

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“MHTMETLIB: A NEW LIBRARY TO MEASURE THE PERFORMANCE OF TRAJECTORY METAHEURISTICS”

JOAQUÍN ANDRÉS GATICA HERNÁNDEZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Nicolás Rojas Morales
Profesor Correferente: Elizabeth Montero

Octubre - 2024

DEDICATORIA

Dedicado a a mi familia y amigos, gracias al apoyo generado durante estos años, me tienen dando este nuevo paso en mi vida.

AGRADECIMIENTOS

Primero que nada, agradecer a mi profesor guía, Nicolas Rojas, por darme la oportunidad de trabajar junto a el en esta memoria, gracias por todo el tiempo, palabras y dedicación, que dieron como resultado este trabajo final con el cual termino mi etapa como estudiante. Junto a esto agradecer a todo los profesores que me formaron durante estos años, no me quedo solo con los conocimientos, si no que tambien con lo humano.

Gracias a toda la gente que logre conocer en esta etapa universitaria, agradecer a mis amigos quimicos, que fueron los primeros con los cuales logre armar lazos de amistad, que hasta el dia de hoy se mantienen, les deseo lo mejor para todos en sus vidas. Por otro lado, como no olvidar a toda la gente del Perro, los cuales no solo fueron grandes compañeros de carrera y trabajos durante estos años, si no que tambien tengo el agrado de decir que son las mejores personas que pude haber conocido durante esta etapa, gracias por todas las juntas, cumpleaños y celebraciones que tuvimos durante estos años, me llevo grandes recuerdos y espero que esta relación siempre continúe.

Por otro lado, agradecer a todos mis amigos que me acompañaron durante estos años, a mi grupo del Karo, a mis compañeros del colegio y otros más, que me ayudaron a despejarme mediante el ocio y esas noches de conversación en Discord.

Por ultimo, agradecer a mi familia que no solo estuvo conmigo estos ultimos años, si no que han estado apoyandome desde siempre. Gracias Juan Nicolas y Rafael, por ser los mejores hermanos que pude haber tenido y poder tenerlos como ejemplos a seguir desde que yo era niño, y por otro lado a mis padres, Juan y Angelica, que siempre estuvieron ahí para entregarme ese animo que me permitio seguir adelante apesar de las adversidades y problemas que surgian, todo este camino recorrido se lo dedico completamente a ustedes y espero que los llene de felicidad al igual que a mi.

RESUMEN

Resumen— Los algoritmos metaheurísticos proponen una buena forma de solucionar problemas, al comparar con técnicas con enfoques determinísticos o técnicas completas de búsqueda, permitiendo disminuir los tiempos de ejecución a cambio de obtener soluciones aproximadas al óptimo. El problema es que su forma estocástica conlleva incerteza para medir su rendimiento, dificultando el análisis de los experimentos y el diseño de ellos. Es por ello que se propone en este trabajo diseñar e implementar una librería para medir el rendimiento de algoritmos metaheurísticos trayectoriales en problemas combinatoriales, permitiendo un análisis completo de ellos en etapas de experimentación. Se valida esta solución con un conjunto de algoritmos trayectoriales clásicos utilizando el framework `IOHprofiler`, permitiendo obtener un análisis completo acerca de distintas características propias de cada algoritmo en la ejecución de distintos problemas *benchmark* de optimización binaria.

Palabras Clave— Algoritmos Metaheurísticos Trayectoriales; Métricas de rendimiento; Problemas de optimización.

ABSTRACT

Abstract— Metaheuristic algorithms offer a good way to solve problems when compared to deterministic approaches or complete search techniques, allowing for reduced execution times in exchange for obtaining solutions that are approximate to the optimal. The problem is that their stochastic nature brings uncertainty when measuring their performance, making it difficult to analyze experiments and design them. Therefore, this work proposes to design and implement a library to measure the performance of trajectory-based metaheuristic algorithms in combinatorial problems, allowing a comprehensive analysis of them during experimentation stages. This solution is validated with a set of classic trajectory-based algorithms using the `IOHprofiler` framework, enabling a complete analysis of various characteristics specific to each algorithm when solving different binary optimization benchmark problems.

Keywords— Trajectory-Based Metaheuristic Algorithms; Performance Metrics; Optimization Problems.

GLOSARIO

ACO: *Ant Colony Optimization*

BSO: *Brain Storm Optimization*

DE: *Differential Evolution*

FEV: *Function evaluations*

HC: *Hill Climbing*

IOH: *Iterative Optimization Heuristics*

RS o RLS: *Randomized Local Search*

PBO: *Pseudo-Boolean Optimization*

PSO: *Particle Swarm Optimization*

SA: *Simulated Annealing*

SFO: *Synergistic Fibroblast Optimization*

TS: *Tabu Search*

TSP: *Travelling Salesman Problem*

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	VIII
ÍNDICE DE TABLAS	X
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	3
1.1 Objetivos	7
1.1.1 Objetivo General	7
1.1.2 Objetivos Específicos	8
CAPÍTULO 2: MARCO CONCEPTUAL	9
2.1 Conceptos metaheurísticos	9
2.2 Algoritmos trayectoriales	10
2.2.1 <i>Hill Climbing</i>	11
2.2.2 <i>Randomized Local Search</i>	13
2.2.3 <i>Simulated Annealing</i>	15
2.3 Métricas de rendimiento	16
2.3.1 Ratio de convergencia	16
2.3.2 Ratio de convergencia geométrico	19
2.3.3 Ratio de error	20
2.3.4 Diversidad	21
2.4 IOHprofiler	24
2.4.1 IOHalgorithm	26
2.4.2 IOHexperimenter	28
2.4.3 IOHanalyzer	28
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	33
3.1 Lectura de logs	35
3.2 Métricas propuestas en MHTMetLib	35
3.2.1 Curva de convergencia	36

3.2.2	<i>Current Best</i>	36
3.2.3	Ratio de Error	38
3.2.4	Ratio de Convergencia	39
3.2.5	Ratio de Convergencia Geométrico	40
3.2.6	Diversidad	41
3.2.7	Diversidad basada en distancia al centro	43
3.2.8	Distancia de Hamming	46
3.2.9	Suma acumulada de proceso de intensificación y diversificación	49
3.2.10	Distribución de mejora o empeoramiento	50
3.2.11	Resumen de métricas implementadas	52
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN		54
4.1	PBO	54
4.1.1	<i>OneMax</i>	54
4.1.2	<i>LeadingOnes</i>	54
4.1.3	<i>Harmonic weights</i>	55
4.1.4	<i>W-model-transformed OneMax/LeadingOnes</i>	55
4.1.5	<i>Low-Autocorrelation Binary Sequence</i>	59
4.1.6	<i>The Ising model</i>	59
4.1.7	<i>Maximum independent vertex set</i>	61
4.1.8	N reinas	62
4.1.9	<i>Concatenated Trap</i>	62
4.1.10	<i>NK landscapes</i>	63
4.2	Escenario experimental	63
4.3	Resultados	64
4.4	Análisis del algoritmo gHC	67
4.5	Análisis del algoritmo RLS	74
4.6	Análisis del algoritmo SA	78
CAPÍTULO 5: CONCLUSIONES		84
ANEXOS		87
REFERENCIAS BIBLIOGRÁFICAS		90

ÍNDICE DE FIGURAS

1	Explosión combinatorial del problema del vendedor viajero (TSP)	3
2	Clasificación de algoritmos metaheurísticos	5
3	Curvas de convergencia para instancias de TSP	18
4	Tasas de convergencia promedio geométrico de SFO, PSO y DE	20
5	Curvas de error relativo según cada función objetivo	21
6	Monitor de diversidad poblacional del algoritmo BSO en funciones uni- modales.	24
7	Módulos y flujo de IOHprofiler	25
8	Curva de tiempo esperado para distintos algoritmos en IOHanalyzer . .	29
9	Histograma de función de masa de probabilidad en IOHanalyzer	30
10	Boxplot de función de masa de probabilidad en IOHanalyzer	31
11	Curva empírica de función de distribución acumulada en IOHanalyzer .	32
12	Integración MHTMetLib con flujo de IOHprofiler	34
13	Formato de archivo de salida <i>logger</i>	35
14	Gráfico de dispersión y <i>boxplot</i> de calidad de solución actual por iteración	36
15	Gráfico de dispersión y <i>boxplot</i> de calidad mejor solución actual por iteración	38
16	Gráfico de dispersión y <i>boxplot</i> de ratio de error por iteración	39
17	Gráfico de dispersión y <i>boxplot</i> de ratio de convergencia por iteración . .	40
18	Gráfico de dispersión y <i>boxplot</i> de ratio de convergencia geométrico por iteración	41
19	Gráfico de dispersión y <i>boxplot</i> de diversidad por iteración	42
20	División de regiones en particiones	45
21	Gráfico de dispersión y <i>boxplot</i> de diversidad basada en distancia al centro por iteración	46
22	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la solución inicial por iteración	48
23	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming entre iteraciones	48
24	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la mejor solución actual por iteración	49
25	Gráfico de dispersión y <i>boxplot</i> de suma acumulada de proceso de inten- sificación y diversificación por iteración	50
26	<i>Boxplot</i> de distribución de mejora y empeoramiento en ejemplo de eje- cución	52
27	Funciones de rugosidad r_1 , r_2 y r_3	58
28	Gráfico de dispersión y <i>boxplot</i> de calidad de solución actual por iteración en LABS (f_{18}) de dimensión 10000	68

29	Gráfico de dispersión y <i>boxplot</i> de ratio de error por iteración en LABS (f_{18}) de dimensión 10000	68
30	Gráfico de dispersión y <i>boxplot</i> de suma acumulada de proceso de intensificación y diversificación por iteración en LABS (f_{18}) de dimensión 10000	69
31	Gráfico de dispersión y <i>boxplot</i> de diversidad por iteración en LABS (f_{18}) de dimensión 10000	69
32	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la solución inicial por iteración en LABS (f_{18}) de dimensión 10000	70
33	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming entre iteraciones en LABS (f_{18}) de dimensión 10000	71
34	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la mejor solución actual por iteración en LABS (f_{18}) de dimensión 10000	71
35	Gráfico de dispersión y <i>boxplot</i> de distribución de <i>delta</i> de mejora de ejecución en LABS (f_{18}) de dimensión 10000	72
36	Gráfico de dispersión y <i>boxplot</i> de distribución de <i>delta</i> de empeoramiento en LABS (f_{18}) de dimensión 10000	72
37	Gráfico de dispersión y <i>boxplot</i> de calidad de solución actual por iteración en OneMax + r_3 (f_{10}) de dimensión 10000	73
38	Gráfico de dispersión y <i>boxplot</i> de diversidad basada en distancia al centro por iteración en OneMax + r_3 (f_{10}) de dimensión 10000	74
39	Gráfico de dispersión y <i>boxplot</i> de diversidad por iteración en OneMax + r_3 (f_{10}) de dimensión 10000	74
40	Gráfico de dispersión y <i>boxplot</i> de calidad de solución actual por iteración en LeadingOne + Epistasis (f_{14}) de dimensión 10000	75
41	Gráfico de dispersión y <i>boxplot</i> de ratio de convergencia por iteración en LeadingOne + Epistasis (f_{14}) de dimensión 10000	75
42	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la solución inicial por iteración en LeadingOne + Epistasis (f_{14}) de dimensión 10000	76
43	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming entre iteraciones en LeadingOne + Epistasis (f_{14}) de dimensión 10000	76
44	Gráfico de dispersión y <i>boxplot</i> de distancia de Hamming sobre la mejor solución actual por iteración en LeadingOne + Epistasis (f_{14}) de dimensión 10000	77
45	Gráfico de dispersión y <i>boxplot</i> de diversidad por iteración en LeadingOne + Epistasis (f_{14}) de dimensión 10000	77
46	Gráfico de dispersión y <i>boxplot</i> de diversidad basada en distancia al centro en LeadingOne + Epistasis (f_{14}) de dimensión 10000	78

47	Gráfico de dispersión y <i>boxplot</i> de curva de convergencia basado en áreas por iteración en <i>IsingRing</i> (f_{19}) de dimensión 10000	79
48	Gráfico de dispersión y <i>boxplot</i> de suma acumulada de proceso de intensificación y diversificación en <i>IsingRing</i> (f_{19}) de dimensión 10000	79
49	Gráfico de dispersión y <i>boxplot</i> de diversidad basada en distancia al centro en <i>IsingRing</i> (f_{19}) de dimensión 10000	80
50	Gráfico de dispersión de curva de convergencia para N Reinas(f_{23}) de dimensión 100, 625 y 5625	81
51	Gráfico de dispersión de distancia de Hamming sobre la solución inicial por iteración en N Reinas(f_{23}) de dimensión 100, 625 y 5625	82
52	Gráfico de <i>boxplot</i> de distancia de Hamming entre iteración en N Reinas(f_{23}) de dimensión 100, 625 y 5625	83

ÍNDICE DE TABLAS

1	Resumen de métricas implementadas en MHTMetLib	53
2	Calidad mejor <i>run</i> en dimensiones de 100 y 625 variables	65
3	Calidad mejor <i>run</i> en dimensiones de 5625 y 10000 variables	66
4	Tabla de resultados para problemas de <i>suite</i> PBO de dimensión 100 . . .	87
5	Tabla de resultados para problemas de <i>suite</i> PBO de dimensión 625 . . .	88
6	Tabla de resultados para problemas de <i>suite</i> PBO de dimensión 5625 . . .	88
7	Tabla de resultados para problemas de <i>suite</i> PBO de dimensión 10000 . .	89

INTRODUCCIÓN

El diseño de algoritmos de optimización es una disciplina que se enfoca en encontrar la mejor solución posible a un problema, dentro de un conjunto de posibles soluciones, bajo ciertas condiciones o restricciones que modelan procesos, tareas o trabajos, que se aplican en diversas áreas, como la ingeniería, economía, inteligencia artificial, biología, y más.

