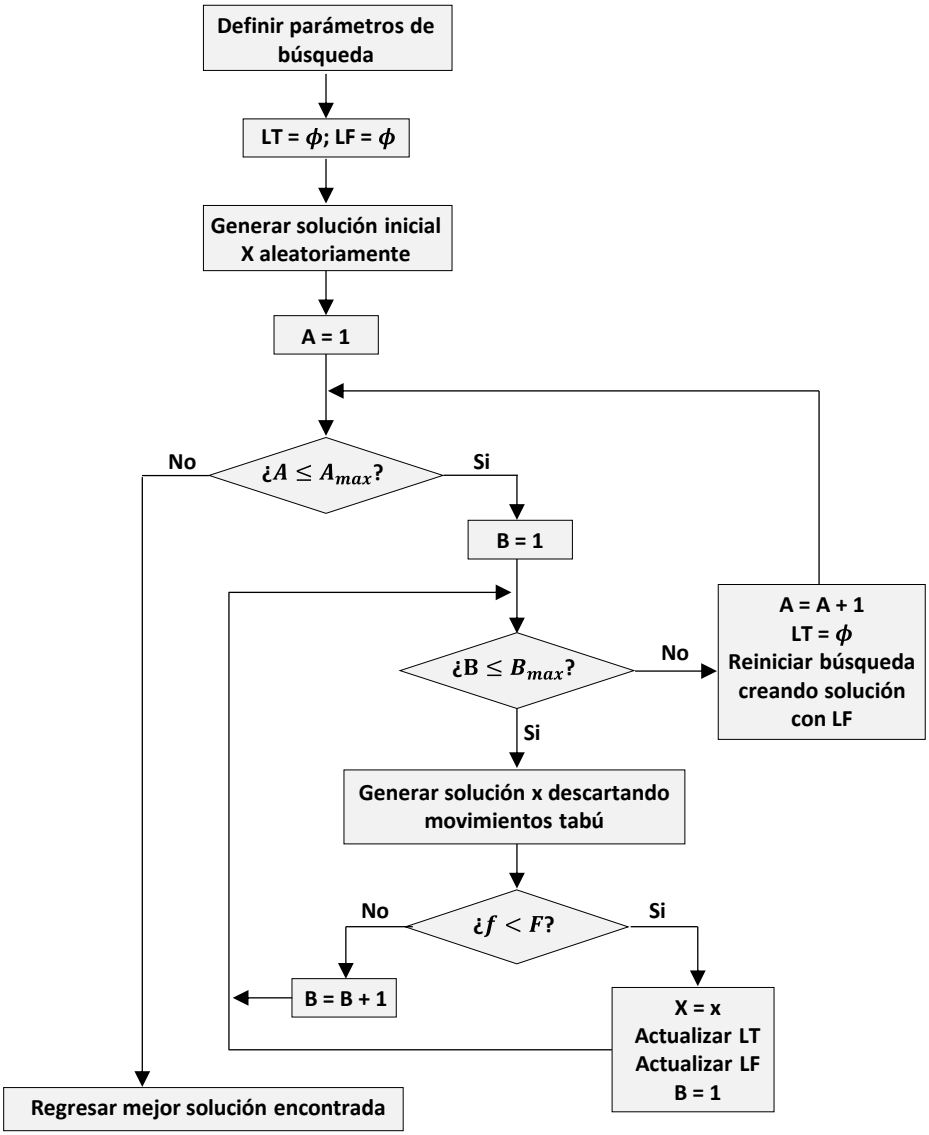


Figura 4.6 Diagrama de flujo de Búsqueda Tabú usado en este ejemplo

Entre las líneas 12 y 16 del código se especifican los parámetros de búsqueda que empleará el algoritmo. Tal como se ha mencionado a lo largo de este libro, los valores de los hiperparámetros se definen a través de un proceso iterativo de prueba y error, no existiendo un método analítico para determinar aquella combinación de valores que brinde el mejor desempeño del algoritmo. Los hiperparámetros de considerados en este caso son:

- Reinicios máximos por medio de la memoria larga (A_{max}).
- Número máximo de soluciones revisadas por vecindario (B_{max}).
- Porcentaje máximo de las variables de la solución actual $\boldsymbol{X}$ modificables para crear la candidata a solución $\boldsymbol{x}$ (pV).
- Número de iteraciones durante las cuales un tabú permanece activo ($Tenure$).
- Porcentaje de las variables de decisión que son almacenadas en la Lista Tabú (LT) en cada iteración (p).

Dos matrices y un vector son creados en las líneas 19 a 21. El vector *MS* es utilizado para almacenar la mejor solución encontrada por el algoritmo; por su parte, las matrices *TL* y *LF* sirven para almacenar la Lista Tabú y la Lista de Frecuencias, respectivamente. La solución inicial $\boldsymbol{X}$ es generada de manera aleatoria entre las líneas 24 y 25. Debido a que todas las soluciones son factibles, para crear la solución inicial basta con tomar un punto aleatorio del espacio de configuraciones. Esto se realiza por medio de la función *randi()*, misma que genera un vector de números enteros aleatorios que es utilizado posteriormente para llamar valores aleatorios del vector $\boldsymbol{V}$.

Para facilitar la evaluación de la Función de Rastrigin en cualquier punto del espacio de configuraciones, se creó un función denominada como *ValorTS()*, la cual toma como argumentos el vector de la solución $\boldsymbol{X}$, el vector $\boldsymbol{o}$ y el valor del mínimo de la función (f_{bias}). La función que evalúa la Función de Rastrigin se encuentra en el archivo llamado "ValorTS.m" el cual se presenta en el Anexo 2.2.

Una vez definida la solución inicial del algoritmo, se comienza con el proceso de optimización por medio de un ciclo *for* y *while*. El ciclo *while* abarca desde la línea 31 hasta la 57, y está controlado por la variable *B*, misma que cuenta el número de soluciones revisadas por el algoritmo sin que se presente una mejora de calidad. Si el contador *B* alcanza el valor límite de soluciones permitidas (B_{max}) sin que se encuentre una de mejor calidad, se considera que el algoritmo ha quedado atrapado en un óptimo local. Cuando esto ocurre, se reinicia el proceso de búsqueda en otro punto del espacio de configuraciones. Dicho proceso está controlado por el ciclo *for* que se desarrolla de la línea 30 hasta la 73. El número máximo de reinicios permitidos por el algoritmo está definido por la variable A_{max}.

En el ciclo *while*, la nueva candidata a solución es generada a partir de la solución actual variando de manera discreta hasta un porcentaje pV de sus variables de decisión, esto se realiza en la función denominada como *MovimientoTS()*. Dicha función regresa la nueva candidata a solución $\boldsymbol{x}$ y el registro de las variables de decisión que fueron modificadas ($\boldsymbol{T}$). Como argumentos, la función toma la dimensión del problema (*D*), el porcentaje máximo de variables a modificar (*pV*), el vector que contiene los posibles valores de las variables de decisión ($\boldsymbol{V}$), la solución actual ($\boldsymbol{X}$), la tenencia, el porcentaje de los cambios que se registran (*p*) y la Lista Tabú (*LT*). El archivo que contiene el código de la función se denomina como "MovimientosTS.m" y es mostrado en el Anexo 2.2 de este libro.

El cambio de una solución a otra se denomina como movimiento. Es importante indicar que, en este ejemplo en particular, no se permite alcanzar cualquier solución del espacio de soluciones por medio de un único movimiento, esto debido a que las alteraciones efectuadas a la solución actual solo realizan incrementos de ± 0.1 a los valores de las variables de decisión seleccionadas para ser modificadas. Tanto las variables de decisión a modificar como la dirección del cambio se definen de manera

aleatoria, teniendo cada variable y dirección de cambio la misma probabilidad de ocurrencia. El moverse por el espacio de soluciones empleando pequeños incrementos o decrementos de valores implica que las soluciones alcanzables por un único movimiento se encuentran en la cercanía de la solución actual. Si por el contrario se variase un alto número de variables de decisión de la solución actual, existirían altas posibilidades de destruir los bloques de variables que brindan una alta calidad a la solución, entorpeciéndose así el proceso de búsqueda.

Al definir los movimientos por el espacio de soluciones, la función "MovimientosTS.m" también se encarga de restringir aquellos cambios que sean considerados como tabús por el algoritmo. Como se mencionó antes, existen dos posibles restricciones al considerar un atributo como tabú: prohibir desechar un atributo recientemente añadido o prohibir añadir nuevamente un atributo recientemente desechado. Por simplificación, en este caso se optó por la primera de las dos opciones.

Una vez que se ha creado la solución $\boldsymbol{x}$, se utiliza la función *ValorTS()* para evaluar la Función de Rastrigin, almacenándose dicho valor en la variable f (línea 33). Si la nueva solución presenta una mayor calidad que la solución actual ($f < F$), en las líneas 36 y 37 se procede a reemplazar la solución actual ($\boldsymbol{X} = \boldsymbol{x}$) y se registra el valor de la Función de Rastrigin en la lista *Fs*, respectivamente. Posteriormente, se almacena la información de la nueva solución en la Lista Tabú (LT) y en la Lista de Memoria Larga o de frecuencias, llamada en el código como LF. El almacenamiento de la información en las estructuras de memoria se realiza en entre las líneas 40 a 53.

En este ejemplo se utilizan dos estructuras de memoria: una Lista Tabú de Memoria Corta (LT) de tipo atributiva, la cual se encarga de dirigir la manera en cómo el algoritmo de BL explora el espacio de soluciones; y una Lista de Memoria Larga (LF), la cual es explicita y registra las frecuencias de aparición de todos los atributos de las soluciones encontradas. La lista LF, actualizada en cada paso en la línea 52, es empleada para permitir al algoritmo escapar de los óptimos locales que encuentre por medio de la información que se ha recabado a lo largo del proceso de exploración. Los registros de la Lista de Memoria Larga pueden ser utilizados de dos formas: 1) creando una solución por medio de la agrupación de bloques de variables con valores que brinden una alta calidad a la solución (Glover, 1989a), o 2) generando una nueva solución en una zona del espacio de búsqueda que por el momento no haya sido explorada (Laguna, 2018). Debido a que ambas opciones presentan enfoques distintos pero complementarios, en este ejemplo se buscó utilizar ambos para escapar de los óptimos locales, con lo cual se estaría empleado una estrategia oscilatoria.

Para tomar en cuenta tanto la frecuencia con la que aparecen algunos atributos como su aporte en la calidad de la solución, se decidió emplear una Lista de Memoria Larga explícita que se actualice en cada paso i y que, para cada variable que se presente en la solución actual, se asigne una cantidad k_i que depende del valor de la función objetivo (F_i) de la solución. Dicha cantidad k_i se suma a la existente hasta ese punto del proceso y se determina por medio de la siguiente expresión:

$$k_i = 1.05^{F_i} \tag{4.5}$$

El colocar el valor de la función objetivo como exponente de un número cercano a 1 es equivalente a incrementar o disminuir la "frecuencia" de aparición del atributo, dependiendo de si este forma parte de una solución con un valor de función objetivo alto o bajo, respectivamente.

Cuando el algoritmo queda atrapado en un óptimo local, se genera una nueva solución escogiendo atributos de manera probabilista, de tal manera que los atributos que presenten en ese momento una baja "frecuencia" de aparición tendrán mayores probabilidades de ser seleccionados para formar parte de la nueva solución de reinicio. Esto ocurre entre las líneas 59 a 71 del código. Esta estrategia permite que, en las primeras iteraciones, el algoritmo reinicie la búsqueda con soluciones ubicadas en zonas inexploradas del espacio de búsqueda, pasando a reinicios con soluciones de mayor calidad a medida que se identifiquen atributos que minimicen la función objetivo. Como en el ejemplo anterior, los atributos escogidos para genera la nueva solución son escogidos por medio de una estructura tipo ruleta. Es importante indicar que este enfoque es susceptible de presentar mejoras y el usuario tiene total libertad para emplear cualquier otro criterio que considere conveniente para almacenar la información en la Lista de Memoria Larga. Por último, el proceso de búsqueda se dará como finalizado una vez que el algoritmo haya realizado un número A_{max} de reinicios.

4.9. Tamaño del espacio de configuraciones

En el problema existen D variables de decisión agrupadas en el vector $\boldsymbol{x} = \{x_1, x_2, \dots, x_D\}$; teniéndose que $x_i \in [-5{,}5]$ con incrementos de 0.1, por lo que existen 101 valores para cada variable de decisión. Tomando en cuenta lo anterior, la función que nos permite determinar el tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) para los diferentes valores de D considerados es:

$$|\boldsymbol{S}(D)| = 101^D \text{ posibles configuraciones} \qquad (4.6)$$

En la Figura 4.7 se muestra graficada la curva que representa el valor del $|\boldsymbol{S}|$ para diferentes valores de D. Nótese que ambos ejes se encuentran en escala logarítmica. Como se puede observar, el espacio de configuraciones del problema crece muy rápidamente y, aún para pequeños valores de D, el valor de $|\boldsymbol{S}|$ resulta ser considerable. A manera de ejemplo, si se considera un valor de dimensiones $D = 5$, existen $1x10^{10}$ posibles configuraciones; por otra parte, si el valor incrementa a $D = 50$, $|\boldsymbol{S}|$ presenta un valor de $1.64x10^{100}$ posibles configuraciones.

Un aspecto importante a recalcar sobre los algoritmos de búsqueda es que el desempeño de estos es dependiente del número de configuraciones posibles en el problema. Para problemas con una dimensionalidad baja, cualquier algoritmo de búsqueda debería ser capaz de encontrar soluciones de calidad alta con relativa facilidad. Esto se debe a que, durante el proceso de optimización, el algoritmo prácticamente estaría revisando todas o un gran porcentaje de las posibles soluciones al problema. Con lo cual la estrategia de búsqueda pasaría a tener una importancia baja. Por otra parte, al incrementarse tanto la dimensión del

problema como las posibles configuraciones, las soluciones de alta calidad se volverán cada vez más escasas y difíciles de encontrar, lo cual repercute negativamente en el desempeño del algoritmo de búsqueda. Es en estas situaciones, donde solo es posible revisar un número reducido de configuraciones, que los algoritmos metaheurísticos cobran relevancia.

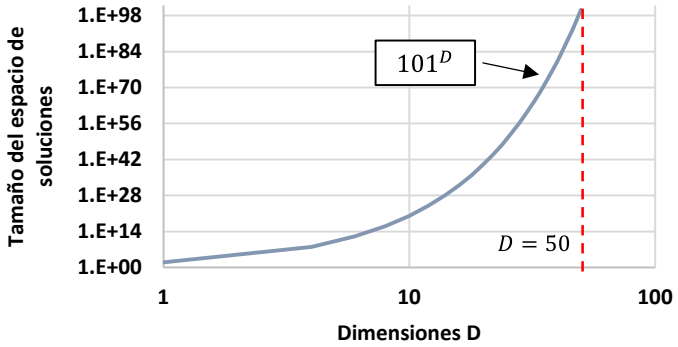


Figura 4.7 Crecimiento del espacio de solución en función del número de variables

4.10. Resultados y discusión

La función de Rastrigin fue minimizada para los valores de $D = 10, 20, 30$ y 50 variables. Para demostrar los beneficios que brinda el utilizar las estructuras de memoria de BT para dirigir el proceso de búsqueda, en todos los casos el mínimo global de la función fue buscado de dos formas: 1) utilizando únicamente BL, y 2) por medio de BT en conjunto con BL. Para cada valor de D, los dos algoritmos fueron ejecutados un total de cinco veces cada uno. Los resultados obtenidos fueron comparados a través de su media y desviación estándar. En ambos algoritmos se permitieron 500 reinicios (A_{max}) y la generación de hasta 5,000 posibles soluciones por vecindario. En el caso de BT, se registraban como tabús hasta el 50% de los atributos recién añadidos a la solución actual, empleándose una tenencia de 2, 3, 5 y 10 iteraciones para los casos de $D = 10, 20, 30$ y 50 variables, respectivamente.

En la Tabla 3.1 se muestran los valores mínimos encontrados de la función objetivo (F) por medio del algoritmo de BL exclusivamente, para todos los casos de D estudiados. De manera adicional se presentan los valores de la media y de la desviación estándar. Por su parte, en la Tabla 3.2 se realiza lo propio, pero con los resultados obtenidos por medio de BT en conjunto con BL

Tabla 3.1 Resultados obtenidos por Búsqueda Local

D	**E1**	**E2**	**E3**	**E4**	**E5**	**Media**	**Desviación estándar**
10	-299.00	-295.00	-311.00	-290.00	-302.00	-299.40	7.89
20	-175.00	-162.28	-183.05	-242.00	-213.00	-195.07	32.20
30	-105.00	-159.00	-103.00	-117.00	-99.00	-116.60	24.63
50	98.00	146.00	149.00	216.58	93.38	140.59	49.79

Tabla 3.2 Resultados obtenidos por Búsqueda Tabú

D	E1	E2	E3	E4	E5	Media	Desviación estándar
10	-306.56	-313.00	-302.88	-313.48	-313.00	-309.78	4.81
20	-252.70	-246.78	-277.08	-263.28	-258.08	-259.58	11.55
30	-176.93	-184.04	-173.18	-205.12	-180.16	-183.89	12.53
50	12.43	4.74	59.00	12.91	63.24	30.46	28.21

Recordando que el mínimo global de la función presenta un valor de -330, se observa que el algoritmo de BL obtuvo valores medios mayores que BT para todos los casos de D considerados. Por lo tanto, el desempeño de BL resultó ser peor que el de BT. En el caso de la Función de Rastrigin construida en 10 dimensiones, ambos algoritmos localizaron mínimos con valores similares; sin embargo, para valores mayores de D, la diferencia entre el desempeño de ambos algoritmos se va haciendo cada vez más pronunciada. Esta disminución en el desempeño a medida que incrementa el espacio de configuraciones se debe a que el conjunto de soluciones de calidad alta no crece al mismo ritmo que el conjunto de posibles soluciones. Por lo tanto, para problemas de gran dimensionalidad, se incrementa la probabilidad de que los algoritmos de búsqueda queden atrapados en óptimos locales de calidad baja.

Cabe destacar que ni BL ni BT fueron capaces de encontrar el mínimo global de la función, aun cuando esta se construyó por medio de 10 variables de decisión. Para poner en perspectiva la forma en que ambos algoritmos exploraron el espacio de soluciones, en la Figura 4.8 se muestran graficados ejemplos de las trayectorias que siguieron BL y BT, esto para D =10 y 50 dimensiones.

Las gráficas de las trayectorias seguidas por los algoritmos de BL y BT, para valores de $D = 10$ y 50, muestran la influencia que tuvo la lista de frecuencias en el proceso de búsqueda. En las primeras iteraciones, BT reinició la búsqueda en zonas que presentaban una baja calidad, motivo por el cual la trayectoria para $D = 50$ presenta valores altos de F; sin embargo, a medida que se desarrolla el proceso de búsqueda, la información obtenida le permitió al algoritmo concentrar su búsqueda en zonas más prometedoras del espacio de configuraciones. Fue esta mayor calidad de las soluciones iniciales en etapas avanzadas de la búsqueda lo que le permitió a BT encontrar mejores óptimos locales que aquellos localizados por BL.

Se observa además que la influencia de la Lista de Frecuencias se incrementó con el número de dimensiones del problema. Para un valor de $D = 10$ dimensiones, las soluciones generadas por medio de la Lista de Frecuencias presentaron una calidad similar. Por su parte, para un valor de $D = 50$ dimensiones, el reinicio del proceso de búsqueda generaba soluciones con valores de la función objetivo ($F(\boldsymbol{X})$) entre 1,400 y 500.

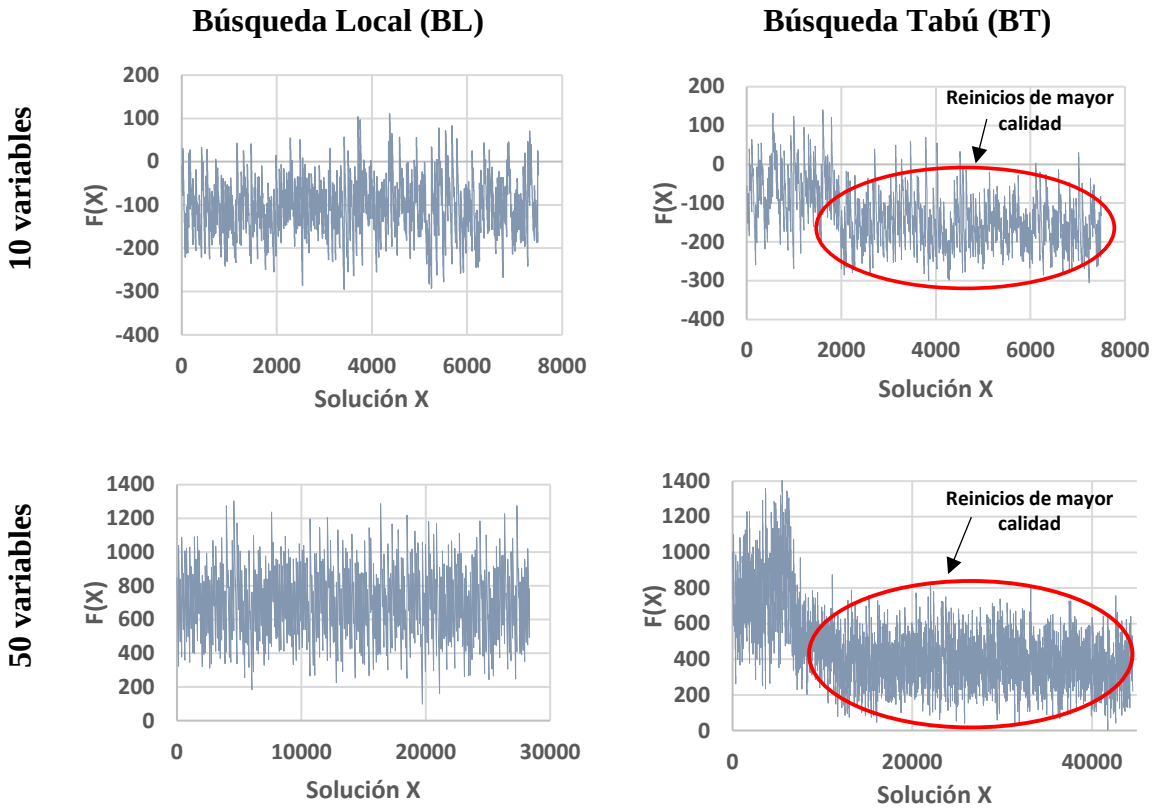


Figura 4.8 Trayectorias seguidas por BL y BT para el caso de 10 y 50 dimensiones

4.11. Conclusiones

En este capítulo se observó que las estructuras de memoria ayudan a mejorar el desempeño de los algoritmos de BL por medio del ejercicio de minimización de la Función de Rastrigin construida en espacios D – dimensionales. Una de las principales características de la Función de Rastrigin es su multimodalidad, es decir, que presenta una gran cantidad de óptimos locales, lo cual dificulta la búsqueda de su mínimo global. Para valores bajos de D, el algoritmo de BL fue capaz de localizar puntos de la función cercanos al mínimo global. Sin embargo, fue necesario reiniciar un gran número de veces el algoritmo para encontrar esos puntos. Por su parte, al utilizar las estructuras de memoria de BT, el algoritmo de búsqueda alcanzó puntos de relativamente mayor calidad a aquellos encontrados al emplear BL por sí sola, manteniéndose esta tendencia para todos los valores de D considerados. Finalmente, se señala que tanto BT como BL fueron incapaces de localizar el mínimo global para valores de $D \geq 10$ dimensiones, observándose una disminución de su desempeño a medida que se incrementaba el valor de D.

5. Optimización topológica de una armadura

En este capítulo se realiza uno de los ejemplos más complejos de la aplicación de los algoritmos metaheurísticos en la ingeniería estructural: la optimización del diseño de una armadura. El algoritmo de optimización empleado es Recocido Simulado (RS) el cual, de manera similar a Búsqueda Tabú (BT), se apoya en un algoritmo de Búsqueda Local (BL) para desplazarse por el espacio de configuraciones. En este problema se toman como variables de decisión no solo las áreas transversales de las barras que componen la armadura sino también los nodos que las unen y la altura del cordón superior de la estructura, haciendo de este ejemplo un problema de optimización topológica.

5.1. Recocido Simulado

Recocido Simulado (en inglés *Simulated Annealing*, *SA*) es un algoritmo de búsqueda basado en una única solución desarrollado por Kirkpatrick *et al*. (1983). La innovación de RS recae en que su formulación permite aceptar soluciones de peor calidad o degradadas durante el proceso de optimización, siendo esto dependiente de una función de probabilidad. RS hace una analogía con el proceso metalúrgico de la obtención de cristales a partir de masas fundidas a altas temperaturas (Yepes *et al*., 2008). En el proceso análogo, si una masa de metal fundido es enfriada de manera súbita, los átomos que la conforman no tendrán tiempo suficiente para crear un patrón organizado, dando lugar a la aparición de tensiones internas que pueden afectar de manera negativa el desempeño mecánico del material resultante. Para evitar esto, es necesario aplicar un programa de enfriamiento controlado que permita llegar a estados ordenados de baja energía. A ese proceso se le conoce en la industria metalúrgica como recocido (o *annealing*, en inglés).

Para emplear los conceptos del recocido en la optimización de problemas, Kirkpatrick empleó la **distribución de Boltzmann**, la cual es perteneciente al campo de la mecánica estadística:

$$P(E_n) = \frac{e^{-\Delta E / k_b T}}{Z} \tag{5.1}$$

donde ΔE es el incremento de energía, T es la **temperatura**, k_b es la constante de Boltzmann, $P(E_n)$ es la probabilidad de encontrar una partícula aleatoria en un estado de energía E_n y Z es la función de partición. Traduciendo esto a un problema de optimización, k_b y Z se vuelven innecesarias, por lo tanto, se consideran con valor igual a la unidad, se conserva el concepto de temperatura T y el incremento de energía ΔE se convierte en la **degradación de la función objetivo**:

$$\Delta E = f(\boldsymbol{x}') - f(\boldsymbol{x}) > 0 \tag{5.2}$$

siendo $f(\boldsymbol{x})$ y $f(\boldsymbol{x}')$ los valores de la función objetivo para la solución actual ($\boldsymbol{x}$) y la solución de menor calidad encontrada ($\boldsymbol{x}'$), respectivamente. Esto para un problema de minimización. Con base en lo anterior, cualquier solución que encuentre el algoritmo es aceptada en base a la siguiente distribución de probabilidad:

$$p(\boldsymbol{x} \rightarrow \boldsymbol{x}') = \min\left(e^{-\left[\frac{f(\boldsymbol{x}') - f(\boldsymbol{x})}{T}\right]},\ 1 \right) \tag{5.3}$$

Obsérvese que en la ecuación 5.3 cualquier solución degradada ($f(\boldsymbol{x}') > f(\boldsymbol{x})$ para un problema de minimización) generará que el evaluar el exponente dé como resultado un valor menor a 1. Por el contrario, si la nueva solución es de mayor calidad, el exponente será positivo y la probabilidad de aceptar la nueva solución será igual a 1, con lo cual RS siempre aceptará soluciones que presenten una mejora en la calidad.

Además de conservar la temperatura como un elemento de su metaheurística, Kirkpatrick también introdujo el concepto de **programa de enfriamiento**, el cual define la forma en que la temperatura va disminuyendo a medida que avanza el proceso de optimización. Debido a que la temperatura del proceso no se mantiene constante, la distribución de probabilidad que define al criterio de aceptación varía durante el proceso de optimización. Este proceso de búsqueda puede ser visto como una sucesión de estados en la cual, el estado sucesor es escogido dependiendo únicamente del estado actual ($\boldsymbol{x} \rightarrow \boldsymbol{x}'$), por lo tanto, la trayectoria que sigue RS resulta ser una **Cadena de Márkov** en la que se revisan un número m de posibles soluciones para cada valor de la temperatura (Blum & Roli, 2003). En la Figura 5.1 se muestra de manera esquemática la probabilidad de aceptación en función del incremento de la función objetivo y considerando diferentes temperaturas. De la figura se observa que la probabilidad de aceptación de una solución degradada es inversamente proporcional al incremento de la función objetivo y disminuye con la temperatura.

Dos conceptos de gran importancia en RS son la temperatura y el programa de enfriamiento. Es a través de la **temperatura** que RS controla su enfoque de exploración. Valores altos de este parámetro provocan que el algoritmo funcione como una búsqueda aleatoria (Winker & Maringer, 2007), mientras que los valores bajos hacen que la exploración se concentre en zonas reducidas del espacio de configuraciones

(Sörensen, 2013). Asimismo, es posible establecer un criterio de parada del algoritmo basado en la temperatura del proceso. Este criterio es conocido como **criterio de congelación** y establece que el proceso de optimización finaliza una vez que la temperatura disminuye a cierto porcentaje de su valor inicial, siendo usual considerar valores del 1% o 2% de esta (Medina, 2001; Yepes *et al*., 2008).

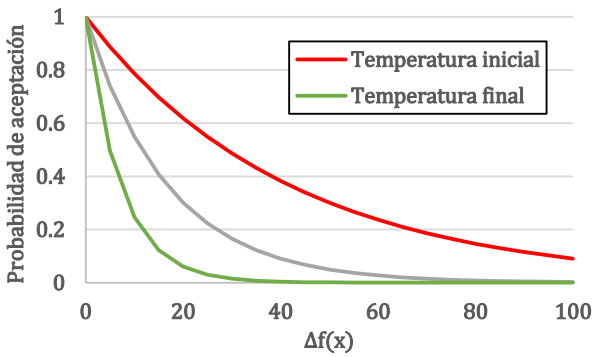


Figura 5.1 Ejemplo de la distribución de Boltzmann utilizada en Recocido Simulado para aceptar soluciones de menor calidad

Por otra parte, es conocido que RS es capaz de converger al óptimo global cuando es ejecutado por un tiempo que tiende al infinito y cuando se emplean ciertos **programas de enfriamiento** (Blum & Roli, 2003; Fleischer, 1995; Granville *et al*., 1994). Lamentablemente, este resultado es de poco interés práctico debido a que no es viable el uso de RS por tiempos extremadamente largos. Por tales motivos, es usual emplear programas de enfriamiento más sencillos, como aquellos de tipo geométrico, en RS (Blum & Roli, 2003). En un decrecimiento geométrico, la temperatura del estado $i + 1$ es proporcional a la temperatura del estado i por medio de un coeficiente de enfriamiento $\alpha < 1$:

$$T_{i+1} = \alpha \cdot T_i \tag{5.4}$$

Tomando en cuenta todo lo expuesto anteriormente, en la Figura 5.2 se presenta el diagrama de flujo genérico de RS. Se puede observar que el proceso de búsqueda parte de una solución factible, la cual puede ser creada por cualquier método que el usuario considere conveniente. Con base en las características de esta solución se definen las temperaturas inicial y final del proceso. Para cada valor de temperatura, se revisa un número determinado de posibles soluciones, pudiendo ser estas creadas por cualquier algoritmo de búsqueda que el usuario desee. Al encontrarse una nueva solución $\boldsymbol{x}'$ que sea de mayor calidad a la solución $\boldsymbol{x}$ o que cumpla con el criterio de aceptación, la primera reemplaza a la segunda. Para revisar el criterio de aceptación se genera un número aleatorio $R < 1$ que sigue una distribución uniforme ($R \sim \text{Unif}\,(0,1)$). Una vez que se ha alcanzado la longitud de la Cadena de Márkov establecida por el usuario, la temperatura se actualiza acorde al programa de enfriamiento y el proceso se repite hasta alcanzar la temperatura final.

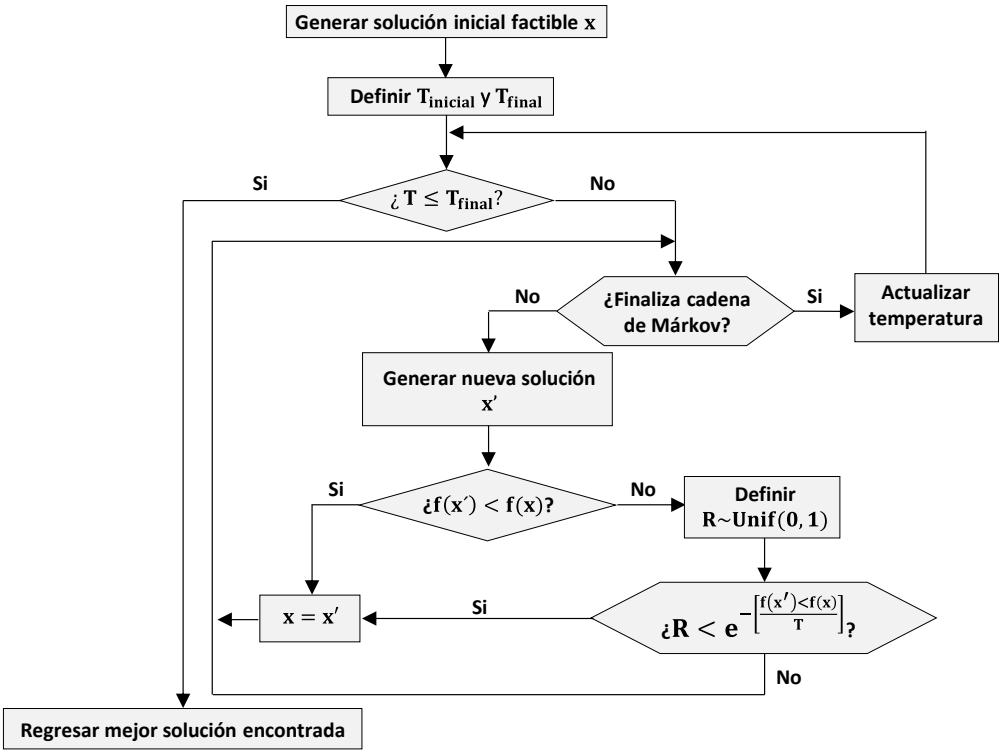


Figura 5.2 Diagrama de flujo de Recocido Simulado

5.2. Optimización de armaduras

Las armaduras son elementos estructurales compuestos por barras que trabajan axialmente y que se suponen unidas de manera articulada a los nodos de la estructura. Debido a su sencillez y facilidad de cálculo son ampliamente utilizadas en problemas de ingeniería estructural. En lo que respecta a la optimización de estructuras, el minimizar el área transversal de los elementos de una armadura es un problema de referencia utilizado muchas veces para demostrar la eficacia de un algoritmo de optimización (D. Goldberg, 1989; Kaveh & Mahdavi, 2014; Kaveh & Zaerreza, 2020; Mirjalili & Lewis, 2016; Mohammadi-Balani *et al*., 2021; Saremi *et al*., 2017). Una versión más compleja del mismo problema considera como variable la geometría de la armadura de tal manera que el objetivo del proceso de optimización es encontrar el diseño con una topología estable y que aproveche al máximo los materiales empleados. Dicho problema de optimización ha sido abordado con anterioridad por Hajela *et al*. (1994), Hasancebi *et al*. (2002), Renkavieski *et al*. (2021), entre otros. De manera general, se puede decir que los problemas de optimización que incluyen de manera conjunta tanto las dimensiones de las barras como la topológica de la estructura resultan ser especialmente desafiantes, siendo necesario utilizar operadores complejos para alcanzar soluciones satisfactorias. En el presente ejemplo se realiza la optimización topológica de una armadura simplemente apoyada con una luz de

36 metros, empleándose RS como algoritmo de optimización. A pesar de que el problema aquí tratado es más sencillo que el presentado por Hasancebi *et al*. (2002), se contrastarán los resultados para identificar las singularidades de este tipo de problemas de optimización.

5.3. Tipo de optimización y función objetivo

En este ejemplo se realiza una optimización de tipo económica. Al tratarse las armaduras de estructuras de acero, la función objetivo a minimizar considerará únicamente el peso total de la estructura (W). Esto es una práctica común en la optimización de estructuras de acero debido a que su costo se ve fuertemente influenciado por la cantidad de material empleado (Balling, 1991; C. Coello & Christiansen, 2000; Degertekin *et al*., 2008). Por lo tanto, la función objetivo es la sumatoria del peso w de todas las k barras que componen a la propuesta de solución:

$$f(\boldsymbol{x}) = W = \sum_{i=1}^{k} w_i \tag{5.5}$$

5.4. Variables de decisión y parámetros del problema

Los diseños de la armadura están delimitados por un arreglo de tipo cuadrícula, donde las posiciones de los nodos se consideran como parámetros del problema. Sin embargo, esto no implica que todos los nodos de la cuadrícula deban de ser utilizados en las propuestas de solución. Debido a que las cargas se consideran aplicadas en el cordón inferior de la armadura, se impondrá la existencia de este en todos los diseños, adicionalmente, la magnitud de las cargas aplicadas será de $P = 100$ kN en el sentido vertical, manteniéndose este valor como constante. Con la finalidad de reducir el tamaño del espacio de configuraciones, se impondrá la condición de simetría con respecto al eje vertical de la estructura. Un esquema de la cuadrícula empleada se muestra en la Figura 5.3.

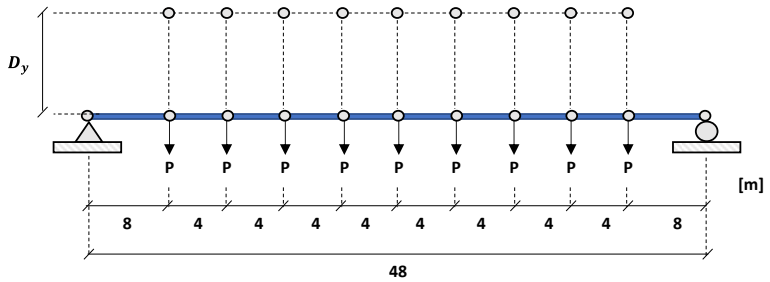


Figura 5.3 Posición de los posibles nodos de la armadura

Como variables de decisión se consideraron la altura del cordón superior (D_y), el área transversal de las barras y el número de barras que componen a la armadura. El valor D_y existe para valores entre 3 y 5 metros, con incrementos discretos de 0.5 metros. En lo que respecta al área transversal de las barras, estas pueden variar entre 5 y 50 cm^2, con incrementos discretos de 5 cm^2. Para simplificar el ejemplo, no se consideraron perfiles comerciales en la definición del área transversal de las barras. Debido a que es una optimización de tipo topológica, el número de barras y los nodos que estas unen varía durante el proceso de optimización. El esquema de conexiones posibles se muestra en la Figura 5.4.

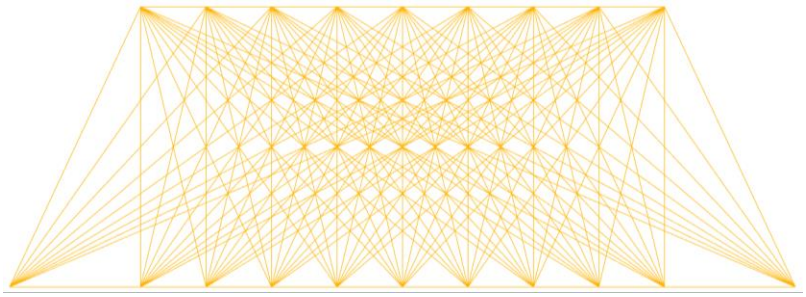


Figura 5.4 Esquema de posibles conexiones de la armadura

5.5. Codificación de los diseños

En este ejemplo las soluciones son codificadas por matrices de tres dimensiones, es decir, $[a_{ijk}]$ con $i, j \in \{1,2,3, \dots, m\}$ y $k \in \{1,2\}$, donde m representa los posibles nodos de la armadura en direcciones horizontales y verticales.

$$\left[a_{ijk}\right] = \begin{bmatrix} x_{11k} & x_{12k} & x_{13k} & & x_{1jk} \\ x_{21k} & x_{22k} & x_{23k} & \cdots & x_{2jk} \\ x_{31k} & x_{32k} & x_{33k} & & x_{3jk} \\ & \vdots & & \ddots & \\ x_{i1k} & x_{i2k} & x_{i3k} & & x_{ijk} \end{bmatrix} \tag{5.6}$$

En la representación matricial, en los elementos con valor $k = 1$ denotan si existe una barra que conecte los nodos i y j, esto por medio de un sistema binario donde 0 representa una ausencia de conexión mientras que 1 indica que existe una barra que vincule ambos nodos. Por otra parte, en los elementos con valores de $k = 2$ se almacenan las áreas trasversales de las barras de conexiones activas, pudiendo estas presentar valores de entre 5 a 50 cm^2 con incrementos de 5 cm^2.

5.6. Restricciones del problema

Para que los diseños de las armaduras producidos por el algoritmo sean considerados como factibles deben de cumplir con los requisitos de estabilidad y resistencia. El requisito de resistencia hace referencia a que ninguna de las barras falle por la acción de los esfuerzos axiales a los que son sometidas. Las fuerzas axiales (N) aplicadas en cada barra son proporcionales a los desplazamientos de sus extremos, tal como indica la siguiente ecuación.

$$N = \frac{AE}{L}\{-Cos\theta \;\; -Sen\theta \;\; Cos\theta \;\; Sen\theta\} \cdot \begin{Bmatrix} D_{ix} \\ D_{iy} \\ D_{fx} \\ D_{fy} \end{Bmatrix} \tag{5.7}$$

donde A, E, L y θ son el área transversal, el módulo de elasticidad del material, la longitud de la barra analizada y el ángulo que esta forma con la horizontal, respectivamente. Por su parte, $D_{ix,y}$ y $D_{fx,y}$ son los desplazamientos de los nodos, inicial y final, de la barra en las direcciones X y Y, tal como se muestra en la Figura 5.5.