El desafío principal en el diseño de estos algoritmos, es desarrollar métodos que sean eficientes, robustos y escalables para resolver problemas complejos en un tiempo razonable, garantizando al mismo tiempo una buena calidad de las soluciones obtenidas. Un algoritmo puede operar como una *caja negra*, es decir, un sistema en el que solo se conoce la entrada y la salida, pero no se tiene una comprensión detallada de los procesos internos o las decisiones que se toman durante su ejecución. Esto dificulta la identificación de áreas de mejora y la comprensión del funcionamiento del algoritmo, ya que no se dispone de información clara sobre su comportamiento, la lógica de sus decisiones o las tendencias que surgen durante los experimentos.

Es por esto, que en este trabajo se propone una librería con un conjunto de métricas recopiladas del estado de arte, las cuales fueron definidas y analizadas para comprender su significado, para luego realizar una prueba de uso en un conjunto de problemas clásicos de optimización binaria, permitiendo obtener resultados y conclusiones claras de la implementación de un grupo de algoritmos trayectoriales de búsqueda.

El presente trabajo se estructura en cinco capítulos, que se detallan a continuación: en el Capítulo 1, se presenta la definición del problema a abordar, las dificultades actuales en torno a dicho problema, la motivación para su estudio, así como los objetivos planteados para el desarrollo de este trabajo. A continuación, en el Capítulo 2, se exponen los conceptos teóricos y las definiciones fundamentales relacionadas con los algoritmos metaheurísticos, además de una revisión del estado del arte sobre las métricas de rendimiento en este ámbito. En el Capítulo 3, se introduce la librería de métricas de rendimiento propuesta para su uso en algoritmos trayectoriales, la cual incluye tanto métricas provenientes del estado del arte como definiciones desarrolladas por el autor de este trabajo. Posteriormente, en el Capítulo 4, se lleva a cabo la validación de la integración de estas métricas en el flujo de un framework de optimización, aplicándolas a tres algoritmos trayectoriales clásicos mediante la resolución de un conjunto de problemas de optimización binaria. Finalmente, en el Capítulo 5, se presentan las conclusiones derivadas del trabajo realizado

y se proponen posibles líneas de investigación para trabajos futuros.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

En el área de la optimización, se trabajan problemas con una alta complejidad computacional, como por ejemplo definir una planificación, determinar una ruta, asignar recursos o secuenciar el genoma humano [Crescenzi *et al.*, 1995]. A menudo, estos problemas involucran un gran número de combinaciones posibles, y a medida que se modelan casos reales, donde aumenta la cantidad de dimensiones que se trata de resolver, la cantidad de soluciones posibles crece de manera exponencial.

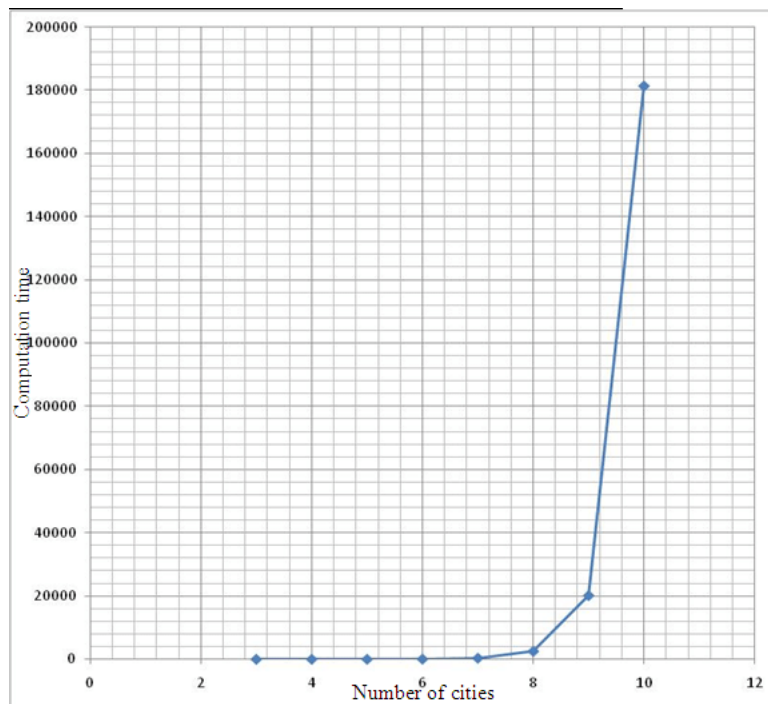


Figura 1: Explosión combinatorial del problema del vendedor viajero (TSP)

Fuente: Using Multi-Agents Systems to Resolve an NP-Complete Problem: Travelling Salesman Problem [Abdoun *et al.*,]

Este crecimiento exponencial del problema, el cual se denomina formalmente como explosión combinatorial, hace inviable evaluar todas las combinaciones, ya que el tiempo requerido para ello, en algunos casos, sobrepasa la propia vida humana [Žerovnik, 2015].

Si bien, existen técnicas de búsqueda completa que filtran este espacio de búsqueda para reducir la cantidad de soluciones a comprobar, como por ejemplo algoritmos de *Forward Checking*, estos siguen siendo intratables. Ante ello, nace un conjunto de algoritmos que busca cambiar la forma de explorar estas soluciones, los cuales se denominan algoritmos de búsqueda incompleta.

Los algoritmos de búsqueda incompleta se pueden dividir en 2 categorías: algoritmos heurísticos y algoritmos metaheurísticos. Ambos métodos están basados en el mismo concepto, establecen estrategias para recorrer el espacio de soluciones del problema transformando de forma iterativa una solución inicial o construyendo soluciones. Las heurísticas tienden a ser dependientes al problema de interés, mientras que las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos, combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos. Además, las metaheurísticas están diseñadas para ser generales y utilizadas en múltiples problemas, ya que no utilizan información o características particulares del problema a resolver.

Los algoritmos metaheurísticos se definen como heurísticas de alto nivel, que permiten encontrar soluciones a un amplio conjunto de problemas combinatoriales [Yang, 2010]. Una característica particular de estos algoritmos es que tienen un factor aleatorio o estocástico. La aleatoriedad ofrece una sólida alternativa a estos algoritmos para evitar quedar atrapados en óptimos locales. Estos algoritmos a menudo se estancan debido a la limitación de soluciones que se encuentran en una región específica. Por otro lado, la búsqueda de nuevas regiones se realiza con el objetivo de encontrar nuevos óptimos, que mejoren la solución actual encontrada. Estas dos propiedades se definen como exploración y explotación, las cuales si existe un balance entre ellos, podrían permitir obtener de forma adecuada soluciones de calidad.

La complejidad de estos problemas hace imposible revisar todas las soluciones posibles del espacio de búsqueda. En ese sentido, se apunta a buscar un conjunto de soluciones, es decir, que cumplan todas las restricciones y condiciones del problema, y que sean de buena calidad en una escala de tiempo aceptable. Además, no se garantiza obtener el óptimo global, sino más bien iterar en múltiples soluciones que se acerquen al óptimo.

Una forma de clasificar estos algoritmos es dividirlos en 3 enfoques según la figura 2, omitiendo el enfoque experimental basado en ancestros que propone el autor en su trabajo, ya que este último es una propuesta del autor que aún no contiene una gran variedad de algoritmos para considerarlo una familia. Los enfoques clásicos son: algoritmos evolutivos, el cual utiliza un conjunto de soluciones candidatas simultáneamente, las cuales permu-

tan en nuevos conjuntos que permiten tener múltiples agentes explorando el espacio de búsqueda, donde destacan algoritmos como *Harmony Search* (HS) [Yang, 2009], *Memetic Algorithm* (MA) [Neri y Cotta, 2012] o *Genetic Algorithm* (GA) [Lambora et al., 2019]; algoritmos trayectoriales, donde trabajan con un único agente que explora iterativamente un vecindario de la solución candidata. Dentro de ellos están *Simulated Annealing* (SA) [Van Laarhoven et al., 1987], *Tabu Search* (TS) [Glover y Laguna, 1998], *Hill Climbing* (HC) [Selman y Gomes, 2006], entre otros; luego algoritmos basados en la naturaleza, donde se trata de simular procesos biológicos o físicos que podrían explorar y explotar los espacios de búsqueda de forma eficiente y efectiva, aquí destacan algoritmos como *Particle Swarm Optimization* (PSO) [Kennedy y Eberhart, 1995], *Black Hole* (BH) [Abualigah et al., 2022] o *Krill Herd Algorithm* (KHA) [Wang et al., 2019]. En esta memoria, se estudiará y experimentará con algoritmos trayectoriales.

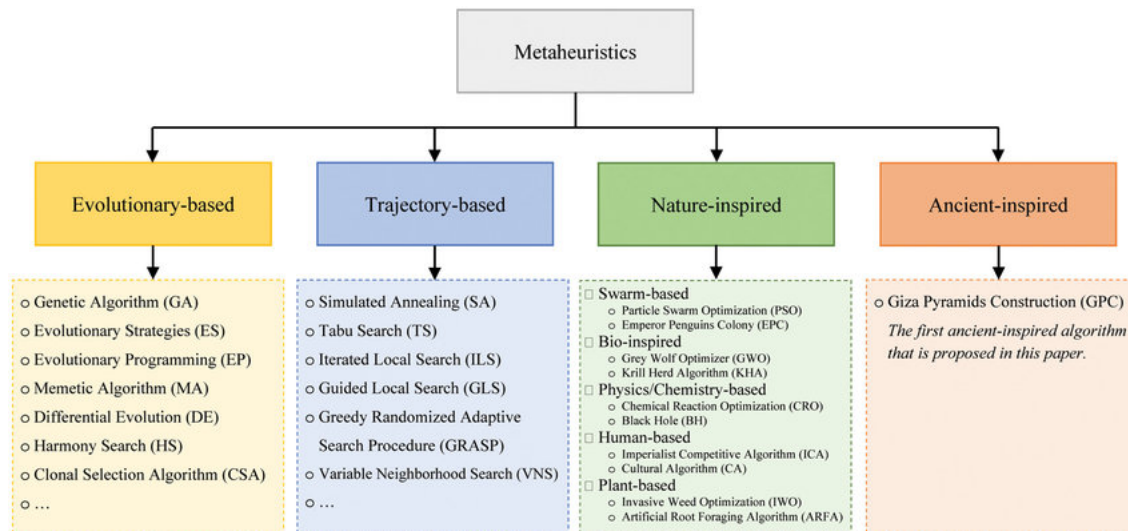


Figura 2: Clasificación de algoritmos metaheurísticos

Fuente: Classifying Metaheuristics: Towards a unified multi-level classification system [Stegherr et al., 2022]

Un diseño adecuado de estos algoritmos es fundamental para su éxito en la etapa de experimentación [Osaba et al., 2021]. En particular, con ciertos componentes que definen su funcionamiento, los cuales son:

- **Representación:** Se refiere a la manera en que se define, estructura y manipula las soluciones dentro del algoritmo. La elección de una representación adecuada es

crucial, ya que afecta la eficiencia y eficacia del algoritmo en la exploración y explotación del espacio de soluciones posibles. Además, afecta a otros componentes del algoritmo, como los operadores de búsqueda o la función de evaluación, incluso permite cubrir ciertas restricciones del problema, encapsulándolas dentro del dominio de las variables.

- **Función de evaluación:** Se encarga de cuantificar la calidad de una solución candidata para un problema específico. La función de evaluación asigna un valor numérico a cada solución candidata según la medida de desempeño o criterio de optimización definido para el problema en cuestión.
- **Operadores de búsqueda:** Estos operadores permiten perturbar las soluciones candidatas, para generar un conjunto nuevo de ellas, que permita explorar el espacio de búsqueda del problema con el objetivo de visitar nuevas regiones o intensificar un *vecindario* local.

Luego de diseñar y construir estos componentes, el algoritmo se ejecuta con el objetivo de resolver un conjunto de instancias, que representan múltiples casos del problema de interés, con distintas características y dimensiones. Esto permite analizar empíricamente el funcionamiento del algoritmo, respecto a la calidad de las soluciones que entrega. Es importante trabajar con un conjunto heterogéneo de instancias, para así evitar sesgos respecto a ciertas condiciones o características del problema, que podrían afectar el funcionamiento del algoritmo con ciertos casos. Los resultados de esta etapa, entregan las soluciones encontradas para cada una de las instancias, junto con la calidad de la solución cuantificada por la función de evaluación diseñada.

Sin embargo, solo conociendo la mejor solución obtenida luego de una ejecución, resulta complejo comprender el diseño del algoritmo, sus parámetros u operadores. Existe mayor información sobre la ejecución del algoritmo, que permite entender y analizar su funcionamiento e implementar mejoras a este. Por ejemplo, se pueden tener dos soluciones candidatas idénticas o similares para una instancia, donde la calidad es la misma, pero una se obtuvo en la etapa inicial del algoritmo y no logro mejorar nunca esa solución, mientras que la otra logró ir mejorando constantemente hasta llegar a la solución final. Determinar en este caso cuál es la mejor ejecución, y el porqué, resulta difícil de identificar solo conociendo el resultado final del algoritmo. En consecuencia, la calidad no resulta ser una buena medida de desempeño por sí sola, y no permite realizar un análisis completo del rendimiento real que generan los algoritmos en cada uno de sus operadores y etapas.

Los algoritmos a menudo tienen parámetros que pueden ser ajustados para adaptarse mejor las características particulares de un conjunto de datos o de un problema dado. Estos pueden controlarse o sintonizarse para poder mejorar las soluciones. Algunos autores estudian los parámetros de los algoritmos [Huang *et al.*, 2019], es decir, ajustan sus configuraciones para mejorar el rendimiento en la resolución del conjunto de instancias. La sintonización busca encontrar la combinación óptima de estos parámetros para lograr un rendimiento mejorado. Aun así, la sintonización se basa en un criterio de calidad, más que en un funcionamiento propio del algoritmo, ya que no utiliza datos de la ejecución, sino más bien del resultado y el conjunto de instancias.

La configuración de algoritmos no se limita únicamente a la sintonización inicial de parámetros, sino que también puede abordarse mediante el control dinámico de estos parámetros durante la ejecución del algoritmo. Un ejemplo claro de esta técnica es en SA, en el cual la temperatura se ajusta gradualmente a medida que avanza el proceso. Este control adaptativo de la temperatura permite al algoritmo equilibrar la exploración y la explotación del espacio de soluciones, ayudando a evitar óptimos locales y a mejorar la probabilidad de encontrar una solución global óptima. La capacidad de ajustar parámetros en tiempo de ejecución permite que el algoritmo responda a las características cambiantes del problema, aumentando su eficiencia y efectividad en la resolución del mismo.

Entonces, por todo lo anteriormente mencionado, resulta pertinente obtener más información respecto al desempeño de las metaheurísticas, en particular con el desarrollo de los operadores de búsqueda, para así construir diseños más eficientes y efectivos, en la resolución de problemas combinatoriales que pueden ser utilizados para resolver situaciones del mundo real.

1.1. Objetivos

1.1.1. Objetivo General

Desarrollar una librería de métricas de rendimiento para algoritmos metaheurísticos trajectoriales.

1.1.2. Objetivos Específicos

Se plantean un conjunto de objetivos específicos con el fin de cumplir el objetivo general descrito:

1. Estudiar el Estado del Arte sobre métricas de rendimiento de algoritmos metaheurísticos.
2. Diseñar métricas de rendimiento para algoritmos metaheurísticos trayectoriales.
3. Implementar la librería en lenguajes *Python* y *C++*.
4. Evaluar la librería desarrollada en algoritmos trayectoriales clásicos como *Tabu Search*, *Simulated Annealing*, *Hill Climbing*, entre otros, en instancias *benchmark* de algún problema combinatorial.

CAPÍTULO 2

MARCO CONCEPTUAL

Este capítulo se centra en la presentación y análisis de los conceptos claves para comprender el diseño de algoritmos. La comprensión de estos conceptos es fundamental para establecer una base común que permita profundizar en las métricas de rendimiento que se proponen en este trabajo.

Primero, se definen conceptos básicos respecto al manejo de algoritmos metaheurísticos, para luego profundizar en los algoritmos trayectoriales, que es el grupo experimental que se utilizara posteriormente dentro de esta memoria. Estos algoritmos son esenciales en la resolución de problemas complejos, y cada uno presenta características y aplicaciones únicas que serán examinadas minuciosamente. Luego, se abordarán las métricas de rendimiento, las cuales son el eje central de la propuesta de esta memoria, ya que estas métricas permiten una comprensión profunda de la eficiencia y efectividad de los algoritmos en diferentes contextos y escenarios, además de cuantificar estas características para la toma de decisiones del diseño en los distintos algoritmos. Finalmente, se presentará un framework llamado *IOHprofiler*, el cual es una herramienta integral para el análisis de algoritmos de optimización. Este apartado cubrirá los componentes principales de este *framework*, detallando sus funcionalidades y cómo facilitan la etapa de experimentación de los algoritmos trayectoriales.

2.1. Conceptos metaheurísticos

- **Solución candidata:** Una solución candidata es una posible solución a un problema de optimización donde todas sus variables tienen asignado un valor. Además, esta solución puede ser factible o no, es decir, puede cumplir todas las restricciones propias del problema a resolver. En algunas familias de algoritmos, se generan múltiples soluciones candidatas, que luego se evalúan y perturban/modifican para encontrar la mejor solución posible dentro del conjunto.
- **Movimiento:** Un movimiento es una transformación que se aplica a una solución candidata x para generar una nueva solución x' . Dependiendo del tipo de representación que tiene las variables del problema, ya sea binaria, entera o continua, se puede utilizar un conjunto de movimientos que perturban las soluciones candi-

datos para poder generar soluciones nuevas del espacio de búsqueda.

- **Vecindario:** El vecindario de una solución candidata es el conjunto de soluciones que se pueden obtener mediante modificaciones de la solución actual. En términos de algorítmicos, el vecindario implica evaluar soluciones cercanas. Este concepto es fundamental para establecer un área cercana de soluciones a visitar, el cual permita trabajar con distintos movimientos, esperando poder experimentar con distintos vecindarios.
- **Exploración:** En el contexto de algoritmos metaheurísticos, se refiere a la búsqueda en diferentes regiones del espacio de soluciones, con el objetivo de descubrir nuevas y prometedoras soluciones, que podrían implicar visitar soluciones candidatas de más baja calidad. Es una estrategia para evitar quedarse atrapado en óptimos locales.
- **Explotación:** Implica centrarse en las áreas del espacio de soluciones, con el objetivo de mejorar aún más esas soluciones. Es una estrategia para refinar y optimizar las soluciones encontradas.
- **Óptimo local:** Un óptimo local es una solución que es mejor que todas las soluciones en su vecindario, pero no necesariamente es la mejor solución posible (óptimo global). Dependiendo del problema, el espacio de búsqueda puede contener varios de estos óptimos locales, generando que la búsqueda sea más compleja por parte del algoritmo, ya que debe evitar estancarse un óptimo local.
- **Función de evaluación:** La calidad de una solución se refiere a cuán buena o adecuada es una solución en comparación con otras posibles soluciones dentro del espacio de búsqueda. Este concepto se cuantifica utilizando una función de evaluación, la cual asigna un valor numérico a cada solución, permitiendo así su comparación. Esta función de evaluación puede estar alineada con una función objetivo, que es el criterio principal que el algoritmo trata de optimizar (maximizar o minimizar). La función de evaluación traduce la función objetivo a un valor que puede ser utilizado para guiar el proceso de búsqueda en el algoritmo.

2.2. Algoritmos trayectoriales

Los algoritmos trayectoriales son una familia de algoritmos metaheurísticos que buscan mejorar la calidad de una solución inicial a lo largo de una *trayectoria* en el espacio de

búsqueda. Estos algoritmos trabajan mediante la modificación iterativa de una solución, explorando localmente el *vecindario* de la solución actual para identificar movimientos que conduzcan a mejoras en la función de evaluación. El concepto de *vecindario* es esencial en los algoritmos trayectoriales, ya que representa el conjunto de soluciones que están próximas a la solución actual en el espacio de búsqueda. La exploración se realiza ajustando la solución actual dentro de este *vecindario*. Estos algoritmos equilibran la exploración del espacio de búsqueda para descubrir nuevas soluciones con la explotación de regiones prometedoras para mejorar la calidad de la solución actual. Dentro de los algoritmos clásicos, se encuentran *Simulated Annealing* (SA) [Van Laarhoven *et al.*, 1987], *Tabu Search* (TS) [Glover y Laguna, 1998], *Hill Climbing* (HC) [Selman y Gomes, 2006], que se explicaran en profundidad en este capítulo.

En contraste, los algoritmos poblacionales, se basan en una población de soluciones y realizan un conjunto de operadores que modifican un subconjunto de ellas. A diferencia de los algoritmos trayectoriales que operan sobre una sola solución, los algoritmos poblacionales mantienen y operan sobre una población de soluciones, lo que les permite explorar y explotar el espacio de búsqueda. Además, los algoritmos poblacionales pueden evitar quedarse atrapados en óptimos locales al mantener diversidad en la población y permitir la introducción de nuevas soluciones mediante operadores de búsqueda. Algunos ejemplos clásicos dentro de esta familia son *Particle Swarm Optimization* (PSO) [Kennedy y Eberhart, 1995] o *Ant Colony Optimization* (ACO) [Dorigo *et al.*, 2006]. Es importante conocer estas diferencias, ya que múltiples autores trabajan sus métricas de rendimiento en ambos tipos de algoritmo, tanto poblacionales como trayectoriales, pero que para este trabajo, se llevara la experimentación solo hacia este último grupo.

2.2.1. *Hill Climbing*

Hill Climbing (HC) [Davis, 1991] es un algoritmo de búsqueda local, el cual su enfoque se asemeja a escalar una colina en busca de la cima, donde la altura representa la calidad de la solución (en términos de la función de evaluación). Este algoritmo comienza de una solución inicial y realiza movimientos locales iterativos para mejorar la calidad de la solución hasta alcanzar un óptimo local.

Este algoritmo tiene dos posibles implementaciones clásicas: HC Alguna Mejora (*First-Improvement*), el cual escoge la primera mejora encontrada en el proceso de generar el *vecindario* en cada iteración; y HC Mejor Mejora (*Best-Improvement*), el cual genera todo el *vecindario* de la solución candidata y escoge la mejor solución de ese conjunto.

Como se puede ver en el algoritmo 1, este comienza con una solución inicial, la que permite generar el primer vecindario de soluciones candidatas, para luego de forma iterativa comparar las soluciones candidatas con la solución actual. En este caso, el algoritmo deja de comparar si encuentra la primera mejora dentro de este vecindario, dejando fuera otras soluciones candidatas que podrían tener mejor calidad que la elegida, con el beneficio de una ejecución más rápida del algoritmo, debido a que con vecindarios de gran tamaño, el costo computacional de comparar cada solución candidata sería muy alto.

Este proceso se realiza de forma continua, hasta que para una solución actual no se genere un vecindario con mejoras respecto a la solución actual, terminando la ejecución con esta solución como resultado óptimo local.

Algoritmo 1 Hill Climbing First-Improvement

```
1: Entrada: Solución inicial  $S_c$ 
2: Salida: Solución óptimo local
3: function HCFirstImprovement( $S_c$ )
4:    $local \leftarrow \text{False}$ 
5:   while not  $local$  do
6:      $local \leftarrow \text{True}$ 
7:      $neighbors \leftarrow \text{generarVecindario}(S_c)$ 
8:     for  $neighbor$  in  $neighbors$  do
9:       if  $f(neighbor)$  es mejor que  $f(S_c)$  then
10:         $S_c \leftarrow neighbor$ 
11:         $local \leftarrow \text{False}$ 
12:        break
13:       end if
14:     end for
15:   end while
16:   return  $S_c$ 
17: end function
```

Para el caso del algoritmo 2, a diferencia del enfoque de primera mejora, este algoritmo evalúa todas las soluciones candidatas y selecciona la que tenga la mejor calidad, es decir, la que maximiza la función de evaluación. Esto asegura que siempre se elija la mejor solución posible dentro del vecindario en cada iteración, aunque a costa de un mayor tiempo de ejecución debido a la necesidad de evaluar todas las soluciones candidatas en cada iteración.