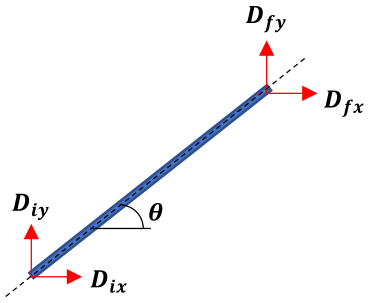


Figura 5.5 Esquema de los desplazamientos y ángulo de la barra

Los desplazamientos en los nodos extremos de cada barra se obtienen por medio del método de rigidez, el cual permite relacionar las fuerzas aplicadas en la estructura, la rigidez aportada por sus elementos y los desplazamientos de sus grados de libertad en arreglos matriciales, permitiendo resolver tanto estructuras estáticamente determinadas como indeterminadas. Matemáticamente:

$$\{\boldsymbol{D}\} = [\boldsymbol{K_G}]^{-1} \cdot \{\boldsymbol{F}\} \tag{5.8}$$

En la ecuación (5.8), la matriz $[\boldsymbol{K_G}]$ es la matriz de rigidez global de la estructura y se construye por medio del ensamble de las matrices de rigidez $[\boldsymbol{K_b}]$ de cada barra i que la componen, obteniéndose estas de la siguiente manera:

$$[\boldsymbol{K_b}]_i = \frac{E_i A_i}{L_i}\begin{bmatrix} Cos^2\theta & Cos\theta Sen\theta & -Cos^2\theta & -Cos\theta Sen\theta \\ Cos\theta Sen\theta & Sen^2\theta & -Cos\theta Sen\theta & -Sen^2\theta \\ -Cos^2\theta & -Cos\theta Sen\theta & Cos^2\theta & Cos\theta Sen\theta \\ -Cos\theta Sen\theta & -Sen^2\theta & Cos\theta Sen\theta & Sen^2\theta \end{bmatrix} \quad (5.9)$$

El esfuerzo axial máximo ($\sigma_{máx}$) que una barra puede tomar depende de su área transversal, del esfuerzo de fluencia del material y del tipo de acción aplicada. En este ejemplo se consideró que todas las barras presentan un esfuerzo de fluencia $f_y = 275\ MPa$. En busca de una simplificación del problema, la disminución de la resistencia de las barras por efectos de pandeo a compresión se consideró por medio de un coeficiente $\chi = 0.7$ en todos los casos, en base a lo anterior $\sigma_{máx}$ se determinó como:

$$\sigma_{\max} = \begin{cases} A \cdot f_y & (Tensión) \\ \chi \cdot A \cdot f_y & (Compresión) \end{cases} \quad (5.10)$$

Por último, el requisito de estabilidad se refiere a que los diseños candidatos a solución no sean estructuras hipostáticas, ni que produzcan matrices de rigidez singulares. Esto se evaluó verificando que la matriz de rigidez global de la estructura fuese definida positiva y que su determinante fuese distinto de cero.

5.7. Descripción del código de Recocido Simulado

En la Figura 5.6 se muestra el diagrama de flujo del algoritmo de RS empleado en este ejemplo. El código fue ejecutado en el programa Matlab y se puede revisar de manera completa en el Anexo 3.1 de este libro. El código comienza eliminando las variables existentes en el programa y limpiando la pantalla, esto con los comandos *clc* y *clear*, respectivamente. Posteriormente, entre las líneas 6 a 10, se definen los parámetros que dirigirán la búsqueda durante el proceso de optimización, dependiendo estos parámetros de las características del algoritmo. Para este caso en particular, se utiliza un programa de enfriamiento geométrico y, siguiendo las recomendaciones de Medina *et al*. (2001), la temperatura inicial del proceso se considera dependiente del costo de la solución inicial. Adicionalmente, para crear nuevas candidatas a solución se utiliza BL. Por tales características, para utilizar RS es necesario definir la relación entre la temperatura inicial y el costo de la solución inicial (T_0), el factor de enfriamiento (α), la temperatura final, la longitud de las cadenas de Márkov (LCM) y el porcentaje máximo de variables de decisión modificables por BL. Como ocurre con los algoritmos de búsqueda anteriormente presentados, los valores de los hiperparámetros deben ser definidos en base a la experiencia y calibrados con algunas pruebas de desempeño. En este ejemplo se encontró aceptable

el considerar que la temperatura inicial fuera 1/4 el valor del costo de la solución inicial, así como una temperatura final menor o igual al 1% de la temperatura inicial. Por su parte, cada Cadena de Márkov generaría 1,000 soluciones para cada valor de temperatura ($LCM = 1{,}000$).

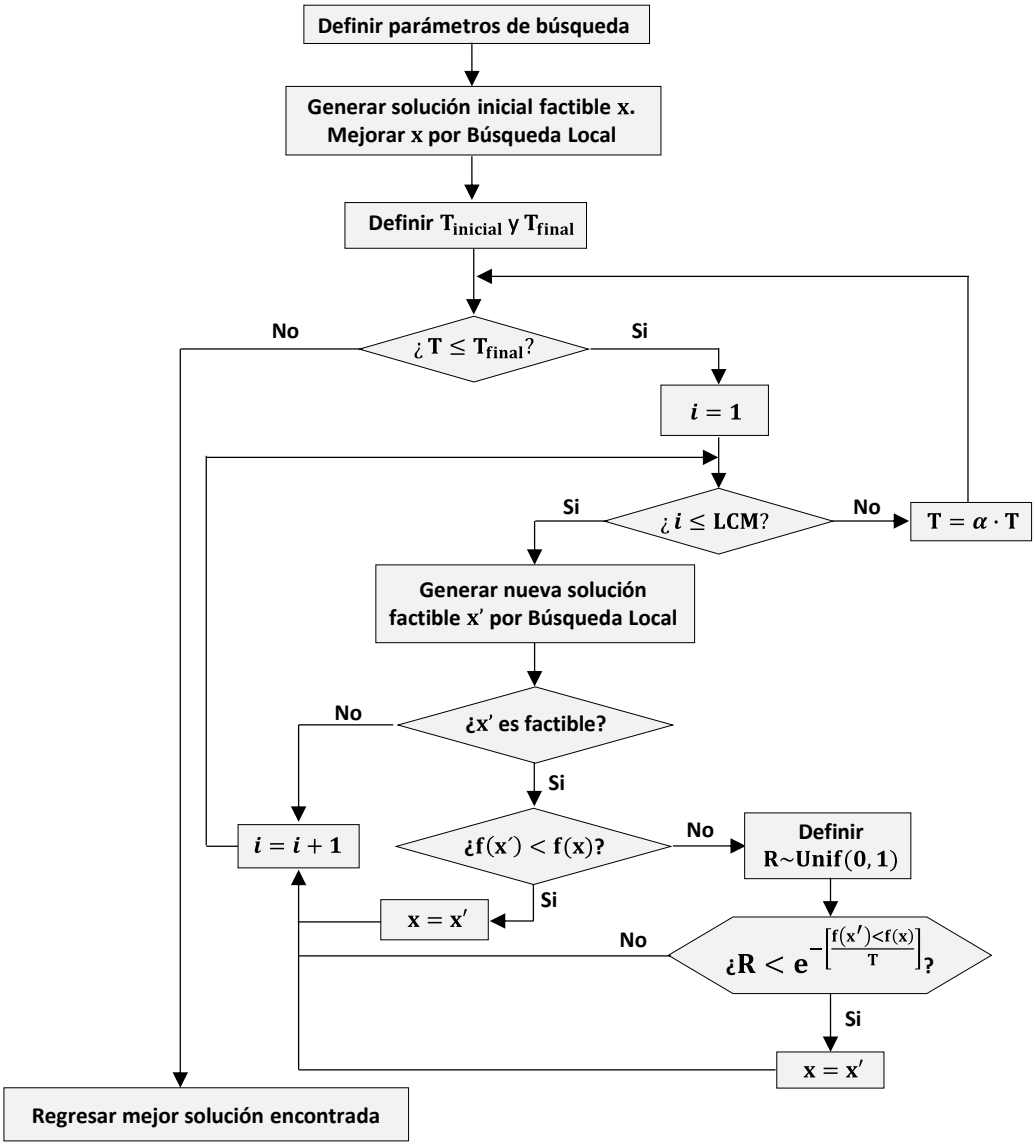


Figura 5.6 Diagrama de flujo de Recocido Simulado usado en este ejemplo

En lo que respecta al programa de enfriamiento, se decidió utilizar un decrecimiento geométrico, el cual está definido por un coeficiente de enfriamiento (α) con valor menor a la unidad ($\alpha < 1$). Cuando α presenta un valor bajo, se obtiene un programa de enfriamiento muy rápido que puede provocar el estancamiento del algoritmo en un óptimo local; por otra parte, si α tiene un valor cercano a 1, el algoritmo puede requerir un tiempo excesivamente largo para finalizar el programa de enfriamiento.

En este ejemplo, el valor de α se definió tomando en cuenta el número de cadenas de Márkov (τ) que era capaz de crear, el cual se determina, para un programa de enfriamiento geométrico, como el menor entero no menor que la siguiente expresión:

$$\tau = \lceil Log_{\alpha}(\beta) + 1 \rceil \qquad (5.11)$$

siendo β el porcentaje de la temperatura inicial (T_o) que define a la temperatura final. En la Tabla 5.1 se muestra el número de cadenas de Márkov que generan diferentes valores de α, considerando el valor de $\beta = 0.01$. Se observa que para un valor de $\alpha = 0.95$ se requieren de 91 cadenas de Márkov para cumplir con el criterio de parada; asimismo, los valores de α igual a 0.70 y 0.75 requieren ambos de menos de 20 cadenas de Márkov para alcanzar la temperatura final. Tomando en cuenta lo anterior, se decidió tomar un valor de $\alpha = 0.90$ el cual genera 45 Cadenas de Márkov.

Tabla 5.1 Cadenas de Márkov generadas por diferentes valores del coeficiente α.

Coeficiente de enfriamiento (α)	**0.70**	**0.75**	**0.80**	**0.85**	**0.90**	**0.95**
Número de Cadenas de Márkov (τ)	14	18	22	30	45	91

De manera similar a BT, RS se apoya en un algoritmo de BL para explorar el espacio de soluciones. En estos algoritmos se explora la vecindad de la solución actual $\boldsymbol{x}$ por medio de cambios discretos en su matriz de diseño teniéndose en cuenta que, si se modifica un porcentaje alto de características, se corre el riesgo de perder los elementos o bloques que le brindan calidad a $\boldsymbol{x}$. Por tal motivo, resulta necesario definir un valor que limite la variabilidad añadida en la creación de soluciones. En el presente ejemplo, esto se realiza por medio de un parámetro denominado como pV el cual indica el porcentaje máximo de características que se pueden modificar para generar la solución $\boldsymbol{x}'$, optándose aquí por alterar hasta el 5% de las características de la solución actual $\boldsymbol{x}$.

Siguiendo el enfoque propuesto por Hasancebi *et al*. (2002), se definió una probabilidad del 80% de disminuir el área transversal de las barras seleccionadas a ser modificadas (P_{dism}), incluyéndose además una probabilidad del 50% de eliminar una conexión seleccionada al azar si la barra que la materializa poseía el área transversal mínima considerada (P_{quit}). Estas probabilidades son definidas en las líneas 11 y 12 del código.

Debido a que en este ejemplo en particular se deben conocer las fuerzas en todas las barras de la armadura para verificar la factibilidad de las soluciones, es necesario definir variables que ayudarán a programar la rutina de análisis estructural. En las líneas 15 a 18 se define el módulo de elasticidad (E), el esfuerzo de fluencia (f_y) y el peso específico del acero (γ_s) empleado en la estructura, asimismo se establece un valor igual a 0.7 como factor de reducción de resistencia por compresión (χ). Posteriormente, los vectores que contienen los posibles valores de las áreas transversales de las barras

y de la altura del cordón superior son generados en las líneas 21 y 22, respectivamente. Posteriormente, entre las líneas 25 a 36 se definen las separaciones de los nodos de la armadura en los ejes X y Y (D_x y D_y, respectivamente), se indican los nodos que presentan restricciones en sus desplazamientos (esto debido a la existencia de apoyos) y se especifican las fuerzas aplicadas en el cordón inferior ($P_x = 0$ kN; $P_y = -100$ kN). Estas últimas son almacenadas en la matriz *F* creada entre las líneas 39 a 41.

Hasancebi *et al*. (2002) reportaron que la modificación de las variables topológicas afecta en mayor medida al diseño resultante que la modificación de las áreas transversales de las barras. Por esta razón, ellos decidieron realizar un proceso de optimización de la solución inicial de manera previa a la aplicación de RS. En este trabajo también se realizó el proceso de optimización en dos fases: en la primera se definió una solución por medio de un algoritmo de BL en el cual es posible modificar tanto las dimensiones de las barras como la altura del cordón superior. Esta fase se desarrolla entre las líneas 48 a 93 del código. En la segunda fase del proceso de optimización, la solución obtenida previamente es tomada como punto de partida para el proceso de optimización dirigido por RS. Las líneas de código que realizan esta segunda etapa del proceso de optimización se encuentran entre las líneas 102 a 143.

En la primera fase del proceso de optimización se realizan 1,000 iteraciones dirigidas por un algoritmo de BL. Para simplificar el proceso, en este código se definieron diversas funciones que son explicadas a continuación y cuyos códigos pueden ser consultados en el Anexo 3.2. En las iteraciones de la primera fase de búsqueda se definen las posiciones de los nodos para un canto de la armadura dado por medio de la función *Cuadricula()*. Esta función recibe como argumentos las separaciones D_x y D_y, además del vector que contiene los posibles valores de D_y. Debido a que en esta etapa del proceso de optimización es posible variar el canto de la armadura, la función *Cuadricula()* modifica este valor de manera aleatoria por medio de cambios discretos. Adicionalmente, la función regresa las matrices *Posi* y *Sime* que se utilizan para definir los nodos de la armadura propuesta como solución. La función *Cuadricula()* es empleada en las líneas 52 y 70 del código.

Posteriormente, la solución inicial es generada en la línea 54 de manera aleatoria por medio de la función *ArmaduraAleatoria()* apoyándose tanto en las matrices *Posi* y *Sime* como en los vectores de restricciones (*R*) y de áreas de las barras (*Areas*) definidos anteriormente. Toda la información necesaria para construir una solución $\boldsymbol{x}$ es almacenada en una matriz de diseño, denominada MD, en la que se registran las uniones activas entre nodos por medio de valores binarios, donde un 1 significa una unión activa y 0 una unión desactivada. Adicionalmente, la misma matriz almacena las áreas transversales de las barras existentes en el diseño actual. Para únicamente considerar armaduras estables en el proceso de optimización, se determina el grado de hiperestaticidad total de las soluciones propuestas y, en base a su valor, se crea una variable llamada *Estab* con valor igual a 1 si la armadura es estable o 0 si es inestable. Tanto la matriz de diseño *MD* como la variable *Estab* de la solución inicial son creadas y devueltas por la función *ArmaduraAleatoria()*.

En caso de que la solución inicial sea inestable, se vuelve a generar una armadura de manera aleatoria. En caso contrario, se procede a realizar el análisis estático de la solución por medio de la función *MetodoRigidez()* en la línea 59. Esta función contiene el procedimiento para aplicar el

método de rigidez y así conocer las demandas en cada una de las barras de la solución analizada, tomando como argumentos el vector de restricciones (*R*) y fuerzas (*F*), las matrices *MD* y *Posi* además del módulo de elasticidad (*E*). La factibilidad de la solución y la cantidad de material empleado son determinadas con ayuda de la función *ComprobacionesCosto()* en la línea 62. Esta función toma como argumentos el coeficiente de reducción de resistencia por pandeo (χ), el esfuerzo de fluencia (f_y), el peso específico del acero (γ_s) y la matriz *Barras* que es donde se almacenan las solicitaciones actuantes en cada barra. Las variables que regresa son *Check*, que presenta un valor de 1 si la solución es factible o de 0 en caso contrario, la variable *Peso* que contiene la evaluación de la función objetivo de la solución y la matriz *Barras* a la que se le añadió el grado de solicitación de las barras.

En las iteraciones subsecuentes (aquellas desarrolladas entre las líneas 68 a 92), el proceso es idéntico, con la única alteración de que, en lugar de generar una nueva solución $\boldsymbol{x}'$ de manera aleatoria, se modifica la solución actual $\boldsymbol{x}$ de manera discreta por medio de BL. El proceso de generación de nuevas soluciones se realiza en la función llamada *MovimientoArmadura()*, la cual crea la matriz de diseño de la nueva candidata a solución (*md*) y determina si dicha solución es estable. Si la calidad de la nueva solución es superior a la solución actual, es decir, se tiene el caso de que $f(\boldsymbol{x}') < f(\boldsymbol{x})$, la solución $\boldsymbol{x}'$ reemplaza a $\boldsymbol{x}$ como solución actual y se procede a la siguiente iteración.

La segunda fase del proceso de optimización es dirigida por RS. En esta etapa se mantiene fijo el canto de la armadura y solo se permite la modificación de las áreas transversales de las barras. El código de la metaheurística se desarrolla entre las líneas 99 y 143. La temperatura inicial del proceso, definida como una fracción del valor de la función objetivo de la solución actual, se define en la línea 99. Posteriormente se crea un ciclo *while* que se ejecuta mientras el valor de la temperatura sea mayor al valor de la temperatura final.

Para iniciar con las Cadenas Márkov se utiliza un ciclo *for* entre las líneas 103 a 140. En este ciclo se hace uso de las funciones *MovimientoArmadura()*, *MetodoRigidez()* y *ComprobacionesCosto()*, las cuales fueron descritas anteriormente. Como en la primera fase del proceso de optimización, en este proceso se utiliza BL para crear una propuesta de solución $\boldsymbol{x}'$ por medio de la modificación de la solución actual $\boldsymbol{x}$. Si la solución es estable, se analiza estáticamente, se obtienen las solicitaciones en sus barras y se verifica si la solución cumple con los requerimientos de resistencia. Si la solución $\boldsymbol{x}'$ es factible y presenta una mayor calidad $\boldsymbol{x}$, la primera sustituye a la segunda; si $\boldsymbol{x}'$ resulta ser factible pero de menor calidad, aún puede volverse la solución actual de acuerdo con la ecuación 5.3.

Una vez que se han revisado un número de soluciones especificadas por la variable *LCM*, se actualiza la temperatura del proceso siguiendo un programa de enfriamiento geométrico. Por último, la solución optimizada es graficada por la función *GraficaArmadura()* empleada en la línea 146 del código. Antes de pasar a discutir los resultados obtenidos por RS, es necesario indicar que el algoritmo aquí utilizado es más sencillo que aquel empleado por Hasancebi *et al*. (2002) para realizar la optimización topológica de su armadura. El motivo de esto reside en la necesidad de no recargar con detalles complejos un ejemplo que tiene una finalidad más orientada a la introducción al campo de la optimización.

5.8. Tamaño del espacio de configuraciones

En este ejemplo en particular, al tratarse de una optimización topológica, el tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) no es constante y resulta ser función del número n de barras empleadas en cada diseño. Cada una de las barras puede tomar como área transversal uno de 10 valores distintos. Asimismo, si dos diseños tienen los mismos valores de áreas transversales, pero en distintas barras, ambos diseños son diferentes. La combinatoria, rama de las matemáticas que estudia la forma de enumerar las distintas ordenaciones que pueden formar un grupo de elementos, nos indica que los diseños posibles a este problema se pueden enumerar por medio de variaciones con repetición. Además de las áreas transversales, existe otra variable, que es la altura del cordón superior. Dicha variable puede tomar cinco valores distintos, con lo cual, hay que multiplicar el número de variaciones por 5. Denominando n al número de barras activas en cada diseño, la función que define a $|\boldsymbol{S}|$ es la siguiente:

$$|\boldsymbol{S}(n)| = 5 \cdot 10^n \tag{5.12}$$

A priori, se desconoce el número mínimo de barras que pueden conformar una solución factible. Sin embargo, es posible determinar un límite superior para este valor. La cuadricula considera en total 20 nodos, cada barra une dos nodos, y ninguna barra puede terminar en el mismo nodo en el que inició, por lo tanto, cada nodo puede conectar como máximo a 19 barras. Bajo estas condiciones, todos los posibles diseños presentan $n \leq 380$ barras activas. Sin embargo, hay que recordar que se impuso la condición de simetría vertical para reducir el espacio de búsqueda, con lo cual solo puede haber 11 nodos distintos, los cuales pueden generar un máximo de 10 barras cada uno. El número de barras n que define a $|\boldsymbol{S}|$ se reduce entonces a solo 110.

En la Figura 5.7 se muestra graficada la curva definida por la función de $|\boldsymbol{S}|$ para $n \in [15{,}110]$. Obsérvese que incluso con la reducción del espacio de búsqueda, en el caso más desfavorable aún existen 10^{109} posibles soluciones a revisar en el problema.

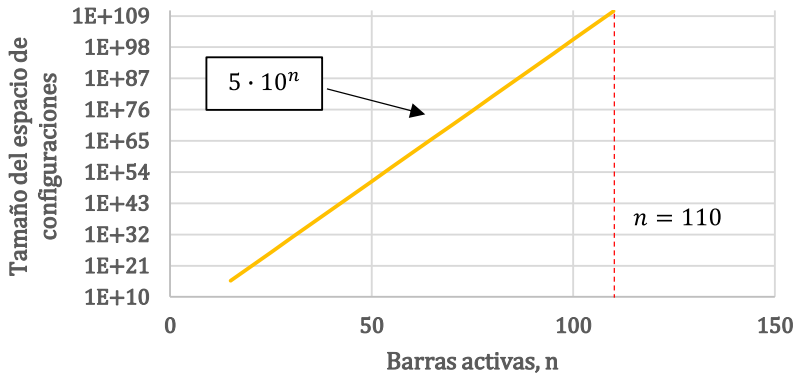
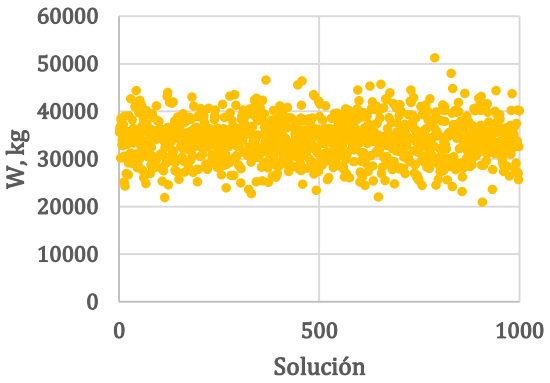


Figura 5.7 Tamaño del espacio de soluciones en función del número de barras activas

5.9. Búsqueda Aleatoria en el espacio de configuraciones

Al ser los algoritmos metaheurísticos una especie de "cajas negras", usualmente resulta difícil determinar si están brindando soluciones de calidad alta a los problemas. Para resolver esta duda, se recomienda generar una muestra aleatoria del espacio de configuraciones de manera preliminar al proceso de optimización. Los resultados obtenidos de dicha búsqueda permiten generar una muestra estadística de los valores de las funciones objetivo que presentan las soluciones creadas con las variables y restricciones consideradas en el problema. En este caso en particular, la muestra generada constó de 14,000 soluciones aleatorias, de las cuales solo 1,000 cumplían con las restricciones, esto implica que la creación de soluciones aleatorias presenta una factibilidad del 7%, la cual se puede considerar como un valor bajo. Los resultados de la Búsqueda Aleatoria (BA) se muestran en la Figura 5.8. Se aprecia que las soluciones generadas presentaban un costo medio de 33,911 kg. Por su parte, el 25% de las soluciones de mayor calidad encontradas por BA tenía un costo menor o igual a 30,643 kg, utilizándose 20,862 kg en la solución de mayor calidad hallada. Con estos valores como referencia, es posible discernir con mayor claridad si nuestro algoritmo es capaz de encontrar o no soluciones de alta calidad.



Estadísticos descriptivos		
Media =	33,911	kg
Desviación estándar =	4,459	kg
Mínimo =	20,862	kg
Primer cuartil =	30,643	kg
Mediana =	34,016	kg
Tercer cuartil =	36,617	kg
Máximo =	51,170	kg

Figura 5.8 Estadísticos de diseños factibles de armaduras generadas aleatoriamente

5.10. Resultados y discusión

De manera similar a lo realizado en el ejemplo anterior con BT, aquí el proceso de optimización fue ejecutado cinco veces, esto tanto para RS como para el algoritmo de BL. En ambos algoritmos se utilizaron los mismos valores de hiperparámetros indicados en la sección 5.7. Los desempeños de ambos algoritmos fueron comparados por medio de las medias estadísticas de sus resultados y sus desviaciones estándar. El peso de las estructuras optimizadas encontradas con ambos algoritmos se muestra en la Tabla 5.2.

Tabla 5.2 Pesos de las soluciones optimizadas halladas por RS y BL, en kg

Algoritmo	E1	E2	E3	E4	E5	Media	D.E.
RS	9,242.6	11,798.1	7,149.7	9,592.1	9,560.9	9,468.7	1,649.2
BL	15,485.9	13,639.9	9,775.6	9,671.1	10,757.5	11,866.0	2,582.0

Como se puede observar, las soluciones optimizadas encontradas por los algoritmos de búsqueda presentan ahorros considerables en el consumo de material. En comparación con los diseños encontrados por la BA (cuyo valor mínimo fue 20,862 kg), RS y BL consiguieron una reducción media del uso de material del 54% y 43%, respectivamente. Otro punto por destacar es que RS obtuvo un desempeño ligeramente superior al mostrado por BL lo cual indica que el espacio de configuraciones presenta óptimos locales que pueden ser evadidos con relativa facilidad.

En la Figura 5.9 se presentan los diseños obtenidos por RS con sus respectivos pesos (W), alturas de cordones superior (D_y) y número de barras empleadas. De esta figura se pueden hacer dos observaciones: los valores de D_y que brindan los pesos menores se encuentran entre 4 y 5 metros; por otra parte, la mayoría de los diseños obtenidos aún muestran arreglos de barras que pueden ser considerados como poco organizados, más aún si se comparan con los obtenidos por Hasancebi *et al*. (2002).

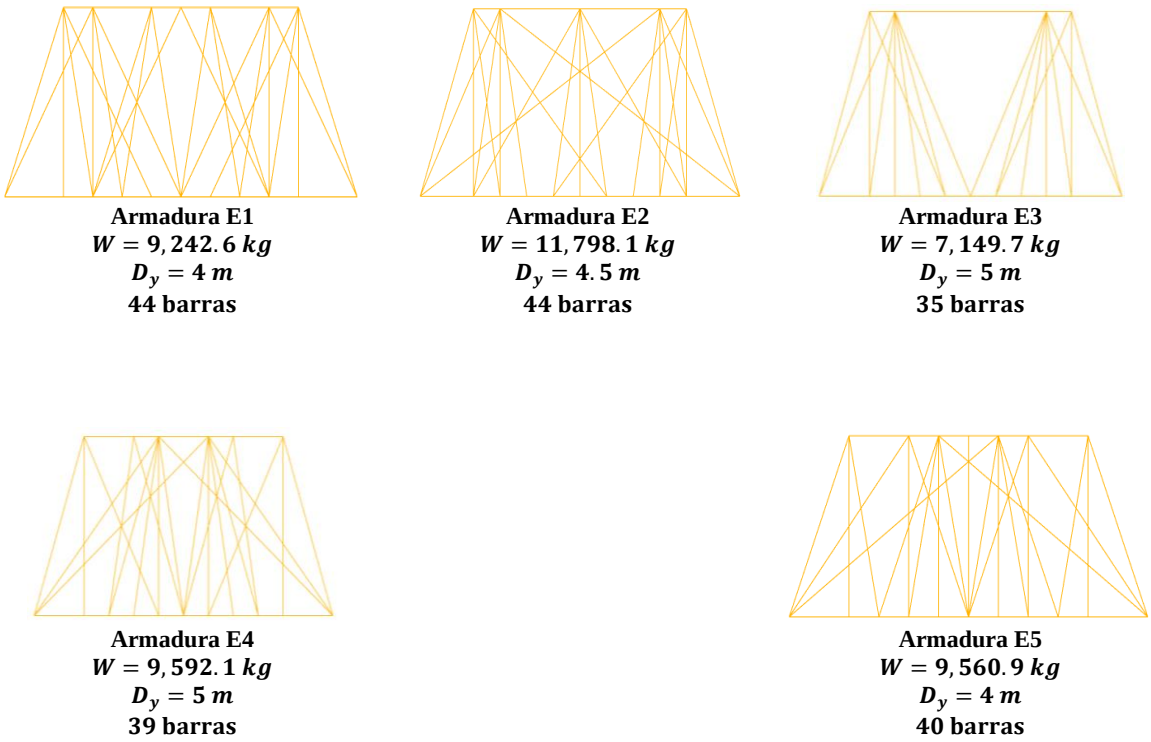


Figura 5.9 Mejores diseños obtenidos por SA

Para tener una mayor comprensión de este problema, en la Figura 5.10 se presentan las trayectorias seguidas por ambos algoritmos de búsqueda en las ejecuciones que brindaron los mejores resultados. Como se puede observar, RS cambió de solución poco más de 5,000 veces mientras que BL solo 80, es decir, RS cambió de solución unas 60 veces más que BL. Esta diferencia de valores se debe a que RS osciló alrededor de soluciones con costos similares, esto gracias a su capacidad de aceptar soluciones de menor calidad. Por su parte, BL cambió de solución un número menor de veces dada su limitante de solo aceptar soluciones de mayor calidad. Adicionalmente, la trayectoria que siguió RS muestra que el algoritmo osciló alrededor de soluciones con un peso $W = 7{,}000$ kg poco después de la solución 1,000, lo cual indica que el algoritmo quedó atrapado en un foso de potencial. En comparación, el algoritmo empleado por Hasancebi *et al*. (2002) generó 76,000 soluciones, esto debido a que se utilizó un programa de enfriamiento muy lento que consideraba la creación de 300 cadenas de Márkov.

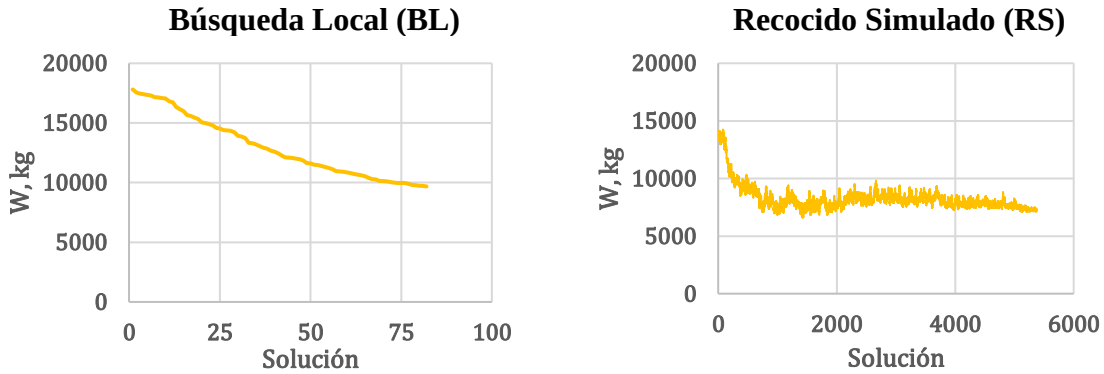


Figura 5.10 Trayectorias seguidas por BL y RS

Para finalizar, se ejecutó RS incrementando $LCM = 2{,}000$ soluciones, con la intención de aumentar el número de diseños revisados. De este proceso de optimización se obtuvo el diseño presentado en la Figura 5.11, mismo que se denomina Armadura E6. Con el incremento en la longitud de las cadenas de Márkov se visitaron aproximadamente 14,000 soluciones, observándose reducciones marginales de la función objetivo después de la solución número 2,000. De este diseño se observa un arreglo más organizado de las barras. Gracias a que se incrementó el número de soluciones visitadas en un factor de 3 se logró reducir el peso de la armadura en 1,000 kg con respecto al mejor diseño encontrado por RS anteriormente, lo cual representa un ahorro de material del 14%. Estos resultados indican que en los problemas de optimización topológica no basta con incrementar el número de soluciones visitadas para obtener diseños de alta calidad, sino que también es necesario utilizar algoritmos de generación de nuevas soluciones más complejos. Un aspecto mejorable de los diseños propuestos es el número de barras que convergen a los nodos. Como se puede observar, los diseños obtenidos presentan nodos que conectan hasta seis barras, lo cual podría resultar impráctico de ejecutar en un proyecto real. Por

simplicidad, este aspecto de las conexiones no fue abordado en el ejemplo, ya que implicaría un refinamiento del algoritmo de creación de soluciones que aportaría poco a comprender el funcionamiento de RS.

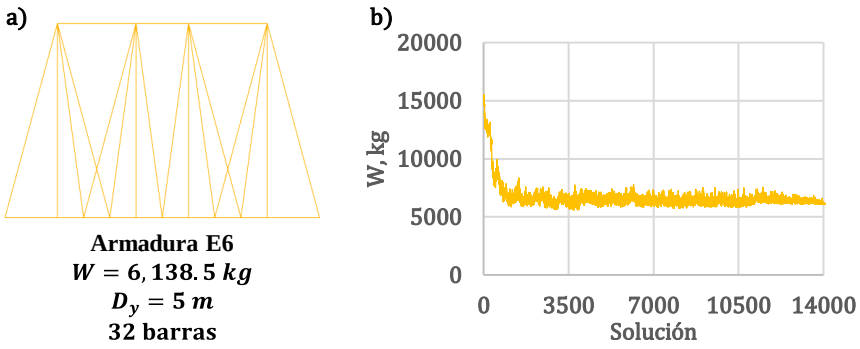


Figura 5.11 a) armadura obtenida con RS y LCM=2,000 soluciones generadas; b) trayectoria seguida por RS

5.11. Conclusiones

De este capítulo se observó que la optimización topológica de una armadura es un problema que resulta extremadamente complejo por el número de variables de decisión que se pueden tomar en cuenta. Si estas son muy numerosas, el tamaño del espacio de configuraciones puede complicar el proceso de optimización, por tal motivo resulta necesario buscar estrategias que permitan reducir el número de variables consideradas. Adicionalmente, el desempeño ligeramente superior de RS sobre BL indica que el espacio de soluciones presenta óptimos locales que pueden ser evadidos con facilidad en las primeras etapas del proceso de búsqueda. Sin embargo, en etapas más avanzadas, incluso RS puede tener problemas para salir de óptimos locales y alcanzar soluciones de mayor calidad.

Por último, se hace hincapié en que el revisar un gran número de soluciones no asegura por sí solo el obtener una solución de alta calidad. En este ejemplo en específico, se requirió utilizar estrategias más refinadas para explorar mejor el espacio de búsqueda. Esto con la finalidad de encontrar diseños de armaduras con una disposición organizada de sus barras)

Esta página ha sido intencionalmente dejada en blanco

6. Diseño óptimo de una viga por Algoritmos Genéticos

En este capítulo se ejemplifica el diseño óptimo de una viga de concreto reforzado por medio del algoritmo metaheurístico conocido como Algoritmos Genéticos (AG). El tipo de optimización a realizar es de tipo económico, es decir, se busca minimizar el costo de la estructura. El ejemplo se resuelve de nueva cuenta en el capítulo 7 por medio del algoritmo conocido como Estrategias Evolutivas (EE). Al comparar los resultados obtenidos por ambos algoritmos, el lector tendrá una noción sobre la influencia que tienen los operadores en la calidad de las soluciones obtenidas de los procesos de optimización por metaheurísticas.

6.1. Algoritmos Genéticos

Los Algoritmos Genéticos (AG) son una metaheurística que se basa en la imitación de los procesos de selección natural para encontrar soluciones de alta calidad en problemas de tipo No Polinomial. Estos algoritmos fueron presentados por primera vez por Holland (1975) y actualmente pertenecen a un grupo de algoritmos más amplio que emplean sus principios básicos y se denominan bajo el nombre común de Algoritmos Evolutivos. En su versión más sencilla, estas metaheurísticas generan soluciones factibles de alta calidad agrupando características prometedoras de soluciones peores por medio de tres operadores: **selección, cruce y mutación** (D. Goldberg, 1989). AG es una metaheurística basada en poblaciones, es decir, trabajan con un grupo de soluciones en cada iteración, pudiéndose denominar a cada una de estas como “individuos”. Cada uno de los individuos es representado por medio del arreglo de caracteres denominado como cromosoma, el cual contiene las características que definen a una solución. Existen dos maneras de representar las características de un individuo por medio de un cromosoma: a nivel **genotípico** y a nivel **fenotípico**. El primer nivel corresponde a la representación de una característica en código binario mientras que el segundo permite otro tipo de variables como números enteros, reales, etc. En un principio los cromosomas se trabajaban en código binario, es decir, las características de cada

solución se almacenaban a nivel genotípico. Sin embargo, lo usual hoy en día es usar fenotipos, lo cual permite establecer directamente el valor correspondiente de cada característica. En la Figura 6.1 se muestra el diagrama de flujo de AG.

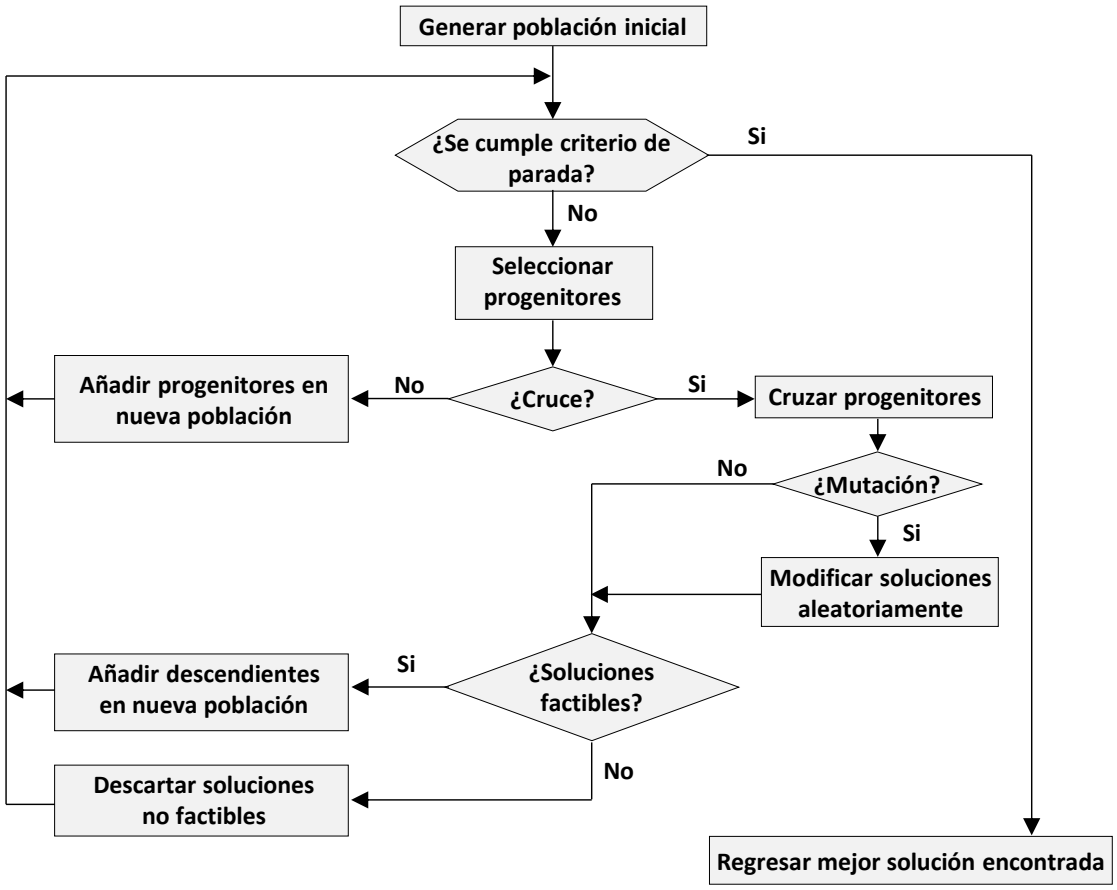


Figura 6.1 Diagrama de flujo de Algoritmos Genéticos

Como se ve en la figura, AG inicia creando una población inicial de manera aleatoria, conformada por un número de individuos definida por el usuario. Esta población semilla es empleada como base para genera mejores soluciones que conformarán a la siguiente generación. Siguiendo la analogía de la selección natural, los individuos de la población actual son escogidos por medio de un **algoritmo de selección** estocástica que favorezca a las soluciones con mayor calidad. La idea de esto es tomar a los mejores individuos para dar origen a una nueva población que compartan sus características con las siguientes generaciones.