Este proceso se realiza de forma continua hasta que no se encuentra ninguna mejora en el vecindario con respecto a la solución actual, momento en el cual el algoritmo termina su ejecución y retorna la solución actual como el resultado óptimo local.

Algoritmo 2 Hill Climbing Best-Improvement

```
1: Entrada: Solución inicial  $S_c$ 
2: Salida: Solución óptimo local
3: function HCBestImprovement( $S_c$ )
4:    $local \leftarrow \text{False}$ 
5:   while not  $local$  do
6:      $local \leftarrow \text{True}$ 
7:      $neighbors \leftarrow \text{generarVecindario}(S_c)$ 
8:      $s_n \leftarrow \text{seleccionarMejorSolución}(neighbors)$ 
9:     if  $f(s_n)$  es mejor que  $f(S_c)$  then
10:       $S_c \leftarrow s_n$ 
11:       $local \leftarrow \text{False}$ 
12:     end if
13:   end while
14:   return  $S_c$ 
15: end function
```

En general, HC de forma estándar no logra explorar de buena forma el espacio de búsqueda, ya que luego de alcanzar un óptimo local, la ejecución se da por finalizada. Ante esto, se incorporan otras técnicas como el *restart*, que permite reiniciar la ejecución desde otro punto del espacio, permitiendo explorar el espacio.

2.2.2. Randomized Local Search

Randomized Local Search (RLS) [Giel y Wegener, 2003] es un algoritmo que busca mejorar una solución iterativamente a través de movimientos locales y aleatorios en un espacio de búsqueda. A diferencia de otros algoritmos, que solo aceptan movimientos que mejoren la solución actual, RLS puede aceptar movimientos que empeoren la solución actual, lo que le permite escapar de óptimos locales y explorar más ampliamente el espacio de soluciones.

La característica particular de RLS es su capacidad para explorar de manera más diversa

el espacio de soluciones, lo que puede conducir a encontrar soluciones mejores o más diversas en comparación con algoritmos más deterministas. Al aceptar movimientos que empeoran la solución actual con cierta probabilidad, puede evitar quedarse atascado en óptimos locales y encontrar soluciones más globales en el espacio de búsqueda.

Dentro del algoritmo 3, se puede observar la estructura general de un RLS. Este algoritmo mantiene dos soluciones: s_{best} , que representa la mejor solución encontrada durante la ejecución, y S_c , que es la solución actual. A diferencia de S_{best} , S_c no siempre almacena la mejor solución encontrada, ya que puede aceptar soluciones de peor calidad para explorar otras zonas del espacio de búsqueda.

Al inicio de la ejecución, se genera un vecino de manera aleatoria, utilizando uno o varios movimientos. Este vecino se compara inicialmente con S_c para verificar si hay una mejora en la calidad. Si existe una mejora, este vecino se convierte en el nuevo S_c . En caso contrario, existe una probabilidad p que permite cambiar S_c por este vecino, aunque tenga una calidad menor. Además, si el nuevo S_c es mejor que s_{best} , s_{best} se actualiza con S_c .

El algoritmo define un criterio de término, que puede ser un número máximo de iteraciones o alcanzar una calidad específica x , indicando así el fin de la ejecución junto con él s_{best} .

Algoritmo 3 Randomized Local Search

```
1: Entrada: Solución inicial  $S_c$ 
2: Salida: Mejor solución encontrada
3: function RandomizedLocalSearch( $S_c$ )
4:    $s_{best} \leftarrow S_c$ 
5:   while no se cumple el criterio de término do
6:      $neighbor \leftarrow \text{generarVecinoAleatorio}(S_c)$ 
7:     if costo( $neighbor$ ) < costo( $S_c$ ) then
8:        $S_c \leftarrow neighbor$ 
9:     else if rand() < probabilidad then
10:       $S_c \leftarrow neighbor$ 
11:     end if
12:     if costo( $S_c$ ) < costo( $s_{best}$ ) then
13:        $s_{best} \leftarrow S_c$ 
14:     end if
15:   end while
16:   return  $s_{best}$ 
17: end function
```

Existen algunos casos, donde no solo es aleatorio la opción de aceptar o no una solución, sino incluso los movimientos para generar los vecindarios resultan ser aleatorios.

2.2.3. *Simulated Annealing*

Simulated Annealing (SA) [Van Laarhoven *et al.*, 1987] está inspirado en el proceso físico del mismo nombre que se utiliza en metalurgia para reducir la dureza de un material mediante un calentamiento controlado y un enfriamiento lento. Este concepto se aplica metafóricamente dentro de SA, ya que la probabilidad de aceptar soluciones peores es alta al comenzar el algoritmo, permitiendo una exploración más amplia del espacio de búsqueda, pero a medida que avanza la búsqueda, la probabilidad disminuye, favoreciendo la aceptación de soluciones que mejoren la función objetivo.

Dentro del algoritmo 4, al igual que en RLS, se mantienen dos soluciones: s_{best} , que representa la mejor solución encontrada durante la ejecución, y S_c , que es la solución actual.

En este algoritmo, se inicializa una temperatura T , que controla la probabilidad de aceptar soluciones peores. Luego, se genera un vecino de manera aleatoria y se calcula la diferencia en calidad (Δ_E) entre el vecino y la solución actual S_c . Si el vecino es mejor que S_c , se acepta como la nueva solución actual. Si no, se acepta con una probabilidad que depende de Δ_E y la temperatura T , permitiendo así escapar de óptimos locales.

Además, si la nueva S_c es mejor que s_{best} , se actualiza s_{best} con S_c . La temperatura T se ajusta en cada iteración para reducir gradualmente la probabilidad de aceptar soluciones peores, es decir, cuando transcurran más iteraciones, menos probable es permitir soluciones de peor calidad.

En este caso, el criterio de término puede ser un número máximo de iteraciones o una temperatura mínima, indicando así el fin de la ejecución y retornando la mejor solución encontrada según s_{best} .

Algoritmo 4 Simulated Annealing

```
1: Entrada: Solución inicial  $S_c$ 
2: Salida: Mejor solución encontrada
3: function SimulatedAnnealing( $S_c$ )
4:   Inicializar temperatura  $T$ 
5:    $s_{\text{best}} \leftarrow S_c$ 
6:   while no se cumpla el criterio de término do
7:      $neighbor \leftarrow \text{generarVecinoAleatorio}(S_c)$ 
8:      $\Delta_E \leftarrow f(neighbor) - f(S_c)$ 
9:     if  $f(neighbor)$  es mejor que  $f(S_c)$  then
10:       $S_c \leftarrow neighbor$ 
11:     else if  $e^{\frac{-\Delta_E}{T}} > \text{random}([0,1])$  then
12:        $S_c \leftarrow neighbor$ 
13:     end if
14:     if  $f(S_c)$  es mejor que  $f(s_{\text{best}})$  then
15:        $s_{\text{best}} \leftarrow S_c$ 
16:     end if
17:      $T \leftarrow \text{cambiarTemperatura}(T)$ 
18:   end while
19:   return  $S_c$ 
20: end function
```

2.3. Métricas de rendimiento

La etapa experimentación es fundamental en el proceso de diseño algorítmico. En particular, busca resolver dos características que son importantes para una buena contribución del algoritmo diseñado: eficiencia, permitiendo obtener soluciones de buena calidad de forma rápida; y efectividad, identificando soluciones de mejor calidad por cada iteración [Barr *et al.*, 1995].

2.3.1. Ratio de convergencia

El ratio de convergencia es una medida que determina que tan rápido la solución converge hacia el óptimo por cada iteración. La convergencia usualmente se puede ver de forma visual, al revisar la variación entre la calidad de la solución actual por cada unidad de

tiempo, iteración o FEV (por sus siglas en inglés, *function evaluations*). Algunos autores [Dhivyaprabha *et al.*, 2018], lo trabajan como la razón de cambio entre la calidad actual de la solución y la calidad de la solución en la iteración anterior.

$$CR = \frac{|f_{i+1} - f_i|}{|f_i - f_{i-1}|} \quad (1)$$

Las irregularidades o patrones inesperados en la curva de convergencia pueden ayudar a identificar posibles problemas en el algoritmo. Por ejemplo, si hay oscilaciones o estancamientos, es decir, zonas donde el algoritmo deja de lograr mejoras, se podría identificar dichas áreas para mejorar, permitiendo ajustar parámetros o modificar operadores. Por otro lado, facilita la comparación de diferentes algoritmos, permitiendo visualizar y comparar cómo se desempeñan en términos de velocidad de convergencia y calidad de solución en un rango de tiempo.

Halim e Ismail utilizan esta métrica para comparar el rendimiento de múltiples algoritmos en el problema TSP [Halim e Ismail, 2019]. En el artículo, se trabajó con los siguientes algoritmos: *Nearest Neighbors* (NN), *Simulated Annealing* (SA), *Tabu Search* (TS), *Genetic Algorithm* (GA), *Ant Colony Optimization* (ACO) y *Tree Physiology Optimization* (TPO). Por otro lado, el conjunto de instancias se divide en 3 grupos: pequeño, donde se trabajan con instancias con menos de 100 ciudades; moderado, donde se encuentran instancias entre 100 y 190 ciudades; y grande, donde están instancias con más de 190 ciudades. Además, se configuraron todos los algoritmos con un criterio de término de 10000 iteraciones.

En la figura 7, se tienen dos secciones, el lado izquierdo es la convergencia de cada algoritmo con un máximo de 10000 iteraciones, y el lado derecho es la convergencia de las primeras 1000 iteraciones, ya que es difícil analizar y tomar una decisión basada en la ejecución completa, ya que todos los algoritmos convergen hacia un óptimo de forma temprana. Se puede observar que entre los algoritmos más rápidos para converger en las primeras 300 iteraciones se encuentran NN, TS y ACO. Como se observa en la figura, SA es el algoritmo más lento en converger en todos los casos. El autor comenta que la mejora para SA también está limitada a la probabilidad respecto a la temperatura, que genera que el algoritmo pueda aceptar soluciones de peor calidad.

Para esta métrica, el autor no define un valor óptimo, ya que se utiliza para comparar distintos algoritmos y se define un umbral, que depende del rendimiento del resto para determinarlo. Aun así, la convergencia permite determinar áreas de estancamiento.

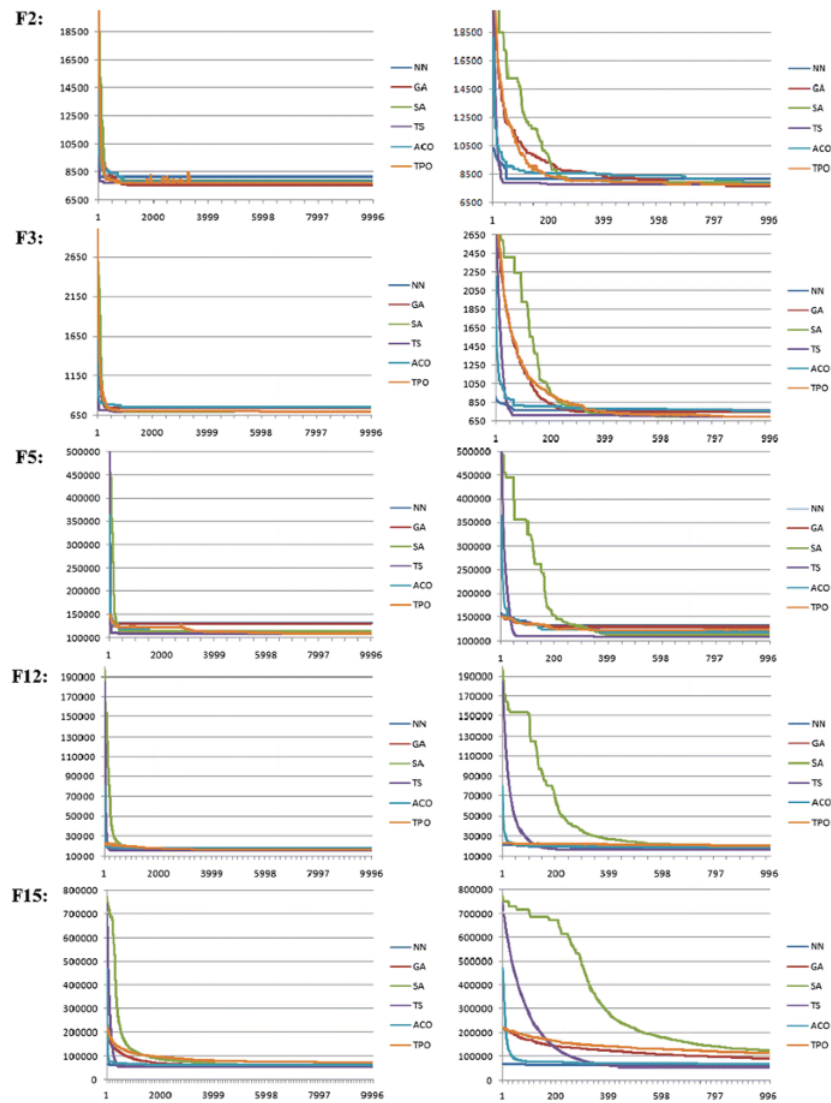


Figura 3: Curvas de convergencia para instancias de TSP

Fuente: *Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem* [Halim e Ismail, 2019]

2.3.2. Ratio de convergencia geométrico

Una variación del ratio de convergencia, es aplicar una media geométrica normalizada respecto a la razón de cambio de cada iteración, entre la calidad del óptimo teórico o experimental de la instancia y la calidad de la solución candidata. Dhivyaprabha ocupa esta métrica dentro de su trabajo respecto a la propuesta de su algoritmo *Synergistic Fibroblast Optimization* (SFO)[Dhivyaprabha et al., 2018], un algoritmo evolutivo inspirado de los organismos celulares fibroblastos en el proceso de cicatrización de heridas dérmicas. Cada célula está buscando continuamente encontrar un óptimo local en el espacio del problema de n dimensiones. En cada iteración, a las células se les asigna un colágeno aleatorio, la calidad de la célula se evalúa utilizando la función de evaluación y se compara con la solución óptima $cbest_{i-1}$ (mejor colágeno anterior). La célula con el mejor valor se considera como la mejor solución $cbest_i$ (mejor colágeno actual). Las células tienen una posición y velocidad dinámica, que trata de ir equilibrando entre el proceso de intensificar y diversificar.