Al haber escogido a los individuos progenitores, la descendencia se forma por medio de un **operador denominado cruce o recombinación**, el cual tiene como finalidad mezclar las características de los progenitores. Posteriormente, las soluciones creadas a través del operador cruce presentarán cierta **probabilidad de mutación**, es decir, de cambiar de manera aleatoria alguna de sus

características. Finalmente, una vez creada la nueva generación de individuos, estos reemplazarán a la generación anterior, volviéndose a su vez los progenitores de la siguiente generación. En el último paso existen las opciones de eliminar completamente la generación anterior o conservar algunos individuos de la población actual para añadirlos a la nueva generación. Este proceso se repite de manera iterativa hasta alcanzar un número determinado de generaciones o hasta cumplir con algún criterio de parada. A continuación, se describen los operadores empleados en la creación de nuevas generaciones con mayor detalle.

Operador Selección. La selección es el operador que define las soluciones que traspasarán sus características a las soluciones de las siguientes generaciones. Existen diferentes formas de escoger a los individuos progenitores, entre las que se encuentran la selección por ruleta, por ranking y por torneo.

- **Selección por ruleta.** La probabilidad que tiene un individuo para ser seleccionado es proporcional al valor de su función objetivo. Este método fue el primero en ser propuesto y es uno de los más utilizado, a pesar de que presenta algunos inconvenientes como el no poder ser usado con valores negativos de la función objetivo o que los individuos de baja calidad pueden presentar probabilidades altas de ser seleccionados (Blickle & Thiele, 1996).
- **Selección por ranking.** En esta formulación, los individuos de la población son ordenados en función de su calidad, de peor a mejor, estableciéndose su probabilidad de ser seleccionados por medio de una función que depende de su posición en el ranking (Blickle & Thiele, 1996). Este método de selección presenta la ventaja de que puede ser utilizado para problemas con funciones objetivo de valores negativos.
- **Selección por torneo.** Se genera un subgrupo de individuos tomando un numero n de estos de la población, de entre esos individuos solo se selecciona al mejor para ser utilizado como progenitor. El proceso se repite hasta que la población de descendientes ha alcanzado el tamaño deseado (D. E. Goldberg & Deb, 1991). Bajo este esquema de selección, un torneo de tamaño grande generará que el operador selección muestre una alta preferencia por soluciones de alta calidad, lo cual a su vez podría disminuir la diversidad de la población. Por tales posibles efectos, el tamaño de torneo apropiado es altamente dependiente del problema en cuestión (Lavinas *et al*., 2018).

Evidentemente, la forma en que se seleccionan las soluciones progenitoras influye en las capacidades de búsqueda del algoritmo. Para demostrar los efectos que puede tener el operador selección en el desempeño de AG, se presenta el ejemplo siguiente: considérese una población formada por 100 individuos divididos en 10 grupos de 10 individuos cada uno. En el primero de estos grupos, que representa a las soluciones de peor calidad, todos los individuos tienen una función objetivo igual a 10; en el segundo grupo, que presenta una calidad ligeramente mayor, la función objetivo de los individuos es igual a 20, y así sucesivamente hasta llegar al último grupo conformado por individuos con una función objetivo de valor 100. Considérese además que la población de la siguiente generación estará conformada igual por 100 individuos. En base a lo anterior, y tomando en cuenta que cada pareja de progenitores crea dos nuevos individuos, será necesario escoger 100 individuos progenitores para crear a la población de la siguiente generación.

En el proceso de selección por ruleta se asignan probabilidades de selección a cada individuo que son iguales al valor de su función objetivo dividido entre la sumatoria de las funciones objetivo de los m individuos que conforman la población, esto es:

$$P(X = x) = \frac{f(x)}{\sum_{i=1}^{m} f(x_i)} \tag{6.1}$$

donde X es la variable aleatoria y $f(x)$ es el valor de la función objetivo del individuo x. Considerando una población con las características antes mencionadas, la probabilidad de seleccionar a alguno de los individuos del grupo de peor calidad (x_{1-10}) por medio de la selección por ruleta es igual a:

$$P(X = x_{1-10}) = \frac{f(x_{1-10})}{\sum_{i=1}^{100} f(x_i)} = \frac{100}{5500} = 0.0182 = 1.8\% \tag{6.2}$$

Las probabilidades de selección que presentarían los grupos restantes por el método de la ruleta se calculan de manera análoga. En la Figura 6.2 se muestra una gráfica con el valor esperado de selección para todos los grupos que conforman a la población considerada.

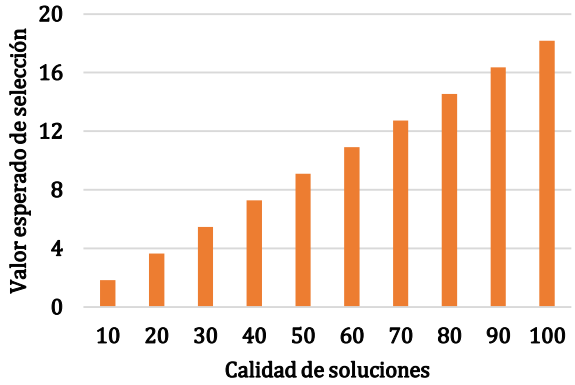


Figura 6.2 Selección de individuos por método de la ruleta

La grafica muestra que el método de selección por ruleta no tiene una fuerte predilección por tomar las soluciones de mayor calidad disponibles ya que, de media y para el caso aquí considerado, las soluciones de mayor calidad solo serían escogidas 18 veces para crear la siguiente generación, lo que representa una probabilidad de selección del 18.2%. Por su parte, aquellas soluciones que presentan valores de función objetivo entre 10 y 50, serían escogidos de media 27 veces como progenitores. Asimismo, las soluciones del grupo de peor calidad, es decir, aquellas con un valor de función objetivo igual a 10, presentan una probabilidad de ser seleccionadas del 1.8%, significando esto que serían escogidas de media 1.8 veces como progenitoras. El valor anterior puede ser un número excesivamente grande si se considera que se trata de soluciones con la peor calidad disponible.

Si en cambio se utilizara una selección por torneo, las probabilidades de seleccionar los grupos antes mencionados cambian considerablemente. En una selección por torneo se realiza un muestreo aleatorio sin reemplazo de la población de tamaño n y se escoge como progenitor a aquel que presente la mayor calidad. Cada solución de la población presenta la misma probabilidad de ser seleccionada para participar en el torneo debido a que aún no se toma en cuenta el valor de la función objetivo en este proceso. Evidentemente, la probabilidad de seleccionar cada grupo de calidad es dependiente del número de participantes n del torneo. Para fines didácticos, considérese un valor $n = 4$ participantes.

Para que sea seleccionada una de las soluciones del grupo de peor calidad por medio del método del torneo sería necesario que el resto de participantes presenten la misma calidad o una inferior, esto es, todas las soluciones escogidas deberían pertenecer al grupo de peor calidad. La probabilidad de que el primer individuo seleccionado para participar en el torneo pertenezca al grupo de peor calidad es igual a 10/100, esto debido a que hay 10 individuos en dicho grupo y la población se compone de 100 individuos. Como es un muestreo sin reemplazo, la probabilidad de que el segundo individuo pertenezca también al grupo de peor calidad es igual a $10/100 \cdot 9/99$, y así sucesivamente con el resto de participantes. Por lo tanto, la probabilidad de seleccionar una de las soluciones del grupo de peor calidad como progenitora en un torneo de tamaño $n = 4$ es igual a:

$$P(X = x_{1-10}) = \left(\frac{10}{100}\right)\left(\frac{9}{99}\right)\left(\frac{8}{98}\right)\left(\frac{7}{97}\right) = 0.005\% \tag{6.3}$$

Por su parte, para tomar una solución del grupo de mayor calidad como progenitora basta con que una o más de ellas sean seleccionadas para participar en el torneo. Por el tamaño del torneo, existen cuatro diferentes casos en lo que esto puede ocurrir: que se seleccionen cuatro, tres, dos o una sola solución del grupo de mayor calidad para participar en el torneo. Dichos eventos tienen una probabilidad de ocurrencia que sigue una distribución hipergeométrica, es decir:

$$P(X = y) = \frac{\binom{k}{y}\binom{m-k}{n-y}}{\binom{m}{n}} \tag{6.4}$$

donde y es el número de éxitos esperados en n intentos, k es el número de individuos que tienen la característica de interés y m es el tamaño de la población. Por ejemplo, la probabilidad de seleccionar una sola de las soluciones del grupo de mayor calidad ($y = 1$) para participar en un torneo de tamaño $n = 4$, considerándose una población de $m = 100$ soluciones y un tamaño del grupo de soluciones de mayor calidad $k = 10$, es igual a:

$$P(X=1)=\frac{\binom{10}{1}\binom{100-10}{4-1}}{\binom{100}{4}}=29.96\% \tag{6.5}$$

De manera análoga, las probabilidades de seleccionar dos, tres y cuatro soluciones del grupo de mayor calidad son iguales a 4.59%, 0.27% y 0.005%, respectivamente. Con lo cual, la probabilidad de que al menos una solución ($y \geq 1$) del grupo de mayor calidad participe en el torneo y, por lo tanto, sea seleccionada para transferir sus características a la siguiente población es igual a 34.83%. Este valor es casi el doble de la probabilidad obtenida por el método de la ruleta. Evidentemente, este es un ejemplo meramente didáctico ya que las poblaciones que aparecen en procesos reales de optimización no presentan calidades tan ordenadas; sin embargo, sí ayuda a visualizar las ventajas que presenta el utilizar la selección por torneo por sobre la selección por ruleta.

Operador Cruce. Al trabajar los AG con varias soluciones simultáneamente, es posible combinar sus características con la esperanza de obtener soluciones de mayor calidad, esto se realiza por medio del operador conocido como cruce o recombinación. Como su nombre lo indica, en el proceso de cruce los cromosomas de dos o más soluciones son fragmentados de manera aleatoria y mezclados de manera posterior. Existen diferentes maneras de definir los puntos donde se fragmentan los cromosomas, pudiéndose utilizar la recombinación por uno, dos y por tres puntos. En el cruce por dos puntos se definen dos posiciones de manera aleatoria por las cuales los cromosomas de ambos individuos progenitores son cortados. Los fragmentos resultantes son combinados de manera posterior para generar dos nuevas soluciones con características de ambos progenitores. La Figura 6.3 muestra de manera esquemática el proceso empleado en el cruce por dos puntos. En este caso, se combinan las características de dos soluciones progenitoras (representadas por medio de triángulos y cruces) para dar origen a dos nuevas soluciones. Los puntos de cruce (P_1 y P_2) se definen de manera aleatoria y son iguales para ambas soluciones progenitoras. Una vez establecidos los puntos de cruce, las características de ambas soluciones progenitoras son mezcladas para dar lugar a dos nuevas soluciones.

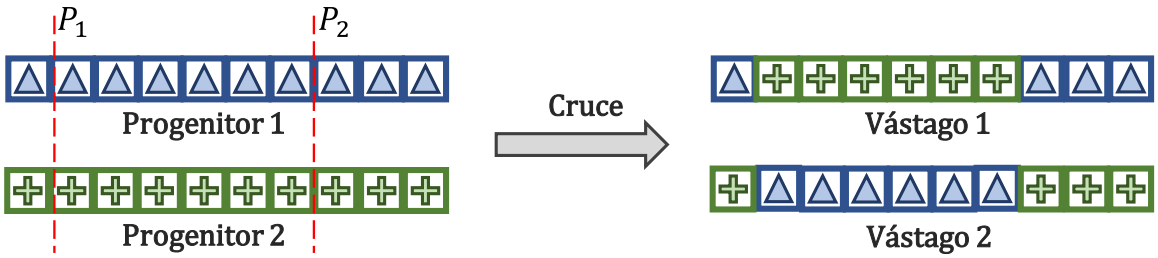


Figura 6.3 Cruce por dos puntos

Spears (2001) indica que, si la creación de nuevas generaciones se realiza únicamente por medio del cruce de sus individuos y por un periodo de tiempo extremadamente largo, la probabilidad que presenta una característica de aparecer en los individuos de las subsecuentes generaciones se vuelve dependiente exclusivamente de la proporción de individuos que mostraban dicha característica en la población inicial. Considerando un cromosoma S conformado por un número L de características, si la población es sometida únicamente al cruce de sus individuos con tiempo infinito ($t \to \infty$), la proporción esperada de S en la población es igual a:

$$\lim_{t\to\infty} p_S(t) = \prod_{i=1}^{L} p_{a_i}(0) \tag{6.6}$$

es decir: el límite de la proporción de *S*, cuando el tiempo tiende a infinito, es igual al producto de p_{a_i}, que es la proporción de la característica a en la posición i de los cromosomas de la población inicial. Lo anterior implica que, por sí solo, el operador cruce es incapaz de añadir diversidad a las nuevas poblaciones y provoca que las poblaciones tiendan a un punto de equilibrio denominado como Equilibrio de Robbins. Para introducir variabilidad en los individuos de las poblaciones, AG utiliza la mutación.

Operador Mutación. La mutación es un proceso por el cual la característica de un individuo cambia de valor de manera aleatoria. Este movimiento tiene dos funciones principales: proteger al algoritmo de pérdidas irrecuperables de características valiosas y a la vez permitir la exploración de una mayor proporción del espacio de soluciones (D. Goldberg, 1989). Bajo este operador, cada nuevo individuo creado presenta una probabilidad τ de mutar. Este valor varía en función de si se utilizan genotipos o fenotipos, encontrándose en la literatura valores de τ menores o iguales al 1% (De Albuquerque *et al*., 2012; Degertekin *et al*., 2008; Hajela & Lee, 1994; Rajeev & Krishnamoorthy, 1998) pero también del 5% (Bojorquez *et al*., 2018; Kaveh & Dadfar, 2008), el 16% (Farhat *et al*., 2009), el 30% (Shook *et al*., 2008) o hasta del 90% (Lepš & Šejnoha, 2003). Al igual de lo que ocurre con el resto de operadores antes descritos, la probabilidad de mutación es un parámetro altamente dependiente del problema, por lo que dicho valor debe ser definido en función de experiencias previas, de procedimientos de prueba y error, o del conocimiento empírico que se tenga del problema (Chiroma *et al*., 2013).

Sobre este operador, Spears (2001) indica que una población sometida únicamente al proceso de mutación tiende a un punto de equilibrio en el cual, la aparición de una característica en un nuevo individuo sigue una distribución de probabilidad uniforme. Es decir, todas las características presentan la misma probabilidad de aparecer. Considerando un cromosoma S conformado por un número L de características, si la población es sometida únicamente a mutaciones, la proporción esperada de S en la población para un tiempo $t \to \infty$ es igual a:

$$\lim_{t\to\infty} p_S(t) = \prod_{i=1}^{L} \frac{1}{C} \tag{6.7}$$

siendo C la cardinalidad de la variable de decisión sobre la cual actúa la mutación. Es importante indicar que el punto de equilibrio alcanzado por medio del proceso de mutación no es el mismo que el equilibrio de Robbins. Sin embargo, al combinar los operadores cruce y mutación se provoca que el segundo converja al punto de equilibrio con distribución uniforme, creciendo la velocidad de convergencia al aumentar la probabilidad de mutación.

Sustitución de la población actual. De manera adicional a los operadores selección y mutación antes descritos, existe la opción de asociar el cruce entre individuos con una probabilidad. Esto se realiza para preservar soluciones de la población actual en la siguiente generación, estableciendo que, si no se concreta el cruce, los individuos progenitores serán copiados directamente en la nueva población. Esta opción tiene ciertas implicaciones en la velocidad de convergencia hacia el equilibrio de Robbins debido a que, tal como indica Spears (2001), los operadores que tienen mayor inclinación hacia la destrucción de los individuos permiten converger al equilibrio con mayor rapidez.

6.2. Diseño de elementos de concreto reforzado

Los elementos estructurales como vigas o columnas se deben diseñar por medio de un proceso iterativo de prueba y error, donde la experiencia del diseñador estructural juega un papel importante. Debido a que en este proceso se deben de revisar múltiples restricciones, en muchas ocasiones los proyectistas conservan la primera solución que cumple con todas las condiciones impuestas por la normativa correspondiente. Al emplear la primera solución factible, el costo de esta dependerá en gran medida de la pericia y experiencia del proyectista que diseña el elemento estructural. Resulta evidente que aquellas y aquellos profesionales con menor experiencia tenderán a presentar diseños con costos de construcción más elevados. En la mayoría de los casos encontrados en la práctica, la diferencia de costos entre los posibles diseños alternativos de los elementos estructurales no tendrá una gran influencia en el costo total del proyecto. Por lo tanto, no resultará ni redituable ni práctico el optimizar los elementos de una estructura in situ. Esto último no aplica en otras industrias, como la de los prefabricados, ya que las empresas de esa área manufacturan una gran cantidad de elementos estructurales que siguen un mismo diseño. Esto implica que los ahorros generados por la optimización de estructuras se multiplicarían por cientos o incluso miles de veces, brindando a las empresas que empleen esta técnica una considerable ventaja competitiva. Además de otras ventajas significativas debidas a la reducción del consumo de materiales.

El encontrar el diseño óptimo de una viga de concreto reforzado es un problema de optimización combinatoria que ha sido planteado anteriormente por Coello *et al*. (1997). De manera similar, en este capítulo se resuelve el problema por medio de AG. Se optimiza el diseño de una viga rectangular bi-apoyada con una luz de 12 metros sometida a su peso propio, además de una carga uniformemente distribuida en servicio de $w = 4.5$ kN/m y a la rotura de $W_m = 6.25$ kN/m. En el diseño del elemento se consideran barras de refuerzo longitudinal tanto en la cara a compresión (al menos dos barras) como en la de tensión (al menos dos barras). El refuerzo trasversal es brindado por medio de estribos a un

ángulo de 90° con respecto al eje principal de la viga. Como datos adicionales se tiene que el elemento presenta un recubrimiento a borde de los estribos de 4 cm y se emplean apoyos de 30 cm de ancho. En la Figura 6.4 se muestra en esquema de la estructura considerada en este ejemplo. Como normativa de referencia se utilizan las Normas Técnicas Complementarias (NTC) del Reglamento de Construcción de la Ciudad de México (2020).

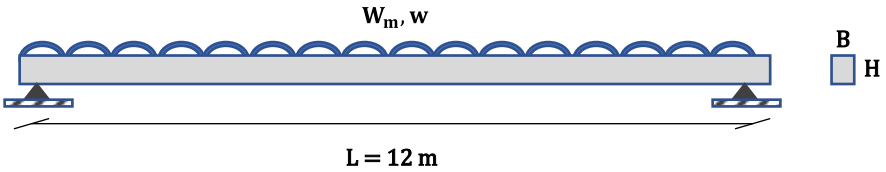


Figura 6.4 Esquema de la viga a optimizar

6.3. Tipo de optimización y función objetivo

La optimización a realizar es de tipo económica, es decir, la función objetivo a minimizar considera únicamente el costo de los materiales empleados en la elaboración de la viga, siendo el costo del elemento igual a la sumatoria del producto de los volúmenes de materiales (V) por el precio unitario (PU) de cada uno.

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} V_i \cdot PU_i \tag{6.8}$$

Por simplificación, solo se toman en cuenta los volúmenes de obra requeridos por cimbra, concreto y acero de refuerzo, tanto longitudinal como transversal. Los costos unitarios de la cimbra y el acero de refuerzo se presentan en la Tabla 6.1. El costo de concreto depende de su resistencia característica ($f'c$). Los precios unitarios de este se presentan a parte en la Tabla 6.2 para cada una de las resistencias a compresión consideradas. Es importante indicar que los valores son sólo ilustrativos para fines de la elaboración de este ejemplo.

Tabla 6.1 Precios unitarios considerador para la cimbra y el acero de refuerzo

Material	Precio unitario	Unidad
Cimbra	220.00	MXN/m^2
Acero de refuerzo	19.64	MXN/kg

Tabla 6.2 Precios unitarios considerador para el concreto

Resistencia $f'c$, MPa	**Precio unitario, MXN/m^3**
25	1,103.20
30	1,253.70
35	1,354.80
40	1,420.30
45	1,502.50
50	1,570.10
55	1,668.20
60	1,722.90
65	1,777.60
70	1,832.30

6.4. Variables de decisión y parámetros del problema

Las variables del problema son las características de la viga que se modifican en busca del diseño óptimo. Por claras razones, su elección e intervalo de valores, para los cuales existen soluciones, determinan los resultados que brindará el proceso de optimización. A continuación, se brinda una lista de las variables que se consideran en este problema.

- **Resistencia a compresión del concreto.** El valor de la resistencia a compresión del concreto ($f'c$) no solo afecta a la resistencia del elemento, sino que también tiene un impacto en su costo. Las NTC consideran que un concreto es de altas prestaciones cuando su resistencia a compresión es mayor a 40 MPa. Por otra parte, no permite utilizar concretos con resistencias mayores a 70 MPa. Basándose en lo anterior y, como se observa en la Tabla 6.2, sólo se consideran valores de resistencia entre 25 a 70 MPa.

$$f'c \in \{25, 30, \dots, 70\}\, MPa \tag{6.9}$$

- **Geometría de la sección.** Se considera una sección rectangular de ancho B y canto H. Tanto para el canto como para el ancho se permitieron valores desde 0.35 a 2.4 m. En ambos casos se utilizaron incrementos discretos de 0.05 m.

$$\begin{aligned} H &\in \{0.35, 0.40, \dots, 2.4\}\, m \\ B &\in \{0.35, 0.40, \dots, 2.4\}\, m \end{aligned} \tag{6.10}$$

- **Acero de refuerzo longitudinal.** El acero de refuerzo longitudinal está compuesto por un lecho en compresión y otro a tensión. En este problema se escogió utilizar una única capa para el lecho en compresión, mientras que se dispuso de dos capas para el lecho en tensión. La cantidad de acero longitudinal se define por el número de barras en cada capa (n) y el diámetro empleado (ϕ), siendo utilizado, por simplicidad, un único diámetro en cada capa. En estas variables el sufijo "c" denota el lecho a compresión y los sufijos "s1" y "s2" los lechos a tensión. El número de barras a utilizar se consideró desde 2 hasta 30 piezas por capa. Los diámetros empleados son aquellos existentes en octavos de pulgada, es decir, desde #3 hasta #12.

$$\begin{gathered} n_c, n_{s1}, n_{s2} \in \{2, 3, \dots, 30\}\ varillas \\ \phi_c, \phi_{s1}, \phi_{s2} \in \{3, 4, 5, 6, 8, 10, 12\}\ octavos\ de\ pulgada \end{gathered} \tag{6.11}$$

- **Acero de refuerzo transversal.** En el caso del acero de refuerzo transversal, este se define por medio del número de estribos a utilizar, el diámetro de estos y su separación en las zonas próximas al apoyo. Debido a que el esfuerzo cortante es reducido en la zona central del claro, se utilizan dos separaciones de estribos en la viga. Durante el proceso de comprobación de los diseños propuestos, se define la separación y la longitud en la que se prolongan los estribos colocados a centro de claro. Al tratarse de refuerzo de cortante, solo se considera disponer uno o dos estribos (n_E), siendo estos de diámetro (ϕ_E) #2.5 o #3. La separación entre estribos (Sp) se consideró entre 7.5 cm y 30 cm, con incrementos discretos de 2.5 cm.

$$\begin{gathered} n_{E1}, n_{E2} \in \{1, 2\}\ varillas \\ \phi_{E1}, \phi_{E2} \in \{2.5, 3\}\ octavos\ de\ in \\ Sp_1, Sp_2 \in \{7.5, 10, \dots, 30\}\ cm \end{gathered} \tag{6.12}$$

En cuanto a los parámetros del problema, estos son las cargas aplicadas, el recubrimiento de la sección, la luz de la viga y el ancho de los apoyos. En la Figura 6.5 se muestra un esquema de las variables consideradas en el problema y su ubicación en los diseños propuestos.

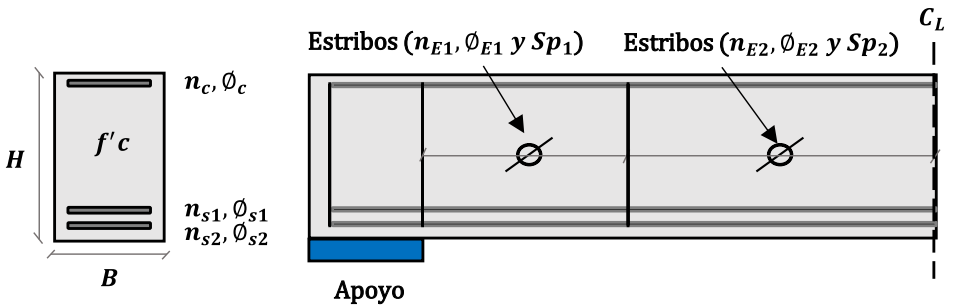


Figura 6.5 Esquema de las variables consideradas en el problema

6.5. Codificación de los diseños

Para este ejemplo, las poblaciones de cada generación fueron almacenadas en matrices bidimensionales $[a_{ij}]$ con $i \in \{1,2,3,\dots,m\}$ y $j \in \{1,2,3,..,n\}$, donde m representa el tamaño de la población mientras que *n* es el número de variables de decisión del problema.

$$\left[a_{ij}\right] = \begin{bmatrix} x_{11} & x_{12} & x_{13} & & x_{1j} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2j} \\ x_{31} & x_{32} & x_{33} & & x_{3j} \\ & \vdots & & \ddots & \\ x_{i1} & x_{i2} & x_{i3} & & x_{ij} \end{bmatrix} \tag{6.13}$$

Bajo esta representación, las características de cada solución de la población son almacenadas en las filas de la matriz a_{ij} por medio de valores discretos que se existen dentro de los intervalos mencionados en la sección 6.4. Por ejemplo, el diseño k-ésimo de la población se encontraría en la fila *k* de la matriz y se definiría por el siguiente conjunto de elementos:

$$\boldsymbol{a_k} = (f'_{ck}, H_k, B_k, n_{ck}, n_{s1k}, n_{s2k}, \phi_{ck}, \phi_{s1k}, \phi_{s2k}, n_{E1k}, n_{E2k}, \phi_{E1k}, \phi_{E2k}, S_{p_1k}, S_{p_2k}) \tag{6.14}$$

6.6. Restricciones del problema

Las vigas creadas durante el proceso de optimización deberán cumplir con las condiciones impuestas en las Normas Técnicas Complementarias del Reglamento de Construcción de la Ciudad de México (2020). Existen diferentes tipos de restricciones que deben de cumplir las vigas. Por simplificación, aquí se clasifican como geométricas, por flexión, por cortante, por deflexiones y por agrietamiento. A continuación, se presentan las restricciones correspondientes a cada clase.

Restricciones geométricas. Para una estructura de ductilidad baja se establece que la relación entre el peralte de la sección (H) y su ancho (B) no debe ser mayor a 6.

$$\frac{H}{B} \leq 6 \tag{6.15}$$

La separación horizontal entre el refuerzo longitudinal no debe ser menor al diámetro (ϕ) de las barras ni a 1.5 el tamaño máximo del agregado (TMA) usado en el concreto. Considerando un TMA = 2.5 cm, se establece que la separación horizontal no debe ser menor que 4 cm.

Restricciones por diseño a flexión. En todos los casos, el acero a tensión (A_s) debe ser mayor o igual a:

$$A_{s,min} = \frac{0.22\sqrt{f'c}}{f_y} B \cdot d \tag{6.16}$$

donde $f'c$ es la resistencia característica del concreto, fy es el esfuerzo de fluencia del acero de refuerzo y d es el peralte efectivo. En todos los casos, el acero a tensión no deberá ser mayor al 90% del que corresponde a la falla balanceada de la sección, en el caso de secciones doblemente armadas, el acero a tensión máximo se obtiene considerando el acero a compresión (A'_s) como:

$$A_{s,max} = 0.9\left(\frac{600\beta_1}{600 + f_y}\frac{f''c}{f_y} \cdot B \cdot d + A'_s\right) \tag{6.17}$$

donde $f''c$ es igual a $0.85 \cdot f'c$, y β_1 es la relación entre la profundidad de la fibra neutra y la altura del bloque de compresiones, que es función de la resistencia característica del concreto y se obtiene de la siguiente manera:

$$\beta_1 = \begin{cases} 0.85 & (f'c \leq 28\ MPa) \\ 1.05 - \dfrac{f'c}{140} & (f'c > 28\ MPa) \end{cases} \tag{6.18}$$

Cuando la sección diseñada es doblemente armada, el momento resistente de la misma se puede determinar por la siguiente ecuación:

$$M_R = FR \cdot [(A_s - A'_s)f_y\left(d - \frac{a}{2}\right) + A'_s \cdot f_y(d - d') \tag{6.19}$$

donde FR es el factor de resistencia, en el caso de flexión $FR = 0.9$; d' es la distancia entre el borde a compresión de la sección y el centroide del acero a compresión, y a es la profundidad equivalente del bloque de compresiones, que se estima como:

$$a = \frac{(A_s - A'_s)f_y}{f''c \cdot B} \tag{6.20}$$

Las NTC indican que la ecuación (6.20) solo es válida si el acero a compresión fluye en el momento en el que se produce la falla, esto debido a que dicho modo de falla corresponde con un comportamiento dúctil, es decir, la estructura presenta deformaciones grandes antes de alcanzar su límite de carga. Esta forma de fallo ocurre cuando:

$$(A_s - A'_s) \geq \frac{600\beta_1}{600 - f_y}\frac{f''c}{f_y} B \cdot d' \tag{6.21}$$

Restricciones por diseño a cortante. La resistencia a cortante de la sección se compone por la fuerza cortante que toma el concreto (V_{CR}) y la fuerza que toma el refuerzo transversal (V_{SR}). La suma de estas fuerzas debe de ser mayor al cortante aplicado mayorado (V_u).

$$V_u \leq V_{CR} + V_{SR} \tag{6.22}$$

V_u se obtiene a una distancia d con respecto al paño del apoyo, y no se permite que este sea mayor a:

$$V_{u,max} = FR\ (0.80)\ \sqrt{f'c}\ B \cdot d \tag{6.23}$$

donde $FR = 0.75$ para cortante. Por su parte, V_{CR} es función de la cuantía del acero a tensión (ρ) con respecto a la sección bruta. El valor de V_{CR} se obtiene por medio de la siguiente expresión:

$$V_{CR} = \begin{cases} FR(0.2 + 20\rho)(0.3)\sqrt{f'c}\ B \cdot d & (\rho < 0.015) \\ \\ FR(0.16)\sqrt{f'c}\ B \cdot d & (\rho \geq 0.015) \end{cases} \tag{6.24}$$

La normativa limita el valor de V_{CR} a un máximo de:

$$V_{CR,max} = FR(0.47)\sqrt{f'c}\ Bd \tag{6.25}$$

Finalmente, el valor de V_{SR} se determina por medio de:

$$V_{SR} = \frac{FR \cdot A_v \cdot f_y \cdot d(Sen\theta + Cos\theta)}{S} \tag{6.26}$$

donde A_v es el área transversal de los estribos (considerando dos ramales por estribo), θ es el ángulo entre los estribos y el eje principal de la viga y S es la separación entre los estribos. Las limitaciones de la separación entre estribos se definen en función de los valores de V_u, V_{CR} y $V_{CR,max}$ de la siguiente manera:

$$S \rightarrow \begin{cases} S_{min} = 60\ mm & (V_u < V_{CR}) \\ S_{max} = 0.5d & (V_{CR} < V_u < V_{CR,max}) \\ S_{max} = 0.25d & (V_{CR,max} < V_u) \end{cases} \tag{6.27}$$

Por último, el área mínima del refuerzo transversal se calcula por medio de la expresión:

$$A_{v,min} = (0.10)\sqrt{f'c}\frac{B \cdot S}{f_y} \tag{6.28}$$

Restricciones por deflexión. La deflexión de la viga es una comprobación en servicio, por lo mismo, se deben de usar los factores de seguridad correspondientes. Para estos casos de carga, se consideran la carga muerta más la carga viva que actúa de manera permanente. La deflexión total (y_T) se compone de la deflexión inmediata (y_o) más la diferida (y_{dif}).

$$y_T = y_o + y_{dif} \tag{6.29}$$

La deflexión inmediata (y_o) se determina por medio de ecuaciones de deflexiones elásticas, en este caso, por tratarse de una viga simplemente apoyada sometida a una carga distribuida, la deflexión inmediata es igual a:

$$y_o = \frac{5wL^4}{384E_cI_{ag}} \tag{6.30}$$

donde E_c es el módulo de elasticidad del concreto, el cual depende del valor de $f'c$, y se obtiene por medio de la siguiente expresión:

$$E_c = \begin{cases} 4{,}400\sqrt{f'c} & (f'c < 40\ MPa) \\ 2{,}700\sqrt{f'c}\ + 11{,}000 & (f'c \geq 40MPa) \end{cases} \tag{6.31}$$

De manera simplificada, es posible utilizar la inercia de la sección transformada agrietada (I_{ag}) para calcular la deflexión inmediata. Al tratarse de una sección rectangular sin carga axial aplicada, I_{ag} se determina como:

$$I_{ag} = \frac{Bx^3}{12} + Bx\left(\frac{x}{2}\right)^2 + (\eta - 1)A'_s(x - d)^2 + \eta(A_s)(x - d)^2 \tag{6.32}$$

donde η es la relación entre los módulos de elasticidad del acero y del concreto ($\eta = E_s/E_c$); y x es la profundidad de la fibra neutral, la cual se obtiene por equilibrio considerando una distribución lineal de esfuerzos del concreto debido a la baja magnitud de las solicitaciones actuantes en servicio:

$$\frac{Bx^2}{2} + A'_s(\eta - 1)(x - d') - A_s\eta(d - x) = 0 \tag{6.33}$$

La deflexión diferida (y_{dif}) se obtiene multiplicando la deflexión inmediata por un factor que es función de la cuantía de acero a compresión (ρ'):

$$y_{dif} = \frac{2}{1 + 50\rho'}\, y_o \tag{6.34}$$

Restricciones por agrietamiento. Las NTC indican que el acero de refuerzo a tensión debe de trabajar, para una condición de servicio, con un esfuerzo menor a un valor dado que depende de la clase de exposición del elemento estructural. Si se cumple la condición, se considera que el diseño satisface los requerimientos por agrietamiento. El esfuerzo límite debe de compararse con el valor obtenido por medio de la siguiente expresión:

$$f_s \cdot \sqrt[3]{d_f \cdot A}\frac{h_2}{h_1} \tag{6.35}$$

donde f_s es el esfuerzo del acero a tensión en condiciones de servicio, d_f es el recubrimiento del concreto de la fibra más traccionada al centro de la barra más próxima, h_1 es la distancia entre el eje neutro y el centroide del refuerzo a tensión, h_2 es la distancia entre el eje neutro y la fibra más esforzada a tensión. Por último, A es el área del concreto a tensión, cuyo centroide coincide con el centroide del acero a tensión, dividida entre el número de barras. Cuando se utilicen barras de diferentes diámetros, el número de barras de calcula de la siguiente manera:

$$\frac{A_{sT}}{A_{s,\emptyset min}} \tag{6.36}$$

siendo A_{sT} el área total del refuerzo a tensión y $A_{s,\emptyset min}$ el área correspondiente a la barra del menor diámetro utilizado. Para una clase de exposición A2, NTC indican un valor máximo para la ecuación (6.35) de 40 MPa.

6.7. Descripción del código de Algoritmos Genéticos

En la Figura 6.6 se muestra el diagrama de flujo de AG empleado en este problema. El código fue ejecutado en el programa Matlab (MathWorks Inc, 2020) y se puede revisar de manera completa en el Anexo 4.1 de este libro. Se recomienda leer de manera paralela el código junto con esta sección para una mejor comprensión de su funcionamiento.

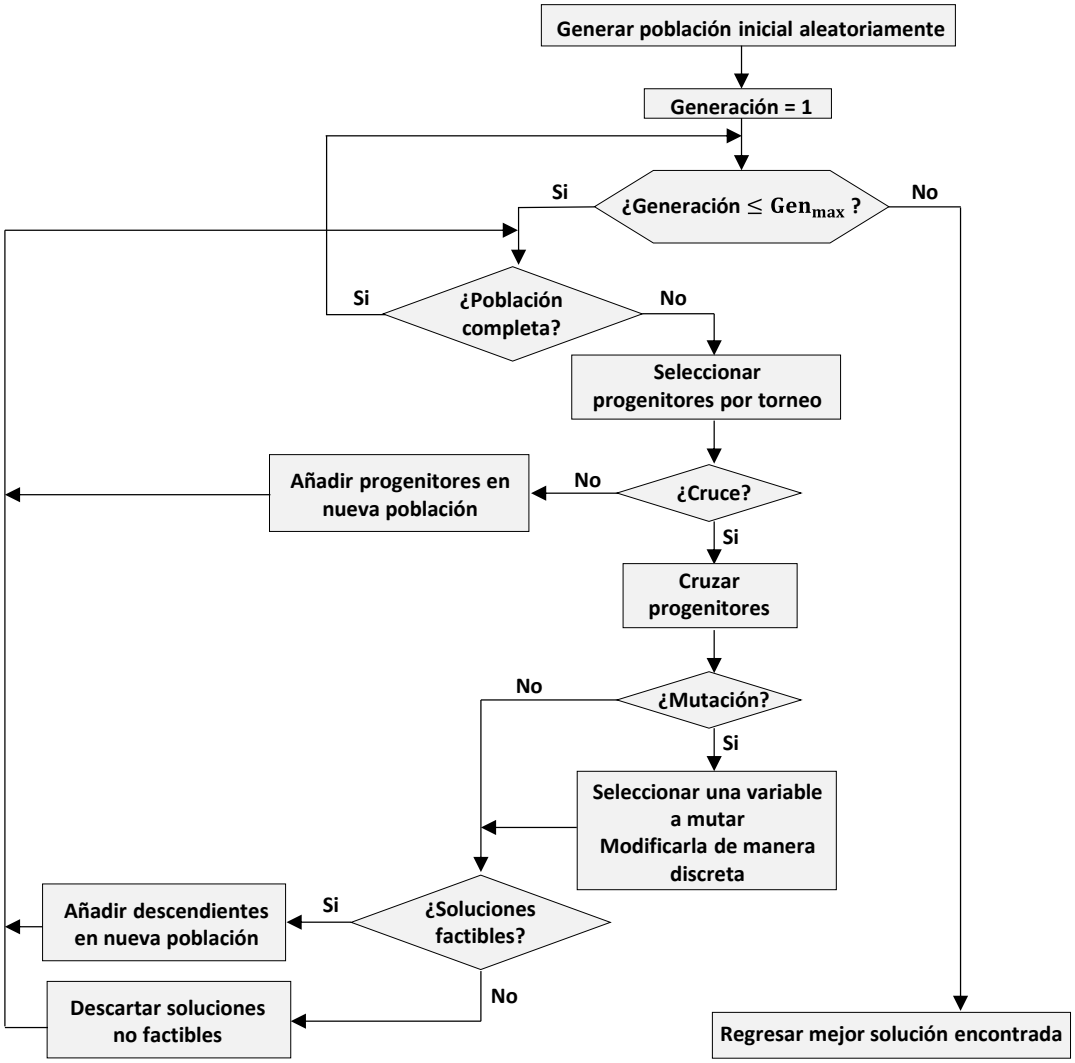


Figura 6.6 Diagrama de flujo de Algoritmos Genéticos empleado en este problema

El código comienza definiendo los intervalos en que existen las variables del problema, es decir, los valores considerados de la resistencia a compresión del concreto (*Vfc*), dimensiones posibles de las vigas (*VH* y *VB*), el número de barras (*Vn*) y diámetro (*Vfi*) del refuerzo longitudinal, y el número (*VnE*), diámetro (*VfiE*) y separación (*VsE*) del refuerzo transversal. Esto se realiza en las líneas 5 a 13 del código. Los valores que se mantienen fijos durante el proceso de optimización, es decir, la luz de la viga (*L*), el ancho de los apoyos (*BApoyo*), el recubrimiento (*rec*) y las cargas actuantes (*w* y *Wm*), se definen en las líneas 16 a 20.

El desempeño del algoritmo depende de los parámetros de búsqueda que se utilicen, los cuales deben de ser calibrados para encontrar la combinación adecuada que permita encontrar las mejores soluciones al problema. Como se observa en las líneas 23 a 28, los parámetros de la búsqueda considerados son los siguientes:

- Tamaño de la población inicial (*TP*).
- Tamaño de la población de descendientes o vástagos (*TV*).
- Tamaño del torneo (*TT*).
- Número de generaciones por búsqueda (*GenMax*).
- Probabilidad de cruce (*PCruce*).
- Probabilidad de mutación (*PMutacion*).

La población inicial define el número de soluciones factibles que se generarán de manera aleatoria y que serán utilizadas para crear las generaciones subsecuentes. Como criterio de parada se utiliza un número máximo de generaciones. Asimismo, el usuario puede definir la probabilidad de cruce y de mutación, esto en función de cualquier criterio que considere conveniente y tomando en cuenta las repercusiones que estos operadores puedan tener en el desempeño del algoritmo. En las siguientes líneas (de la 31 a la 38) se definen una serie de matrices necesarias para el funcionamiento del algoritmo y que almacenarán los resultados de la búsqueda. Estas son: la matriz *Población*, que registra las características de todas las soluciones actuales del algoritmo. Por su parte, *Torneo* guarda los identificadores de las soluciones que participan en el proceso de selección por torneo además de sus respectivos costos. La matriz *Participantes* también es utilizada en el proceso de selección por torneo. La matriz *Vv* agrupa los valores posibles de las variables de decisión, siendo los valores asignados a esta matriz entre las líneas 45 y 57. Los costos de todas las soluciones de la población actual son almacenados en el vector *Costos*, mientras que la población siguiente y sus respectivos costos se guardan en las variables *Vástagos* y *CostosV*, respectivamente. Por último, los mejores costos de cada generación serán registrados en el vector *Cmin*.