En este trabajo, se define la métrica $R(t)$ como la tasa de convergencia promedio (geométrica) de un algoritmo evolutivo para t generaciones:

$$R(t) = 1 - \left(\left| \frac{f_{opt} - f_1}{f_{opt} - f_0} \right| \left| \frac{f_{opt} - f_2}{f_{opt} - f_1} \right| \dots \left| \frac{f_{opt} - f_t}{f_{opt} - f_{t-1}} \right| \right)^{\frac{1}{t}} = \left(\left| \frac{f_{opt} - f_t}{f_{opt} - f_0} \right| \right)^{\frac{1}{t}} \quad (2)$$

Como se puede ver en la figura 4, se realiza un análisis de este algoritmo con otros dos algoritmos conocidos: *Particle Swarm Optimization* (PSO) y *Differential Evolution* (DE). Para este análisis, se trabajó con un conjunto de 5 funciones *benchmark*, los cuales arrojaron que SFO logro un ratio de convergencia superior al de sus pares, aunque es un porcentaje pequeño, es suficiente para evidenciar cambios en la eficiencia general del algoritmo en toda su ejecución.

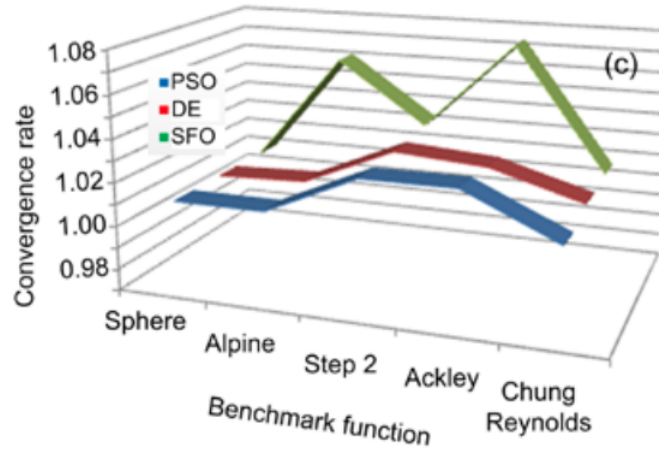


Figura 4: Tasas de convergencia promedio geométrico de SFO, PSO y DE
Fuente: *Synergistic fibroblast optimization: a novel nature-inspired computing algorithm*
[Dhivyaprabha et al., 2018]

2.3.3. Ratio de error

A diferencia del ratio de convergencia, el de error busca medir qué tan alejada está la calidad de la solución de cada iteración respecto a la solución óptima, ya sea esta última teórica o experimental. Idealmente, se espera que a medida que avanzan las iteraciones, este índice tienda a 0. Sin embargo, en algunos algoritmos el error puede oscilar debido a la capacidad de explorar el espacio de búsqueda que tenga el algoritmo.

He Jun utiliza esta métrica dentro de su trabajo para determinar qué tan buenas soluciones pueden producir los algoritmos evolutivos [He, 2016]. En este trabajo el objetivo es proporcionar un análisis analítico de la calidad de la solución en términos del error de aproximación relativa, que es donde nace el ratio de error. Jun define esta métrica como:

$$E_i = 1 - \frac{f_i}{f_{opt}} \quad (3)$$

Donde f_i es la calidad de la solución o generación de la iteración i del algoritmo, y f_{opt} el óptimo teórico o experimental del problema. En ese sentido, el dominio de esta métrica va entre $[0, 1]$, a medida que el valor se acerca a cero, la calidad se aproxima a la óptima.

En este trabajo [He, 2016], se utilizó un algoritmo evolutivo denominado *Strictly Elitist EA*, que tiene un operador de mutación *bit-flip*, que crea una población de individuos, para luego escoger a los mejores individuos, con el objetivo de generar una nueva población realizando este proceso durante cada iteración. Para este algoritmo, se experimentó con 3 funciones *benchmark* clásicas: *OneMax*, *Square* y *Logarithmic*. Para estas funciones, se puede ver cómo el ratio de error converge rápidamente a cero utilizando este algoritmo, en donde se puede ver en la figura 5, donde después de las 10 de iteraciones los valores del E_i son cercanos a cero.

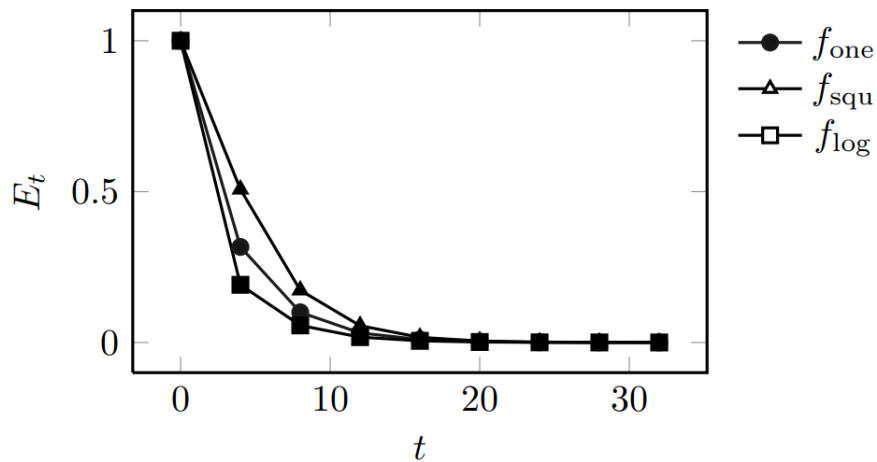


Figura 5: Curvas de error relativo según cada función objetivo

Fuente: *An analytic expression of relative approximation error for a class of evolutionary algorithms* [He, 2016]

2.3.4. Diversidad

La diversidad en un algoritmo se refiere a cómo están distribuidos los individuos en el espacio de búsqueda y refleja cuánta información se está explorando y cuánta se está explotando a lo largo de las iteraciones. Una diversidad alta indica una capacidad de exploración, mientras que una baja sugiere una tendencia a explotar el espacio de búsqueda. Por lo tanto, la medida de diversidad tiene un impacto significativo en cómo converge el algoritmo, y una buena diversidad puede prevenir caer en óptimos locales de baja calidad. Un algoritmo eficaz debe encontrar un equilibrio óptimo entre estos dos objetivos de búsqueda.

Cheng utiliza esta métrica con el objetivo de medir la capacidad de exploración y explotación del algoritmo *Brain Storm Optimization* (BSO) [Cheng *et al.*, 2014], un algoritmo donde las soluciones se separan en varios grupos, y las mejores soluciones de cada grupo se conservan para la próxima iteración. Además, se pueden generar nuevos individuos basados en uno o dos individuos en los grupos. La capacidad de explotación se mejora cuando el nuevo individuo está cerca de la mejor solución hasta ahora, mientras que la capacidad de exploración se mejora cuando el nuevo individuo se genera aleatoriamente, o se genera a partir de individuos en dos grupos.

Con este algoritmo, Cheng utiliza la siguiente definición para calcular la diversidad en las distintas ejecuciones:

$$\begin{aligned}\bar{x}_j &= \frac{1}{m} \sum_{i=1}^m x_{ij} \\ Div_j &= \frac{1}{m} \sum_{i=1}^m |x_{ij} - \bar{x}_j| \\ Div &= \sum_{j=1}^b w_j \cdot Div_j\end{aligned}$$

Donde j es la dimensión o posición del vector solución, i es el vector de solución de la iteración o generación i y m es la cantidad de iteraciones totales del algoritmo. Por otro lado, $\bar{x}_j$ es el promedio de todos los valores en la dimensión j para cada las soluciones de cada iteración. Con esto se genera la diversidad para cada dimensión, donde w_j es un peso asociado a la dimensión j , que por lo general se toma equitativo para todas las dimensiones de la solución, es decir:

$$w_j = \frac{1}{n}, \forall j \in [1, n]$$

Siendo n es la dimensión del problema. Por lo tanto, la definición final para esta métrica sería:

$$Div = \frac{1}{n} \sum_{j=1}^b Div_j \quad (4)$$

La métrica tiene un dominio entre $[a, b]$, donde a el menor valor que pueda tomar la variable de decisión en cada una de sus dimensiones, mientras que b es el mayor valor, es

decir, comparte del dominio de la representación del problema. Por ejemplo, si se trabaja con un problema TSP de 5 ciudades, una representación clásica es que cada ciudad se le asigne un número del 1 al 5, generando que el dominio de la variable de decisión x_i sea 1, 2, 3, 4, 5. Para este caso, el menor valor que puede tomar la variable de decisión es $a = 1$, mientras que el mayor valor es $b = 5$, por lo tanto, el dominio de la métrica Div sería $[1, 5]$.

En el trabajo de Cheng, se experimentó con 11 funciones de optimización, 5 unimodales y 6 multimodales, todas con objetivo de minimizar la función definida. Todas las funciones se ejecutan 50 veces para garantizar un resultado estadístico razonable, es decir, se obtienen 50 resultados para cada función. La dimensión utilizada es de $n = 50$ en todas las funciones.

Para la etapa de experimentación, se trabajó con 4 variantes del algoritmo BSO: La primera es la versión original donde después de escoger los *clusters*, no se reinician las soluciones, es decir, no se escogen nuevas soluciones al azar; la versión *half* introduce que cada cierta cantidad de iteraciones, se reinician la mitad de las soluciones, con el objetivo de explorar mejor el espacio de búsqueda; por otro lado, la variante *decrease* también reinician la mitad de las soluciones, pero esta cantidad va disminuyendo linealmente realizando este proceso con menos soluciones; y por último, la variante *increase* aumenta linealmente la cantidad de soluciones que reinicializa, es decir, realiza lo contrario que la variante *decrease*.

En la figura 6, se puede ver en la curva segmentada de color azul, como la métrica de diversidad a medida que se va iterando durante la ejecución del algoritmo, se va estancando en una zona en particular del espacio de búsqueda. Mientras que con las variantes que buscan explorar más zonas del espacio, logran una variabilidad mejor de esta métrica. Con esto, el autor concluye que añadir la componente de reinicialización podría ayudar a generar mayor diversidad en el espacio de búsqueda de las soluciones candidatas.

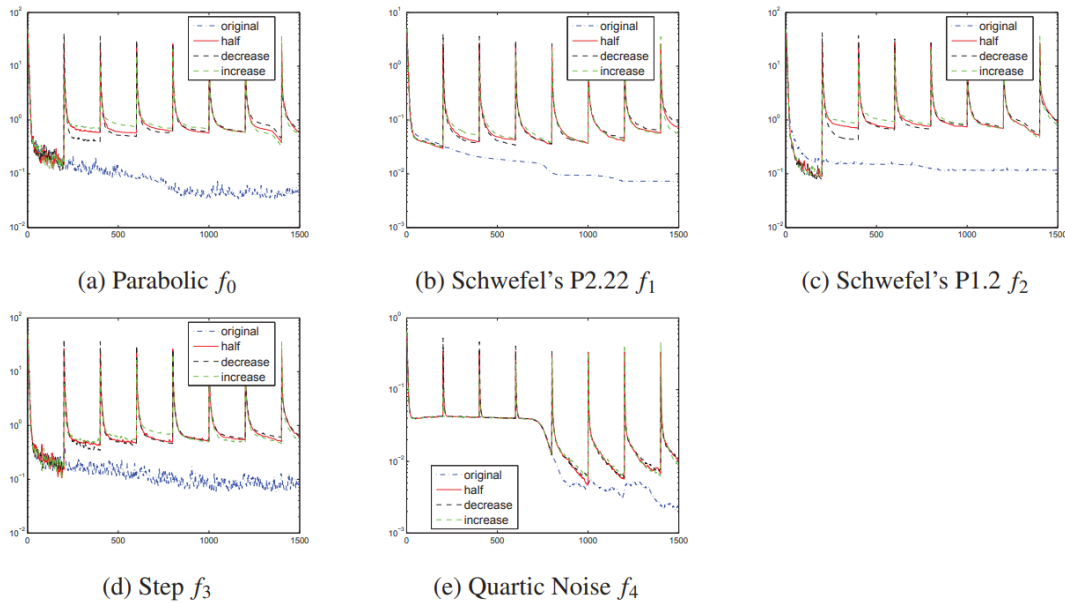


Figura 6: Monitor de diversidad poblacional del algoritmo BSO en funciones unimodales.

Fuente: *Population Diversity Maintenance In Brain Storm Optimization Algorithm*
[Cheng et al., 2014]

2.4. IOHprofiler

IOHprofiler es un *framework* de evaluación del rendimiento de heurísticas de optimización iterativa (IOH), desarrollado para el lenguaje C++¹. El objetivo es integrar varios elementos de todo el proceso de evaluación algorítmica, que incluye generadores de problemas (instancias), marcos de trabajo modulares para algoritmos, técnicas automatizadas de configuración de algoritmos y métodos de extracción de características, hasta la experimentación, el análisis de datos y la visualización. La plataforma se compone de los siguientes componentes principales:

- IOHalgorithm para la implementación eficiente de varios algoritmos de optimización clásicos.

¹<https://iohprofiler.github.io>

- **IOHproblem** para proporcionar una colección de funciones de prueba.
- **IOHexperimenter** para generar conjuntos de evaluación, que producen datos experimentales.
- **IOHalyzer** para el análisis estadístico y la visualización de los datos experimentales.
- **IOHdata** para alojar los conjuntos de datos de evaluación de *IOHexperimenter* y otras plataformas, como BBOB/COCO y *Nevergrad*.

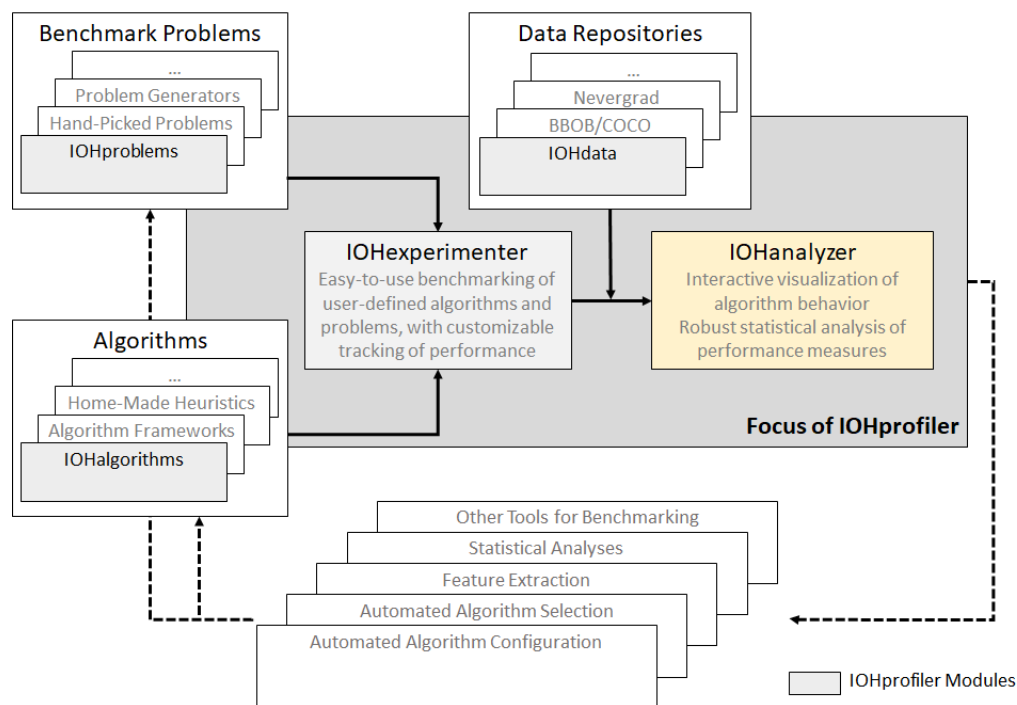


Figura 7: Módulos y flujo de IOHprofiler

Fuente: *IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics* [Doerr et al., 2018]

Para comprender la propuesta de solución, y las mejoras que ésta incluye respecto al estado actual de este *framework*, a continuación se explicarán los módulos de IOHalgorithms, IOHexperimenter y IOHalyzer.

2.4.1. IOHalgorithm

En este módulo, se implementan distintos algoritmos metaheurísticos, enfocados en resolver problemas de optimización booleana, es decir, problemas donde la representación de la solución es un vector $x \in \{0, 1\}^n$, esto debido a la implementación de operadores como *bit-flip*. Además, se encuentran tanto algoritmos trayectoriales como poblacionales, donde se presentan algoritmos como *Greedy Hill Climber*, *Randomized Local Search*, *Fast Genetic Algorithm*, *Vanilla Genetic Algorithm*, entre otros.

Para la etapa de validación de esta memoria, se utilizan los algoritmos trayectoriales que este módulo provee. El primero es *Greedy Hill Climber*, donde en esta implementación explicada en el algoritmo 5 se va cambiando de forma iterativa cada una de las variables de la solución actual, hasta encontrar una solución candidata que mejore la solución actual. Se trata de asimilar a un HC Alguna Mejora, donde se acepta la primera mejora que se encuentra en el *vecindario*, solo que aquí no existe el concepto de *vecindario*.

Algoritmo 5 Greedy hill climber (gHC)

```
1: Entrada: Solución inicial  $x \in \{0, 1\}^n$  generada al azar
2: for  $t = 1, 2, 3, \dots$  do
3:    $x^* \leftarrow x$ 
4:   Realiza bit-flip vector  $x^*$  en la posición  $1 + (t \bmod n)$ 
5:   if  $f(x^*) > f(x)$  then
6:      $x \leftarrow x^*$ 
7:   end if
8: end for
9: return  $x$ 
```

El segundo es *Randomized Local Search*, el cual se presenta en el algoritmo 6, donde se realiza un bit-flip en cada posición con una probabilidad p del 50 %, siempre aceptando esta solución como nueva solución candidata, sin importar si es mejor o no que la solución actual. Este permite que se exploren varias soluciones de forma iterativa, pero resulta difícil que pueda converger rápido a un óptimo local.

Algoritmo 6 Randomized local search (RLS)

```
1: Entrada: Solución inicial  $x \in \{0, 1\}^n$  generada al azar
2: for  $t = 1, 2, 3, \dots$  do
3:    $x^* \leftarrow x$ 
4:   Realiza bit-flip vector  $x^*$  en cada posición con una probabilidad  $p$  del 50 %
5:   if  $f(x^*) > f(x)$  then
6:      $x \leftarrow x^*$ 
7:   end if
8: end for
9: return  $x$ 
```

Y por último, la implementación de *Simulated Annealing* presente en el algoritmo 7, la cual es la única que genera una solución candidata donde existe más de un bit modificado a la vez por iteración. En este caso, se itera por cada posición de la solución actual, y existe una probabilidad p de realizar un bit-flip en esa posición, siendo $p = 1/n$, donde n es la dimensión de la instancia del problema a resolver, permitiendo un movimiento que perturba varias partes de la solución actual. Por otro lado, al ser SA, permite aceptar soluciones candidatas que no son mejores que la solución actual, lo que genera que se pueda explorar varias zonas del espacio de búsqueda.

Algoritmo 7 Simulated annealing (SA)

```
1: Entrada: Solución inicial  $x \in \{0, 1\}^n$  generada al azar
2:  $p \leftarrow 1/n$ 
3:  $T \leftarrow$  Temperatura Inicial
4: for  $t = 1, 2, 3, \dots$  do
5:    $x^* \leftarrow x$ 
6:   while  $x^* \neq x$  do
7:     for  $i = 1, 2, 3, \dots, n$  do
8:       Realizar bit-flip en  $x_i$  con una probabilidad  $p$ 
9:     end for
10:  end while
11:  if  $f(x^*) > f(x)$  or  $e^{\frac{-\Delta E}{T}} > \text{random}([0,1])$  then
12:     $x \leftarrow x^*$ 
13:  end if
14: end for
15: return  $x$ 
```

2.4.2. IOHexperimenter

Este módulo es el *core* del framework, ya que se encarga de unir los módulos de IOHalgorithm y IOHproblem para ejecutar múltiples instancias de un problema o una *suite* de problemas, permitiendo obtener resultados experimentales de forma completa. Permite modificar distintos parámetros de ejecución como la cantidad de *runs* o ejecuciones, la cantidad máxima de iteraciones por ejecución y por último, la semilla o *seed* que utiliza el algoritmo para generar los números aleatorios.

Al momento de ejecutar las instancias, se generan dos archivos de salida que permiten ser utilizados posteriormente en el módulo IOHanalyzer: el primero es un *logger* que captura información relevante de la ejecución del algoritmo, como lo son las distintas iteraciones, la solución actual con su calidad, y otros parámetros propios del algoritmo; y el segundo es un archivo de salida que contiene la mejor solución encontrada en las distintas ejecuciones junto con la iteración en que fue encontrada.

2.4.3. IOHanalyzer

Finalmente, IOHanalyzer es un módulo de análisis y visualización, que permite utilizar los datos obtenidos en IOHexperimenter para generar análisis estadístico. Todo esto se realiza en un servidor web que puede ser ejecutado localmente, para ingresar los datos experimentales generados anteriormente, permitiendo obtener distintos tipos de métricas de rendimiento.

Dentro de las métricas que se implementan se encuentra: el resumen de media, mejor y peor ejecución en una tabla comparativa; tiempo esperado para alcanzar un valor de la función de evaluación $f(x)$; probabilidad de masa respecto a un *target* t relacionado con la calidad de la solución; función de distribución acumulada empírica (ECDF) que describe la curva de probabilidad respecto a la cantidad de iteraciones necesarias para alcanzar un *target*; entre otros test estadísticos como Kolmogorov-Smirnov para verificar la similitud en la distribución de dos algoritmos muestrales.

Además, este módulo genera gráficos comparativos que permite entender de mejor forma el rendimiento de cada algoritmo respecto a distintas características.

En la figura 8, se puede apreciar un gráfico que compara la mejor solución encontrada versus la cantidad de iteraciones o funciones de evaluación que se realizaron, permitiendo

comparar entre múltiples algoritmos cuáles son los que tienen mejor rendimiento.

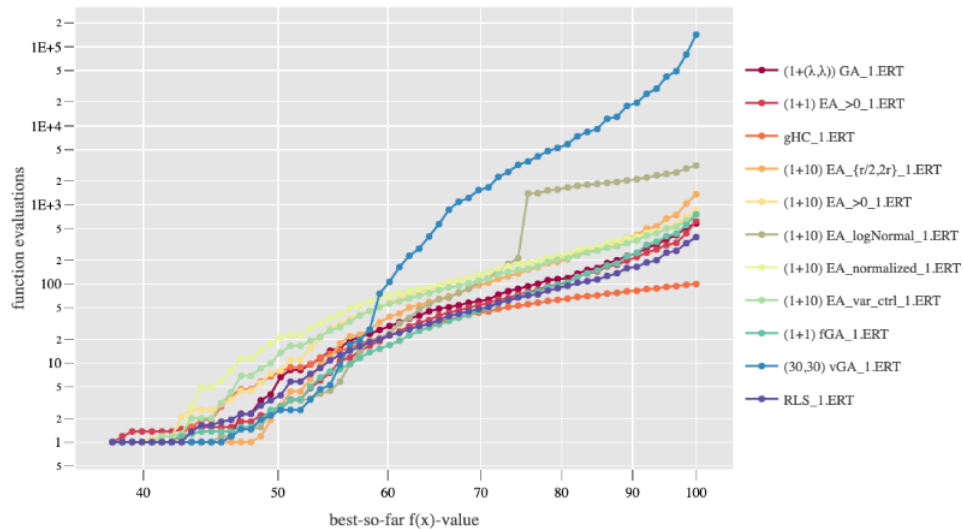


Figura 8: Curva de tiempo esperado para distintos algoritmos en IOHalyzer
Fuente: *IOHalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics*
[Wang et al., 2022]

Por otro lado, en la figura 9 se presenta un histograma de función de masa de probabilidad, se muestra la cantidad de *runs*, que son ejecuciones distintas del algoritmo, que se encuentran entre $[a, b]$ funciones de evaluación para alcanzar un valor α de calidad. En este caso, se presenta un histograma por cada algoritmo, y los rangos se calculan automáticamente según la distribución de cada uno. Esta misma idea se puede llevar a un *boxplot* como se presenta en la figura 10, donde se puede ver la distribución de cada algoritmo para alcanzar el valor α en una cantidad n de *runs*.