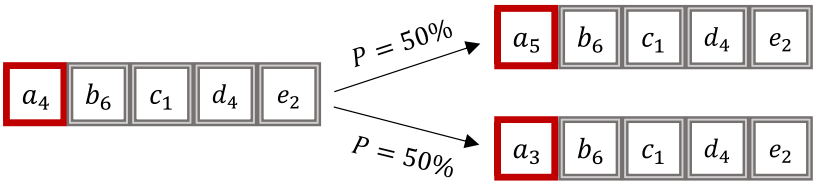
El código del proceso de optimización se desarrolla entre las líneas 61 a 229. Como en el ejemplo anterior, aquí se definen una serie de funciones cuyo código puede ser consultado a profundidad en el Anexo 4.2.

Dado que no todos los diseños generados cumplirán con las restricciones del problema, para crear la población es necesario utilizar un ciclo *while* que finalice una vez que se haya alcanzado el tamaño de la población especificado por el usuario, sin importar el número de iteraciones que esto tome. El ciclo *while* que crea la población inicial se desarrolla entre las líneas 61 a 89. Para generar dicha población, se crea un diseño a la vez tomando valores de manera aleatoria con ayuda de la matriz Vv. Una vez que el diseño ha sido creado, se utiliza la función ComprobacionesAG() para verificar su factibilidad. Las restricciones consideradas por la función son aquellas descritas en la sección 6.6 y abarcan de la ecuación 6.15 hasta la 6.36. La función ComprobacionesAG() toma como argumentos la solución a verificar y los parámetros del problema (luz, recubrimiento, cargas, etc.) y genera una serie de variables que describen a la solución. De estas variables, la denominada como check presenta un valor igual a cero si la solución es factible o uno si no lo es, el resto de las variables se utilizan para calcular el costo de la solución. En caso de que la solución sea factible, se utiliza la función *CostoViga()* para determinar su costo. Posteriormente, el costo de la solución es almacenado en el vector *Costos*. Este proceso se repite hasta que se completa la población inicial.

Habiéndose generado la población inicial, se procede con la creación de las siguientes generaciones por medio de los operadores selección, cruce y mutación. Para generar las generaciones subsecuentes se emplea un ciclo *for* que abarca de la línea 95 a la 229. En este caso se aplica una estrategia elitista la cual preserva para la siguiente generación la solución de mayor calidad en la población actual, esto ocurre en las líneas 97 y 98. Para la creación del resto de soluciones de la siguiente generación, las soluciones progenitoras son seleccionadas por medio de un torneo con cuatro participantes, los cuales son tomados de manera aleatoria de la población actual. Para evitar que se escoja la misma viga dos veces en el proceso de selección, el torneo se define en dos ocasiones: una primera considerando la población total y una segunda descartando a la primera viga seleccionada. El resultado del torneo se define de manera determinista, tomando como ganador a la solución que presente la mayor calidad entre todos los participantes. Este proceso es realizado entre las líneas 106 a 145.

De manera posterior, en la línea 148, se emplea un condicional *if* que da inicio al proceso de creación de nuevas soluciones si un número generado de manera aleatoria y con valor entre 0 y 1 es menor a la probabilidad de cruce establecida por el usuario anteriormente. Si ocurre el cruce entre las soluciones progenitoras, se crean dos nuevas soluciones que presentan cierta probabilidad de mutar, esto es, de presentar cambios aleatorios en sus características. Tanto el proceso de cruce como de mutación ocurren dos veces en el código, una por cada nueva solución generada por iteración. El primer cruce entre soluciones progenitoras se desarrolla entre las líneas 150 a 163, y se emplea el método del corte en dos puntos de los cromosomas descrito en la Figura 6.3. La primera solución nueva, creada por el cruce de las soluciones progenitoras, es almacenada en la matriz *Vastagos*, esto entre las líneas 159 a 163.

La solución nueva creada será mutada a través de la función *Mutacion()*, considerando una probabilidad *PMutacion*. Se decidió que el operador mutación solo modifica una variable del cromosoma de la nueva solución, escogiéndose la variable a modificar de manera aleatoria. Todos los valores que pueden tomar las variables del problema se encuentran almacenadas en el arreglo matricial *Vv*. Habiéndose escogido la variable a modificar, basta con identificar su posición en la matriz y definir el movimiento a realizar. En este caso se consideraron solamente dos movimientos: reducir o incrementar en una unidad la posición de la variable a mutar. Este movimiento discreto de pequeño alcance se utilizó para evitar que la mutación destruyera la factibilidad de la nueva solución. Obsérvese que un movimiento aleatorio a cualquier otra posición del vector asociado a la variable hubiera incrementado la probabilidad de que los diseños mutados no cumplieran con las restricciones del problema debido al cambio brusco de sus características. En la Figura 6.7 se muestra de manera esquemática el proceso de mutación. La variable de decisión que fue seleccionada para mutar resultó ser a_4, pudiendo esta cambiar al valor a_3 o a_5 únicamente. Si en cambio la variable a mutar fuera c_1, esta solo podría cambiar a c_2 ya que no existe una posición anterior a 1. En caso de ser activada, la función *Mutacion()* recibe como argumentos la solución creada recientemente y la matriz *Vv* que contiene todos los posibles valores de las variables de decisión del problema, y devuelve la característica a mutar y su nuevo valor, esto a través de las variables *Gen* y *Mut*, respectivamente. Esto ocurre entre las líneas 166 a 169.



$$a^T = \{a_1, a_2, a_3, a_4, a_5, a_6\}$$

Figura 6.7 Esquema del proceso de mutación empleado en este problema

Una vez que las nuevas soluciones han sido creadas y, en caso de ser necesario, mutadas, estas son sometidas en la línea 172 a la verificación de las restricciones impuestas por normativa por medio de la función *ComprobacionesAG()* descrita anteriormente. Si la nueva solución no resultase factible, es descartada; en caso contrario se evalúa el costo de la solución en la línea 177 por medio de la función *CostoViga()* y se almacena en población de la descendencia. En este punto cabe señalar que esta no es la única estrategia disponible para lidiar con diseños que no cumplen las restricciones ya que también es posible utilizar funciones de penalización que disminuyen la probabilidad de los diseños infactibles de participar en la creación de la siguiente población (Gen & Cheng, 1996).

Entre las líneas 189 a 212 ocurre la creación de la segunda nueva solución por medio de los operadores cruce y mutación. En caso de que no se concrete el cruce entre las dos soluciones progenitoras, estas son copiadas de manera íntegra en la población de la descendencia. Dicho proceso se encuentra entre las líneas 214 a 220 del código. Este proceso de creación de soluciones factibles es repetido hasta completar el tamaño de la población de la generación siguiente. Cuando esto ocurre, la población de progenitores es reemplazada por la población creada recientemente, se registra el costo de la solución de mayor calidad presente en la población actual y se procede con la creación de la siguiente generación. Como se puede observar, el código de AG es relativamente sencillo y se basa en tres operadores muy básicos, los cuales permiten ciertas modificaciones para mejorar el desempeño del proceso de búsqueda.

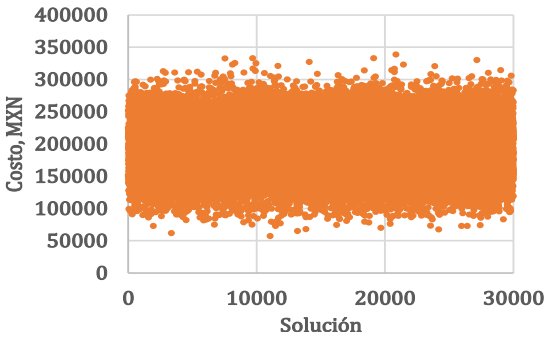
6.8. Tamaño del espacio de configuraciones

El tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) del problema es igual a la multiplicación de la cardinalidad de cada una de las variables consideradas. Como se observa en el código del algoritmo (líneas 5 a 13), se utilizan 10 valores de resistencia a compresión del concreto, 42 valores de peralte, *H*; 42 valores de ancho, *B*; 29 valores de números de barras de refuerzo longitudinal, 7 valores de diámetros de barras (estas últimas dos variables se repiten tres veces debido al refuerzo a compresión y a las dos capas de refuerzo a tensión), 2 valores de diámetro y número de estribos utilizados y 10 valores de separaciones entre estribos. Considerando lo anterior, se tiene un espacio de soluciones de tamaño igual a:

$$|\boldsymbol{S}| = 10 \cdot 42^2 \cdot 29^3 \cdot 7^3 \cdot 2^2 \cdot 10 = 5.90x10^{12} \text{ configuraciones posibles} \qquad (6.37)$$

6.9. Búsqueda Aleatoria en el espacio de configuraciones

De manera previa al proceso de optimización, se realizó una Búsqueda Aleatoria (BA) en el espacio de configuraciones, generando 30,000 vigas factibles y registrando los costos que estas presentaban. Los resultados obtenidos junto con los estadísticos descriptivos de la muestra se presentan en la Figura 6.8.



Estadísticos descriptivos		
Media =	197,712	MXN
Desviación estándar =	38,343	MXN
Mínimo =	57,598	MXN
Primer cuartil =	172,598	MXN
Mediana =	200,161	MXN
Tercer cuartil =	224,042	MXN
Máximo =	338,961	MXN

Figura 6.8 Búsqueda aleatoria del espacio de soluciones

Se observa que las soluciones factibles encontradas por BA presentan un costo máximo de 338,961 MXN y un valor medio igual a $197,712 MXN. La muestra también indica que las vigas cuyo costo sea menor a $57,600 son extremadamente infrecuentes. Debido a este resultado, se buscó calibrar el algoritmo para conseguir soluciones con costos menores a $60,000 MXN. A continuación, se presentan los resultados obtenidos de algunas ejecuciones del algoritmo para resaltar las capacidades y limitaciones que tiene AG para encontrar soluciones de alta calidad.

6.10. Resultados y discusión

Para determinar la influencia de los valores de los hiperparámetros en el desempeño de AG, se realizaron múltiples pruebas del algoritmo con probabilidades de mutación entre 0 y 1, considerando incrementos del 10%. Para cada probabilidad de mutación se obtuvo una muestra de 10 soluciones optimizadas donde se definió un tamaño de población de 200 individuos, un tamaño de torneo de 4 individuos, una probabilidad de cruce del 90% y 100 generaciones por ejecución. Las calidades de las soluciones encontradas se muestran en la figura 6.9 a través de diagramas de cajas y bigotes. En estos

diagramas, la caja central de los diagramas representa todos aquellos valores de la muestra que se encuentran entre los percentiles 25 y 75, indicando la barra intermedia la localización de la mediana o percentil 50. Por su parte, los bigotes inferiores y superiores representan los valores que se encuentran debajo del percentil 25 y por arriba del percentil 75, respectivamente.

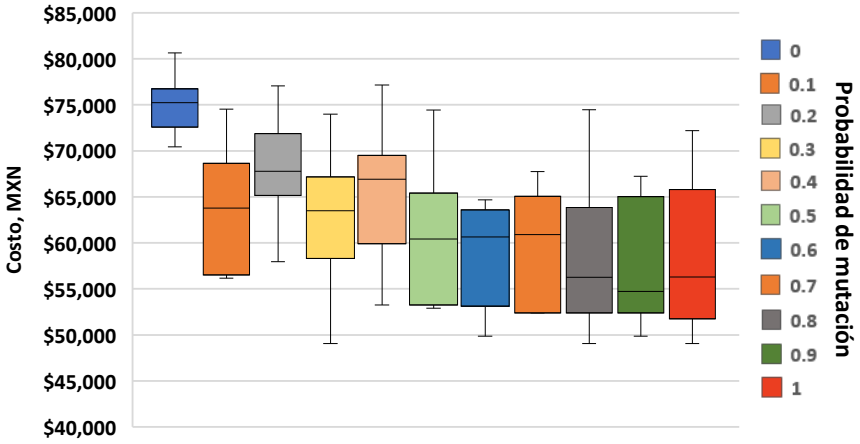


Figura 6.9 Costos de los soluciones encontradas por Algoritmos Genéticos en función de la probabilidad de mutación.

Tal como se puede apreciar, una probabilidad del 0% de mutación afecta de manera desfavorable el desempeño del algoritmo al punto que devuelve soluciones con un costo medio de $72,000 MXN. A pesar de que incrementar la probabilidad de mutación tiene un impacto benéfico en la calidad de las soluciones encontradas, esta influencia no es sostenida. Por ejemplo, el costo medio de las soluciones encontradas al utilizar probabilidades de mutación del 10, 30 y 40% resultó bastante similar, presentando un valor aproximado de $64,000 MXN. Para el caso de una probabilidad de mutación del 20% el costo medio se incrementó a $68,000 MXN, lo cual se atribuye a la variabilidad de resultados producida por la naturaleza estocástica de las metaheurísticas.

En lo que respecta al desempeño de AG con probabilidades de mutación del 50% o mayores, el algoritmo encontró soluciones con costos medios de entre $61,000 y $57,000 MXN. Estos resultados parecen indicar que una probabilidad de mutación alta resulta ser beneficiosa para este problema en particular; sin embargo, es necesario estudiar cómo tales probabilidades interactúan con el resto de hiperparámetros del problema.

Debido a la imposibilidad de estudiar todas las posibles combinaciones de hiperparámetros, en este ejemplo nos limitaremos a investigar la interacción entre la probabilidad de mutación y el tamaño de la población, así como entre la probabilidad de mutación y el tamaño del torneo. En la tabla 6.1 se presentan los costos medios obtenidos de muestras de 10 soluciones optimizadas al variar la probabilidad de mutación y el tamaño de la población. Las muestras fueron obtenidas al considerar un tamaño de torneo de 4, una probabilidad de cruce del 90% y 100 generaciones por ejecución. En este

caso se observa que la calidad de las soluciones incrementa con la probabilidad de mutación y el tamaño de la población. Para poblaciones pequeñas, el desempeño de AG resulta deficiente debido a que no cuenta con una variabilidad lo suficientemente alta como para explorar el espacio de configuraciones de manera efectiva. Adicionalmente, se observa que el incrementar el tamaño de la población de 300 individuos hasta 450 comienza a generar incrementos marginales de desempeño. Esto se observa más claramente en las muestras obtenidas con probabilidades de mutación del 80 y 100%. En el primer caso, la calidad de las soluciones disminuyó ya que el costo medio se incrementó de $58,775 a $59,332 MXN; mientras que el segundo caso el costo medio cambió de $57,187 a $56,751 MXN, lo cual representó una disminución del 0.8%.

Tabla 6.1. Costos medios. Probabilidad de mutación y tamaño de la población

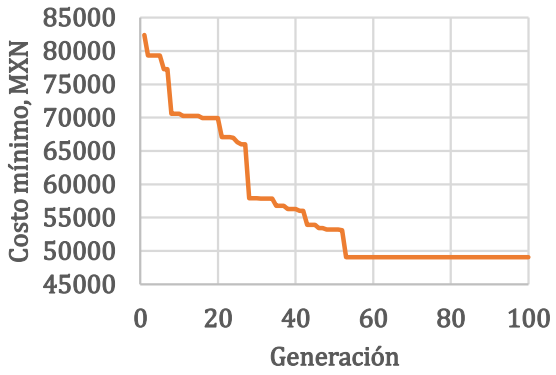
Tamaño de la población \ Probabilidad de mutación	0	0.2	0.4	0.6	0.8	1
50	$105,864	$81,654	$86,980	$85,431	$74,185	$81,815
150	$79,523	$79,749	$73,297	$67,520	$61,141	$61,965
300	$78,689	$66,645	$63,993	$67,302	$58,775	$57,187
450	$72,437	$68,487	$62,125	$59,184	$59,332	$56,751

En la tabla 6.2 se presentan los costos medios obtenidos al variar el tamaño del torneo y la probabilidad de mutación. Nuevamente, los costos medios fueron obtenidos de muestras de 10 soluciones optimizadas, las cuales se obtuvieron al ejecutar AG considerando un tamaño de población de 200 individuos, una probabilidad de mutación del 90% y 100 generaciones por ejecución. En este caso se observó que la calidad de las soluciones es relativamente indiferente al tamaño del torneo. Por ejemplo, para una probabilidad de mutación del 100%, el costo medio de las soluciones se mantuvo entre $61,928 y $66,576 MXN. Estos datos indican una diferencia entre los valores máximo y mínimo del 7%. Más aún, se observó que la combinación de valores que encontró las mejores soluciones fue aquella que empleaba una probabilidad de mutación del 60% y un tamaño del torneo de 3 individuos. Estos resultados contradicen en primera instancia lo hallado previamente, es decir, que el desempeño de AG mejoraba al incrementar la probabilidad de mutación; sin embargo, esto muestra lo complejas que son las interacciones entre los hiperparámetros de una metaheurística.

Tabla 6.2. Costos medios. Probabilidad de mutación y tamaño del torneo

Tamaño del torneo \ Probabilidad de mutación	0	0.2	0.4	0.6	0.8	1
2	$86,311	$72,441	$67,382	$61,152	$59,882	$65,062
3	$86,788	$72,903	$66,523	$60,960	$66,722	$61,928
4	$79,174	$71,148	$66,152	$68,738	$66,100	$66,576
5	$78,563	$72,827	$67,806	$71,890	$63,946	$65,517

Tomando en cuenta los resultados anteriores, se decidió tomar como valores de hiperparámetros una población de 450 individuos, un tamaño de torneo de 2 individuos, una probabilidad de cruce del 90%, una probabilidad de mutación del 80% y la creación de 100 generaciones. En este ejercicio el código de AG se ejecutó 10 veces. En la Figura 6.9 se muestra la trayectoria seguida por la ejecución que encontró la solución de mayor calidad, la cual presentaba un costo igual a $49,063 MXN. Los costos del resto de los óptimos locales encontrados también son presentados.



Costo Min, MXN					Media, MXN	Desviación estándar, MXN
E1	**E2**	**E3**	**E4**	**E5**	55,036	5,797
56,303	65,034	65,034	52,390	49,063		
E6	**E7**	**E8**	**E9**	**E10**		
52,390	56,303	49,063	52,390	52,390		

Figura 6.10 Costos de los soluciones encontradas por Algoritmos Genéticos y trayectoria de la solución de mayor calidad encontrada

Basándose en el costo medio de las 10 soluciones optimizadas encontradas por AG ($55,036 MXN), se puede concluir que el algoritmo es capaz de encontrar soluciones de alta calidad; sin embargo, la alta dispersión de los costos también señala una tendencia de AG por quedar atrapado en óptimos locales de baja calidad. El motivo detrás de este desempeño inconsistente está asociado a uno de los operadores empleados por AG y se describe en el capítulo 7. Para finalizar este capítulo, en la Figura 6.10 se muestra un esquema de la mejor solución encontrada por AG. Es importante señalar que el diseño obtenido no requería utilizar una separación distinta de estribos para el centro de la viga y las zonas próximas a los apoyos.

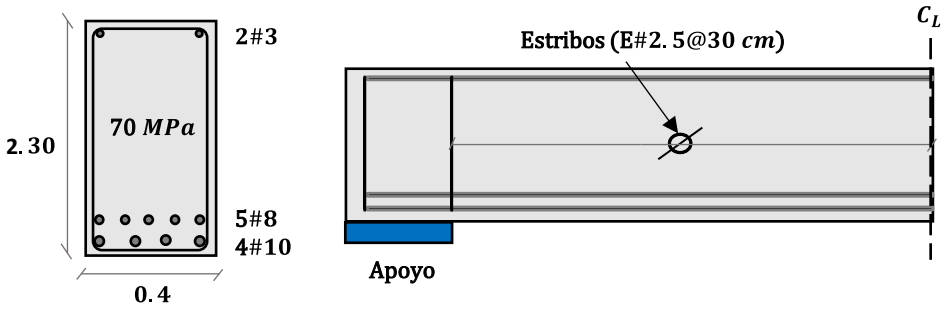


Figura 6.11 Diseño óptimo encontrado por Algoritmos Genéticos en la mejor solución

6.11. Conclusiones

Para el ejemplo de identificar el diseño óptimo de una viga de concreto reforzado se encontró que AG era capaz de encontrar soluciones con un costo de $49,063 MXN, valor que es un 15% menor que la solución de mayor calidad encontrada por una Búsqueda Aleatoria. No obstante estos resultados, caben señalar dos puntos muy importantes: en primera instancia, para alcanzar soluciones de alta calidad fue necesario emplear una probabilidad de mutación del 80%, un valor considerablemente alto si se compara con lo reportado en otros estudios (De Albuquerque *et al*., 2012; Degertekin *et al*., 2008; Kaveh & Dadfar, 2008). Por otra parte, también se encontró que AG presentaba cierta tendencia por quedar atrapado en soluciones de baja calidad ($65,034 MXN). Este problema de desempeño inconsistente tiene su origen en la forma en que AG se mueve por el espacio de configuraciones. En el capítulo 7 se resolverá el mismo problema, pero esta vez con el algoritmo metaheurístico conocido como Estrategias Evolutivas. Este último presenta operadores muy similares a AG; sin embargo, se observará que incluso pequeñas diferencias entre operadores pueden incrementar la calidad de las soluciones encontradas al problema.

Esta página ha sido intencionalmente dejada en blanco

7. Diseño óptimo de una viga por Estrategias Evolutivas

Se resuelve el ejercicio del capítulo 6 por medio de la metaheurística conocida como Estrategias Evolutivas (EE), la cual pertenece, junto con Algoritmos Genéticos (AG), a la familia de metaheurísticas conocida como Algoritmos Evolutivos. Los operadores que emplea EE siguen principios similares a aquellos introducidos por AG, recibiendo de igual manera los nombres de selección, cruce y mutación. Este ejemplo tiene como objetivo demostrar que algunos de los operadores empleados por AG en el capítulo anterior restringen las zonas del espacio de configuraciones que pueden ser alcanzadas, afectando a su vez el desempeño del algoritmo. Sobre la relación entre operadores y el espacio de configuraciones se hablará más en el capítulo 8.

7.1. Estrategias Evolutivas

Las Estrategias Evolutivas (EE) fueron creadas por Rechenberg (1973) en la Universidad Técnica de Berlín, y posteriormente desarrolladas por Bäck, Hoffmeister y Schwefel (1991). Al igual que AG, esta metaheurística aplica los principios de la selección natural en la optimización de problemas; sin embargo, los operadores utilizados difieren ligeramente entre ambos algoritmos.

Como ya se mencionó antes, las metaheurísticas que hacen uso de analogías con la selección natural fueron de los primeros algoritmos metaheurísticos en ser creados. En un principio, el algoritmo de EE era extremadamente básico. Esto es: haciendo uso de una única solución se daba origen a otra por medio de una mutación, de la solución original y la mutada se conservaba aquella que presentase mejor calidad y se repetía el proceso (T Bäck *et al*., 1991). Esta versión del algoritmo es conocida como Estrategias Evolutivas de dos miembros. Posteriormente, se introdujo el concepto de población, lo cual permitió generar nuevas soluciones por medio del cruce. A esta nueva versión se le denominó como EE - $(\mu + 1)$, donde μ hace referencia al tamaño de la población, la cual crea una única descendencia en cada iteración. En este algoritmo, la nueva solución creada es comparada con la peor de las

soluciones en la población actual, reemplazándola en caso de presentar una mejor calidad, con lo cual siempre hay μ individuos en la población.

Existen otras dos versiones de la metaheurística, las cuales fueron propuestas por Schwefel (T Bäck *et al*., 1991) en la década de los 70 y 80, y son llamadas como EE - $(\mu + \lambda)$ y EE - (μ, λ). En ambas nomenclaturas se utiliza μ para referirse al tamaño de la población, siendo λ el número de descendientes creados por iteración. En EE - $(\mu + \lambda)$ se añade un número λ de descendientes a una población conformada por μ individuos, eliminándose las λ peores soluciones, de tal manera que el tamaño de la población siempre es igual a μ. En la versión EE - (μ, λ) del algoritmo, la población de progenitores, conformada por μ individuos, da origen a una población de descendientes, conformada por λ individuos, siendo $\lambda > \mu$. En esta versión, siempre se elimina completamente a la población de progenitores, sin importar lo buena o mala que sea, y es reemplazada por la población de descendientes (Beyer & Schwefel, 2002).

En la Figura 7.1 se muestra el diagrama de flujo general de EE. Al ser EE una metaheurística basada en poblaciones, el primer paso del algoritmo es crear un número determinado de soluciones que conformarán su población inicial.

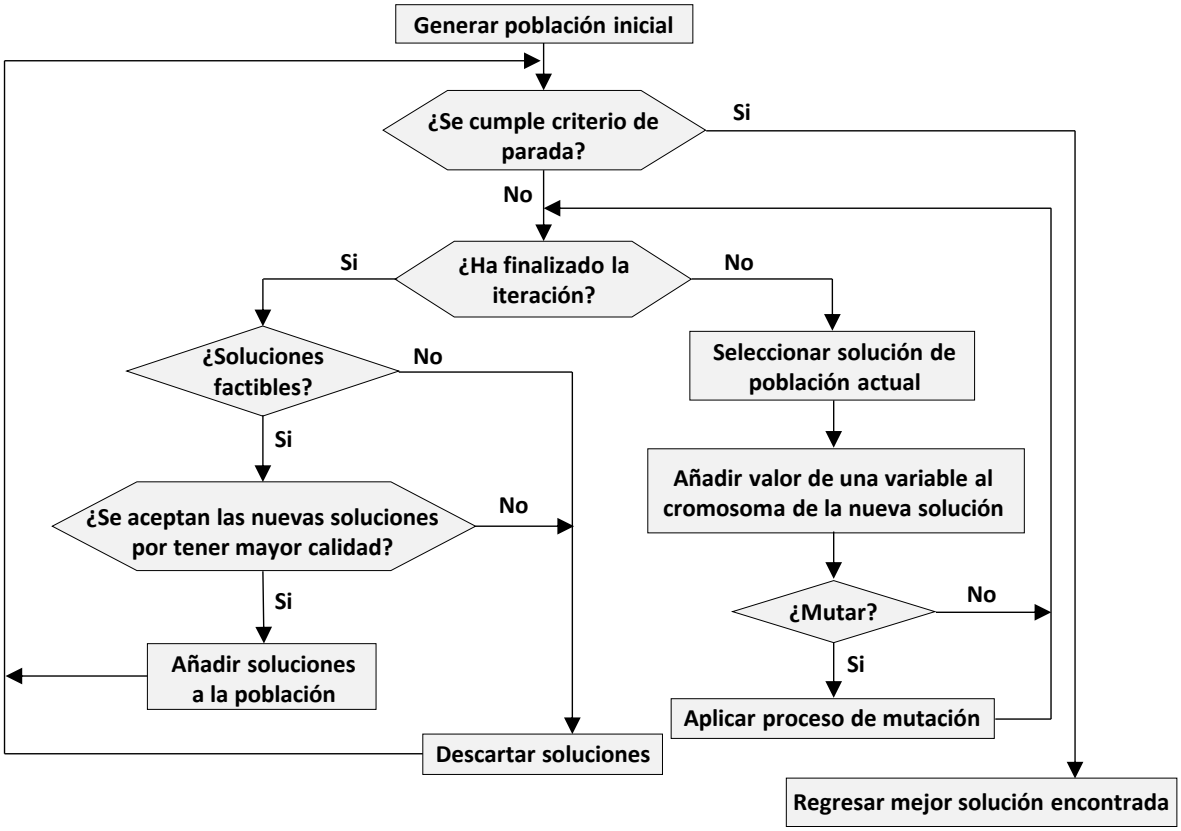


Figura 7.1 Diagrama de flujo de Estrategias Evolutivas

Por cada iteración del algoritmo, y con base en dicha población inicial, el algoritmo crea poblaciones nuevas con una calidad cada vez mayor por medio de los operadores selección, cruce y mutación. A diferencia de lo que ocurre en AG, donde una solución es creada uniendo bloques de variables de decisión provenientes de dos o más soluciones, en EE las nuevas soluciones son creadas tomando valores individuales de las variables de decisión de soluciones escogidas aleatoriamente de la población actual. Cada vez que se añade un nuevo valor al cromosoma de la solución en proceso de creación, existe una probabilidad de que dicho valor añadido mute. El proceso de creación de la nueva solución termina una vez que cada una de sus variables de decisión tiene un valor asignado; sin embargo, la iteración del algoritmo no finaliza hasta que se han creado un número $\lambda \geq 1$ de nuevas soluciones. Como se mencionó anteriormente, existen versiones de EE donde las nuevas soluciones creadas por iteración son comparadas con las peores λ soluciones de la población actual, reemplazando las primeras a las segundas si las nuevas soluciones presentan una calidad superior. También existe una versión donde las nuevas soluciones reemplazan completamente a la población anterior, es decir, no se realiza un proceso de comparación entre las calidades de la nueva población y la población actual. Una vez que se ha actualizado la población, ya sea por la adición de nuevos miembros o su reemplazo total, el algoritmo procede con la siguiente iteración hasta cumplir con su criterio de parada que suele ser un número determinado de iteraciones. Los elementos más importantes de EE son descritos a continuación.

Operador Selección. El proceso de selección empleado en EE es distinto a aquel utilizado en AG. En esta metaheurística, la selección hace referencia a la forma de determinar cuáles soluciones son conservadas y cuáles son eliminadas al finalizar una iteración, afectando de manera indirecta a la población de soluciones disponibles para efectuar el proceso de cruce. En EE, la selección presenta dos enfoques: conservar en la siguiente generación solo las mejores soluciones o eliminar completamente a la población de la generación anterior. En el caso de EE - $(\mu + 1)$ y EE - $(\mu + \lambda)$, el proceso de selección empleado es de naturaleza determinística debido a que solo las μ mejores soluciones son seleccionadas para continuar en la población. Las técnicas de selección que siguen este enfoque son conocidas como elitistas y son una condición suficiente para proveer de convergencia al algoritmo de búsqueda (Beyer & Schwefel, 2002). Por su parte, en EE - (μ, λ) las nuevas soluciones creadas en cada iteración del algoritmo reemplazan en su totalidad a la población actual, sin importar si existe o no una de calidad superior.

Operador Cruce. A diferencia de lo que ocurre en AG, donde se cruzan dos o más soluciones progenitoras para generar nuevas soluciones, el operador cruce empleado en EE da origen a una solución única. En este proceso, se escoge de manera aleatoria una de las μ soluciones presentes en la población actual y se toma el valor de una de sus variables de decisión para añadirlo en el cromosoma de la nueva solución generada (T Bäck *et al*., 1991; Thomas Bäck *et al*., 1997; Thomas Bäck & Schwefel, 1993). Dicho de otra manera, las nuevas soluciones generadas por este algoritmo se forman agrupando bloques unitarios de información de la población actual, una variable de decisión a la vez. Para ejemplificar este operador, en la Figura 7.2 se muestra de manera esquemática el proceso de creación de una solución nueva por EE a partir de una población. Para crear la solución nueva, cuyas características están almacenadas en el arreglo de color azul, se toma una solución de manera aleatoria

de la población actual, teniendo todas las soluciones de la población la misma probabilidad de ser seleccionadas. En este ejemplo, la segunda solución de la población fue seleccionada para transferir su valor b_3 a la nueva solución, el cual es almacenado en la tercera posición del arreglo que define la nueva solución. Como se puede observar, la nueva solución presenta también valores provenientes de las soluciones cuarta y quinta, mismos que están representados por los valores d_1 y e_2, respectivamente. Este proceso de tomar características unitarias de la población se repite hasta que todas las características de la nueva solución tienen un valor asignado.

Operador Mutación. En EE el operador mutación es aplicado a cada una de las variables que componen a la solución nueva en proceso de creación, pudiendo ocurrir esto de dos maneras: 1) por medio de una mutación ligera hacia un valor en la vecindad inmediata del valor actual de la variable, o 2) por medio de un cambio aleatorio en el que todo el alfabeto que compone a la variable presenta la misma probabilidad de ser escogido como valor nuevo. Estos dos procesos de mutación son ampliamente usados y pueden emplearse de manera combinada (Eiben, 1997; Fogel & Atmar, 1990; Michalewicz *et al*., 1992), teniendo cada uno su propia probabilidad de ocurrencia.

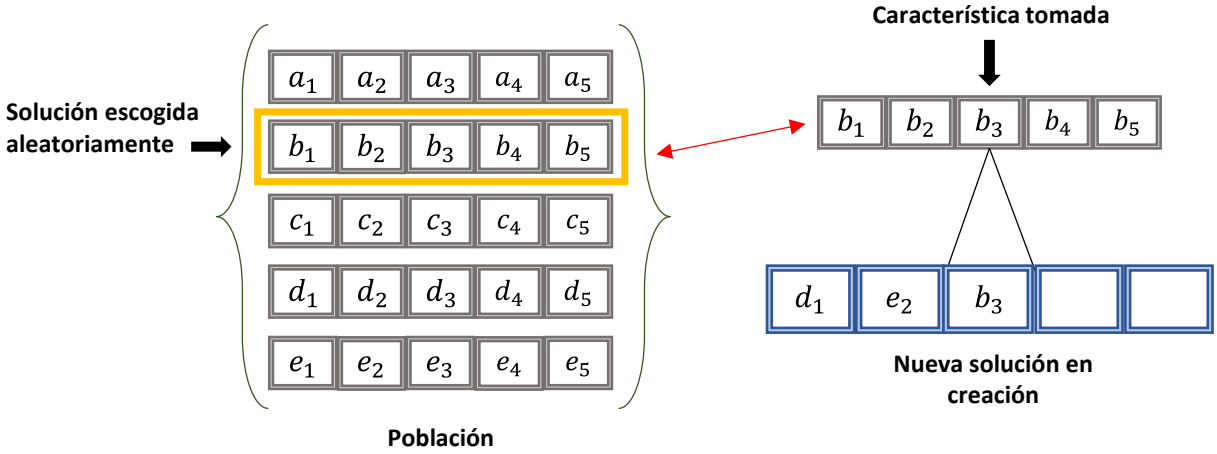


Figura 7.2 Operador cruce empleado en Estrategias Evolutivas

Como se puede observar, a pesar de que los AG y las EE pertenecen a la misma familia de metaheurísticas y comparten los mismos principios básicos, los enfoques que ambos algoritmos utilizan son diferentes, permitiendo esto considerarlos como dos metaheurísticas distintas. En las secciones siguientes se resuelve una vez más (igual que en el capítulo 6) el ejemplo de optimizar el diseño de una viga de concreto reforzado. Dado que este problema ya fue resuelto en el capítulo 6, en las secciones subsecuentes sólo se da un resumen breve de las características del problema.

7.2. Tipo de optimización y función objetivo

El problema de optimización que se resuelve en este capítulo es el mismo al presentado entre las secciones 6.2 a 6.6. Se trata de dimensionar una viga simplemente apoyada, sujeta a una carga variable uniformemente distribuida y considerado como variables de decisión las dimensiones de la sección transversal, las características del armado de refuerzo y el tipo de concreto utilizado.

El tipo de optimización es de tipo económica en la que se busca reducir el costo de fabricación de una viga de concreto armado. Los costos considerados son aquellos relacionados con el costo del acero, el concreto y la cimbra requerida, mismos que fueron mostrados en el capítulo 6.

7.3. Variables de decisión y parámetros del problema

Las variables de decisión en este problema son las mismas consideradas en el ejemplo del capítulo 6. A manera de resumen, estas consisten en la resistencia a compresión del concreto ($f'c$), el ancho (B) y alto (H) de la sección, el número (η) y diámetro (ϕ) del refuerzo longitudinal y el número (η_E), diámetro (ϕ_E) y separación (Sp) de los estribos de la viga. En cuanto a los parámetros del problema, estos son las cargas aplicadas, el recubrimiento de la sección, la luz de la viga y el ancho de los apoyos.

7.4. Restricciones del problema

Las restricciones en este problema son las mismas consideradas en el capítulo 6. Se recomienda al lector revisar la sección 6.6 para más detalles. De manera resumida, se toman en cuenta las condiciones de diseño que imponen las NTC para vigas de concreto, las incluyen aspectos geométricos, de resistencia a flexión, a cortante y limitaciones en deflexiones y agrietamiento.

7.5. Descripción del código de Estrategias Evolutivas

En la Figura 7.3 se muestra el diagrama de flujo del algoritmo de EE utilizado en este ejemplo, el cual corresponde a la versión conocida como EE - ($\mu + 1$) donde se mantiene constante el tamaño de la población a un número μ de individuos. El código fue ejecutado en el programa Matlab (MathWorks Inc, 2020) y se puede revisar de manera completa en el Anexo 5.1. Se recomienda leer de manera paralela el código junto con esta sección para una mejor comprensión de su funcionamiento. Las líneas 1 a 20 del código son las mismas instrucciones empleadas por el código de AG, presentado en el capítulo 6. En estas líneas se borra la memoria del programa, se definen los posibles valores de las

variables de decisión y se establecen los parámetros del problema. Entre las líneas 23 a 25 se definen los hiperparámetros empleados por EE. Obsérvese que en esta metaheurística se redujo el número de parámetros de búsqueda empleados, siendo necesario definir sólo tres valores:

- Tamaño de la población (μ).
- El número de generaciones máximas ($GenMax$).
- Probabilidad de mutación (PM).

Entre las líneas 28 y 32 se crean matrices y vectores que almacenan información durante el proceso de optimización. La matriz *Poblacion* es aquella donde se guardarán las características de las soluciones de la población actual, *Vv* es una matriz que almacena todos los valores permitidos de las variables de decisión del problema, mismos que son asignados entre las líneas 39 a 51 del código. Asimismo, *Costos*, *Solucion* y *Cmin* son vectores que registran los costos de la población actual, la nueva solución creada en la iteración y los costos de las soluciones de mayor calidad, presentes en la población actual de cada iteración, respectivamente.

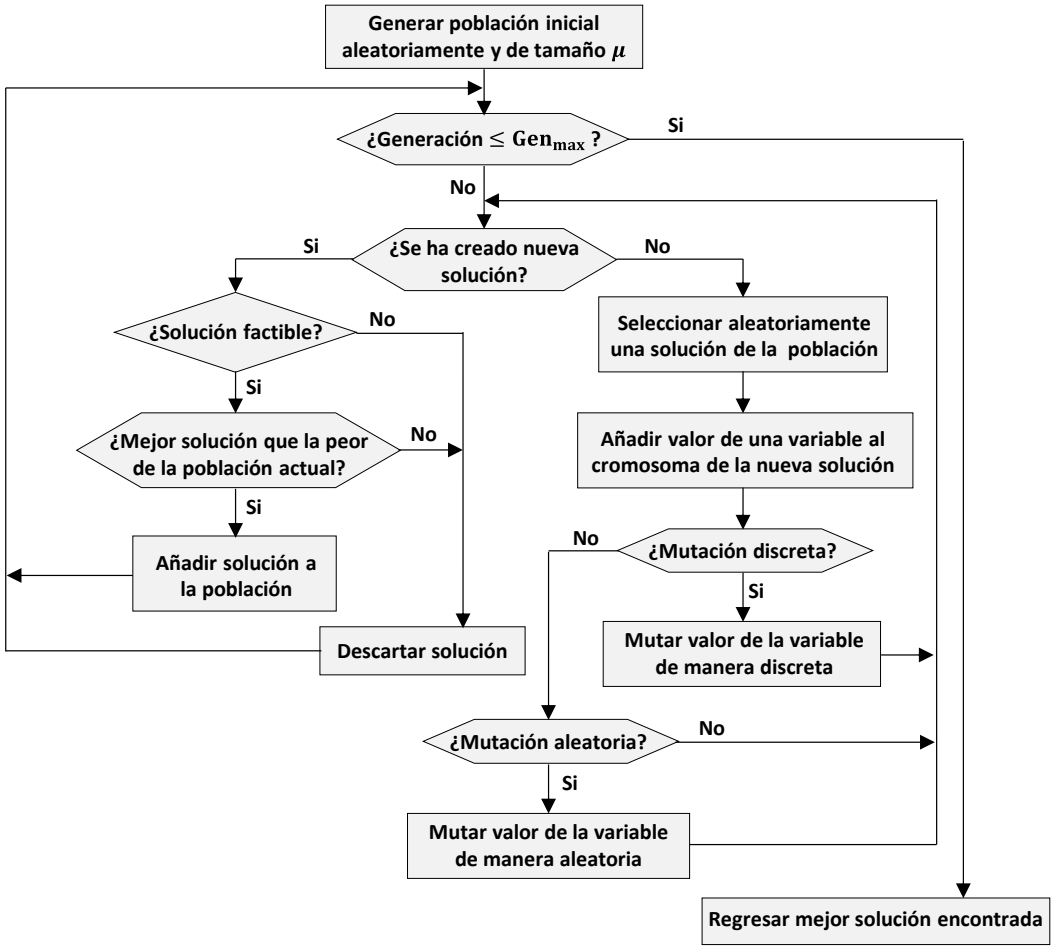


Figura 7.3 Diagrama de flujo de EE – (μ+1) empleado en este problema

El código de EE - $(\mu + 1)$ se desarrolla entre las líneas 55 a 124 del código. Obsérvese que, al solo crearse una solución por generación, el código de EE se simplifica considerablemente en comparación con el de AG. El primer paso del algoritmo es crear de manera aleatoria una población inicial factible de tamaño μ. Esto se realiza por medio de un ciclo *while* que abarca de la línea 55 hasta la 83. Este ciclo *while* inicia creando una solución compuesta por 14 valores tomados de manera aleatoria de la matriz *Vv*. Dicha solución es posteriormente sometida a la verificación de las restricciones consideradas; de manera similar a lo realizado en el ejemplo anterior, aquí se utiliza la función denominada como *ComprobacionesAG()* para esta tarea. Si la solución resulta ser factible, en la línea 78 se llama la función *CostoViga()* para determinar su costo y el resultado es almacenado en el vector *Costos*. Las funciones *ComprobacionesAG()* y *CostoViga()* empleadas en este código son las mismas a las ocupada en el ejemplo del capítulo 6, por tal motivo se omite una descripción detallada de su funcionamiento y se recuerda al lector que el código de ambas funciones puede ser revisado en su totalidad en el Anexo 4.2 de este libro.