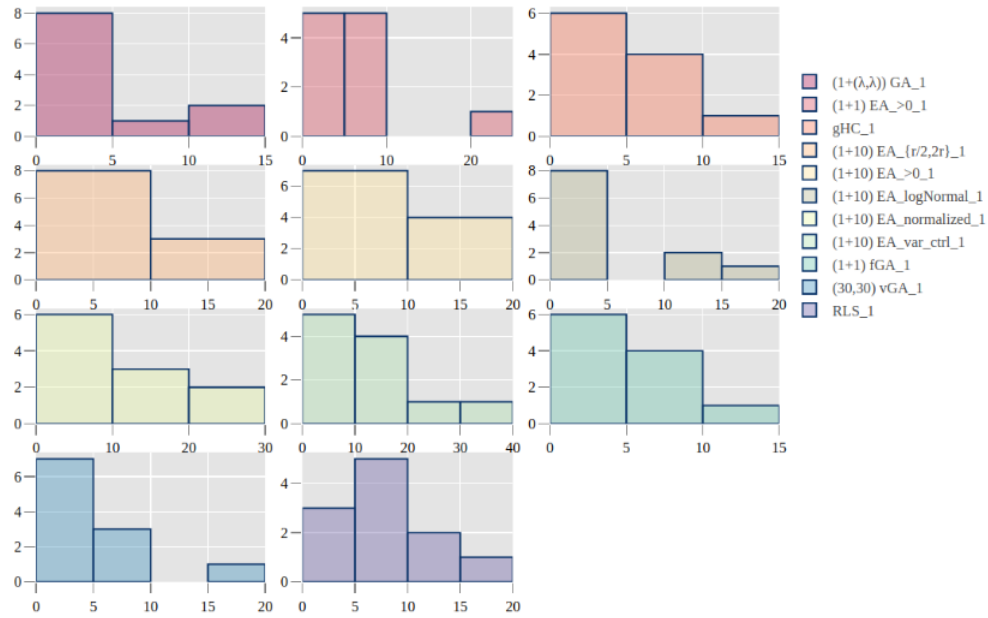


Figura 9: Histograma de función de masa de probabilidad en IOHalyzer
Fuente: *IOHalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics*
[Wang et al., 2022]

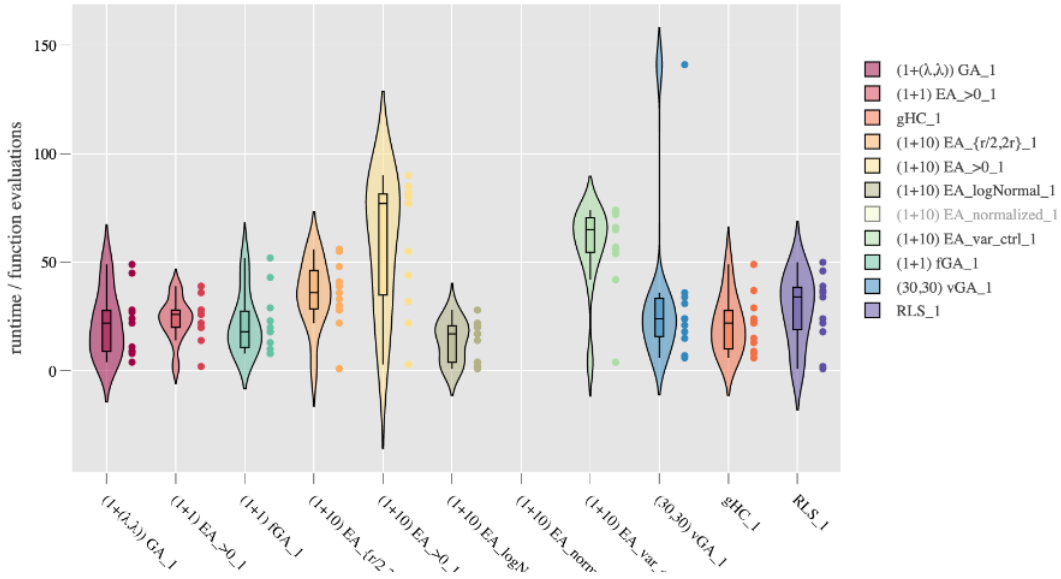


Figura 10: Boxplot de función de masa de probabilidad en IOHalyzer

Fuente: *IOHalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics* [Wang et al., 2022]

Por último, se presenta una curva empírica de función de distribución acumulada en la figura 11, donde colocando un límite $f_{\min}$ y $f_{\max}$, junto con un δ que indica la distancia entre cada f generado dentro del rango $[f_{\min}, f_{\max}]$, se puede generar una curva para cada algoritmo que representa la proporción de valores del espacio que se pueden alcanzar con x funciones de evaluación. Con esto, permite revisar la dificultad propia de cada algoritmo para mejorar la calidad de las soluciones, a medida que la calidad es mayor.

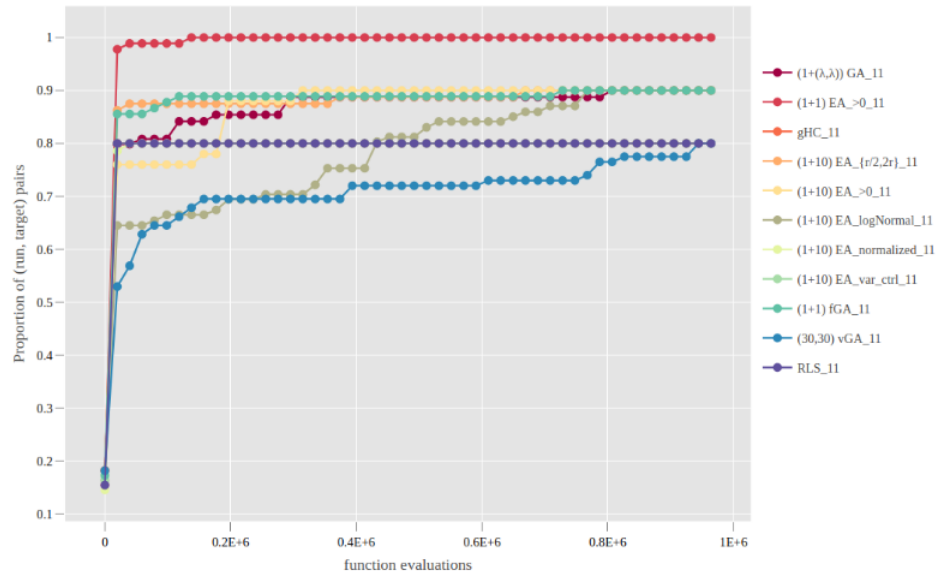


Figura 11: Curva empírica de función de distribución acumulada en IOHalyzer
Fuente: *IOHalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics*
[Wang et al., 2022]

En su mayoría, las métricas implementadas en este módulo se basan en análisis estadístico post-ejecución de los distintos algoritmos, por lo cual, no entregan la información necesaria para entender aspectos de diseño respecto a la ejecución como tal de los algoritmos. Esto se debe a que estas métricas solo se basan en el resultado final de múltiples ejecuciones experimentales, que generan una muestra de datos para poder realizar cálculos de distribución, medidas de tendencia y test de hipótesis. Es por esto, que en el siguiente capítulo, se presenta una propuesta de solución que implementa métricas de rendimiento, que logran captar la información de la ejecución de los algoritmos y determinar patrones de comportamiento en los operadores o las toma de decisiones del algoritmo.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

La propuesta de esta memoria es una librería de métricas de rendimiento para algoritmos metaheurísticos trayectoriales. El objetivo es permitir al usuario (que diseña, crea, prueba y analiza estos algoritmos), reportar y analizar lo que ocurre en la ejecución de estos algoritmos y entregar mayor información respecto a características como la eficiencia y la efectividad. Es por esto que nace MHTMetLib, la librería que incluye métricas de rendimiento que buscan cuantificar ciertas características propias del algoritmo y permita ser comparado con sus pares de forma sencilla.

Para esta memoria, se decide trabajar con el *framework* de IOHprofiler, ya que se observa una oportunidad de mejora respecto al manejo de métricas de rendimiento que se pueden integrar dentro del módulo de análisis. Además, este *framework* presenta una buena base para trabajar con algoritmos ya definidos, conocidos en la literatura y que cuentan con una buena documentación para comenzar a realizar experimentos de forma rápida y sencilla. Por otro lado, la integración que se plantea realizar con la librería, se simplifica debido al manejo de *logs* de ejecución que viene implementado en IOHexperimenter, lo que permite obtener los datos necesarios para calcular las métricas de rendimiento que se implementan en MHTMetLib.

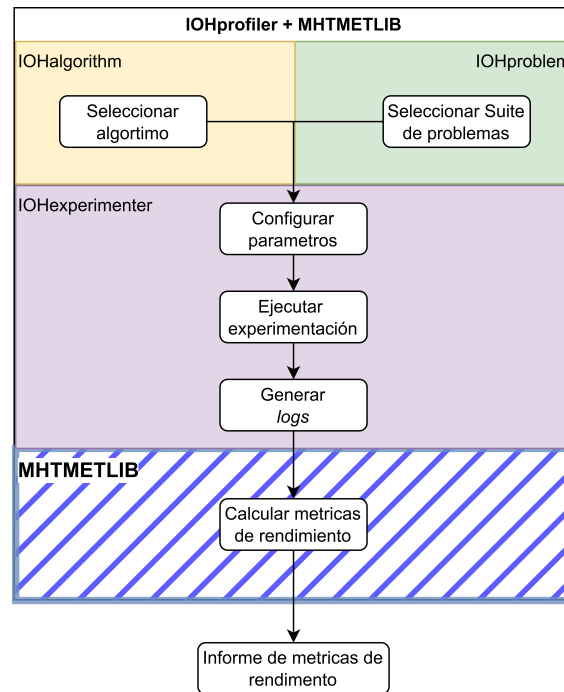


Figura 12: Integración MHTMetLib con flujo de IOHprofiler
Fuente: Elaboración propia

La figura 12 presenta el flujo algorítmico que tiene IOHprofiler comprende el uso de IOHalgorithm para seleccionar o implementar algoritmos de búsqueda incompleta, ya sea de tipo trayectorial o poblacional. Luego, éste integra una suite de problemas a resolver mediante IOHproblem, permitiendo elegir un conjunto de problemas *benchmark* de distintas suites de prueba. Luego, se configuran los parámetros de ejecución y se entregan los resultados obtenidos al resolver las distintas instancias configuradas en la suite de problemas. En esta etapa, IOHexperimenter permite generar un archivo de *logs*, donde se puede capturar variables propias del algoritmo, que permiten obtener información respecto a la solución, la calidad y los operadores que se utilizan en la resolución de las distintas instancias. Con este archivo, se pueden calcular las distintas métricas en MHTMetLib, para luego entregar los resultados de forma gráfica o numérica mediante un informe de salida para las instancias que se indiquen, permitiendo realizar un análisis completo de la ejecución de múltiples algoritmos en un mismo problema.

3.1. Lectura de *logs*

Al utilizar el módulo de `IOHexperimienter`, existe la opción de integrar un *logger* que registra cierta información respecto a la ejecución de los algoritmos. Dentro de la información, se puede registrar las iteraciones realizadas, además de la solución candidata y la calidad en cada una de ellas, lo cual permite tener la información necesaria para medir las métricas de rendimiento que se implementan más adelante en esta propuesta.

Es importante aclarar que si bien este *logger* solo funciona cuando se integran algoritmos desde el módulo de `IOHAlgorithm`, se puede crear un *logger* simple en cualquier lenguaje de programación, que imprima los valores en el formato que indica la figura 13, es decir, la primera columna es la iteración actual del algoritmo, luego la calidad o valor de la función de evaluación (y_i) para la solución candidata que se imprime el valor de cada variable en las siguientes n columnas, donde n representa la dimensión de la instancia que se está resolviendo.

```
i yi x1 x2 x3 ... xn
1 0 0 0 0 ... 0
2 6 1 1 0 ... 0
3 5 0 1 0 ... 0
...
100 20 0 1 1 ... 0
```

Figura 13: Formato de archivo de salida *logger*
Fuente: Elaboración propia

La idea es que cuando se ejecuta el algoritmo para resolver un conjunto de instancias, `IOHexperimenter` genera un conjunto de *logs* que es leído por `MHTMetLib`, capturando toda esta información de ejecución, que termina siendo el *input* para las distintas métricas de rendimiento que se implementan en esta librería.