Una vez creada la población inicial factible, esta es utilizada para generar nuevas soluciones en las iteraciones subsecuentes del algoritmo desarrolladas en el ciclo *while* que abarca de las líneas 90 hasta 124. Es importante recordar que el operador cruce empleado por EE difiere de aquel usado en GA en que el primero crea soluciones introduciendo valores de una variable de decisión a la vez, mientras que el último agrupa cadenas de características de soluciones presentes en la población actual. Para codificar el operador cruce de EE se utiliza un ciclo *for* que se desarrolla entre las líneas 92 a 104. En dicho ciclo, se toma el valor de la característica *i+1* de una solución seleccionada de manera aleatoria y es introducida en la nueva solución en el proceso de creación. Adicionalmente, cada vez que se agrega el valor de una variable a la nueva solución, existe una probabilidad PM de que dicho valor presente una mutación. Como se mencionó antes, existen dos tipos de mutaciones a utilizar: una que cambia el valor de la variable hacia otro valor en la vecindad inmediata del mismo; y una mutación que realiza un cambio aleatorio hacia cualquier otro valor del intervalo para el cual existe la variable correspondiente. El primer caso es una mutación de corto alcance que sirve para refinar las características de las nuevas soluciones. Esta mutación es ejecutada en las líneas 96 y 97 por medio de la función *EEMutacion()*. Obsérvese que esta mutación es idéntica a aquella utilizada en el código de AG, la cual es explicada de manera esquemática en la Figura 6.7. Si la variable no presentó la primera mutación, existe una probabilidad $(1 - PM)$ de presentar la mutación aleatoria. En este segundo proceso de mutación, el cual es realizado por la función *EEMutacion2()* en las líneas 100 y 101, la variable de decisión puede cambiar a cualquier otro valor siguiendo una distribución de probabilidad uniforme. Los códigos de ambas funciones pueden ser consultados en el Anexo 5.2 de este libro.

Cuando se ha formado la nueva solución, esta es sometida a las comprobaciones pertinentes (líneas 106 y 107) y, si resulta ser un diseño factible, su costo es evaluado (líneas 112 y 113). Solo cuando se crea una solución factible el algoritmo da por finalizada la iteración actual. Entre las líneas 115 a 119 la nueva solución es comparada con la peor de las soluciones presentes en la población actual por

medio del valor de su función objetivo, si la primera resulta peor que la segunda, la nueva solución es descartada, en caso contrario, la nueva solución reemplaza a la peor de la población actual. Las instrucciones por las cuales se registra el nuevo costo mínimo de la población y se reinicia la variable *Solucion* ocurren entre las líneas 120 a 122. Este proceso de creación de soluciones y actualización de la población se repite hasta que el algoritmo ha realizado un número de iteraciones definido por la variable $GenMax$.

7.6. Espacio de configuraciones

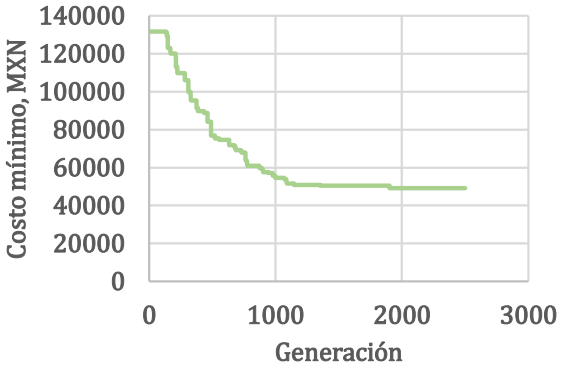
El tamaño del espacio de soluciones de este problema fue determinado en el capítulo 6. En total, es posible crear $5.90x10^{12}$ diseños de vigas con las variables de decisión consideradas. Resulta obvio que solo una fracción de estas son soluciones factibles. Tal como se presenta en el capítulo anterior, la muestra generada por una Búsqueda Aleatoria (BA) indica que las soluciones factibles con costos inferiores a los 57,000 MXN son extremadamente infrecuentes. Nuevamente, aquí se buscará que el algoritmo de optimización brinde diseños cuyo costo sea menor al mejor resultado obtenido por BA.

7.7. Resultados y discusión

Variando los parámetros de búsqueda de EE, se encontró que la combinación $\mu = 12$ individuos, $GenMax = 2{,}500$ generaciones y $PM = 0.5$ era la que brindaba resultados superiores. El algoritmo fue ejecutado un total de 10 veces, encontrándose un diseño con un costo de $49,063 MXN, el cual resulta ser también el mejor diseño encontrado por AG.

En la Figura 7.4 se muestra la trayectoria seguida por el algoritmo en su mejor ejecución en conjunto con el resto de los valores de los óptimos locales encontrados. Evidentemente, EE parece ser capaz de localizar soluciones de mayor calidad de una manera mucho más consistente que AG. Para verificar este resultado, se generaron dos muestras adicionales de 50 óptimos locales cada una, las cuales se obtuvieron por ambas metaheurísticas.

Los estadísticos descriptivos de ambas muestras son presentados gráficamente por medio diagramas de cajas y bigotes en la Figura 7.5. Como se puede observar, EE presenta tiende a converger a soluciones con calidad similar, las cuales presentan costos de entre $49,000 MXN y $50,300 MXN, aproximadamente. Por su parte, las calidades de los diseños obtenidos por AG existen en un intervalo mucho más amplio, encontrando este algoritmo soluciones con costos entre $49,000 MXN y $68,000 MXN. Por su parte, la versión originalmente considerada de AG mostró un costo medio de $56,883 MXN.



Costo Min, MXN					Media, MXN	Desviación estándar, MXN
E1	**E2**	**E3**	**E4**	**E5**	49,461	338
49,862	49,727	49,325	49,170	49,331		
E6	**E7**	**E8**	**E9**	**E10**		
49,727	49,063	49,170	50,014	49,218		

Figura 7.4 Óptimos locales encontrados por Estrategias Evolutivas

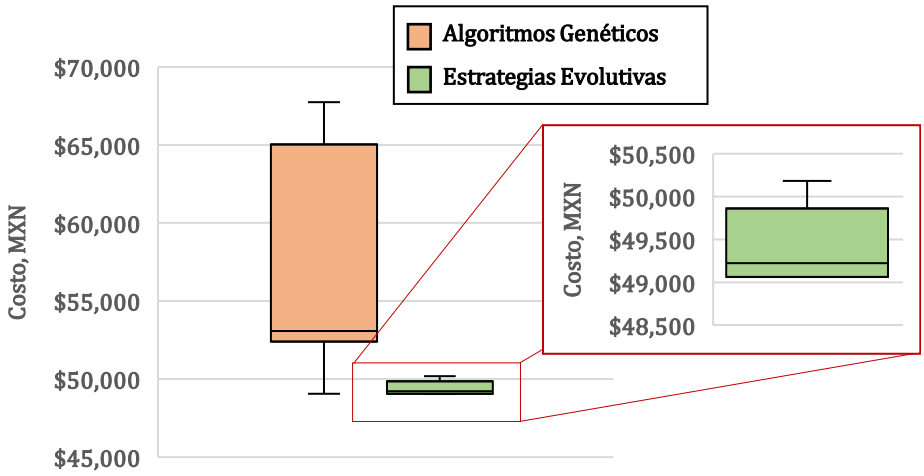


Figura 7.5 Comparación de los desempeños de AG y EE

Para comprender el motivo de los desempeños diferenciados de ambos algoritmos es necesario considerar los siguientes factores: en primera instancia, los diseños de vigas creados de manera aleatoria tienden a ser ineficientes, lo que implica que utilizan cuantías de armado elevadas, resistencias de concreto altas y

secciones transversales de medidas considerables para garantizar el cumplimiento de las restricciones. Al avanzar el proceso de optimización, los volúmenes de materiales comenzarán a disminuir para abaratar los costos de los diseños, mientras que la resistencia del concreto tenderá a aumentar para compensar el decremento en la capacidad de los diseños. Asimismo, por la aplicación constante de los operadores selección, cruce y mutación, las características presentes en los individuos de la población comenzarán a acentuarse hasta alcanzar un punto de equilibrio donde la diversidad resulte limitada. En tal situación, el operador de mutación sería responsable de añadir diversidad a la población; sin embargo, el operador utilizado en este ejemplo tenía la limitación de que solo podía modificar una característica de los diseños a la vez, cambiando su valor únicamente a uno cercano, ya sea el inmediatamente superior o inferior al valor actual. Debido a estas condiciones de operación, si ocurría el caso de que la población convergiera a diseños con elevados volúmenes de materiales en conjunto con una resistencia baja del concreto, existía el riesgo de que el algoritmo quedara atrapado en tales óptimos locales. Para escapar de dicha situación se requeriría una disminución de los volúmenes de materiales en simultáneo con un incremento de la resistencia del concreto. Al no contar el operador mutación con dicha opción, AG solo podía tomar dos caminos: incrementar la resistencia del concreto, encareciendo el diseño y afectando de manera negativa su probabilidad para ser seleccionado como progenitor; o disminuir los volúmenes de materiales y correr el riesgo de que la resistencia el elemento no fuera suficiente para cumplir con las restricciones de diseño.

Para demostrar la influencia del operador mutación en el desempeño de AG, en la figura 7.6 se presenta una comparativa de desempeños al considerarse operadores mutación que modifican una característica (AG_mut_1) y dos características (AG_mut_2) del diseño. La modificación del operador mutación para conseguir alterar dos características del diseño cada vez que es llamado resulta bastante sencilla, y se deja como tarea para el lector. La comparativa entre ambas versiones de AG se realiza por medio de los diagramas de cajas y bigotes obtenidos al crear 50 soluciones por ambos algoritmos. Cabe señalar que la distribución mostrada para la versión de AG_mut_1 es la misma que se presentó en la figura 7.5. Como se puede observar, la modificación del operador mutación permitió alcanzar mejores desempeños ya que el costo medio de las soluciones encontradas por la versión de AG que alteraba dos características a través de la mutación se redujo a $52,707 MXN. Este cambio en el costo medio de las soluciones representa una disminución del 7.3% en comparación con la versión de AG que solo modificada una característica a través del operador mutación.

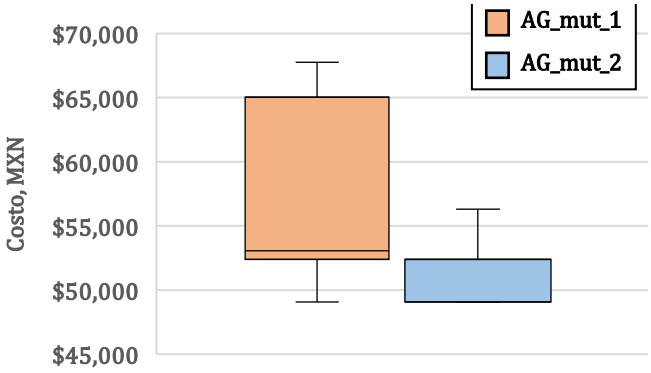


Figura 7.6 Comparación de desempeños de AG en función del operador mutación

A pesar de la mejora del desempeño, es importante señalar que el costo medio de AG aún se encuentra un 6% por arriba del costo medio de las soluciones halladas por EE. Para mejorar aún más el desempeño de AG, el lector podría intentar modificar los hiperparámetros del algoritmo, es decir, variar el tamaño de la población o el número de iteraciones, además de alterar las probabilidades de mutación y cruce. Sin embargo, resulta necesario mencionar que los resultados previamente obtenidos sobre la interacción de los valores de los hiperparámetros ya no son igualmente válidos para esta nueva versión de AG. Para comprender mejor esto, considere una versión de AG con un operador mutación que sea capaz de modificar todas las características del diseño hacia cualquier valor considerado en las variables de decisión. En tal situación, una probabilidad de mutación del 100% generaría que el algoritmo funcionase como una búsqueda aleatoria, con lo cual su desempeño se vería afectado de manera negativa. No obstante, en este ejemplo se utilizó una probabilidad de mutación elevada y se observó un desempeño adecuado. Esto, que en primera instancia puede sonar contradictorio, se vuelve bastante razonable si se recuerda que el operador mutación empleado originalmente tenía un mínimo de alcance y solo afectaba una característica, lo cual provocaba que inclusive una probabilidad de mutación alta impactara de manera limitada la estabilidad del algoritmo. Por lo tanto, se puede concluir que el desempeño de una metaheurística depende de la compleja interacción entre las características del problema, los valores empleados en los hiperparámetros y el funcionamiento de los operadores.

7.8. Conclusiones

Recordando el teorema de *No Free Lunch* (Wolpert & Macready, 1997), este capítulo no pretende afirmar que EE es un algoritmo superior a AG, en cambio, se brinda evidencia empírica sobre cómo los operadores pueden influenciar fuertemente el desempeño de una metaheurística. Este resultado, que puede parecer trivial a primera vista, abre la puerta a reflexionar sobre la influencia que tienen los operadores en el propio relieve del espacio de configuraciones. Incluso más, la cuestión resulta ser tan relevante que es la parte medular de la metaheurística presentada en el capítulo 8, conocida como Búsqueda por Vecindario Variable (BVV) o *Variable Neighborhood Search* (*VNS*).

Esta página ha sido intencionalmente dejada en blanco

8. Calibración de un modelo de elementos finitos

En este capítulo se realiza un ejemplo de calibración de un modelo de elementos finitos por medio de la metaheurística conocida como Búsqueda por Vecindario Variable (BVV). La calibración de un modelo de elementos finitos consiste en la modificación de los parámetros que controlan la respuesta del modelo matemático de una estructura con el objetivo de buscar una coincidencia entre dicha respuesta y aquella obtenida en campo o por ensayos experimentales. Tal como se observó en el capítulo 7, los operadores empleados por una metaheurística definen las soluciones que pueden ser alcanzadas por el algoritmo en cada movimiento. Basándose en este principio, BVV utiliza una serie de operadores, o vecindarios, definidos por el usuario para explorar de manera eficiente el espacio de configuraciones.

8.1. Búsqueda por Vecindario Variable

Búsqueda por Vecindario Variable (en inglés *Variable Neighborhood Search, VNS*) es un algoritmo propuesto por Mladenović *et al*. (1997) y actualmente existen muchas versiones de esta metaheurística en la literatura (Pierre Hansen *et al*., 2017). BVV es un algoritmo basado en una solución única que requiere la definición de un número k de vecindarios ($N_1, \dots, N_k$). El algoritmo inicia generando una primera solución $\boldsymbol{x}$ de manera aleatoria o por medio de otro procedimiento. Posteriormente, se genera de manera aleatoria una nueva solución $\boldsymbol{x}'$ en alguno de los vecindarios de la solución actual $\boldsymbol{x}$ definidos anteriormente. A la solución $\boldsymbol{x}'$ se le aplica un algoritmo de Búsqueda Local (BL) con la esperanza de encontrar una solución $\boldsymbol{x}''$ cuya calidad sea superior a la de $\boldsymbol{x}$, es decir, $f(\boldsymbol{x}'') < f(\boldsymbol{x})$. Si no es posible hallar una solución $\boldsymbol{x}''$ de mayor calidad, se repite el proceso, pero con otro vecindario. Estos pasos se repiten hasta que ya no sea posible encontrar una nueva solución $\boldsymbol{x}''$ en ninguno de los k vecindarios definidos (Boussaïd *et al*., 2013; Duarte *et al*., 2018).

En la Figura 8.1 se presenta el diagrama de flujo de BVV. Del diagrama de flujo se desprende que BVV explora el espacio de soluciones por medio de dos movimientos, uno en el que se aplica un procedimiento de mejora y otro conocido como *shaking*. En la fase de mejora, como su nombre lo indica, se busca aumentar la calidad de la solución actual $\boldsymbol{x}$, mientras que el movimiento conocido como *shaking* se aplica sobre la solución $\boldsymbol{x}'$ y se utiliza para salir de óptimos locales (P Hansen & Mladenović, 2018; Pierre Hansen *et al*., 2017).

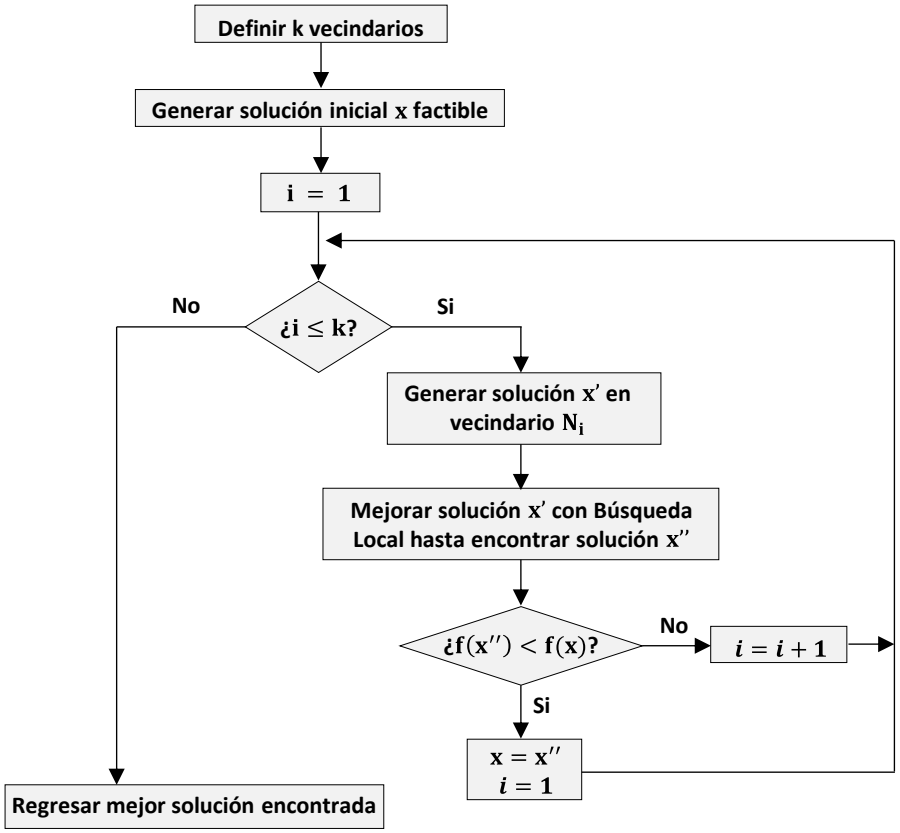


Figura 8.1 Diagrama de flujo del código de Búsqueda por Vecindario Variable

En la Figura 8.2 se muestra de forma esquemática el proceso de optimización de BVV. Como se mencionó antes, el algoritmo explora el espacio de configuraciones partiendo de una solución inicial $\boldsymbol{x}$ (denotada como un punto rojo), posteriormente se exploran los k diferentes vecindarios definidos por el usuario en busca de una mejor solución x' (denotada como un punto negro). Cuando tal solución es encontrada, se aplica una BL para crear una solución $\boldsymbol{x''}$ de mayor calidad (denotada como un punto anaranjado) y que será utilizada como punto de partida para la siguiente iteración del algoritmo.

Para comprender mejor el funcionamiento de BVV es necesario definir de manera precisa el término de vecindario y cómo este afecta al proceso de búsqueda. De manera formal, un vecindario se puede definir de la siguiente forma:

$$N(\boldsymbol{x^c}) = \{\boldsymbol{x^n} \in \bar{\boldsymbol{X}} \mid \delta(\boldsymbol{x^c}, \boldsymbol{x^n}) \leq \alpha\} \tag{8.1}$$

donde $\boldsymbol{x^c}$ es el punto alrededor del cual se define el vecindario, $\boldsymbol{x^n}$ es otro punto cualquiera, $\bar{\boldsymbol{X}}$ es el conjunto de soluciones factibles, α es un valor positivo dado y $\delta(\boldsymbol{x^c}, \boldsymbol{x^n})$ es una función cuasi métrica o función de distancia(Pierre Hansen *et al*., 2017). La función δ también puede ser representada de la siguiente forma (Winker & Maringer, 2007):

$$\delta(\boldsymbol{x^c}, \boldsymbol{x^n}) \rightarrow \|\boldsymbol{x^n} - \boldsymbol{x^c}\| \tag{8.2}$$

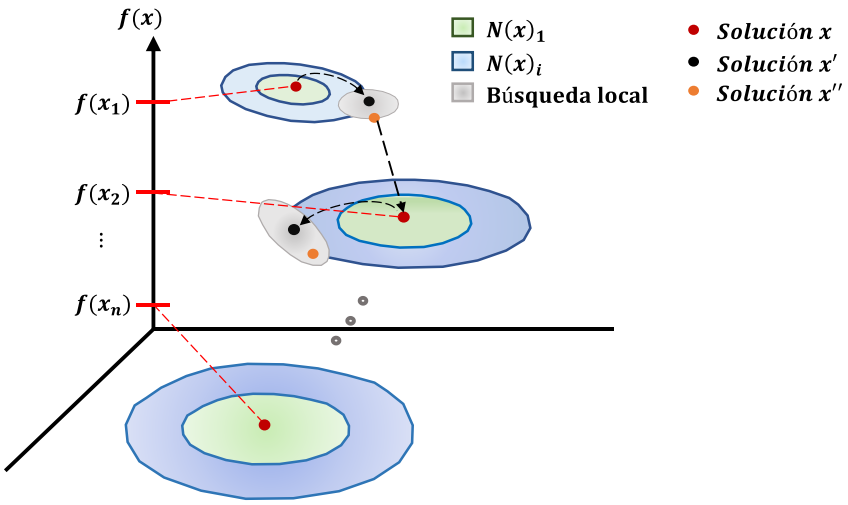


Figura 8.2 Esquema del patrón de búsqueda de Búsqueda por Vecindario Variable. Basada en (P Hansen & Mladenović, 2018)

De manera simplificada, el vecindario de $\boldsymbol{x^c}$ son todas aquellas soluciones $\boldsymbol{x^n}$ que se encuentran dentro del alcance de los operadores empleados por el algoritmo de búsqueda. El establecer la vecindad de una solución adquiere una relevancia especial al momento de definir conceptos como "máximos" o "mínimos". En primera instancia, y recordando las clases básicas de cálculo, el mínimo de una función se puede definir llanamente como aquel punto donde la derivada de la función es igual a cero. Sin embargo, tal como señala Jones (1995), una definición más apropiada para este campo sería la de "aquel punto que se encuentran por debajo de todos aquellos de su vecindad", por su parte, el concepto de máximo puede definirse de la misma manera, pero en términos contrarios. La idea de que los vecindarios (y, por lo tanto, los operadores) definen los mínimos y máximos del espacio de soluciones

fue abordada por Jones en su artículo titulado “One operator, one landscape” (Jones, 1995) publicado en el año de 1995. En este trabajo, Jones afirma que la función objetivo no es quien define la forma del relieve del espacio de búsqueda sino más bien es el operador utilizado por el algoritmo. Por lo tanto, no solo el relieve es independiente de la función objetivo, sino que además existe uno por cada operador empleado.

Para ejemplificar esta idea, en la Figura 8.3 se muestra una curva que representa los valores de la función objetivo $f(\boldsymbol{x})$. Como se puede observar, la solución $\boldsymbol{x}$ se encuentra en un mínimo local. Sin embargo, esta situación puede variar en función del operador que se utilice para alcanzar la siguiente solución $\boldsymbol{x}'$. En primera instancia, imaginemos que se utiliza un operador que solo permite alcanzar los puntos de color rojo. En este caso, es claro que la solución $\boldsymbol{x}$ conserva su título de mínimo local. Si se cambia a un segundo operador, que ahora permita alcanzar los puntos en azul, la situación daría un giro, pues se observaría que $\boldsymbol{x}'$ alcanza un mínimo y $\boldsymbol{x}$ en realidad es un máximo. Por último, si se emplea un tercer operador, el cual solo permita moverse a los puntos de color verde, se volvería al caso inicial, es decir, que $\boldsymbol{x}$ se encuentra de nueva cuenta en un mínimo local.

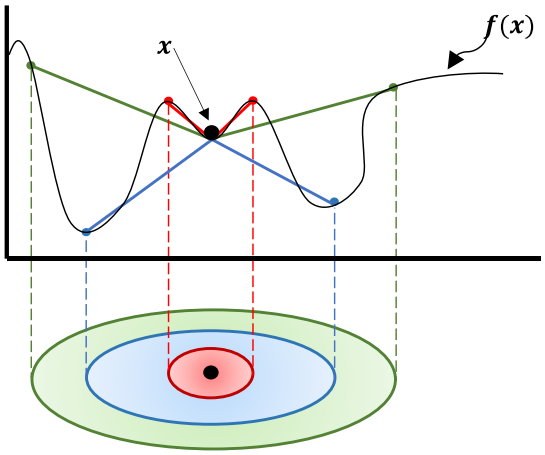


Figura 8.3 Esquema del patrón de búsqueda de Búsqueda por Vecindario Variable. Basada en (P Hansen & Mladenović, 2018)

Como se observa, el utilizar múltiples vecindarios en BVV es la estrategia que le permite al algoritmo salir de óptimos locales. Si hay algún vecindario que indique que la solución se encuentra en un mínimo, utilizando otro vecindario se podría permitir algún movimiento que lleve a mejores soluciones (Blum & Roli, 2003). De manera general, las ideas, sobre las cuales se basa BVV, se pueden enunciar de la siguiente manera (P Hansen & Mladenović, 2018; Pierre Hansen *et al*., 2009):

1. Un mínimo local de un vecindario no es necesariamente el mismo para otra estructura de vecindario.
2. Un mínimo global es un mínimo local con respecto a todas las estructuras de vecindario posibles.

3. Para muchos problemas, el mínimo local de uno o varios vecindarios se encuentran relativamente cerca entre ellos.

Como ya se mencionó, BVV utiliza un conjunto de k vecindarios ($N_i, (i = 1, \dots, k)$). La manera en que estos vecindarios son utilizados en el proceso de búsqueda puede ser de manera aleatoria, aunque una secuencia ordenada con una cardinalidad creciente, como la siguiente, también es común (Blum & Roli, 2003):

$$|N_1| < |N_2| < \cdots < |N_k| \tag{8.3}$$

El tamaño del vecindario también es una cuestión importante ya que define la manera en cómo se comporta el algoritmo. Mientras que los vecindarios grandes permiten un desplazamiento rápido a través del espacio de soluciones, los vecindarios pequeños permiten enfocar la búsqueda en una zona reducida (Winker & Maringer, 2007). De igual manera a como ocurre con BL, aquí existen dos criterios para tomar la siguiente solución $\boldsymbol{x}'$ durante el proceso de búsqueda. La primera se conoce como *"best improvement"* que consiste en tomar la mejor solución $\boldsymbol{x}'$ de todas las posibles. Por supuesto, esto implica revisar todas las soluciones que existen en todos los vecindarios considerados. Ello es una estrategia costosa en términos computacionales. Por otra parte, se tiene la estrategia conocida como *"first improvement"*, en la cual se toma la primera solución $\boldsymbol{x}'$ que se encuentra y que brinde una mejora en el valor de la función objetivo. Por otra parte, es bien sabido que la calidad de la solución inicial afecta el desempeño de los algoritmos de búsqueda local. Por esa razón, existen dos formas de localizar una solución inicial: por medio de una búsqueda aleatoria o utilizando algún algoritmo codicioso (Duarte *et al*., 2018).

Por último, a continuación se describen de manera breve algunas de las versiones que actualmente existen de BVV (Duarte *et al*., 2018; P Hansen & Mladenović, 2018; Pierre Hansen *et al*., 2017):

- **Variable Neighborhood Descent (VND)**: En esta versión, el cambio de los vecindarios durante el proceso de búsqueda se realiza de forma determinística.
- **Reduced Variable Neighborhood Search (RVNS)**: Esta versión omite la fase de BL.
- **Basic Variable Neighborhood Seacrh (BVNS)**: En BVNS el cambio de vecindario se realiza por métodos tanto estocásticos como determinísticos.
- **Skewed Variable Neighborhood Search (SVNS)**: Esta variante permite aceptar soluciones peores en función de que tan alejada está la nueva posible solución de la actual.
- **Variable Neighborhood Decomposition Search (VNDS)**: Es una extensión de BVV a dos niveles, en los cuales se descompone el problema.
- **Fixed Neighborhood Search (FNS)**: En esta versión solo se utiliza un vecindario para generar una perturbación que permite salir de mínimos locales. Esto tiene dos ventajas:
  I. Mejor efectividad: Algunos atributos de la solución actual con buenos valores se mantienen.
  II. Mejor eficiencia: La siguiente aplicación de BL tendrá un mejor desempeño, ya que la solución actual se encontrará en un valle más profundo del espacio de soluciones.

8.2. Optimización de estructuras complejas

En el capítulo 5 se observó que el optimizar el diseño de una armadura era una tarea complicada debido a que se requería programar el método de rigidez para resolver cualquier armadura candidata a solución. En aquellos casos en que la estructura, objeto de optimización, es más compleja se vuelve necesario crear códigos adicionales que permitan tomar en cuenta aspectos más refinados como las relaciones esfuerzo – deformación de los materiales, los comportamientos no lineales, la acumulación de daño estructural, etc. Para evitar el inconveniente de tener que elaborar de manera artesanal un código tan complejo, es posible recurrir a programas de análisis por elementos finitos, mismos que permiten modelar estructuras con diferentes grados de detalle y de manera confiable.

Uno de los programas de elementos finitos más utilizados en el área de la investigación hoy en día es OpenSees (PEER, 2006) (del inglés, *Open System for Earthquake Engineering Simulation*) el cual es un programa de licencia libre que permite analizar estructuras y modelos geotécnicos bajo cargas gravitatorias y dinámicas. Esta herramienta tiene sus orígenes en la tesis doctoral de Francis McKenna (1997). Actualmente, gracias a las aportaciones de investigadores de diferentes universidades y al patrocinio y desarrollo del programa PEER (del inglés, *Pacific Earthquake Engineering Research Center*) de los Estados Unidos de Norteamérica, OpenSees cuenta con un gran abanico de comandos para definir materiales, elementos y procedimientos de análisis. Debido a esto, es recomendable consultar el libro de OpenSees de Velasco y Guerrero (2020), disponible en línea, para comenzar a utilizar esta herramienta.

Para demostrar el uso, de manera conjunta de los algoritmos metaheurísticos y OpenSees, en este capítulo se realiza la calibración de un modelo de elementos finitos teniendo como base el modelo de una columna de suelo compuesta por diferentes estratos de arcillas. Un ejemplo similar a este fue presentado por Gu *et al*. (2012), donde se modeló una columna de suelo sometida a una excitación sísmica. En este caso se considera que se cuenta con un registro de la respuesta dinámica de la columna de suelo y se desea elaborar un modelo de elementos finitos que represente fielmente dicha respuesta. La columna de suelo cuenta con siete estratos y es mostrada de manera esquemática en la Figura 8.4. Asimismo, la columna es sometida a las aceleraciones de la componente Este-Oeste del sismo de Michoacán, México, del 19/sept/1985, registrado en la estación SCT. Para modelar los estratos del suelo, se utilizaron elementos finitos rectangulares con una relación esfuerzo – deformación definida por el material *PressureIndependMultiYield*, disponible en OpenSees. La información del material se puede revisar ampliamente en el manual de la referencia (OpenSees, 2021).

El material *PressureIndependMultiYield* presenta dos comportamientos distintos: uno de tipo elástico al ser sometido a cargas gravitaciones, y otro elastoplástico frente a cargas dinámicas. Los datos que controlan la respuesta del material son: la densidad del suelo (ρ), el módulo a cortante (G), el módulo de compresibilidad o módulo de Bulk ($Bulk$), la cohesión del terreno (c), la deformación a cortante para el máximo esfuerzo (γ_{max}) y la presión de confinamiento efectiva ($refPress$). El diagrama esfuerzo – deformación del comportamiento elastoplástico del material es mostrado en la Figura 8.5. Se puede apreciar que el material presenta una relación esfuerzo – deformación no lineal,

la cual podría resultar complicada de modelar sin el uso de elementos finitos. Para hacerlo de manera eficiente, la estrategia más conveniente es apoyarse de programas especializados en el análisis estructural como OpenSees.

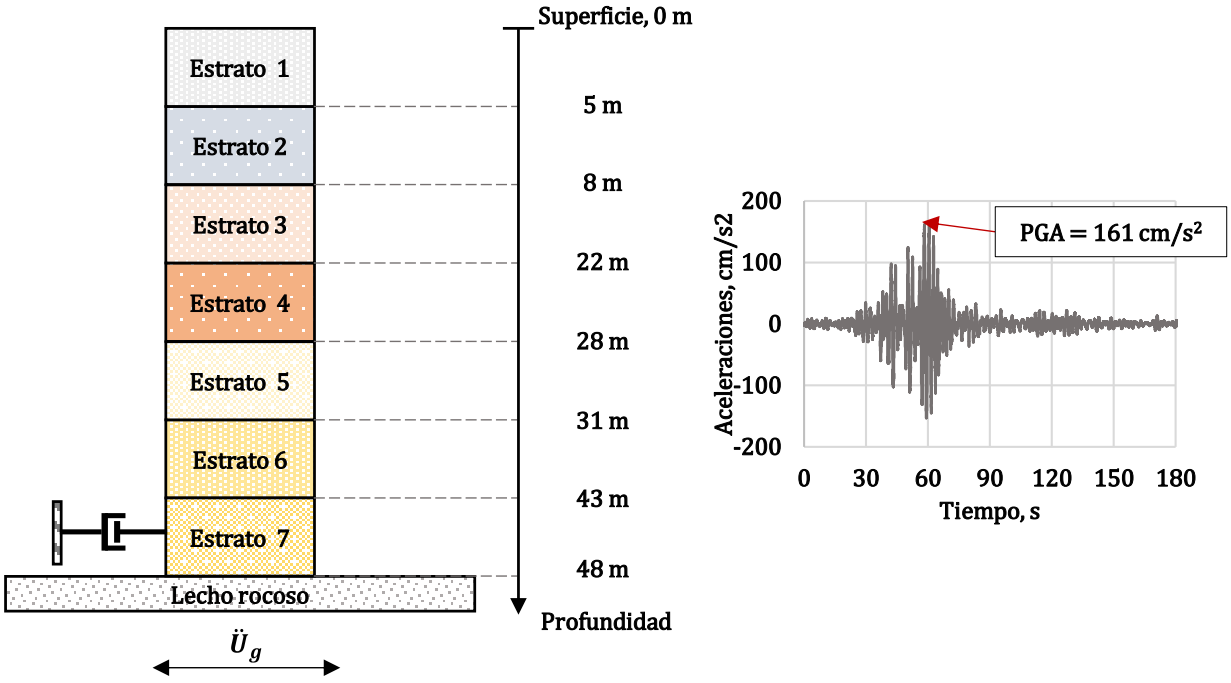


Figura 8.4 Esquema de la columna de suelo analizada

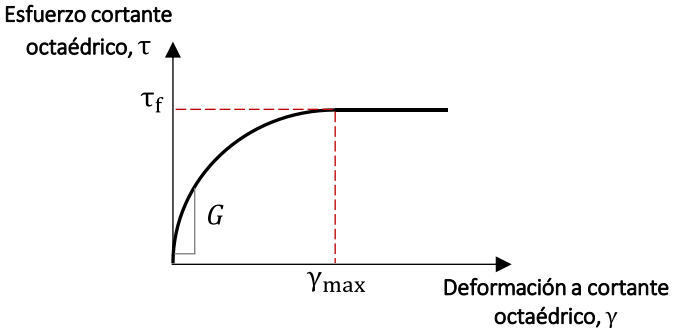


Figura 8.5 Diagrama esfuerzo - deformación del material *PressureIndependMultiYield* (adaptada de (OpenSees, 2021))

En este ejemplo se considera que la columna de suelo se apoya sobre un lecho rocoso con una densidad de 2,400 kg/m^3 y una velocidad de propagación de ondas de 800 m/s. Todos los estratos son de tipo arcilloso, con una deformación a cortante para el máximo esfuerzo (γ_{max}) igual a 0.1. El resto de los parámetros necesarios para definir el modelo computacional son presentados en la Tabla 9.1.

Tabla 9.1. Datos de los estratos de suelo empleados en el modelo numérico

Estrato	Profundidad, m	Espesor H, m	Densidad kg/m^3	G, kPa	$Bulk, kPa$	Cohesión c, kPa	Presión efectiva, kPa
1	5.0	5.0	1,300	50,000	190,000	25.0	10
2	8.0	3.0	1,500	60,000	210,000	30.0	50
3	22.0	14.0	1,400	75,000	220,000	35.0	50
4	28.0	6.0	1,600	65,000	200,000	25.0	50
5	31.0	3.0	1,600	75,000	250,000	30.0	80
6	43.0	12.0	1,600	80,000	350,000	45.0	100
7	48.0	5.0	1,800	90,000	400,000	50.0	100

En el modelo de la columna de suelo, los desplazamientos laterales de los nodos que se ubican a la misma profundidad se encuentran vinculados por medio de la restricción *equalDOF*, lo que implica que siempre se desplazan la misma distancia. Adicionalmente, se definen dos tipos de amortiguamiento: el primero es un amortiguamiento de Rayleigh, y el segundo es uno brindado por un elemento con comportamiento viscoso, de longitud cero, cuyo coeficiente de amortiguamiento es dependiente de las propiedades del lecho rocoso. En el análisis del modelo, primero se aplican las cargas gravitacionales producidas por el peso propio de los estratos y, posteriormente, se aplican las aceleraciones en los nodos inferiores de la columna de suelo. El código completo utilizado para generar el modelo en OpenSees se muestra en el Anexo 6.3.

8.3. Tipo de optimización y función objetivo

El problema de optimización considerado en este capítulo consiste en la calibración de un modelo de elementos finitos. Por lo tanto, se debe de buscar que la respuesta obtenida del modelo numérico de la columna de suelo se asemeje lo más posible a la respuesta de referencia. Para cuantificar el error entre ambas respuestas se emplea el Error Cuadrático (EC) que es definido por medio de la ecuación (8.4):

$$f(\boldsymbol{x}) = EC = \sum_{i=1}^{n} (X_i - Y_i)^2 \tag{8.4}$$

donde X_i es la respuesta del modelo numérico y Y_i es la respuesta de referencia, ambos para el paso i del registro de la respuesta. Se aclara que en este caso se comparan los desplazamientos laterales registrados en la parte superior de la columna de suelo. En la Figura 8.6 se muestran comparados los registros de desplazamientos cuando la estructura se sometió al movimiento sísmico de la Figura 8.4. En rojo se muestra la respuesta de referencia y en gris aquella calculada por el programa de elementos

finitos. Se puede observar que, a pesar de que ambas respuestas presentan cierto grado de similitud, el modelo numérico aún no representa de manera exacta el comportamiento de referencia. Por lo anterior, en este ejemplo se utiliza BVV para minimizar las diferencias entre ambas respuestas.

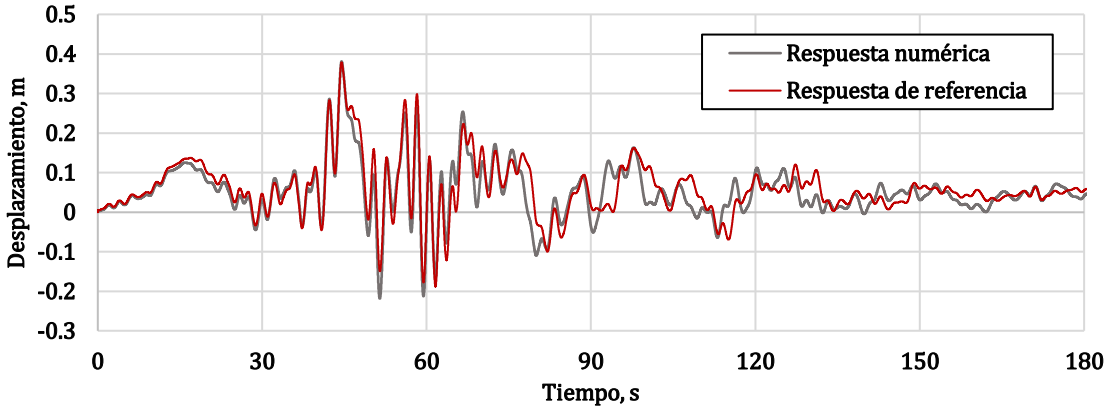


Figura 8.6 Comparación entre la respuesta de referencia y la numérica

8.4. Variables de decisión y parámetros del problema

Como parámetros del problema se consideran las características del lecho rocoso sobre el que se apoya la columna de suelo y la deformación a cortante, γ_{max}. Como variables de decisión se toman los datos que controlan la respuesta del modelo computacional, es decir, el espesor del estrato (H), la densidad del suelo (ρ), el módulo a cortante (G), el módulo de compresibilidad o módulo de Bulk ($Bulk$), la cohesión del terreno (c) y la presión de confinamiento efectiva ($refPress$). Para todas las variables de decisión, con excepción del espesor H, se definieron en un intervalo cerrado con incrementos discretos de diferentes valores. En lo que respecta a la variable H, al desconocerse el espesor asociado al óptimo del problema, se permitió que su valor se desplazara en un intervalo no definido con incrementos de 0.10 m. Los intervalos para el resto de las variables, con sus respectivos incrementos, fueron los siguientes:

$$\begin{gathered} \rho \in \{1300, 1310, \dots, 2000\}\ kg/m^3 \\ G \in \{50000, 50500, \dots, 100000\}\ kPa \\ Bulk \in \{190000, 190500, \dots, 420000\}\ kPa \\ c \in \{20, 21, \dots, 60\}\ kPa \\ refPress \in \{5, 6, \dots, 110\}\ kPa \end{gathered} \tag{8.5}$$

8.5. Codificación de los parámetros del modelo

Cada candidato a solución de este problema es representado por arreglos matriciales de 7x6 en donde las filas denotan cada estrato del modelo mientras que las columnas indican las propiedades mecánicas mencionadas en la sección 8.4. Por ejemplo, las características de la solución actual $\boldsymbol{x}$ se almacenarían en la matriz $[a_x]$ de la siguiente forma:

$$[a_X] = \begin{bmatrix} H_1 & \rho_1 & G_1 & \text{Bulk}_1 & c_1 & \text{refPress}_1 \\ H_2 & \rho_2 & G_2 & \text{Bulk}_2 & c_2 & \text{refPress}_2 \\ & & & \vdots & & \\ H_7 & \rho_7 & G_7 & \text{Bulk}_7 & c_7 & \text{refPress}_7 \end{bmatrix} \tag{8.6}$$

8.6. Restricciones del problema

La única restricción considerada en este problema consistió en que los modelos candidatos a solución no deben de presentar problemas de inestabilidad numérica durante el análisis. Esto se realiza impidiendo que el algoritmo de optimización genere modelos en los que los estratos de suelos se encuentren traslapados entre sí.

8.7. Descripción del código de Búsqueda por Vecindario Variable

Para resolver este ejemplo se utiliza un algoritmo de BVV que únicamente emplea BL en combinación con diferentes estructuras de vecindario de cardinalidad creciente. El diagrama de flujo que representa de manera gráfica el código del algoritmo de BVV se muestra en la Figura 8.7. De manera similar a lo realizado en los demás ejemplos, en el Anexo 6.1 se brinda el código utilizado para realizar el proceso de optimización en el programa Matlab.

El código comienza definiendo los parámetros que controlan el proceso de optimización en las líneas 6 a 8. Esta metaheurística requiere definir el porcentaje máximo de variables a modificar (pV), las estructuras de los vecindarios (*Vecindarios*) y los intentos máximos que tendrá el algoritmo para encontrar una solución de mayor calidad en cada vecindario (*IterMax*). Para este ejemplo se consideraron 6 tipos de vecindarios ordenados por medio de una cardinalidad creciente los cuales permiten buscar soluciones a ±1, ±2, ±3, ±5, ±7 y ±10 movimientos discretos de la solución actual. Los últimos cinco vecindarios, al tener un mayor alcance, le permiten al algoritmo salir de óptimos locales. Adicionalmente, se permite que el algoritmo genere como máximo 50 candidatas a solución

cada vez que se emplee el primer vecindario, disminuyendo el número de soluciones máximas para los vecindarios subsecuentes con mayor alcance.

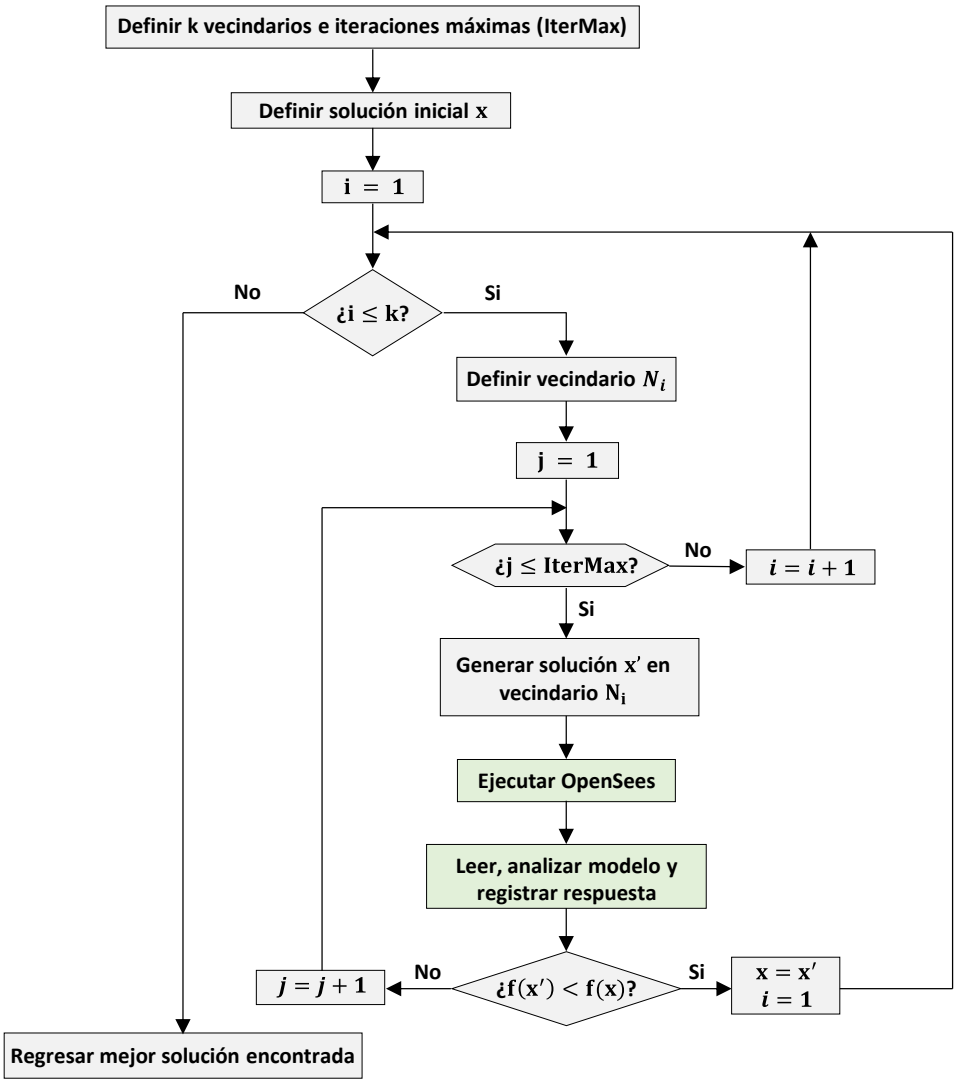


Figura 8.7 Diagrama de flujo del algoritmo de Búsqueda por Vecindario Variable usado

En las líneas siguientes se definen los intervalos para los cuales existen las variables de decisión (líneas 11 a 16). Asimismo, se definen diversas variables que son utilizadas más adelante por el algoritmo (líneas 19 a 24). Estas variables son: las matrices *MD* y *md* que almacenan las características de las soluciones actuales $\boldsymbol{x}$ y $\boldsymbol{x'}$, la matriz *VD* donde se agrupan todos los valores posibles de las variables de decisión (asignados entre las líneas 38 y 42), el vector *H* que almacena los espesores de los diferentes estratos, la matriz *A* que indica las variables de decisión que se modifican en el proceso de creación de nuevas soluciones y la variable *Error* que registra el valor del Error Cuadrático en cada iteración. De manera posterior, la solución inicial es

establecida entre las líneas 29 a 34 del código y su respuesta es cargada a la variable *DespI* en la línea 35. Por último, la respuesta de referencia es almacenada en la variable *DespR* y se calcula el Error Cuadrático entre las respuestas iniciales y de referencia en las líneas 46 y 47.

El proceso de optimización se desarrolla dentro de dos ciclos *for* anidados entre las líneas 49 a 118 del código. El primer ciclo *for*, que inicia en la línea 49, sirve para variar las estructuras de vecindarios empleados en la creación de nuevas soluciones, guardando el vecindario actual en la variable n (línea 50). Los vecindarios se revisan por medio de un orden de cardinalidad creciente y solo se pasa al siguiente vecindario si se ha generado un número *IterMax* de soluciones que no muestren una mejora para el proceso de optimización. El segundo ciclo *for*, que controla el número de soluciones creadas por cada estructura de vecindario, inicia en la línea 51 y termina en la 117.

Entre las líneas 53 a 93 se crean las nuevas propuestas a solución variando, según el vecindario empleado, diferentes valores de la solución actual. Tanto las variables de decisión que son escogidas para ser modificadas como la dirección del cambio que tendrá lugar se definen de manera aleatoria. De manera similar a los ejemplos anteriores, si el movimiento implica que la variable de decisión se sale del intervalo para el cual existe, el movimiento es rechazado y el valor no es modificado. Hay que recordar que en este ejemplo se trabaja de manera conjunta con Matlab y OpenSees, por tal motivo, las nuevas propuestas a solución son guardadas en un archivo denominado como "*SueloDinamico*" con extensión *.tcl*, mismo que es ejecutable en OpenSees. Esto se realiza en la línea 100 del código con la ejecución del archivo "*VariablesSuelo.m*", el cual crea el archivo del modelo que lee OpenSees. Al ser un archivo únicamente de escritura, el código de "*VariablesSuelo.m*" es relativamente sencillo y puede ser consultado en el Anexo 6.2.

Para leer y analizar la nueva solución propuesta con OpenSees (pasos marcados de color verde en el diagrama de flujo), es necesario almacenar el ejecutable de OpenSees en la carpeta de trabajo de Matlab, además de emplear el comando siguiente para llamar a OpenSees desde la interfaz de Matlab:

```
!OpenSees.exe "SueloDinamico.tcl"
```

El modelo de OpenSees de la columna de suelo puede ser revisado en el Anexo 6.3 de este libro; sin embargo, es necesario contar con algunos conocimientos básicos sobre los comandos y el lenguaje *TCL* empleado por el programa para su interpretación correcta. Una vez que se ha ejecutado OpenSees y se ha analizado la nueva propuesta a solución, los resultados del análisis son almacenados por OpenSees en un archivo llamado "*DespTop*", con extensión *.out* y que puede ser leído por Matlab. De esta manera se cierra el proceso de transferencia de información entre Matlab y OpenSees.

Entre las líneas 103 a 113 del código se determina el Error Cuadrático entre la respuesta de referencia y el modelo candidato a solución. Si el Error Cuadrático de esta nueva solución es menor el error que presenta la solución actual, es decir, si la nueva solución es de mayor calidad, esta se vuelve la solución actual y se reinicia el ciclo de búsqueda, en caso contrario se continua con la siguiente iteración. Finalmente, se presenta la última solución encontrada por el algoritmo con las instrucciones de las líneas 121 a 123.

8.8. Tamaño del espacio de configuraciones

Para este problema de optimización, el tamaño del espacio de configuraciones ($|\boldsymbol{S}|$) es igual al producto de las variaciones con repetición de las diferentes variables de decisión consideradas en los siete estratos. Es importante mencionar que el número de posibles soluciones también es dependiente de las posibles estratificaciones de la columna de suelo, siendo estas función del espesor y discretización de cada estrato, además de la altura total de la columna de suelo. La cardinalidad de las variables de decisión consideradas, y omitiendo por simplicidad las posibles combinaciones generadas por la variación del espesor de los estratos, es: 71 elementos para la densidad del suelo (ρ), 101 elementos para el módulo a cortante (G), 461 elementos para el módulo de compresibilidad ($Bulk$), 41 elementos para la cohesión del terreno (c) y 106 elementos para la presión de confinamiento efectiva ($refPress$). Por lo anterior, $|\boldsymbol{S}|$ presenta como mínimo un valor igual a:

$$|\boldsymbol{S}| > 71^7 \cdot 101^7 \cdot 461^7 \cdot 41^7 \cdot 106^7 = 1.263x10^{71} \text{ configuraciones posibles} \qquad (8.7)$$

8.9. Resultados y discusión

El algoritmo de BVV descrito en la sección 8.7 se aplicó para realizar la calibración del modelo de elementos finitos de una columna estratificada de suelo. Durante el proceso de optimización, se observó que el algoritmo era capaz de encontrar soluciones de alta calidad al problema con gran facilidad. En la Figura 8.8 se muestran comparadas la respuesta de referencia (en rojo) y la respuesta numérica obtenida del modelo actualizado de elementos finitos (en gris). Obsérvese que ambos registros de desplazamientos son prácticamente idénticos, lo cual era lo que se buscaba por medio del proceso de optimización. Para la ejecución del algoritmo que originó la respuesta numérica mostrada en la Figura 8.8, el Error Cuadrático obtenido presentó un valor igual a 0.231 m^2. A manera de comparación, el Error Cuadrático del modelo original era igual a 20.26 m^2, lo cual representa una disminución de las diferencias entre ambas respuestas del 99%.

En la Tabla 9.2 se muestran comparados los datos de la respuesta de referencia, del modelo de elementos finitos inicial y del calibrado para los siete estratos de suelo del modelo. Los datos de la respuesta de referencia se encuentran resaltados con letras oscuras para facilitar la comparación. Se observa que el algoritmo de optimización fue capaz de encontrar modelos con una respuesta muy parecida a la del modelo de referencia. Sin embargo, la similitud variaba entre las variables de decisión consideradas. Mientras que para los espesores y la densidad la solución actualizada presenta valores que se asemejan al modelo de referencia, para la cohesión y la presión efectiva se observan estratos en los que hay una diferencia considerable entre ambos modelos. Para finalizar, se hace la observación de que, a pesar de estas diferencias entre los modelos numéricos, el algoritmo fue capaz de hallar

múltiples soluciones con Errores Cuadráticos menores a 0.5 m^2. De este resultado se desprende la conclusión de que el modelo está tomando en cuenta variables de decisión que tiene poca influencia en la respuesta numérica del modelo.

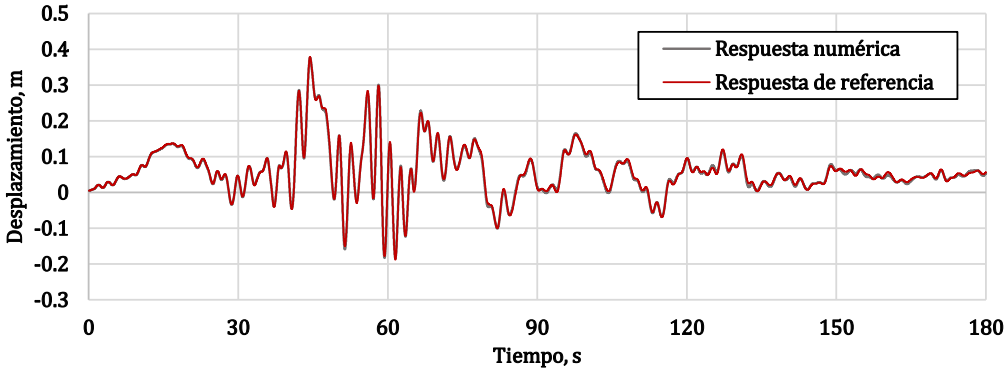


Figura 8.8 Comparación entre la respuesta de referencia y la numérica del modelo de elementos finitos actualizado

Tabla 9.2 Comparación entre el modelo inicial, actualizado y de referencia

Estrato	Respuesta	Profundidad, m	Espesor H, m	Densidad kg/m^3	G, kPa	$Bulk, kPa$	Cohesión c, kPa	Presión efectiva, kPa
1	Inicial	5.0	5.0	1,300	50,000	190,000	25.0	10
	Calibrada	5.6	5.6	1,460	50,000	196,500	20.0	18
	Referencia	**5.5**	**5.5**	**1,450**	**57,500**	**205,000**	**23.2**	**5**
2	Inicial	8.0	3.0	1,500	60,000	210,000	30.0	50
	Calibrada	6.9	1.3	1,620	58,000	201,000	52.0	38
	Referencia	**7.0**	**1.5**	**1,570**	**63,000**	**227,000**	**29.0**	**30**
3	Inicial	22.0	14.0	1,400	75,000	220,000	35.0	50
	Calibrada	21.3	14.4	1,590	73,500	221,000	51.0	52
	Referencia	**23.0**	**16.0**	**1,490**	**82,000**	**235,000**	**33.5**	**70**
4	Inicial	28.0	6.0	1,600	65,000	200,000	25.0	50
	Calibrada	29	7.7	1,480	72,500	210,000	30.0	36
	Referencia	**28.8**	**5.8**	**1,540**	**69,500**	**215,000**	**26.1**	**80**
5	Inicial	31.0	3.0	1,600	75,000	250,000	30.0	80
	Calibrada	31.8	2.8	1,740	78,000	254,000	22.0	74
	Referencia	**32.6**	**3.8**	**1,650**	**77,500**	**277,000**	**28.9**	**85**
6	Inicial	43.0	12.0	1,600	80,000	350,000	45.0	100
	Actualizada	42.6	10.8	1,680	75,500	347,500	40.0	110
	Referencia	**44.5**	**11.9**	**1,520**	**83,500**	**359,000**	**46.5**	**95**
7	Inicial	48.0	5.0	1,800	90,000	400,000	50.0	100
	Calibrada	47.9	5.3	1,820	78,000	406,000	60.0	107
	Referencia	**51.5**	**7.0**	**1,910**	**94,700**	**415,000**	**52.3**	**105**

8.10. Conclusiones

Se utilizó la metaheurística conocida como Búsqueda por Vecindario Variable para realizar la calibración de un modelo de elementos finitos de una columna de suelo estratificado. Este proceso consistió en la modificación de los parámetros que controlan la respuesta del modelo para buscar semejanza entre la respuesta numérica y una respuesta de referencia registrada en campo. Los resultados obtenidos por medio del algoritmo de optimización demuestran que es posible obtener un alto grado de similitud entre la respuesta del modelo y la respuesta de referencia por medio de este procedimiento. Adicionalmente, en este capítulo se presentó una manera sencilla y eficaz para emplear conjuntamente los algoritmos metaheurísticos y un programa de elementos finitos (en este caso OpenSees), lo que permite, en última instancia, el optimizar estructuras con un alto grado de complejidad.

Esta página ha sido intencionalmente dejada en blanco

9. Conclusiones generales

Los algoritmos metaheurísticos son herramientas poderosas que permiten resolver una amplia variedad de problemas complejos de optimización en tiempos razonables. Debido a que las metaheurísticas utilizan procesos estocásticos, lo normal es que, en cada ejecución, estos algoritmos encuentren una solución diferente al problema en cuestión. Más aún, los algoritmos metaheurísticos no son capaces de encontrar la solución exacta al problema de optimización ni tampoco es posible determinar con seguridad la similitud o cercanía entre las soluciones encontradas por el algoritmo y el óptimo global. A pesar de estas evidentes desventajas, las metaheurísticas gozan hoy en día de una gran popularidad entre la comunidad científica debido a que suelen ser el único método viable para resolver problemas de tipo no polinomial.

En lo que respecta a la ingeniería estructural, las metaheurísticas muestran una gran capacidad para generar diseños resilientes y sostenibles, motivo por el cual su uso en dicha área ha sido ampliamente estudiado en las últimas décadas. Sin embargo, la escasez de programas de estudio que aborden su uso obliga a que las y los estudiantes aprendan de este tema de manera autónoma. Para reducir esta complicación, el presente libro ejemplificó el uso de las metaheurísticas en la resolución de diversos tipos de problemas relacionados con la Ingeniería Estructural a la vez que proporciona un sustento teórico de los algoritmos mostrados.

Para concluir este libro cabe remarcar algunos aspectos relevantes sobre el uso de las metaheurísticas. En primera instancia, se debe señalar que las metaheurísticas no presentan una relación restrictiva entre algoritmos y problemas de optimización, esto significa que un mismo problema puede ser resuelto por diferentes metaheurísticas. Asimismo, las metaheurísticas descritas en este libro no están limitadas al campo de la ingeniería estructural por lo que pueden ser empleadas en otros campos ingenieriles, necesitándose cambios mínimos en su codificación para resolver problemas poco o nada relacionados con la optimización de estructuras. Por lo anterior, se recomienda al lector investigar otros ejemplos de aplicación de esta clase de algoritmos en su área de interés, esto con la finalidad de que se desarrollen trabajos de investigación que exploten sus utilidades prácticas.

Finalmente, resulta necesario mencionar que el teorema de No Free Lunch (Wolpert & Macready, 1997) indica que ningún algoritmo de optimización mostrará siempre un desempeño superior al resto de posibles algoritmos. Tal resultado advierte de la poca rigurosidad científica de las decenas de artículos científicos publicados cada año en donde se presenta una nueva metaheurística con un desempeño definido como “abrumadoramente sobresaliente” por sus propios autores. A pesar de sus implicaciones tan importantes, el teorema de No Free Lunch no es capaz de indicar en qué problemas determinado algoritmo mostrará un buen desempeño y en cuáles uno malo. Para resolver tal pregunta, es importante considerar los requerimientos propios de cada problema así como los componentes que se utilizarán en la construcción de la metaheurística con la cual se buscará resolverlo.

10. Referencias

Abioye, S. O., Oyedele, L. O., Akanbi, L., Ajayi, A., Davila Delgado, J. M., Bilal, M., Akinade, O. O., & Ahmed, A. (2021). Artificial intelligence in the construction industry: A review of present status, opportunities and future challenges. *Journal of Building Engineering*, *44*(April 2020), 103299. https://doi.org/10.1016/j.jobe.2021.103299

ACM Transactions on Evolutionary Learning and Optimization. (2022). *Author Guidelines*. https://dl.acm.org/journal/telo/author-guidelines

Aranha, C., Camacho Villalón, C. L., Campelo, F., Dorigo, M., Ruiz, R., Sevaux, M., Sörensen, K., & Stützle, T. (2021). Metaphor-based metaheuristics, a call for action: the elephant in the room. *Swarm Intelligence*, *0123456789*. https://doi.org/10.1007/s11721-021-00202-9

Ashwini R. Kulkarni, Vijaykumar Bhusare, Ashwini R. Kulkarni, & Vijaykumar Bhusare. (2016). Structural optimization of reinforced concrete structures. *International Journal of Engineering Research And*, *V5*(07), 3–8. https://doi.org/10.17577/ijertv5is070156

Atabay, Ş. (2009). Cost optimization of three-dimensional beamless reinforced concrete shear-wall systems via genetic algorithm. *Expert Systems with Applications*, *36*(2 PART 2), 3555–3561. https://doi.org/10.1016/j.eswa.2008.02.004

Atashpaz-Gargari, E., & Lucas, C. (2007). Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition. *2007 IEEE Congress on Evolutionary Computation, CEC 2007*, 4661–4667. https://doi.org/10.1109/CEC.2007.4425083

Bäck, T, Hoffmeister, F., & Schwefel, H. (1991). A survey of evolution strategies. *Proceedings of the Fourth International Conference on Genetic Algorithms*, *January*, 2–9.

Bäck, Thomas, Rudolph, G., & Schwefel, H.-P. (1997). Evolutionary Programming and Evolution Strategies: Similarities and Differences. *Proceedings of the Second Annual Conference on Evolutionary Programming*.

Bäck, Thomas, & Schwefel, H.-P. (1993). An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*, *1*(1), 1–23. https://doi.org/10.1162/evco.1993.1.1.1

Balling, R. (1991). *Optimal Steel Frame Desgin by Simulated Annealing*. *117*(6), 1780–1795.

Bekdaş, G., Nigdeli, S. M., & Yang, X. S. (2018). A novel bat algorithm based optimum tuning of mass dampers for improving the seismic safety of structures. *Engineering Structures*, *159*(April 2017), 89–98. https://doi.org/10.1016/j.engstruct.2017.12.037

Beyer, H.-G., & Schwefel, H.-P. (2002). Evolution strategies - A comprehensive introduction. *Natural Computing*, *1*, 3–52. https://doi.org/10.1023A

Birattari, M., Paquete, L., Stützle, T., & Varrentrapp, K. (2003). *Classification of metaheuristics and design of experiments for the analysis of components. Reporte: AIDA – 01 – 05.*

Biyanto, T., & at al. (2016). Duelist Algorithm: An Algorithm Inspired by How Duelist Improve Their Capabilities in a Duel. In *En: Tan Y., Shi Y. y Niu B. (eds) Advances in Swarm Intelligence. ICSI 2016. Lecture Notes in Computer Science* (Vol. 9712). Springer, Cham.

Biyanto, T. R., Matradji, Irawan, S., Febrianto, H. Y., Afdanny, N., Rahman, A. H., Gunawan, K. S., Pratama, J. A. D., & Bethiana, T. N. (2017). Killer Whale Algorithm: An Algorithm Inspired by the Life of Killer Whale. *Procedia Computer Science*, *124*, 151–157. https://doi.org/10.1016/j.procs.2017.12.141

Bland, J. A. (2001). Optimal structural design by ant colony optimization. *Engineering Optimization*, *33*(4), 425–443. https://doi.org/10.1080/03052150108940927

Blickle, T., & Thiele, L. (1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, *4*(4), 361–394. https://doi.org/10.1162/evco.1996.4.4.361

Blum, C., & Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, *35*, 268–308. https://doi.org/10.1145/937503.937505

Bojorquez, E., Leyva, H., Reyes, A., & Fernández, E. (2018). Diseño Óptimo Multiobjetivo de Edificios de Concreto Reforzado Usando Algoritmos Genéticos. *Revista de Ingeniería Sísmica*, *99*, 23–47.

Boussaïd, I., Lepagnot, J., & Siarry, P. (2013). A survey on optimization metaheuristics. *Information Sciences*, *237*, 82–117. https://doi.org/10.1016/j.ins.2013.02.041

Brandeau, M. L., & Chiu, S. S. (1993). Sequential location and allocation: worst case performance and statistical estimation. *Computers & Operations Research*, *1*, 289– 298.

Camacho-Villalón, C. L., Dorigo, M., & Stützle, T. (2019). The intelligent water drops algorithm: why it cannot be considered a novel algorithm: A brief discussion on the use of metaphors in optimization. *Swarm Intelligence*, *13*(3–4), 173–192. https://doi.org/10.1007/s11721-019-00165-y

Camp, C. V., Bichon, B. J., & Stovall, S. P. (2005). Design of low-weight steel frames using ant colony optimization. *Journal of Structural Engineering*, *131*(3), 369–379. https://doi.org/10.1061/40700(2004)158

Chiroma, H., Abdulkareem, S., Abubakar, A., Zeki, A., Ya 'u Gital, A., & Usman, M. J. (2013). Correlation study of genetic algorithm operators: Crossover and mutation probabilities. *International Symposium on Mathematical Sciences and Computing Research*, *2013*(December), 6–7.

Chou, J. S., & Nguyen, N. M. (2020). FBI inspired meta-optimization. *Applied Soft Computing Journal*, *93*. https://doi.org/10.1016/j.asoc.2020.106339

Chu, S., Tsai, P., & Pan, J. (2006). Cat Swarm Optimization. In *En: Yang Q. y Webb G. (eds) PRICAI 2006: Trends in Artificial Intelligence. PRICAI 2006. Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg.

Coello, C., & Christiansen, A. D. (2000). Multiobjective optimization of trusses using genetic algorithms. *Computers and Structures*, *75*(6), 647–660. https://doi.org/10.1016/S0045-7949(99)00110-8

Coello, C. (2018). Multi-objective Optimization. In *En: Martí R., Pardalos P. y Resende M. (eds) Handbook of Heuristics.* Springer, Cham.

Coello, Carlos, Hernández, F., & Farrera, F. (1997). Optimal design of reinforced concrete beams using genetic algorithms. *Expert Systems with Applications*, *12*(1), 101–108. https://doi.org/10.1016/S0957-4174(96)00084-X

Colorni, A., Dorigo, M., & Maniezzo, V. (1991). Distributed Optimization by ant colonies. *Proceedings of the First European Conference on Artificial Life*.

Dassault Systemes. (2017). *Abaqus FEA*.

De Albuquerque, A. T., El Debs, M. K., & Melo, A. M. C. (2012). A cost optimization-based design of precast concrete floors using genetic algorithms. *Automation in Construction*, *22*, 348–356. https://doi.org/10.1016/j.autcon.2011.09.013

De Oliveira, S., Bezerra, L., Stützle, T., Dorigo, M., Wanner, E., & De Souza, S. (2021). A computational study on ant colony optimization for the traveling salesman problem with dynamic demands. *Computers and Operations Research*, *135*(July 2020). https://doi.org/10.1016/j.cor.2021.105359

Degertekin, S. O., Saka, M. P., & Hayalioglu, M. S. (2008). Optimal load and resistance factor design of geometrically nonlinear steel space frames via tabu search and genetic algorithm. *Engineering Structures*, *30*(1), 197–205. https://doi.org/10.1016/j.engstruct.2007.03.014

Del Ser, J., Osaba, E., Molina, D., Yang, X. S., Salcedo-Sanz, S., Camacho, D., Das, S., Suganthan, P. N., Coello Coello, C. A., & Herrera, F. (2019). Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, *48*, 220–250. https://doi.org/10.1016/j.swevo.2019.04.008

Dokeroglu, T., Sevinc, E., Kucukyilmaz, T., & Cosar, A. (2019). A survey on new generation metaheuristic algorithms. *Computers and Industrial Engineering*, *137*. https://doi.org/10.1016/j.cie.2019.106040

Dorigo, M. (2016). Swarm Intelligence : A few things you need to know if you want to publish in this journal. *Swarm Intelligence*.

Duarte, A., Sánchez, J., Mladenović, N., & Todosijević, R. (2018). Variable Neighborhood Descent. In *En: Martí R., Pardalos P. y Resende M. (eds). Handbook of Heuristics.* Springer, Cham.

Eiben, A. E. (1997). Genetic algorithms + data structures = evolution programs. In *Artificial Intelligence in Medicine* (Vol. 9, Issue 3, pp. 283–286). https://doi.org/10.1016/s0933-3657(96)00378-8

Eusuff, M., Lansey, K., & Pasha, F. (2007). Shuffled frog-leaping algorithm: a memetic meta-heuristic for discrete optimization. *Engineering Optimization*, *38*, 129–154.

Farhat, F., Nakamura, S., & Takahashi, K. (2009). Application of genetic algorithm to optimization of buckling restrained braces for seismic upgrading of existing structures. *Computers and Structures*, *87*(1–2), 110–119. https://doi.org/10.1016/j.compstruc.2008.08.002

Feo, T. A., & Resende, M. G. C. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, *6*, 109–133. https://doi.org/10.1007/BF01096763

Fleischer, M. (1995). Simulated annealing: Past, present and future. *Proceedings of the 1995 Winter Simulation Conference.*

Fogel, D. B., & Atmar, J. W. (1990). Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, *63*(2), 111–114. https://doi.org/10.1007/BF00203032

Geem, Z., Hoon, J., & Loganathan, G. (2001). A New Heuristic Optimization Algorithm: Harmony Search. *Simulation.*, *76*, 60–68.

Gen, M., & Cheng, R. (1996). A survey of penalty techniques in genetic algorithms. *Proceedings of IEEE International Conference on Evolutionary Computation*, 804–809. https://doi.org/10.1109/ICEC.1996.542704

Gholizadeh, S., & Mohammadi, M. (2017). Reliability-based seismic optimization of steel frames by metaheuristics and neural networks. *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part A: Civil Engineering*, *3*(1), 1–11. https://doi.org/10.1061/ajrua6.0000892

Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences*, *8*, 156–166.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, *13*, 533–549.

Glover, F. (1989a). Tabu Search - Part I. *ORSA Journal on Computing*, *1*(3), 190–206.

Glover, F. (1989b). Tabu Search - Part II. *Orsa Journal on Computing*, *1*(3), 190–206.

Glover, F., & Laguna, M. (1999). Tabu Search. *ORSA Journal on Computing*, *1*(3). https://doi.org/10.1287/ijoc.1.3.190

Glover, F., Taillard, E., & De Werra, D. (1993). A user's guide to tabu search. *Annals of Operations Research*, *41*(1), 1–28. https://doi.org/10.1007/BF02078647

Goldberg, D. (1989). Genetic algorithms in search, optimization, and machine learning. In *Addison Wesley*.

Goldberg, D. E., & Deb, K. (1991). A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Foundations of Genetic Algorithms* (Vol. 1). Morgan Kaufmann Publishers, Inc. https://doi.org/10.1016/b978-0-08-050684-5.50008-2

Granville, V., Rasson, J. P., & Krivánek, M. (1994). Simulated Annealing: A Proof of Convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(6), 652–656. https://doi.org/10.1109/34.295910

Gu, Q., Barbato, M., Conte, J. P., Gill, P. E., & McKenna, F. (2012). OpenSees-SNOPT Framework for Finite-Element-Based Optimization of Structural and Geotechnical Systems. *Journal of Structural Engineering*, *138*(6), 822–834. https://doi.org/10.1061/(asce)st.1943-541x.0000511

Guerrero, H., Ji, T., Teran-Gilmore, A., & Escobar, J. (2016). A method for preliminary seismic design and assessment of low-rise structures protected with buckling-restrained braces. *Engineering Structures*, *123*, 141–154. https://doi.org/10.1016/j.engstruct.2016.05.015

Hajela, P., & Lee, E. (1994). Genetic algorithms in truss topological optimization. *International Journal of Solids and Structures*, *32*(22), 3341–3357. https://doi.org/10.1016/0020-7683(94)00306-H

Hansen, P, & Mladenović, N. (2018). *Variable Neighborhood Search* (p. 2018). En: Martí R., Pardalos P. y Resende M. (eds) Handbook of Heuristics. Springer, Cham.

Hansen, Pierre, Mladenović, N., & Hansen, P. (1997). Variable neighborhood search. *Computers and Operations Research.*, *24*, 1097–1110. https://doi.org/10.1007/978-3-319-07124-4_19

Hansen, Pierre, Mladenović, N., & Moreno Pérez, J. A. (2009). Variable neighborhood search: Methods and applications. In *Annals of Operations Research* (Vol. 175). https://doi.org/10.1007/s10288-008-0089-1

Hansen, Pierre, Mladenović, N., Todosijević, R., & Hanafi, S. (2017). Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, *5*, 423–454. https://doi.org/10.1007/s13675-016-0075-x

Harifi, S., Khalilian, M., Mohammadzadeh, J., & Ebrahimnejad, S. (2019). Emperor Penguins Colony: a new metaheuristic algorithm for optimization. *Evolutionary Intelligence*, *12*(2), 211–226. https://doi.org/10.1007/s12065-019-00212-x

Hasançebi, O, & Çarbaş, S. (2011). Ant colony search method in practical structural optimization. *International Journal of Optimization in Civil Engineering*, *1*, 91–105.

Hasançebi, Oguzhan, & Erbatur, F. (2002). Layout optimisation of trusses using simulated annealing. *Advances in Engineering Software*, *33*(7–10), 681–696. https://doi.org/10.1016/S0965-9978(02)00049-2

Hashim, F. A., & Hussien, A. G. (2022). Snake Optimizer: A novel meta-heuristic optimization algorithm. *Knowledge-Based Systems*, *242*, 108320. https://doi.org/10.1016/j.knosys.2022.108320

Hayyolalam, V., & Pourhaji Kazem, A. A. (2020). Black Widow Optimization Algorithm: A novel meta-heuristic approach for solving engineering optimization problems. *Engineering Applications of Artificial Intelligence*, *87*. https://doi.org/10.1016/j.engappai.2019.103249

Heidari, A. A., Mirjalili, S., Faris, H., Aljarah, I., Mafarja, M., & Chen, H. (2019). Harris hawks optimization: Algorithm and applications. *Future Generation Computer Systems*, *97*, 849–872. https://doi.org/10.1016/j.future.2019.02.028

Hofmeister, B., Bruns, M., & Rolfes, R. (2019). Finite element model updating using deterministic optimisation: A global pattern search approach. *Engineering Structures*, *195*, 373–381. https://doi.org/10.1016/j.engstruct.2019.05.047

Holland, J. (1962). Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, *9*(3), 297–314. https://doi.org/10.1145/321127.321128

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, *79*, 2554–2558. https://doi.org/10.1073/pnas.79.8.2554

Instituto de Desarrollo Local. (2021). *Pueblos Mágicos de España*. https://www.pueblosmagicos.es/

Jones, T. (1995). *One Operator , One Landscape* (Issue March 1995).

Journal of Heuristics. (2015). *Policies on heuristic search*. https://www.springer.com/journal/10732/updates/17199246

Karaboga, D., & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *Journal of Global Optimization*, *39*, 459–471. https://doi.org/10.1007/s10898-007-9149-x

Kaveh, A., & Dadfar, B. (2008). Optimum Seismic Design of Steel Moment Resisting Frames By Genetic Algorithms. *Asian Journal of Civil Engineering (Building and Housing)*, *9*(2), 23.

Kaveh, A., & Mahdavi, V. R. (2014). Colliding bodies optimization: A novel meta-heuristic method. *Computers and Structures*, *139*, 18–27. https://doi.org/10.1016/j.compstruc.2014.04.005

Kaveh, A., & Nasrollahi, A. (2014). Performance-based seismic design of steel frames utilizing charged system search optimization. *Applied Soft Computing Journal*, *22*, 213–221. https://doi.org/10.1016/j.asoc.2014.05.012

Kaveh, A., & Zaerreza, A. (2020). Shuffled shepherd optimization method: a new Meta-heuristic algorithm. *Engineering Computations*, *37*(7), 2357–2389. https://doi.org/10.1108/EC-10-2019-0481

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Kocer, F. Y., & Arora, J. S. (1996). Optimal design of prestressed concrete transmission poles. *Proceedings of the Conference on Analysis and Computation*, *9445*(July), 111–122. https://doi.org/10.1061/(ASCE)0733-9445(1996)122

Koza, J. (1989). Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*.

Koza, J. (1994). Genetic programming: On the programming of computers by means of natural selection,. In *Biosystems* (Vol. 33, Issue 1). Massachusetts Institute of Technology. https://doi.org/10.1016/0303-2647(94)90062-0

Laguna, M. (2018). Tabu Search. In *Martí R., Pardalos P., Resende M. (eds) Handbook of Heuristics. Springer, Cham.* https://doi.org/https://doi.org/10.1007/978-3-319-07124-4_24

Lakshmanaprabu, S. K., Shankar, K., Sheeba Rani, S., Abdulhay, E., Arunkumar, N., Ramirez, G., & Uthayakumar, J. (2019). An effect of big data technology with ant colony optimization based routing in vehicular ad hoc networks: Towards smart cities. *Journal of Cleaner Production*, *217*, 584–593. https://doi.org/10.1016/j.jclepro.2019.01.115

Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., & Dizdarevic, S. (1999). Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, *13*(2), 129–170. https://doi.org/10.1023/A:1006529012972

Lavinas, Y., Aranha, C., Sakurai, T., & Ladeira, M. (2018). Experimental analysis of the tournament size on genetic algorithms. *Proceedings - 2018 IEEE International Conference on Systems, Man, and Cybernetics*, 3647–3653. https://doi.org/10.1109/SMC.2018.00617

Lepš, M., & Šejnoha, M. (2003). New approach to optimization of reinforced concrete beams. *Computers and Structures*, *81*(18–19), 1957–1966. https://doi.org/10.1016/S0045-7949(03)00215-3

Liao, W., Lu, X., Fei, Y., Gu, Y., & Huang, Y. (2024). Generative AI design for building structures. *Automation in Construction*, *157*(June 2023). https://doi.org/10.1016/j.autcon.2023.105187

Lin, B., & Miller, D. C. (2004). Tabu search algorithm for chemical process optimization. *Computers and Chemical Engineering*, *28*(11), 2287–2306. https://doi.org/10.1016/j.compchemeng.2004.04.007

Lin, N., Fu, L., Zhao, L., Hawbani, A., Tan, Z., Al-Dubai, A., & Min, G. (2022). A novel nomad migration-inspired algorithm for global optimization. *Computers and Electrical Engineering*, *100*(October 2020), 107862. https://doi.org/10.1016/j.compeleceng.2022.107862

Mackenna, F. (1997). *Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing.* Universidad de California, EE. UU.

Manoharan, S., & Shanmuganathan, S. (1999). A comparison of search mechanisms for structural optimization. *Computers and Structures*, *73*(1–5), 363–372. https://doi.org/10.1016/S0045-7949(98)00287-9

Martínez-Gavara, A., Corberán, T., & Martí, R. (2021). GRASP and tabu search for the generalized dispersion problem. *Expert Systems with Applications*, *173*(January), 0–1. https://doi.org/10.1016/j.eswa.2021.114703

Martínez, F. J., González-Vidosa, F., Hospitaler, A., & Alcalá, J. (2011). Design of tall bridge piers by ant colony optimization. *Engineering Structures*, *33*(8), 2320–2329. https://doi.org/10.1016/j.engstruct.2011.04.005

Martínez, F. J., González-Vidosa, F., Hospitaler, A., & Yepes, V. (2010). Heuristic optimization of RC bridge piers with rectangular hollow sections. *Computers and Structures*, *88*(5–6), 375–386. https://doi.org/10.1016/j.compstruc.2009.11.009

Marzband, M., Yousefnejad, E., Sumper, A., & Domínguez-García, J. L. (2016). Real time experimental implementation of optimum energy management system in standalone Microgrid by using multi-layer ant colony optimization. *International Journal of Electrical Power and Energy Systems*, *75*, 265–274. https://doi.org/10.1016/j.ijepes.2015.09.010

MathWorks Inc. (2020). *MATLAB and Optimization Toolbox*.

Medina, J. (2001). Estimation of incident and reflected waves using simulated annealing. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, *127*, 213–221.

Michalewicz, Z., Janikow, C. Z., & Krawczyk, J. B. (1992). A modified genetic algorithm for optimal control problems. *Computers and Mathematics with Applications*, *23*(12), 83–94. https://doi.org/10.1016/0898-1221(92)90094-X

Michiels, W., Aarts, E., & Korst, J. (2018). Theory of Local Search. In *Martí R., Pardalos P., Resende M. (eds) Handbook of Heuristics. Springer, Cham.* https://doi.org/https://doi.org/10.1007/978-3-319-07124-4_6

Mirjalili, S., & Lewis, A. (2016). The Whale Optimization Algorithm. *Advances in Engineering Software*, *95*, 51–67. https://doi.org/10.1016/j.advengsoft.2016.01.008

Mohammadi-Balani, A., Dehghan Nayeri, M., Azar, A., & Taghizadeh-Yazdi, M. (2021). Golden eagle optimizer: A nature-inspired metaheuristic algorithm. *Computers and Industrial Engineering*, *152*. https://doi.org/10.1016/j.cie.2020.107050

Mokarram, V., & Banan, M. R. (2018). An improved multi-objective optimization approach for performance-based design of structures using nonlinear time-history analyses. *Applied Soft Computing Journal*, *73*, 647–665. https://doi.org/10.1016/j.asoc.2018.08.048

Nguyen, T.-H., & Jung, J. J. (2021). Ant colony optimization-based traffic routing with intersection negotiation for connected vehicles. *Applied Soft Computing*, *112*, 107828. https://doi.org/10.1016/j.asoc.2021.107828

NTC - Concreto. (2023). *NTC-23. Normas Técnicas Complementarias para diseño y construcción de estructuras de concreto* (Issue 220).

Ohsaki, M., & Nakajima, T. (2012). Optimization of link member of eccentrically braced frames for maximum energy dissipation. *Journal of Constructional Steel Research*, *75*, 38–44. https://doi.org/10.1016/j.jcsr.2012.03.008

OpenSees. (2021). *Command Manual*. https://opensees.berkeley.edu/wiki/index.php/OpenSees_User

Osaba, E., Villar-Rodriguez, E., Del Ser, J., Nebro, A. J., Molina, D., LaTorre, A., Suganthan, P. N., Coello Coello, C. A., & Herrera, F. (2021). A tutorial on the design, experimentation and application of metaheuristic algorithms to real-World optimization problems. *Swarm and Evolutionary Computation*, *64*(August 2020), 100888. https://doi.org/10.1016/j.swevo.2021.100888

Pareto, V. (1896). *Cours D'Economie Politique, vol. I and II. F. Rouge, Lausanne, vol. I and II.*

PEER. (2006). *OpenSees, open source finite element platform for earthquake engineering simulations.* Pacific Earthquake Engineering Research Center. University of California, Berkeley.

Polya, G. (1945). *How to solve it: A new aspect of mathematical model.* Princeton University Press,.

Rajeev, S., & Krishnamoorthy, C. S. (1998). Genetic algorithm-based methodology for design optimization of reinforced concrete frames. *Computer-Aided Civil and Infrastructure Engineering*, *13*(1), 63–74. https://doi.org/10.1111/0885-9507.00086

Rashedi, E., Nezamabadi-pour, H., & Saryazdi, S. (2009). GSA: A Gravitational Search Algorithm. *Information Sciences*, *179*(13), 2232–2248. https://doi.org/10.1016/j.ins.2009.03.004

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung techischer Systeme nach Prinzipien der biologischen Evolution. Frommann Holzboog Verlag, Stuttgart.*

Renkavieski, C., & Parpinelli, R. S. (2021). Meta-heuristic algorithms to truss optimization: Literature mapping and application. *Expert Systems with Applications*, *182*(October 2020), 115197. https://doi.org/10.1016/j.eswa.2021.115197

Saremi, S., Mirjalili, S., & Lewis, A. (2017). Grasshopper Optimisation Algorithm: Theory and application. *Advances in Engineering Software*, *105*, 30–47. https://doi.org/10.1016/j.advengsoft.2017.01.004

Secretaria de Turismo. (2021). *Pueblos Mágicos de México*. https://www.gob.mx/sectur/articulos/pueblos-magicos-206528

Seo, H., Kim, J., & Kwon, M. (2018). Optimal seismic retrofitted RC column distribution for an existing school building. *Engineering Structures*, *168*, 399–404. https://doi.org/10.1016/j.engstruct.2018.04.098

Sevaux, M., Sörensen, K., & Pillay, N. (2018). Adaptative and multilevel metahueristics. In *En: Martí R., Pardalos P. y Resende M. (eds). Handbook of Heuristics.* Springer, Cham.

Shah, H. (2008). Intelligent water drops algorithm: A new optimization method for solving the multiple knapsack problem. *International Journal of Intelligent Computing and Cybernetics.*, *1*, 193–212.

Shan, W., Liu, J., & Zhou, J. (2023). Integrated method for intelligent structural design of steel frames based on optimization and machine learning algorithm. *Engineering Structures*, *284*, 115980. https://doi.org/10.1016/j.engstruct.2023.115980

Shook, D. A., Roschke, P. N., Lin, P. Y., & Loh, C. H. (2008). GA-optimized fuzzy logic control of a large-scale building for seismic loads. *Engineering Structures*, *30*(2), 436–449. https://doi.org/10.1016/j.engstruct.2007.04.008

Sörensen, K. (2013). Metaheuristics-the metaphor exposed. *International Transactions in Operational Research*, *22*, 3–18. https://doi.org/10.1111/itor.12001

Sörensen, K., & Glover, F. (2013). *Sörensen K, Glover FW (2013) Metaheuristics. En: Gass S. y Fu M. (eds) Encyclopedia of operations research and management science, 663 3rd edn, pp 960 – 970, Springer, Boston, MA*. Springer, Boston, MA.

Sörensen, K., Sevaux, M., & Glover, F. (2018). A History of Metaheuristics. In *In: Martí R., Pardalos P. y Resende M. (eds). Handbook of Heuristics.* Springer, Cham.

Spears, W. M. (2001). The equilibrium and transient behavior of mutation and recombination. *Foundations of Genetic Algorithms 6*, 241–260. https://doi.org/10.1016/b978-155860734-7/50096-2

Suganthan, P. N., Hansen, N., Liang, J. J., Deb, K., Chen, Y. P., Auger, A., & Tiwari, S. (2005). Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. In *Technical Report, Nanyang Technological University, Singapore* (Issue May 2005).

Tamura, K., & Yasuda, K. (2016). Spiral Optimization Algorithm Using Periodic Descent Directions. SICE Journal of Control, Measurement, and System Integration. Vol. 9; pp: 134 – 43. *SICE Journal of Control, Measurement, and System Integration*, *9*, 134–143.

Tapeh, A. T. G., & Naser, M. Z. (2023). Artificial intelligence, machine learning, and deep learning in structural engineering: A scientometrics review of trends and best practices. In *Archives of Computational Methods in Engineering* (Vol. 30, Issue 1). Springer Netherlands. https://doi.org/10.1007/s11831-022-09793-w

Tzanetos, A., & Dounias, G. (2021). Nature inspired optimization algorithms or simply variations of metaheuristics? *Artificial Intelligence Review*, *54*(3), 1841–1862. https://doi.org/10.1007/s10462-020-09893-8

United Nations. (2021). *Sustainable Development*. https://www.un.org/sustainabledevelopment/es/

Velasco, L., & Guerrero, H. (2020). *Introducción al análisis estructural con OpenSees*. Serie de docencia. Instituto de Ingeniería, Universidad Nacional Autónoma de México.

Velasco, L., Guerrero, H., & Hospitaler, A. (2022). Can the global optimum of a combinatorial optimization problem be reliably estimated through extreme value theory? *Swarm and Evolutionary Computation*, *75*. https://doi.org/10.1016/j.swevo.2022.101172

Velasco, L., Guerrero, H., & Hospitaler, A. (2023). A Literature Review and Critical Analysis of Metaheuristics Recently Developed. *Archives of Computational Methods in Engineering*. https://doi.org/10.1007/s11831-023-09975-0

Velasco, L., Guerrero, H., Hospitaler, A., & Saura, H. (2024). Comparative study on the performance of metaheuristics in the optimization of seismic designs. *18th World Conference on Earthquake Engineering*, 1–12.

Velasco, L., & Hospitaler, A. (2021). *Diseño óptimo del refuerzo estructural, mediante disipadores CRP, para la adecuación del desempeño sísmico de estructuras aporticadas de hormigón armado.* [Universidad Politecnica de Valencia]. http://hdl.handle.net/10251/177909

Velasco, L., Hospitaler, A., & Guerrero, H. (2022a). Optimal design of the seismic retrofitting of reinforced concrete framed structures using BRBs. *Bulletin of Earthquake Engineering*. https://doi.org/10.1007/s10518-022-01394-z

Velasco, L., Hospitaler, A., & Guerrero, H. (2022b). Optimización metaheurística del refuerzo sísmico de estructuras aporticadas de hormigón armado con contravientos restringidos al pandeo. *XXIII Congreso Nacional de Ingeniería Sísmica*.

Wang, Y., & Han, Z. (2021). Ant colony optimization for traveling salesman problem based on parameters optimization. *Applied Soft Computing*, *107*, 107439. https://doi.org/10.1016/j.asoc.2021.107439

Wedyan, A., Whalley, J., & Narayanan, A. (2017). Hydrological Cycle Algorithm for Continuous Optimization Problems. *Journal of Optimization*, *10*(2), 1–25. https://doi.org/10.1155/2017/3828420

Wetter, M. (2001). GenOpt - A Generic Optimization Program. *Seventh International IBPSA Conference*, *August*, 601–608.

Weyland, D. (2010). A Rigorous Analysis of the Harmony Search Algorithm: How the Research Community can be Misled by a "Novel" Methodology. *International Journal of Applied Metaheuristic Computing (IJAMC)*, *1*(2), 50–60. https://doi.org/10.4018/jamc.2010040104

Weyland, D. (2015). A critical analysis of the harmony search algorithm-How not to solve sudoku. *Operations Research Perspectives*, *2*, 97–105. https://doi.org/10.1016/j.orp.2015.04.001

Winker, P., & Maringer, D. (2007). *The threshold accepting optimization algorithm in economics and statistics. In: Kontoghiorghes E. y Gatu C. (eds). Optimization, Econometric and Financial Analysis. Advances in Computational Management Science. Vol 9. Sprin*. Springer, Berlin, Heidelberg.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82. https://doi.org/10.1109/4235.585893

Xiao, Y., Yue, F., Zhang, X., & Shahzad, M. M. (2022). Aseismic optimization of mega-sub controlled structures based on Gaussian process surrogate model. *KSCE Journal of Civil Engineering*, *26*(5), 2246–2258. https://doi.org/10.1007/s12205-022-0832-8

Xie, W., Deng, Z., Ding, B., & Kuang, H. (2015). Fixture layout optimization in multi-station assembly processes using augmented ant colony algorithm. *Journal of Manufacturing Systems*, *37*, 277–289. https://doi.org/10.1016/j.jmsy.2014.08.005

Yang, X. (2010). *A New Metaheuristic Bat-Inspired Algorithm. In: González J., Pelta D., Cruz C., Terrazas G. y Krasnogor N. (eds) Nature Inspired Cooperative Strategies for Optimization (NICSO 2010). Studies in Computational Intelligence, vol 284. Springer*. Springer, Berlin, Heidelberg.

Yang, X., & Deb, S. (2009). (2009). Cuckoo search via L'evy fligts. En Proc. the World Congress on Nature and Biologically Inspired Computing, pp: 210 – 214. *Proc. the World Congress on Nature and Biologically Inspired Computing*, 210–214.

Yannis, M., Magdalene, M., & Nikolaos, M. (2010). *Yannis M., Magdalene M., y Nikolaos M. (2010). A Bumble Bees Mating Optimization Algorithm for Global Unconstrained Optimization Problems. En: González J., Pelta D., Cruz C., Terrazas G. y Krasnogor N. (eds) Nature Inspired Cooperative Strategies for Opti* (Vol. 284). Springer, Berlin, Heidelberg.

Yeo, D., & Potra, F. A. (2015). Sustainable Design of Reinforced Concrete Structures through CO2 Emission Optimization. *Journal of Structural Engineering*, *141*(3). https://doi.org/10.1061/(asce)st.1943-541x.0000888

Yepes, V., Alcala, J., Perea, C., & González-Vidosa, F. (2008). A parametric study of optimum earth-retaining walls by simulated annealing. *Engineering Structures*, *30*(3), 821–830. https://doi.org/10.1016/j.engstruct.2007.05.023

Zavala, G. R., Nebro, A. J., Luna, F., & Coello Coello, C. A. (2014). A survey of multi-objective metaheuristics applied to structural optimization. *Structural and Multidisciplinary Optimization*, *49*(4), 537–558. https://doi.org/10.1007/s00158-013-0996-4

Zervoudakis, K., & Tsafarakis, S. (2020). A mayfly optimization algorithm. *Computers and Industrial Engineering*, *145*. https://doi.org/10.1016/j.cie.2020.106559

Anexo 1. Código de Optimización por Colonia de Hormigas

```
% Se limpian variables
clc
clear

% Se definen parámetros iniciales
alfa=2.0;           % Coef. intensidad
beta=2.5;           % Coef. distancia
ro=0.95;            % Coef. evaporación
Q=10^5;             % Feromonas
a=4;                % Hormigas por pueblo
tao=0.1;            % Intensidad inicial
CiclosTotales=25;   % Iteraciones a realizar
R=6367.5;           % Radio estimado de la Tierra
cRestr=1;           % 1=Sí, 0=No

% Se definen las posiciones de cada pueblo
X=load('PueblosMagicos.txt');
X=transpose(X);

% Restricciones
CRest=[8,43,45,55,96,116,118,17,22,42,72,73,83,101];
CY=[8,43,45,55,96,116,118];

% Se definen dimensiones
n=size(X,2); % N de pueblos
m=a*n;       % N de hormigas totales

% Matrices de ceros
D=zeros(n,n);              % De distancia
Nu=zeros(n,n);             % De visibilidad
P=zeros(n,n);              % De probabilidad
L=zeros(m,n);              % De longitudes
MResultados=zeros(m,n);    % De resultados

% Se define las matrices de intensidad
T=tao*ones(n,n);
DT=zeros(n,n);

% Mejores resultados
MejorL=10^9;
MejorRuta=zeros(1,n);

% Se definen matriz de distancia
for i=1 : n
    for j=1 : n
        D(i,j)=2*R*asin(((sind((X(1,j)-X(1,i))/2))^2+
              cosd(X(1,i))*cosd(X(1,j))*(sind((X(2,j)-X(2,i))/2))^2)^(1/2));

        if cRestr==1
            % Se entra a Yucatán
            if size(find(j==CY),2)~=0
                if size(find(i==CRest),2)==0
                    D(i,j)=1000*D(i,j);
                end
```

```
            end

            % Se sale de Yucatán
            if size(find(i==CY),2)~=0
                if size(find(j==CRest),2)==0
                    D(i,j)=1000*D(i,j);
                end
            end

        end

    end
end

% Se define la matriz de visibilidad
for i=1 : n
    for j=1 : n
        if i==j
            Nu(i,j)=0;
        else
            Nu(i,j)=1/D(i,j);
        end
    end
end

% Se define la matriz de probabilidad
for i=1 : n
    for j=1 : n

       P(i,j)=(T(i,j)^alfa)*(Nu(i,j)^beta)/(dot(T(i,:).^alfa,Nu(i,:).^beta));
    end
end

% Se define la matriz de ciudades
b=diag(ones(1,n));
Tabu = b;
for i=1 : a-1
    Tabu=[Tabu;b];
end

% Se aplica el algoritmo, actualización de feromonas por ciclo
for Ciclo=1 : CiclosTotales
    for i=1 : m
        Cont=1; % Contador para almacenar resultados
        d=find(1==Tabu(i,:),1);  % Localiza pueblo inicial i de la hormiga
        MResultados(i,Cont)=d;   % Se inicia ruta

        for j=1 : n
            % Se obtiene lista de posibles movimientos
            if j==n
                U=Tabu(i,:);
                U(1,MResultados(i,1))=0;
            else
                U=Tabu(i,:);
            end

            % Se cambian valores (1=0) y (0=1)
            U(U==0)=2;
            U(U==1)=0;
            U(U==2)=1;
```

```
            % Se descartan las probabilidades de los movimientos no permitidos
            V=U.*(P(d,:));

            % Se crean los intervalos de probabilidad que definen qué
              movimiento realizar
            for k=1 : n
                W(1,k)=sum(V(1,[1:k]));
            end

            % Se normaliza la probabilidad hasta 1
            W=W./W(1,n);

            % c es el pueblo de salida
            c=d;

            % d es el pueblo destino
            d=find(rand<W,1);

            % Se actualiza la matriz de movimientos permitidos
            Tabu(i,d)=1;

            % Se actualiza matriz de distancia y resultados
            Cont=Cont+1;
            L(i,Cont)=D(c,d);
            MResultados(i,Cont)=d;
        end

        % Se determina distancia total recorrida por la hormiga
        L(i,1)=sum(L(i,[2:n+1]));

    end % Se ha completado una iteración

    % Se guardar los mejores resultados de la iteración
    if min(L(:,1))<MejorL
        MejorL= min(L(:,1));
        MejorRuta=MResultados(find(L(:,1)==MejorL),:);
        figure(1)
        comet(X(2,MejorRuta(1,:)),X(1,MejorRuta(1,:)),0.05)
    end

    min(L(:,1))
    % Se actualizan las feromonas generadas en esta iteración
    for i=1 : m
        for j=1 : n-1
            pinicial=MResultados(i,j); % posición inicial
            pfinal=MResultados(i,j+1); % posición final

            DT(pinicial,pfinal)=DT(pinicial,pfinal)+Q/L(i,1);
            T(pinicial,pfinal)=ro*T(pinicial,pfinal)+DT(pinicial,pfinal);
        end
    end

    % Se actualiza la matriz de probabilidad
    for i=1 : n
        for j=1 : n
           P(i,j)=(T(i,j)^alfa)
                  *(Nu(i,j)^beta)/(dot(T(i,:).^alfa,Nu(i,:).^beta));
        end
    end
```

```
174         % Se limpia la matriz de ciudades
175         b=diag(ones(1,n));
176         Tabu = b;
177         for i=1 : a-1
178             Tabu=[Tabu;b];
179         end
180     end
181
182     % Comprobación de la mejor respuesta
183     for i=1 : n
184         MejorD(1,i)=D(MejorRuta(1,i),MejorRuta(1,i+1));
185     end
186     sum(MejorD(1,:))
187
188     % Regresar resultado final
189     Y=importdata('ListaPueblos.txt');
190
191     for i=1 : n
192         CiudadesOrdenadas(1,i)=Y(MejorRuta(1,i),1);
193     end
194
195     CiudadesOrdenadas=transpose(CiudadesOrdenadas);
196
197     % Se grafica la mejor solución encontrada
198     Y=load('Pais.txt');
199
200     % Se define el tipo de gráfica y elementos auxiliares
201     figure(1)
202     plot(Y(:,2),Y(:,1),'color',[0.25 0.8 1])
203     hold on
204     comet(X(2,MejorRuta(1,:)),X(1,MejorRuta(1,:)),0.05)
205     xlabel("Longitud, grados","fontSize",16)
206     ylabel("Latitud, grados","fontSize",16)
207     get(gca,'fontname')
208     set(gca,'fontname','Cambria Math')
209     hold off
```

Anexo 2.1 Código de Búsqueda Tabú

```
% Minimización de una función por medio de BT
clear;
clc,

% Variables del problema
D=10;                           % Dimensiones del problema
f_bias=-330;                    % Valor del mínimo
V=-5:0.1:5;                     % Valores de las D variables
o=V(randi(size(V,2),1,D));      % Posición del mínimo

% Se definen los parámetros de búsqueda
A_max=500;   % Reinicios máximos usando memoria larga
B_max=5000;  % Número máximo de soluciones sin obtener mejora
pV=0.15;     % Porcentaje de variables a modificar
Tenure=1;    % Movimientos que dura un tabú activo
p=0.5;       % Porcentaje de soluciones guardadas

% Matrices de ceros
MS=zeros(1,D);                  % Mejor solución encontrada
TL=zeros(Tenure,round(p*pV*D)); % Lista Tabú
LF=ones(size(V,2),D).*0.05;     % Lista de frecuencias

% Generación de solución inicial aleatoria
X=V(randi(size(V,2),1,D));
F=ValorTS(X,o,f_bias);

% Valores para iniciar el while
B=1; C=1; Fs=[];

for A=1:A_max
   while B<=B_max
       [x,T]=MovimientoTS(D,pV,V,X,Tenure,p,TL);
       f=ValorTS(x,o,f_bias);

       if(f<F)
          X=x; F=f; B=1;
          Fs=[Fs;F];

          % Se activan tabús
          M=T(1,unique(randi(size(T,2),1,ceil(p*size(T,2)))));
          TL(C,:)=0;
          TL(C,1:size(M,2))=M;

          if (C==Tenure)
              C=1;
          else
              C=C+1;
          end

          % Se guardan frecuencias
          for i=1:D
              LF(find(V==X(1,i)),i)=LF(find(V==X(1,i)),i)+1.05^F;
          end
```

```
        else
            B=B+1;
        end
    end

    if(A<A_max)
        B=1;
        TL=zeros(Tenure,round(p*pV*D)); % Reinicio de Lista Tabú

        for i=1:D
            Ruleta=LF(:,i).^-2;
            Ruleta=cumsum(Ruleta./sum(Ruleta(:,1)));
            X(1,i)=V(1,find(rand<Ruleta,1));
        end

        F=ValorTS(X,o,f_bias);
        Fs=[Fs;F];
    end

end

min(Fs)
```

Anexo 2.2 Funciones utilizadas en el capítulo 4

```
function [f] = ValorTS(x,o,f_bias)
% Esta función valora la solución propuesta por TS
x=x-o;
f = sum(x.^2-10.*cos(2.*pi.*x)+10,2)+f_bias;
end
```

```
function [x,Posiciones] = MovimientoTS(D,pV,V,x,Tenure,p,TL)
% Se genera una solución de la vecindad de X
n=randi(round(D*pV)); % Número de variables a modificar

Cambios=randi(2,1,n); % Cambios a realizar
Cambios(Cambios==2)=-1;
Posiciones=unique(randi(D,1,n)); % Variables a modificar

% Se eliminan movimientos tabús
for h=1:size(Posiciones,2)
   for i=1:Tenure
      for j=1:round(p*pV*D)
         if(TL(i,j)==Posiciones(1,h))
            Posiciones(1,h)=0;
         end
      end
   end
end

if(size(find(Posiciones~=0,2)~=0))
    Posiciones(Posiciones==0)=Posiciones(find(Posiciones~=0,1));
    Posiciones=unique(Posiciones);

    i=1;
    m=find(x(1,Posiciones(1,i))==V);

    if(m==1)
        if(Cambios(1,i)~=-1)
            m=m+Cambios(1,i);
        end
    elseif(m==size(V,2))
        if(Cambios(1,i)~=1)
            m=m+Cambios(1,i);
        end
    else
        m=m+Cambios(1,i);
    end

    x(1,Posiciones(1,i))=V(1,m);

end

end
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 3.1. Código de Recocido Simulado

```
% Se limpia la memoria del programa
clc
clear

% Se definen los parámetros del proceso de optimización
T0=1/4;      % Fracción del costo inicial como temperatura inicial
Alfa=0.90;   % Factor de enfriamiento
pf=0.01;     % Fracción de la temperatura inicial como criterio de parada
LCM=1000;    % Longitud de la cadena de Márkov
pV=0.05;     % Porcentaje de variables a modificar en cada movimiento
Pdism=0.80;  % Probabilidad de disminuir áreas transversales
Pquit=0.50;  % Probabilidad de eliminar una barra con área mínima

% Se definen características de las barras
E=200000;    % Módulo de elasticidad, MPa
fy=275;      % Límite de fluencia, MPa
GammaS=7850; % Peso específico del acero, kg/m3
Chi=0.7;     % Disminución de resistencia por compresión

% Vectores que definen valores de las variables consideradas
Areas=(5:5:50);     % Áreas en cm2
DyV=(3.0:0.5:5.0);  % Vector con los posibles valores de Dy

% Se definen los incrementos de los nodos
Dx=8;        % Incrementos en x, en m
Dy=max(DyV); % Incrementos en y, en m

% Se definen los apoyos
% Nodo, Rx, Ry; (1=Restringido, 0=Libre)
R=[1,1,1;
   11, 0, 1];

% Se definen las fuerzas en los nodos, KN
Fx=0;        % Fuerza en X, kN
Fy=-100;     % Fuerza en Y, kN
F=[];

%  Nodo, Fx, Fy
for i=2 : R(2,1)-1
    F=vertcat(F,[i Fx Fy]);
end

% Por el ciclo inicial
Check=0;

TodosPesos=[];
% Creación del diseño inicial
for i=1 : LCM
    if i==1
        while Check==0
            % Se define la distribución inicial de los nodos
            [Posi,Sime,Dy]=Cuadricula(Dx,DyV,Dy);
```

```
            % Se genera la armadura inicial
            [MD,Estab]=ArmaduraAleatoria(R,Areas,Posi,Sime);

            % Solo si es estable se revisa
            if Estab==1
                % Se resuelve la armadura por método de rigidez
                [Barras,Nodos]=MetodoRigidez(R,F,MD,Posi,E);

                % Se realizan comprobaciones y se evalúa la función objetivo
                [Check,Peso,Barras]=ComprobacionesCosto(Chi,fy,GammaS,Barras);

                % Se registra peso de la solución inicial
                TodosPesos=Peso;
            end
        end
    else
        % Se modifica la distribución de los nodos
        [posi,sime,dy]=Cuadricula(Dx,DyV,Dy);

        % Se genera una solución en la vecindad de la MD actual
        [md,Estab]=MovimientoArmadura(pV,Pdism,Pquit,R,Areas,MD,sime);

        % Solo si es estable se revisa
        if Estab==1
            % Se resuelve la armadura por método de rigidez
            [barras,nodos]=MetodoRigidez(R,F,md,posi,E);

            % Se realizan comprobaciones y se evalúa la función objetivo
            [Check,peso,barras]=ComprobacionesCosto(Chi,fy,GammaS,barras);

            % Se comparan funciones objetivos
            if peso<Peso && Check==1
                MD=md;
                Peso=peso;
                Dy=dy;
                Posi=posi;
                Sime=sime;
            end
        end
    end
end

% Se registra peso de la solución antes de SA
TodosPesos(2,1)=Peso;

% Optimización del diseño inicial por SA
Ti=Peso*T0;  % Temperatura inicial del proceso
T=Ti;        % Temperatura actual del proceso

while T>Ti*pf
    for i=1 : LCM
        % Se genera una solución en la vecindad de la MD actual
        [md,Estab]=MovimientoArmadura(pV,Pdism,Pquit,R,Areas,MD,Sime);

        % Solo si es estable se revisa
        if Estab==1
            % Se resuelve la armadura por método de rigidez
            [barras,nodos]=MetodoRigidez(R,F,md,Posi,E);

            % Se realizan comprobaciones y se evalúa la función objetivo
            [Check,peso,barras]=ComprobacionesCosto(Chi,fy,GammaS,barras);
```

```
114             % Se comparan funciones objetivos
115             if peso<Peso && Check==1
116                 MD=md;
117                 Peso=peso;
118                 Barras=barras;
119                 Nodos=nodos;
120
121                 % Se registra peso de la solución actual
122                 if TodosPesos(size(TodosPesos,1),1)~=Peso
123                     TodosPesos(size(TodosPesos,1)+1,1)=Peso;
124                 end
125             else
126                 % Se revisa criterio de aceptación
127                 if rand()<exp(-((peso-Peso)/T)) && Check==1
128                     MD=md;
129                     Peso=peso;
130                     Barras=barras;
131                     Nodos=nodos;
132
133                     % Se registra peso de la solución actual
134                     if TodosPesos(size(TodosPesos,1),1)~=Peso
135                         TodosPesos(size(TodosPesos,1)+1,1)=Peso;
136                     end
137                 end
138             end
139         end
140     end
141     % Se actualiza la temperatura del proceso
142     T=Alfa*T;
143 end
144
145 % Se genera una representación gráfica de la armadura
146 GraficaArmadura(Posi,Barras,Nodos)
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 3.2. Funciones utilizadas en el capítulo 5

```
function [Posiciones,Simetria,Dy] = Cuadricula(Dx,DyV,Dy)
% Se definen las posiciones de los nodos
Px1=[0,(1:0.5:5),6];
Px2=(1:0.5:5);
Px=horzcat(Px1,Px2);

dy=find(Dy==DyV);

if dy~=1 && dy~=size(DyV,2)
    if rand<0.5
        Dy=DyV(1,dy+1);
    else
        Dy=DyV(1,dy-1);
    end
else
    if dy==1 && rand<0.5
        Dy=DyV(1,dy+1);
    elseif dy==size(DyV,2) && rand<0.5
        Dy=DyV(1,dy-1);
    end
end

Px1=vertcat(Dx*Px1,0*Dy*ones(1,size(Px1,2)));
Px2=vertcat(Dx*Px2,1*Dy*ones(1,size(Px2,2)));

% Matriz que almacena las posiciones de los nodos
Posiciones=(1:1:size(Px,2));
Px=horzcat(Px1,Px2);
Posiciones=transpose(vertcat(Posiciones,Px));

% Se define una matriz para aplicar simetría
A=unique(Posiciones(:,3));
Simetria=[];

for i=1 : size(A,1)
   B=find(Posiciones(:,3)==A(i,1));
   for j=1 : (size(B,1)-1)/2
      Simetria=vertcat(Simetria,[Posiciones(B(j,1),1),...
          Posiciones(max(B)-j+1)]);
   end
end

% Se introducen nodos sin simetría
Simetria=vertcat(Simetria,[setdiff(Posiciones(:,1),Simetria),...
    setdiff(Posiciones(:,1),Simetria)]);
end
```

```
function [MD,Estabilidad] = ArmaduraAleatoria(R,Areas,Posiciones,Simetria)
% Esta función define la solución inicial del proceso de optimización
% Se genera la base de la MD
MD=zeros(size(Posiciones,1),size(Posiciones,1));
% La solución inicial no tiene todas las conexiones activas
```

```
for i=1 : size(MD,1)
    for j=1 : size(MD,2)
        if j==i+1
            MD(i,j:size(MD,2))=round(rand(1,size(MD,2)-j+1));
        end
        if MD(i,j,1)==1
            MD(i,j,2)=Areas(randi(size(Areas,2)))/10000;
        end
    end
end

% Se impone simetría en la estructura
for i=1 : size(MD,1)
    [a1,b1]=find(Simetria==i);
    X1=Simetria(a1(1,1),:);

    if size(find(X1~=i),2)==0
        c1=X1(1,1);
    else
        c1=X1(1,find(X1~=i,1));
    end

    for j=1 : size(MD,2)
        if j>=i+1
            [a2,b2]=find(Simetria==j);
            X2=Simetria(a2(1,1),:);

            if size(find(X2~=j),2)==0
                c2=X2(1,1);
            else
                c2=X2(1,find(X2~=j,1));
            end

            MD(min(c1,c2),max(c1,c2),:)=MD(i,j,:);
        end
    end
end

% Se determina si la armadura es estable
% Para calcular el grado de hiperestaticidad total
% GHT = GHI + GHE

% GHE = N incógnitas - N restricciones
GHE=3-sum(sum(R(:,2:3)));

% GHI = Coaxiones internas - GDLI
% GDLI = 3*(Barras totales - 1)
GDLI=3*(sum(sum(MD(:,:,1)))-1);

% Coacciones internas = 2*Nodos*(Barras que convergen al nodo - 1)
% Se determinan las barras que convergen a cada nodo
Conv=sum(MD(:,:,1),1)+transpose(sum(MD(:,:,1),2));
CI=zeros(1,max(Conv)-1);

% Si el modelo generado tiene un nodo conectado a una única barra
if find(Conv==1,2)~=0
    Estabilidad=0;
    return
else
    % Se calculan las coacciones internas
    for i=2:max(Conv)
```

```
113             if size(find(Conv==i),2)~=0
114                 CI(1,i-1)=2*size(find(Conv==i),2)*(i-1);
115             end
116         end
117
118         % Cálculo de GHT
119         CI=sum(CI);
120         GHI=CI-GDLI;
121         GHT=GHI+GHE;
122
123         if GHT<=0
124             Estabilidad=0;
125         else
126             Estabilidad=1;
127         end
128     end
129
130     end



131     function [Barras,Nodos] = MetodoRigidez(R,F,MD,Posiciones,E)
132     % Se aplica el método de rigidez para resolver la armadura
133
134     % Se define el vector de barras por MD
135     Barras=[];
136
137     for i=1 : size(Posiciones,1)
138         for j=1 : size(Posiciones,1)
139             if MD(i,j)==1
140                 Barras=cat(1,Barras,[i,j,MD(i,j,2),E]);
141             end
142         end
143     end
144
145     % Se define el vector de nodos por MD
146     Nact=transpose(unique(Barras(:,(1:2))));
147     Nodos=[];
148
149     for i=1: size(Nact,2)
150         X1=find(Posiciones(:,1)==Nact(1,i));
151         Nodos=cat(1,Nodos,[Nact(1,i),Posiciones(X1,(2:3))]);
152     end
153
154     % Se agregan elementos a la matriz nodos
155     Nodos=horzcat(Nodos,zeros(size(Nodos,1),2));
156
157     % Se enumeran los grados de libertad
158     for i=1:size(Nodos,1)
159         % Si el nodo no tiene restricción
160         if size(find(Nodos(i,1)==R(:,1)),1)==0
161             Nodos(i,1:5)=horzcat(Nodos(i,1:3),...
162                 1+max(max(Nodos(:,4:5))),2+max(max(Nodos(:,4:5))));
163
164         % Si esta parcialmente restringido
165         else
166             if size(find(0==R(R(:,1)==Nodos(i,1),2:3)),2)==1
167                 Nodos(i,1:5)=horzcat(Nodos(i,1:3),...
168                     1+max(max(Nodos(:,4:5))),1+max(max(Nodos(:,4:5))));
169                 Nodos(i,3+find(1==R(R(:,1)==i,2:3)))=0;
170             end
```

```
    end
end

% Resto de nodos
z=transpose(find(0==Nodos(:,4:5)));
for i=1:size(z,2)
    Nodos(3*size(Nodos,1)+z(1,i))=1+max(max(Nodos(:,4:5)));
end

% Se crea la matriz de fuerzas en cada nodo
FN=zeros(2*size(Nodos,1),1);

% Se asocian fuerzas a los GDL de los nodos
for i=1:size(F,1)
    FN(Nodos(F(i,1),4))=F(i,2);
    FN(Nodos(F(i,1),5))=F(i,3);
end

Barras=horzcat(Barras,zeros(size(Barras,1),4));

% Matriz que almacena todas la matrices de rigidez
Kb=zeros(5,4,size(Barras,1));

% Se definen las matrices de rigidez de cada barra
for i=1:size(Barras,1)
    X1=find(Nodos(:,1)==Barras(i,1));
    X2=find(Nodos(:,1)==Barras(i,2));

    L=((Nodos(X2,2)-Nodos(X1,2))^2+(Nodos(X2,3)-Nodos(X1,3))^2)^(1/2);
    Cx=(Nodos(X2,2)-Nodos(X1,2))/L;
    Sy=(Nodos(X2,3)-Nodos(X1,3))/L;

    Barras(i,5)=L;
    Barras(i,6)=Cx;
    Barras(i,7)=Sy;

    % Se registran los GDL de cada barra
    for j=1:2
        X3 = find(Nodos(:,1)==Barras(i,j));
        Kb(1,2*j-1,i)=(Nodos(X3,4));
        Kb(1,2*j,i)=(Nodos(X3,5));
    end

    % Se introducen resto de datos
    Kb(2:5,:,i)=Barras(i,3)*Barras(i,4)*10^3/L...
        *[Cx^2,Cx*Sy,-(Cx^2),-Cx*Sy;
        Cx*Sy,Sy^2,-Cx*Sy,-(Sy^2);
        -(Cx^2),-Cx*Sy,Cx^2,Cx*Sy;
        -Cx*Sy,-(Sy^2),Cx*Sy,Sy^2];
end

% Se ensambla la matriz de rigidez de la armadura
KG=zeros(2*size(Nodos,1));
for h=1:size(Kb,3)          % Barra
    for i=1:size(Kb,2)      % Filas
       for j=1:size(Kb,2)   % Columnas
           KG(Kb(1,i,h),Kb(1,j,h))=KG(Kb(1,i,h),Kb(1,j,h))+Kb(i+1,j,h);
       end
    end
end
% Se obtienen los desplazamientos
```

```
DLsup=size(KG,1)-size(find(R(:,2:3)==1),1);
DL=pinv(KG(1:DLsup,1:DLsup))*FN(1:DLsup,1);

% Se calculan las reacciones
FRinf=DLsup+1;
FR=KG(FRinf:size(KG,1),1:DLsup)*DL;

% Calculo de fuerzas axiales
DL=vertcat(DL,zeros(size(KG,1)-DLsup,1,1));

% Para asegurar la no singularidad de la matriz
if round(sum(FR(2:3,1)))==round(abs(sum(FN)))
%     Valores negativos indican compresión de la barra
    for h=1:size(Barras,1)
        DNb=[DL(Kb(1,1,h),1);DL(Kb(1,2,h),1);DL(Kb(1,3,h),1);
            DL(Kb(1,4,h),1)];
        Barras(h,8)=Barras(h,3)*Barras(h,4)*10^3/Barras(h,5)*...
            [-Barras(h,6),-Barras(h,7),Barras(h,6),Barras(h,7)]*...
            DNb;
    end
else
    for h=1:size(Barras,1)
        Barras(h,8)=10^10;
    end
end
end



function [Check,Peso,Barras] = ComprobacionesCosto(Chi,fy,GammaS,Barras)
% Comprobaciones de la resistencia de las barras
Peso=0; % Por si no cumple comprobaciones

for i=1 : size(Barras,1)
    if Barras(i,8)<0 % A compresión
        if abs(Barras(i,8))<Chi*fy*10^3*Barras(i,3)
            Check=1; % Cumple
            Barras(i,9)=abs(Barras(i,8))/(Chi*fy*10^3*Barras(i,3));
        else
            Check=0; % No cumple
            return
        end
    else % A tensión
        if Barras(i,8)<fy*10^3*Barras(i,3)
            Check=1; % Cumple
            Barras(i,9)=Barras(i,8)/(fy*10^3*Barras(i,3));
        else
            Check=0; % No cumple
            return
        end
    end
end

% Se determina el peso total de la estructura
Peso=zeros(size(Barras,1),1);

for i=1 : size(Barras,1)
   Peso(i,1)=Barras(i,3)*Barras(i,5)*GammaS;
end
Peso=sum(Peso);
end
```

```
function [MD,Estabilidad] = MovimientoArmadura(pV,Pdism,Pquit,R,Areas,MD,Simetria)
% Se generar una nueva solución en una vecindad de MD actúal
VectorMD=[];

for i=1 : size(MD,1)
    for j=1 : size(MD,2)
        if j==i+1
            VectorMD=horzcat(VectorMD,MD(i,j:size(MD,2),2)*10000);
        end
    end
end

% Se hace que valor de MD sean enteros
VectorMD=fix(VectorMD);

% Vector que selecciona variables a modificar aleatoriamente
m=unique(randi(size(VectorMD,2),1,round(randi(size(VectorMD,2))*pV)));

% Proceso de modificación
for i=1 : size(m,2)
    if VectorMD(1,m(1,i))~=0 && VectorMD(1,m(1,i))~=5 &&...
            VectorMD(1,m(1,i))~=max(Areas)
        if rand()<Pdism
            VectorMD(1,m(1,i))=Areas(1,find(Areas==VectorMD(1,m(1,i)))-1);
        else
            VectorMD(1,m(1,i))=Areas(1,find(Areas==VectorMD(1,m(1,i)))+1);
        end
    else
        if VectorMD(1,m(1,i))==0 && rand<1-Pdism
            VectorMD(1,m(1,i))=min(Areas);
        elseif VectorMD(1,m(1,i))==5 && rand<Pdism
            VectorMD(1,m(1,i))=0;
        end
    end
end

% Se actualiza MD
a=1;
for i=1 : size(MD,1)
    for j=1 : size(MD,2)
        if j==i+1
            b=size(MD,2)-j+a;
            MD(i,j:size(MD,2),2)=VectorMD(1,a:b)/10000;
            a=b+1;
        end
        if MD(i,j,2)~=0
            MD(i,j,1)=1;
        elseif MD(i,j,2)==0
            MD(i,j,1)=0;
        end
    end
end

% Se elimina una barra con área transversal mínima
if rand()<Pquit
    [a,b]=find(MD(:,:,2)==min(Areas)/10000);
    c=horzcat(a,b);
    if size(c,1)~=0
        Borrar=randi(size(c,1));
        MD(c(Borrar,1),c(Borrar,2),:)=0;
    end
```

```
end

% Se determinan las barras que convergen a cada nodo
Conv=sum(MD(:,:,1),1)+transpose(sum(MD(:,:,1),2));

% Se elimina un nodo aleatorio con dos barras
if size(find(Conv==2),2)~=0
    Borrar=find(Conv==2);
    Borrar=Borrar(1,randi(size(Borrar,2)));
    if Borrar>7
        MD(Borrar,:,:)=0;
        MD(:,Borrar,:)=0;
    end
end

% Se impone la existencia del cordón inferior
for h=1:5
    if MD(h,h+1,1)==0
        MD(h,h+1,1)=1;
        MD(h,h+1,2)=Areas(randi(size(Areas,2)))/10000;
    end
end

% Se impone simetría en la estructura
for i=1 : size(MD,1)
    [a1,b1]=find(Simetria==i);
    X1=Simetria(a1(1,1),:);

    if size(find(X1~=i),2)==0
        c1=X1(1,1);
    else
        c1=X1(1,find(X1~=i,1));
    end

    for j=1 : size(MD,2)
        if j>=i+1
            [a2,b2]=find(Simetria==j);
            X2=Simetria(a2(1,1),:);

            if size(find(X2~=j),2)==0
                c2=X2(1,1);
            else
                c2=X2(1,find(X2~=j,1));
            end

            MD(min(c1,c2),max(c1,c2),:)=MD(i,j,:);
        end
    end
end

% Se determina si la armadura es estable
% Para calcular el grado de hiperestaticidad total
% GHT = GHI + GHE

% GHE = N incognitas - N restricciones
GHE=3-sum(sum(R(:,2:3)));

% GHI = Coaxiones internas - GDLI
% GDLI = 3*(Barras totales - 1)
GDLI=3*(sum(sum(MD(:,:,1)))-1);

```

```
412 % Coacciones internas = 2*Nodos*(Barras que convergen al nodo - 1)
413 % Se determinan las barras que convergen a cada nodo
414 Conv=sum(MD(:,:,1),1)+transpose(sum(MD(:,:,1),2));
415 CI=zeros(1,max(Conv)-1);
416
417 % Si el modelo generado tiene un nodo conectado a una única barra
418 if find(Conv==1,2)~=0
419     Estabilidad=0;
420     return
421 else
422     % Se calculan las coaxiones internas
423     for i=2:max(Conv)
424         if size(find(Conv==i),2)~=0
425             CI(1,i-1)=2*size(find(Conv==i),2)*(i-1);
426         end
427     end
428
429     % Cálculo de GHT
430     CI=sum(CI);
431     GHI=CI-GDLI;
432     GHT=GHI+GHE;
433
434     if GHT<=1
435         Estabilidad=0;
436     else
437         Estabilidad=1;
438     end
439 end
440 end
```

```
441 function [] = GraficaArmadura(Posiciones,Barras,Nodos)
442 % Se grafica la solución actual
443 close(figure(1));
444
445 figure(1)
446 title('Solución actual')
447 hold on
448
449 % Para escalar ejes X y Y
450 x0=[-2,max(Posiciones(:,2))+2];
451 y0=[-1,-1];
452 plot(x0,y0,('w'))
453
454 x0=[0,0];
455 y0=[-1,0.15*max(Posiciones(:,2))];
456 plot(x0,y0,('w'))
457
458 for i=1: size(Barras,1)
459     x=[Nodos(find(Nodos==Barras(i,1),1),2),...
460         Nodos(find(Nodos==Barras(i,2),1),2)];
461     y=[Nodos(find(Nodos==Barras(i,1),1),3);...
462         Nodos(find(Nodos==Barras(i,2),1),3)];
463     plot(x,y,'color',[1 0.7 0])
464 end
465
466 hold off
467 end
```

Anexo 4.1. Código de Algoritmos Genéticos

```
% Optimización de una viga por Algoritmos Genéticos
clear

% Se definen los valores de las variables
Vfc=(25:5:70);              % fc = Resistencia del concreto
VH=(0.35:0.05:2.4);         % H = Peralte de la sección
VB=(0.35:0.05:2.4);         % B = Ancho de la sección
Vn=(2:1:30);                % nc,n1,n2 = N barras (compresión, tensión sup, tensión inf)
Vfi=[3,4,5,6,8,10,12];      % fic,fi1,fi2 = Diámetro acero (compresión, tensión
                              sup, tensión inf)
VnE=[1,2];                  % nE = N de estribos
VfiE=[2.5,3];               % fiE = Diámetro de los estribos
VsE=(0.075:0.025:0.30);     % sE = Separación de los estribos (Apoyos y centro)

% Parámetros del problema
L=12;          % Longitud viga, m
BApoyo=0.30;   % Ancho de los apoyos, m
rec=0.04;      % Recubrimiento, m
w=4.5;         % Carga distribuida servicio, kN/m
Wm=6.25;       % Carga distribuida rotura, kN/m

% Parámetros del algoritmo de búsqueda
TP=450;          % Tamaño de la población inicial
TV=450;          % Tamaño de la población de descendientes
TT=2;            % Tamaño del torneo
GenMax=100;      % Número máximo de generaciones
PCruce=0.9;      % Probabilidad de cruce
PMutacion=0.8;   % Probabilidad de mutación

% Matrices de ceros
Poblacion=zeros(TP,15);
Torneo=zeros(2,TT);
Participantes=zeros(1,TT);
Vv=zeros(14,max([size(Vfc,2),size(VH,2),size(VB,2),size(Vn,2),size(VsE,2)]));
Costos=zeros(TP,1);
Vastagos=zeros(TP,15);
CostosV=zeros(TV,1);
Cmin=zeros(GenMax,1);

% Se determina el espacio de soluciones del problema
EspacioSoluciones = size(Vfc,2)*size(VH,2)*size(VB,2)*size(Vn,2)^3*...
    size(Vfi,2)^3*size(VnE,2)*size(VfiE,2)*size(VsE,2)

% Matriz con todos los valores de las variables
Vv(1,1:size(Vfc,2))=Vfc;
Vv(2,1:size(VH,2))=VH;
Vv(3,1:size(VB,2))=VB;
Vv(4,1:size(Vn,2))=Vn;
Vv(5,1:size(Vfi,2))=Vfi;
Vv(6,1:size(Vn,2))=Vn;
Vv(7,1:size(Vfi,2))=Vfi;
Vv(8,1:size(Vn,2))=Vn;
Vv(9,1:size(Vfi,2))=Vfi;
Vv(10,1:size(VnE,2))=VnE;
```

```
Vv(11,1:size(VfiE,2))=VfiE;
Vv(12,1:size(VsE,2))=VsE;
Vv(13,1:size(VsE,2))=VsE;

% Se genera la población inicial
i=1; % Para iniciar el while
while i<=TP
    Poblacion(i,1)=0;
    Poblacion(i,2)=Vv(1,randi(size(Vfc,2)));
    Poblacion(i,3)=Vv(2,randi(size(VH,2)));
    Poblacion(i,4)=Vv(3,randi(size(VB,2)));
    Poblacion(i,5)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,6)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,7)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,8)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,9)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,10)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,11)=Vv(10,randi(size(VnE,2)));
    Poblacion(i,12)=Vv(11,randi(size(VfiE,2)));
    Poblacion(i,13)=Vv(12,randi(size(VsE,2)));
    Poblacion(i,14)=Vv(12,randi(size(VsE,2)));

    % Comprobaciones de la solución
    [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro]=...
        ComprobacionesAG(Poblacion(i,1:14),L,rec,BApoyo,Wm,w);
    Poblacion(i,1)=Check;
    Poblacion(i,15)=Le;

    if(Poblacion(i,1)==0)
        [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,
                sECentro,Vfc);
        Costos(i,1)=Costo;
        i=i+1;
    end
end

CostoMin=min(Costos(:,1))
Cmin(1,1)=CostoMin;

% Repeticion de las generaciones
for Generacion=1:GenMax
    % Se aplica estrategia elitista
    Vastagos(1,1:15)=Poblacion(find(CostoMin==Costos,1),1:15);
    CostosV(1,1)=Costos(find(CostoMin==Costos,1),1);

    % Creación de una nueva generación
    i=2; % Para iniciar el while
    while i<=TV
        % Se escoge pareja 1
        a=0;
        % Se determinan los participantes del torneo
        while a==0
            Participantes=randi(TV,1,TT);
            if size(unique(Participantes),2)==TT
                a=1;
            end
        end

        % Se define el torneo
        for j=1:TT
           Torneo(1,j)=Participantes(1,j);
```

```
116             Torneo(2,j)=Costos(Participantes(1,j),1);
117         end
118
119         % Primera pareja
120         N1=find(Torneo(2,:)==min(Torneo(2,:)),1);
121         N1=Torneo(1,N1);
122         Pareja1=Poblacion(N1,1:size(Vv,1));
123
124         % Se escoge pareja 2
125         a=0;
126         % Se determinan los participantes del torneo
127         while a==0
128             Participantes=randi(TV,1,TT);
129             if size(find(Participantes==N1),2)==0
130                 if size(unique(Participantes),2)==TT
131                     a=1;
132                 end
133             end
134         end
135
136         % Se define el torneo
137         for j=1:TT
138             Torneo(1,j)=Participantes(1,j);
139             Torneo(2,j)=Costos(Participantes(1,j),1);
140         end
141
142         % Segunda pareja
143         N2=find(Torneo(2,:)==min(Torneo(2,:)),1);
144         N2=Torneo(1,N2);
145         Pareja2=Poblacion(N2,1:size(Vv,1));
146
147         % Inicia proceso de cruce
148         if(rand()<PCruce)
149             % Puntos de corte entre variables
150             a1=0;
151             while a1==0
152                 Union=sort(randi(size(Vv,1)-1,1,2));
153                 if size(unique(Union),2)==2
154                     a1=1;
155                 end
156             end
157
158             % Primera descendencia
159             Vastagos(i,1:Union(1,1))=Pareja1(1,1:Union(1,1));
160             Vastagos(i,Union(1,1)+1:Union(1,2))=Pareja2(1,Union(1,1)+
161                                                 1:Union(1,2));
162             Vastagos(i,Union(1,2)+1:size(Vv,1))=Pareja1(1,Union(1,2)+
163                                                 1:size(Vv,1));
164
165             % Proceso de mutación
166             if(rand()<PMutacion)
167                 [Gen,Mut]=Mutacion(Vastagos(i,1:14),Vv);
168                 Vastagos(i,Gen)=Mut;
169             end
170
171             % Se revisa la validez de la viga
172             [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro]=...
173                 ComprobacionesAG(Vastagos(i,1:14),L,rec,BApoyo,Wm,w);
174             Vastagos(i,1)=Check;
175             Vastagos(i,15)=Le;
```

```
            if(Vastagos(i,1)==0
                [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,
                        sEApoyo,sECentro,Vfc);
                CostosV(i,1)=Costo;
                i=i+1;
            end

            % Para no crear soluciones adicionales
            if(i>TV)
                break
            end

            % Segunda descendencia
            Vastagos(i,1:Union(1,1))=Pareja2(1,1:Union(1,1));
            Vastagos(i,Union(1,1)+1:Union(1,2))=Pareja1(1,Union(1,1)+
                                                1:Union(1,2));
            Vastagos(i,Union(1,2)+1:size(Vv,1))=Pareja2(1,Union(1,2)+
                                                1:size(Vv,1));

            % Proceso de mutación
            if(rand()<PMutacion)
                [Gen,Mut]=Mutacion(Vastagos(i,1:14),Vv);
                Vastagos(i,Gen)=Mut;
            end

            % Se revisa la validez de la viga
            [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro]=...
                ComprobacionesAG(Vastagos(i,1:14),L,rec,BApoyo,Wm,w);
            Vastagos(i,1)=Check;
            Vastagos(i,15)=Le;

            if(Vastagos(i,1)==0)
                [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,
                        sEApoyo,sECentro,Vfc);
                CostosV(i,1)=Costo;
                i=i+1;
            end
        else
            Vastagos(i,1:15)=Poblacion(N1,1:15);
            CostosV(i,1)=Costos(N1,1);
            i=i+1;
            if(i<TV)
                Vastagos(i,1:15)=Poblacion(N2,1:15);
                CostosV(i,1)=Costos(N2,1);
            end
        end
    end

    Poblacion=Vastagos(1:TV,:);
    Vastagos=zeros(TV,1);
    Costos=CostosV;
    CostoMin=min(Costos(:,1))
    Cmin(Generacion,1)=CostoMin;
end
```

Anexo 4.2. Funciones utilizadas en el capítulo 6

```
function [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro] =
ComprobacionesAG(Viga,L,rec,BApoyo,Wm,w)
% Esta función determina la validez de la viga y define la longitud
% donde se colocarán los estribos en el apoyo

% Se llaman los valores de la viga
fc   =Viga(1,2);
H    =Viga(1,3);
B    =Viga(1,4);
nc   =Viga(1,5);
fic  =Viga(1,6)*(2.54/800);
ns1  =Viga(1,7);
fis1 =Viga(1,8)*(2.54/800);
ns2  =Viga(1,9);
fis2 =Viga(1,10)*(2.54/800);
nE   =Viga(1,11);
fiE  =Viga(1,12)*(2.54/800);
sEApoyo=Viga(1,13);
sECentro=Viga(1,14);

% Calculos de apoyo
fcc=0.85*fc;    % Es f''c, MPa
if(fc>28)       % Altura efectiva bloque de compresión, m
    beta1=1.05-fc/140;
else
    beta1=0.85;
end
if(fc>=40)      % Módulo elasticidad concreto, MPa
    Ec=2700*fc^(1/2)+11000;
else
    Ec=4400*fc^(1/2);
end

fy=420;     % Esfuerzo de fluencia del acero, MPa
Es=200000;  % Módulo de elasticidad del acero, MPa

Nu=Es/Ec;   % Relación entre módulos

Asc=nc*pi()/4*(fic)^2;      % Área a compresión, m2
As1=ns1*pi()/4*(fis1)^2;    % Área a tensión superior, m2
As2=ns2*pi()/4*(fis2)^2;    % Área a tensión inferior, m2
Av=2*nE*pi()/4*(fiE)^2;     % Área estribos apoyo, m2

ro=(As1+As2)/(B*H);       % Cuantía a tensión
roc=Asc/(B*H);            % Cuantía a compresión


sepV = 0.04;        % Separación entre armaduras verticales, m
BApoyo = 0.30;      % Ancho de los apoyos, m

Yg=(As1*(sepV+fis1/2+fis2/2))/(As1+As2); % Centro de gravedad de las armaduras, m
d=H-rec-fiE-As2/2-Yg; % Es d (peralte efectivo), m
dc=rec+fiE+fic/2; % Es d', m
Le=0;
```

```
Check=1;

% Relación geométrica
if(H/B>6)
    Check=1;
    return
end

% Separación entre refuerzos longitudinales
if((B-2*rec-2*fiE-nc*fic)/(nc-1)<0.04)
    Check=2;
    return
end

if((B-2*rec-2*fiE-ns1*fis1)/(ns1-1)<0.04)
    Check=3;
    return
end

if((B-2*rec-2*fiE-ns2*fis2)/(ns2-1)<0.04)
    Check=4;
    return
end

% Revisión flexión
a=(As1+As2-Asc)*fy/(fcc*B); % Profundidad bloque de compresión
Mflex=1.5*Wm*L^2/8+1.3*(25*H*B)*L^2/8;

% Acero a tensión mínimo
if(As1+As2<(0.22*(fc*10^3)^(1/2)*B*d/(fy*10^3)))
    Check=5;
    return
end

% Acero máximo a tensión
if(As1+As2>0.90*(600*beta1/(600+fy)*fcc*B*d/(fy)+Asc))
    Check=6;
    return
end

% Fluencia del acero a compresión
if(As1+As2-Asc<600*beta1*fcc*B*dc/((600-fy)*fy))
    Check=7;
    return
end

% Momento de rotura
if(Mflex>0.90*((As1+As2-Asc)*fy*(d-a/2)+Asc*fy*(d-dc))*10^3)
    Check=8;
    return
end

% Revisión cortante
Vu=1.5*Wm*L/2-1.5*Wm*(BApoyo/2+d)+1.3*(25*B*H)*L/2-1.3*(25*B*H)*(BApoyo/2+d);
if(ro<0.015)
    Vcr=0.75*(0.2+20*ro)*0.3*(fc*10^3)^(1/2)*B*d;
else
    Vcr=0.75*0.16*(fc*10^3)^(1/2)*B*d;
end
VsR=0.75*Av*fy*(10^3)*d/sEApoyo;

```

```
% Vcr máximo
if(Vcr>0.75*(0.47)*(fc*10^3)^(1/2)*B*d)
    Check=9;
    return
end

% Vu máximo
if(Vu>0.75*(0.80)*(fc*10^3)^(1/2)*B*d)
    Check=10;
    return
end

% Separaciones máximas estribos
if(Vu>0.75*(0.47)*(fc*10^3)^(1/2)*B*d)
    if(sEApoyo>0.25*d)
        Check=11;
        return
    end
else
    if(Vu>Vcr)
        if(sEApoyo>0.50*d)
            Check=12;
            return
        end
    end
end

% Resistencia a cortante
if(Vcr+VsR<Vu)
    Check=13;
    return
end

% Se define cambio de separación de estribos
VsRc=0.75*Av*fy*(10^3)*d/sECentro/nE;
Le=(1.5*Wm*L/2+1.3*25*B*H*L/2-Vcr-VsRc)/(1.5*Wm+1.3*25*B*H);

% Comprobaciones de deflexión
AFN=B/2;
BFN=Asc*(Nu-1)+(As1+As2)*Nu;
CFN=-Asc*(Nu-1)*dc-(As1+As2)*Nu*d;
FN=(-BFN+(BFN^2-4*AFN*CFN)^(1/2))/(2*AFN);

Iag=B*FN^3/12+B*FN^3/4+(Nu-1)*Asc*(FN-dc)^2+Nu*(As1+As2)*(FN-d)^2;

% Deflexión elástica
Delas=5*w*L^4/(384*Ec*10^3*Iag);

% Deflexión diferida
Ddifer=2/(1+50*roc)*Delas;

% Deflexión total
if(Delas+Ddifer>L/240)
    Check=14;
    return
end

% Comprobaciones de agrietamiento
fsMax=40000;
fsAcero=(w*L^2/8)*(d-FN)*Nu/Iag;
df=rec+fiE+fis2/2;
```

```
Afis=B*(Yg+df)*2/((As1+As2)/min(As1/ns1,As2/ns2));
h1=d-FN;
h2=H-FN;

if(fsMax<fsAcero*(df*Afis)^(1/3)*h2/h1)
    Check=15;
    return
end

Check=0;

% Fin de la función
end



function [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro,Vfc)
% Esta función determina el costo de la viga actual

% Costos de los materiales
Cfc = [1103.2, 1253.7, 1354.8, 1420.3, 1502.5, 1570.1, 1668.2, 1722.9, 1777.6, 1832.3]; %
MXN/m3
Cacero = 19.64;     % MXN/kg
Ccimbra = 220.0;    % MXN/m2

% Costo del concreto
Concreto=(L+BApoyo)*B*H*Cfc(1,find(Vfc==fc));

% Costo del acero longitudinal
Longitudinal=(L+BApoyo+H)*(Asc+As1+As2)*Cacero*7860;

% Costo estribos
if(Le-BApoyo/2-d<0)
    Lc=L-BApoyo;
    EstribosApoyo=2*(1+1)*(B-2*rec+H-2*rec)*Av*Cacero*7860;
    EstribosCentro=(round(Lc/sECentro)-1)*(B-2*rec+H-2*rec)*Av*Cacero*7860;
else
    Lc=L-(2*round((Le-BApoyo/2-d)/sEApoyo)*sEApoyo+d+BApoyo/2);
    EstribosApoyo=2*(round((Le-BApoyo/2)/sEApoyo)+1)*(B-2*rec+H-2*rec)*Av*Cacero*7860;
    EstribosCentro=(round(Lc/sECentro)-1)*(B-2*rec+H-2*rec)*Av*Cacero*7860;
end

% Costo cimbra
Cimbra=(L+BApoyo)*(B+2*H)*Ccimbra;

% Costo total
Costo=Concreto+Longitudinal+EstribosApoyo+EstribosCentro+Cimbra;
end



function [Gen,Mut] = Mutacion(Vastago,Vv)
% Función que define la variable a mutar y cómo hacerlo
a=randi(13);    % Variable a mutar
Gen=a+1;        % Posición en el código de la viga
Xx=Vv(a,1:find(Vv(a,:)==0,1)-1); % Valores de la variable

if(size(Xx,2)==0)
    Xx=Vv(a,:);
end

```

```
232     Go=Vastago(1,Gen);  % Valor actual
233     p=find(Go==Xx);     % Posición en el vector
234     m=randi(2);         % Dirección del cambio
235
236     if(m==1)
237         if(p~=1)
238             Mut=Xx(1,p-1);  % Nuevo valor
239         else
240             Mut=Xx(1,p);    % No se altera el valor
241         end
242     else
243         if(p~=size(Xx,2))
244             Mut=Xx(1,p+1);  % Nuevo valor
245         else
246             Mut=Xx(1,p);    % No se altera el valor
247         end
248     end
249
250     end
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 5.1. Código de Estrategias Evolutivas

```
% Optimización de una viga por Estrategias Evolutivas
clear

% Se definen los valores de las variables
Vfc=(25:5:70);              % fc = Resistencia del concreto
VH=(0.35:0.05:2.4);         % H = Peralte de la sección
VB=(0.35:0.05:2.4);         % B = Ancho de la sección
Vn=(2:1:30);                % nc,n1,n2 = N barras (compresión, tensión sup, tensión inf)
Vfi=[3,4,5,6,8,10,12];      % fic,fi1,fi2 = Diámetro acero (compresión, tensión
                              sup, tensión inf)
VnE=[1,2];                  % nE = N de estribos
VfiE=[2.5,3];               % fiE = Diámetro de los estribos
VsE=(0.075:0.025:0.30);     % sE = Separación de los estribos (Apoyos y centro)

% Parámetros fijos del problema
L=12;          % Longitud viga, m
BApoyo=0.30;   % Ancho de los apoyos, m
rec=0.04;      % Recubrimiento, m
w=4.5;         % Carga distribuida servicio, kN/m
Wm=6.25;       % Carga distribuida rotura, kN/m

% Parámetros del algoritmo de búsqueda
Mu=12;         % Tamaño de la población
GenMax=2500;   % Número máximo de generaciones
PM=0.50;       % Probabilidad mutación hacia la vecindad

% Matrices de ceros
Poblacion=zeros(Mu,15);
Vv=zeros(13,max([size(Vfc,2),size(VH,2),size(VB,2),size(Vn,2),size(VsE,2)]));
Costos=zeros(Mu,1);
Solucion=zeros(1,15);
Cmin=zeros(GenMax,1);

% Se determina el espacio de soluciones del problema
EspacioSoluciones = size(Vfc,2)*size(VH,2)*size(VB,2)*size(Vn,2)^3*...
    size(Vfi,2)^3*size(VnE,2)*size(VfiE,2)*size(VsE,2)

% Matriz con todos los valores de las variables
Vv(1,1:size(Vfc,2))=Vfc;
Vv(2,1:size(VH,2))=VH;
Vv(3,1:size(VB,2))=VB;
Vv(4,1:size(Vn,2))=Vn;
Vv(5,1:size(Vfi,2))=Vfi;
Vv(6,1:size(Vn,2))=Vn;
Vv(7,1:size(Vfi,2))=Vfi;
Vv(8,1:size(Vn,2))=Vn;
Vv(9,1:size(Vfi,2))=Vfi;
Vv(10,1:size(VnE,2))=VnE;
Vv(11,1:size(VfiE,2))=VfiE;
Vv(12,1:size(VsE,2))=VsE;
Vv(13,1:size(VsE,2))=VsE;

% Se genera la población inicial
i=1; % Para iniciar el while
```

```
while i<=Mu
    Poblacion(i,1)=0;
    Poblacion(i,2)=Vv(1,randi(size(Vfc,2)));
    Poblacion(i,3)=Vv(2,randi(size(VH,2)));
    Poblacion(i,4)=Vv(3,randi(size(VB,2)));
    Poblacion(i,5)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,6)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,7)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,8)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,9)=Vv(4,randi(size(Vn,2)));
    Poblacion(i,10)=Vv(5,randi(size(Vfi,2)));
    Poblacion(i,11)=Vv(10,randi(size(VnE,2)));
    Poblacion(i,12)=Vv(11,randi(size(VfiE,2)));
    Poblacion(i,13)=Vv(12,randi(size(VsE,2)));
    Poblacion(i,14)=Vv(12,randi(size(VsE,2)));

    % Comprobaciones de la solución
    [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro]=...
        ComprobacionesAG(Poblacion(i,1:14),L,rec,BApoyo,Wm,w);
    Poblacion(i,1)=Check;
    Poblacion(i,15)=Le;

    if(Poblacion(i,1)==0)
        [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,
                          sEApoyo,sECentro,Vfc);
        Costos(i,1)=Costo;
        i=i+1;
    end
end

CostoMin=min(Costos(:,1))
Cmin(1,1)=CostoMin;

% Repeticion de las generaciones
Generacion=1; % Para iniciar el while
while Generacion<GenMax
    % Creación de una nueva solución
    for i=1 : size(Vv,1)
        Solucion(1,i+1)=Poblacion(randi(Mu),i+1);

        if(rand()<=PM)
            [Mut]=EEMutacion(Solucion,Vv,i);
            Solucion(1,i+1)=Mut;
        else
            if(rand()<=1-PM)
                [Mut]=EEMutacion2(Vv,i);
                Solucion(1,i+1)=Mut;
            end
        end
    end
    % Se revisa la validez de la viga
    [Check,Le,B,H,fc,d,Asc,As1,As2,Av,sEApoyo,sECentro]=...
        ComprobacionesAG(Solucion,L,rec,BApoyo,Wm,w);

    Solucion(1,15)=Le;

    if(Check==0)
        [Costo]=CostoViga(L,Le,rec,BApoyo,B,H,fc,d,Asc,As1,As2,Av,
                          sEApoyo,sECentro,Vfc);
        Generacion=Generacion+1;
```

```
115             if(max(Costos)>Costo)
116                 j=find(max(Costos)==Costos,1);
117                 Poblacion(j,:)=Solucion;
118                 Costos(j,1)=Costo;
119             end
120             CostoMin=min(Costos(:,1))
121             Cmin(Generacion,1)=CostoMin;
122             Solucion=zeros(1,15);
123         end
124     end
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 5.2. Funciones utilizadas en el capítulo 7

```
function [Mut] = EEMutacion(Solucion,Vv,i)
% Función que define la variable a mutar y cómo hacerlo
Xx=Vv(i,1:find(Vv(i,:)==0,1)-1); % Valores de la variable

if(size(Xx,2)==0)
    Xx=Vv(i,:);
end

p=find(Solucion(1,i+1)==Xx);    % Posición en el vector
m=randi(2);                     % Dirección del cambio

if(m==1)
    if(p~=1)
        Mut=Xx(1,p-1);  % Nuevo valor
    else
        Mut=Xx(1,p);    % No se altera el valor
    end
else
    if(p~=size(Xx,2))
        Mut=Xx(1,p+1);  % Nuevo valor
    else
        Mut=Xx(1,p);    % No se altera el valor
    end
end

end

function [Mut] = EEMutacion2(Vv,i)
% Función que define la variable a mutar y cómo hacerlo
Xx=Vv(i,:);     % Valores de la variable

if(size(find(Xx==0,1),2)==0)
    Mut=Xx(1,randi(size(Xx,2)));        % Mutación aleatoria
else
    Mut=Xx(1,randi(find(Xx==0,1)-1));   % Mutación aleatoria
end
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 6.1. Código de Búsqueda por Vecindario Variable

```
% Se limpian variables
clc
clear

% Se definen parámetros del problema
pV=0.40;
Vecindarios=[1,2,3,5,7,10];
IterMax=[50,35,35,25,25,20];

% Se definen intervalos de las variables de decisión
dH=0.1;
rho=(1300:10:2000);
G=(50000:500:100000);
Bulk=(190000:500:420000);
Coh=(20:1:60);
refPress=(5:1:110);

% Matrices con tamaño predefinido
MD=zeros(6,7);
md=zeros(6,7);
VD=zeros(5,size(Bulk,2)+1);
H=zeros(1,size(MD,2));
A=2*ones(size(MD,1),size(MD,2));
Error=[];

% Se carga el modelo inicial y se almacena en MD
% 1)Profundidad; 2)Densidad; 3)G; 4)Bulk; 5)Cohesión; 6)Presión de
% referencia
MD(1,:)=[5.0 8.0 22.0 28.0 31.0 43.0 48.0];
MD(2,:)=[1300 1500 1400 1600 1600 1600 1800];
MD(3,:)=[50000 60000 75000 65000 75000 80000 90000];
MD(4,:)=[190000 210000 220000 200000 250000 350000 400000];
MD(5,:)=[25.0 30.0 35.0 25.0 30.0 45.0 50.0];
MD(6,:)=[10.0 50.0 50.0 50.0 80.0 100.0 100.0];
DespI=load('DespTopInicial.out');

% Agrupamos las variables de decisión en una matriz
VD(1,1:size(rho,2))=rho;
VD(2,1:size(G,2))=G;
VD(3,1:size(Bulk,2))=Bulk;
VD(4,1:size(Coh,2))=Coh;
VD(5,1:size(refPress,2))=refPress;

% Se cargan las desplazamientos reales
DespR=load('DespTopReal.out');
EC=sum((DespI-DespR).^2);
Error=EC;

for k=1:size(Vecindarios,2)
    n=Vecindarios(1,k);
    for m=1:IterMax(1,k)
        % Matriz con valores a modificar aleatoriamente
        a=round(rand(1,randi(round(pV*size(MD,1)*size(MD,2)))));
        for i=1:size(a,2)
```

```
            A(randi(size(MD,1)),randi(size(MD,2)))=a(1,i);
        end

        % Creación de nueva solución en vecindad
        md=MD;
        for i=1:size(md,2)
            if i==1
                H(1,i)=md(1,i);
            else
                H(1,i)=md(1,i)-md(1,i-1);
            end
            for j=1:size(md,1)
                if j==1
                    if A(j,i)==0
                        if H(1,i)-n*dH>0
                            md(j,i)=md(j,i)-n*dH;
                        end
                    elseif A(j,i)==1
                        if i==size(md,2)
                            md(j,i)=md(j,i)+n*dH;
                        else
                            if md(j,i)+n*dH<md(j,i+1)
                                md(j,i)=md(j,i)+n*dH;
                            end
                        end
                    end
                else
                    Var=VD(j-1,1:(find(VD(j-1,:)==0)-1));
                    B=find(md(j,i)==Var);
                    if A(j,i)==0
                        B=max(1,B-n);
                        md(j,i)=Var(1,B);
                    elseif A(j,i)==1
                        B=min(size(Var,2),B+n);
                        md(j,i)=Var(1,B);
                    end
                end
            end
        end

        % Se genera un nuevo modelo para OpenSees
        VariablesSuelo;

        % Se analiza el modelo en OpenSees
        !OpenSees.exe "SueloDinamico.tcl"
        Desp=load('DespTop.out');

        % Se determina el Error Cuadrático
        if size(Desp,1)==size(DespR,1)
            ec=sum((Desp-DespR).^2);
        end

        if EC>ec
            k=1;
            m=1;
            MD=md;
            EC=ec;
            Error(1,size(Error,2)+1)=EC;
        end
```

```
114             % Reinicio de A
115             A=2*ones(size(MD,1),size(MD,2));
116
117         end
118     end
119
120     % Resultado final
121     md=MD;
122     VariablesSuelo;
123     !OpenSees.exe "SueloDinamico.tcl"
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 6.2. Funciones utilizadas en el capítulo 8

```
% Se crea el archivo con los datos del nuevo modelo
file = 'VariablesDecision.tcl';

% Se modifica el archivo de las variables de decisión para OpenSees
fileID = fopen(file,'w');
fprintf(fileID, '# Variables de decisión de nuestro modelo. Arcilla media\n');

fprintf(fileID, '# Lista de profundidades de capas de suelo, en m\n')
fprintf(fileID, 'set H {%2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f};\n\n', md(1,:));

fprintf(fileID, '# Densidad del terreno en kg/m3\n')
fprintf(fileID, 'set rho {%d %d %d %d %d %d %d};\n\n', md(2,:));

fprintf(fileID, '# Módulo a cortante en kPa\n')
fprintf(fileID, 'set Gr {%d %d %d %d %d %d %d};\n\n', md(3,:));

fprintf(fileID, '# Módulo de Bulk en kPa\n')
fprintf(fileID, 'set Br {%d %d %d %d %d %d %d};\n\n', md(4,:));

fprintf(fileID, '# Cohesión del terreno en kPa\n')
fprintf(fileID, 'set c {%2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f};\n\n', md(5,:));

fprintf(fileID, '# Presión de referencia en kPa\n')
fprintf(fileID, 'set refPress {%2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f};\n\n', md(6,:));

type(file);
fclose(fileID);
```

Esta página ha sido intencionalmente dejada en blanco

Anexo 6.3. Código del modelo en OpenSees

```
# Se modela la respuesta dinámica de una columna de suelo arcilloso, los valores
  utilizados
# son de referencia y se toman del manual en línea de OpenSees. Link:
# https://opensees.berkeley.edu/wiki/index.php/PressureIndependMultiYield_Material

# Se crea un modelo de 2 dimensiones y dos grados de libertar por nodo
wipe
model BasicBuilder -ndm 2 -ndf 2
set pi 3.141592654;

# Se definen los parámetros del modelo
set N 7;            # Número de capas de suelo
set gammaPeak 0.1;  # Deformación a cortante máxima
set phi 0.0;        # Ángulo de fricción
set presCoef 0.0;   # Coeficiente de presión de referencia
set nu 0.0;         # Módulo de Poisson
set B 3.0;          # Ancho columna considerada en m
set Esp 1.0;        # Espesor elementos en m

# Variables de decisión de las capas de suelo
source VariablesDecision.tcl

# Valores del terreno subyacente
set rockVs 800;    # Velocidad de ondas en m/s
set rockDen 2.4e3; # Densidad del lecho rocoso en kg/m3

# Se define los nodos del modelo
set Hd [lindex $H [expr $N-1]]
for {set j 1} {$j <= [expr (2*$N)+1]} {incr j 2} {
       if {$j==1} {
              node $j          0.0 0.0
              node [expr $j+1] $B  0.0
       } elseif {$j==[expr (2*$N)+1]} {
              node $j          0.0 $Hd
              node [expr $j+1] $B  $Hd
       } else {
              node $j          0.0 [expr $Hd-[lindex $H [expr ($N-2)-($j-3)/2]]]
              node [expr $j+1] $B  [expr $Hd-[lindex $H [expr ($N-2)-($j-3)/2]]]
       }; # Fin if
}; # Fin for

# Se definen apoyos fijos en la base de la columna de suelo
fix 1 0 1
fix 2 0 1

# Se ligan los GDL traslacionales de la columna de suelo
for {set k 3} {$k <= [expr (2*$N)+1]} {incr k 2} {
    equalDOF  $k  [expr $k+1]  1 2
}

# Definir nodos para el amortiguamiento, elemento de longitud cero
node 100 0.0 0.0
node 101 0.0 0.0
```

```
# Definir los apoyos como fijos del amortiguamiento
fix 100 1 1
fix 101 0 1

# Se ligan los desplazamientos laterales de los nodos
equalDOF 1   2 1
equalDOF 1 101 1

# Se definen los materiales de los elementos
for {set j 0} {$j <= [expr $N-1]} {incr j 1} {
     set k [expr $N-1-$j]
     nDMaterial PressureIndependMultiYield 10$j 2 [lindex $rho $k] [lindex $Gr $k] [lindex
$Br $k] [lindex $c $k] $gammaPeak $phi [lindex $refPress $k] $presCoef
}

# Material uniaxial viscoso
set C [expr $B*$rockVs*$rockDen]
uniaxialMaterial Viscous 201 $C 1

# Se definen los elementos del modelo
for {set j 0} {$j <= [expr $N-1]} {incr j 1} {
     set k [expr $N-1-$j]
     set Wx 0.0;
     set Wy [expr [lindex $rho $k]*-9.81/1000];
     element quad 30$j [expr 2*$j+1] [expr 2*$j+2] [expr 2*$j+4] [expr 2*$j+3] $Esp
"PlaneStrain" 10$j 0.0 [lindex $rho $k] $Wx $Wy
}

# Elemento de longitud zero para el amortiguamiento
element zeroLength 401 100 101 -mat 201 -dir 1

# Se define la ventana que permitirá visualizar el modelo.
    recorder display "Columna suelo" 10 10 600 600 -wipe
    prp 0 0 50
    vup 0 1 0
    vpn 0 0 1
    display 1 2 10

# Se definen los registros a guardar
recorder Node -file DespTop.out -node 15 -dof 1 disp;

# Se realiza el análisis estático
system BandGeneral
numberer RCM
constraints Plain
test NormDispIncr 1.0e-2 25;
integrator LoadControl 0.1
algorithm Newton
analysis Static
analyze 10

# Se realiza análisis dinámico
setTime 0.0
wipeAnalysis

set Factor 0.01; # El registro de aceleraciones se encuentra en cm/s2, este factor lo
                   convierte a m/s2.
set dt 0.01;        # Diferencial de tiempo del acelerograma.
set Npuntos 20000;  # Número de puntos del acelerograma.
set AccelDataFile "SCT18509.txt"; # Se indica el registro a utilizar.
set DirX 1;         # Dirección en la que se aplicarán las aceleraciones.
```

```
115  # Se define el patrón de aceleraciones.
116  set accelSeries "Series -dt $dt -filePath $AccelDataFile -factor $Factor";
117  pattern UniformExcitation 2 $DirX -accel $accelSeries
118
119  # Se define el amortiguamiento de la estructura.
120  set dampRatio 0.02
121  set omega1 [expr 2*$pi*0.2];     # lower frequency
122  set omega2 [expr 2*$pi*20];      # upper frequency
123  set a0 [expr 2*$dampRatio*$omega1*$omega2/($omega1 + $omega2)]; # coeficientes de
124  amortiguamiento
125  set a1 [expr 2*$dampRatio/($omega1 + $omega2)];
126  rayleigh $a0 $a1 0.0 0.0
127
128  # Se definen los parámetros del análisis estático.
129  system UmfPack
130  numberer RCM
131  constraints Plain
132  test NormDispIncr 1.0e-8 10
133  integrator Newmark 0.5 0.25
134  algorithm Newton
135  analysis Transient
136  analyze $Npuntos $dt
```

Las Series del Instituto de Ingeniería describen los resultados de algunas de las investigaciones más relevantes de esta institución. Con frecuencia son trabajos in extenso de artículos que se publican en revistas especializadas, memorias de congresos, etc.

Cada número de estas Series se edita con la aprobación técnica del Comité Editorial del Instituto, basada en la evaluación de árbitros competentes en el tema, adscritos a instituciones del país y/o el extranjero.

Actualmente hay tres diferentes Series del Instituto de Ingeniería:

SERIE INVESTIGACIÓN Y DESARROLLO

Incluye trabajos originales sobre investigación y/o desarrollo tecnológico. Es continuación de la Serie Azul u Ordinaria, publicada por el Instituto de Ingeniería desde 1956, la cual actualmente tiene nueva presentación y admite textos en español e inglés.

SERIE DOCENCIA

Está dedicada a temas especializados de cursos universitarios para facilitar a estudiantes y profesores una mejor comprensión de ciertos temas importantes de los programas de estudio.

SERIE MANUALES

Abarca manuales útiles para resolver problemas asociados con la práctica profesional o textos que describen y explican el estado del arte o el estado de la práctica en ciertos temas. Incluye normas, manuales de diseño y de laboratorio, reglamentos, comentarios a normas y bases de datos.

Las Series del Instituto de Ingeniería pueden consultarse gratuitamente desde la dirección electrónica del Instituto http://www.ii.unam.mx (http://aplicaciones.iingen.unam.mx/ConsultasSPII/Buscarpublicacion.aspx) y pueden grabarse o imprimirse en formato PDF desde cualquier computadora.

INSTITUTO DE INGENIERÍA UNAM®