3.2. Métricas propuestas en `MHTMetLib`

Luego de capturar la información del *logger*, se genera un conjunto de métricas que se explicaran en esta sección, que adicionalmente se pueden presentar de forma gráfica, ya sea mediante un *scatter* y un *boxplot*, esto mediante la librería `Matplotlib` de Python.

3.2.1. Curva de convergencia

Para comenzar, se captura la información sin depurar del *logger*. En este caso, se enfoca en la calidad de la solución candidata visitada por iteración. Esto permite obtener una primera vista respecto a la distribución de la calidad de las soluciones candidatas y la toma de decisiones propia del algoritmo.

Como se ve en la figura 14, se puede revisar un patrón preliminar respecto a como se va realizando el proceso de intensificar y diversificar dentro del algoritmo, analizando el cambio de calidad en cada iteración, y además con el *boxplot* revisar la distribución de la calidad de la solución dentro de la ejecución. En este caso, el algoritmo oscila dentro de un mismo rango durante las primeras 5000 iteraciones, para luego comenzar a mejorar las soluciones candidatas de forma considerable en las iteraciones restantes.

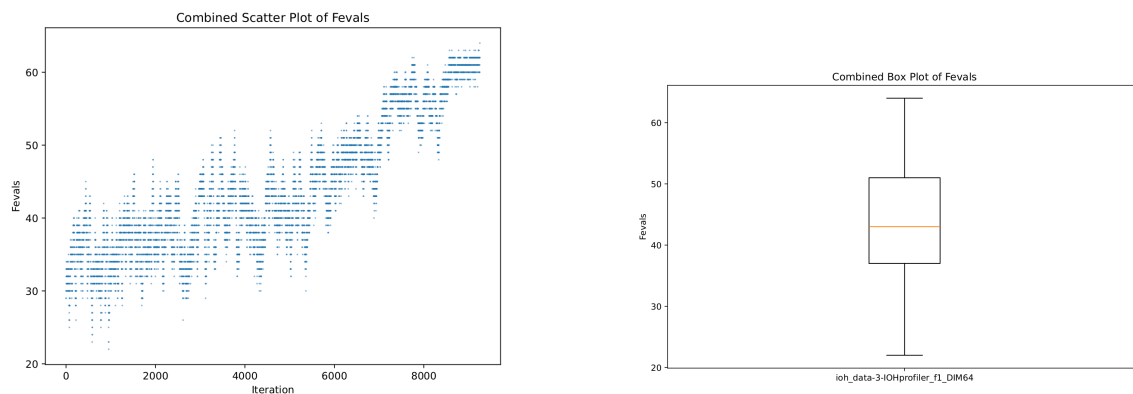


Figura 14: Gráfico de dispersión y *boxplot* de calidad de solución actual por iteración
Fuente: Elaboración propia

3.2.2. Current Best

En un algoritmo trayectorial, el *current best* (mejor actual) se refiere a la mejor solución encontrada hasta un momento dado de la ejecución. Es una referencia a la solución que, entre todas las evaluadas hasta ese punto, tiene el valor óptimo (ya sea mínimo o máximo) de la función objetivo.

Si se tiene una función de evaluación $f(x)$ que se debe optimizar (minimizar o maximizar) y un conjunto de soluciones candidatas S hasta la iteración i . El *current best* cb_i en el caso

de una minimización se puede expresar como:

$$cb_i = \min_{j=1,2,\dots,i} f(x_j)$$

En el caso de una maximización, sería:

$$cb_i = \max_{j=1,2,\dots,i} f(x_j)$$

donde:

- i es la iteración actual del algoritmo.
- $f(x_j)$ es la función de evaluación evaluada en la solución candidata de la iteración j .

Al obtener el cb_i para las distintas iteraciones, permite medir cada cuanto tiempo el algoritmo logra mejoras, la curva que genera y el nivel de mejora que logra gráficamente mediante el gráfico de dispersión.

Como se puede ver en los gráficos de la figura 15, se puede visualizar estancamiento cuando no existen mejoras, como se presenta entre las iteraciones 4000 y 6000, mientras que la calidad se distribuye entre 47 y 53, según la distribución de la función de evaluación que entrega el gráfico *boxplot*.

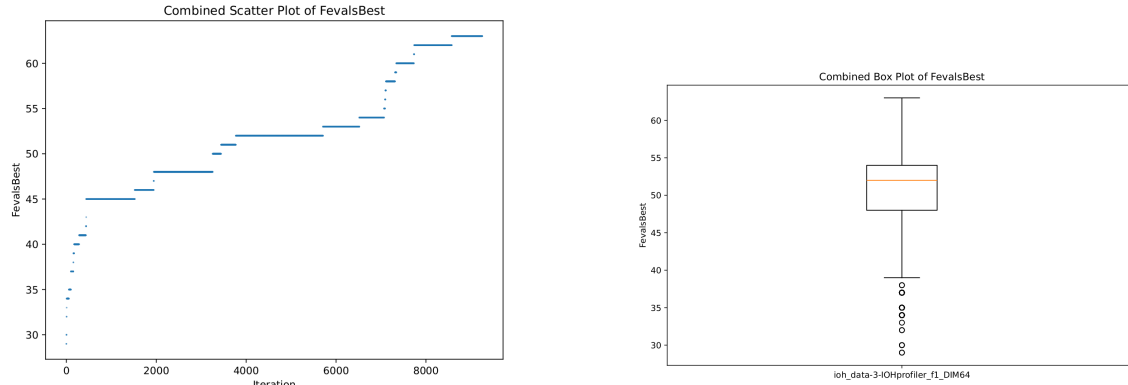


Figura 15: Gráfico de dispersión y *boxplot* de calidad mejor solución actual por iteración
Fuente: Elaboración propia

3.2.3. Ratio de Error

El ratio de error permite determinar la distancia de error que existe entre la calidad de la solución actual s_i y la mejor solución conocida, ya sea teórica o experimental del problema. En este caso, cuando el error es cercano a 1, más lejos está de la solución óptima (ya sea local o global), mientras que el error cercano a 0 indica que la solución tiene una calidad cercana al óptimo.

Si se busca maximizar la función de evaluación:

$$ER_{\max} = 1 - \frac{f_i}{f_{\text{best}}} \quad (5)$$

En cambio, si se busca minimizar la función de evaluación:

$$ER_{\min} = 1 - \frac{f_{\text{best}}}{f_i} \quad (6)$$

En este caso, en la figura 16 se puede revisar que tan alejado está del óptimo teórico o experimental, entregando un patrón a medida que avanza la ejecución del algoritmo. En este caso, se ve que el algoritmo converge cada vez más a ese óptimo y el error va disminuyendo hasta llegar a 0.

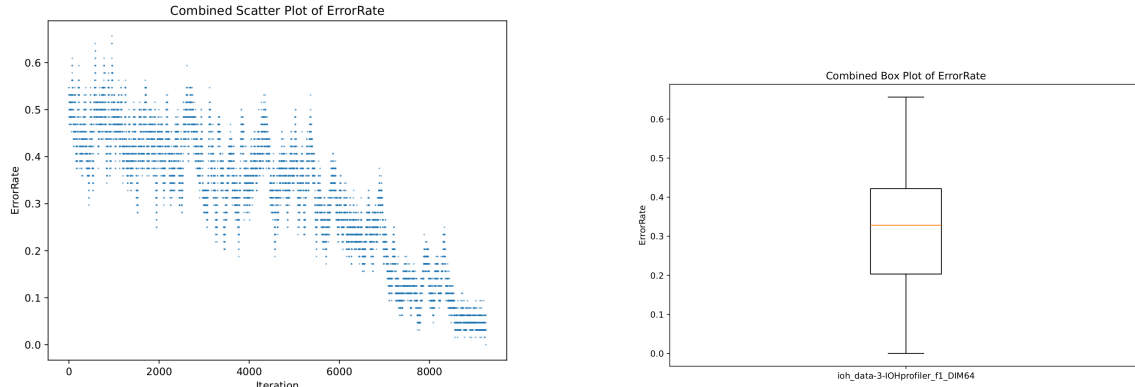


Figura 16: Gráfico de dispersión y *boxplot* de ratio de error por iteración
Fuente: Elaboración propia

3.2.4. Ratio de Convergencia

En este caso, se implementa el ratio de convergencia presentado en el marco conceptual, donde se calcula la razón de cambio entre la calidad de la solución actual en la iteración i , con la calidad de la solución en la iteración anterior s_{i-1} y posterior s_{i+1} , esto basado en la fórmula:

$$CR = \frac{|f_{i+1} - f_i|}{|f_i - f_{i-1}|} \quad (7)$$

En la figura 17, se aprecia como el gráfico de dispersión entrega un patrón complejo de analizar respecto a cuanto es el nivel de mejora o empeoramiento en general. En cambio, con el gráfico *boxplot* se puede lograr revisar una distribución completa de los ratios y se logra encontrar una evidencia para determinar cuanto en general mejoró o empeoró la calidad en las distintas iteraciones del algoritmo. En este caso, las soluciones por iteración muestran un empeoramiento con respecto a su iteración anterior a la mayoría de iteraciones.

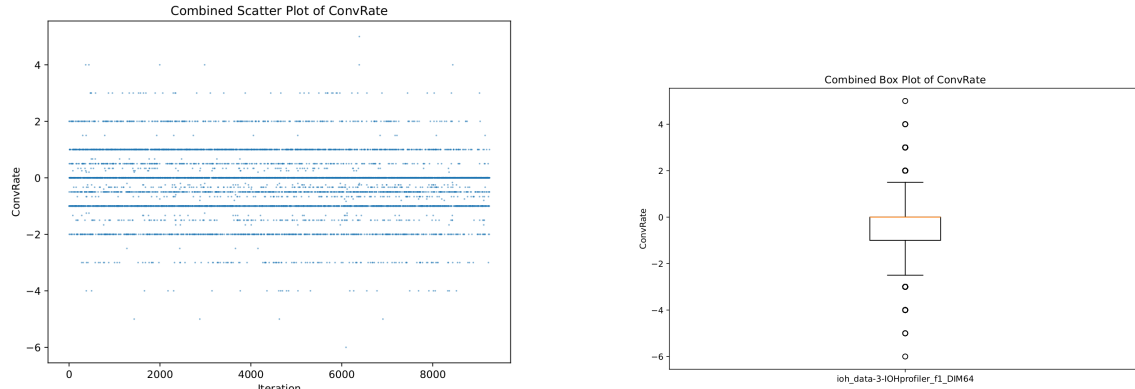


Figura 17: Gráfico de dispersión y *boxplot* de ratio de convergencia por iteración
Fuente: Elaboración propia

3.2.5. Ratio de Convergencia Geométrico

Para el ratio de convergencia geométrico, se añade como parámetro adicional a la fórmula (7), la iteración donde se calcula el ratio de convergencia, esto para darle mayor peso a las variaciones logradas en etapas tempranas del algoritmo, con respecto a las etapas finales. Es por esto, que a medida que avanzan las iteraciones, se podrían obtener ratios de convergencia iguales en distintas iteraciones, pero el ratio geométrico será distinto para cada uno dependiendo de la iteración que represente. Las variaciones logradas en las primeras iteraciones suelen ser más significativas, ya que el algoritmo está explorando y haciendo ajustes importantes. Al darles más peso, se reconoce su impacto en la mejora global. En las etapas finales, las mejoras suelen ser más pequeñas, ya que el algoritmo se aproxima a una solución óptima. Si estas mejoras menores fueran valoradas igual que las iniciales, podría dar la impresión de que el algoritmo sigue mejorando a un ritmo constante, cuando en realidad se está acercando a su límite.

Para este caso, al igual que el ratio de convergencia sin signo, no afecta si el objetivo es maximizar o minimizar la función, y además se utiliza la mejor solución teórica o experimental del problema para calcular el nivel de cambio entre cada iteración:

$$GCR = 1 - \left(\frac{|f_{\text{opt}} - f_i|}{|f_{\text{opt}} - f_0|} \right)^{\frac{1}{i}} \quad (8)$$

Como se ve en la figura 18, a medida que avanzan las iteraciones, la medida de GCR va disminuyendo considerablemente, acercándose cada vez más a 0, pero luego logra en la última iteración del algoritmo alcanzar el valor 1, evidenciando que logró alcanzar el óptimo teórico o experimental de la instancia del problema.

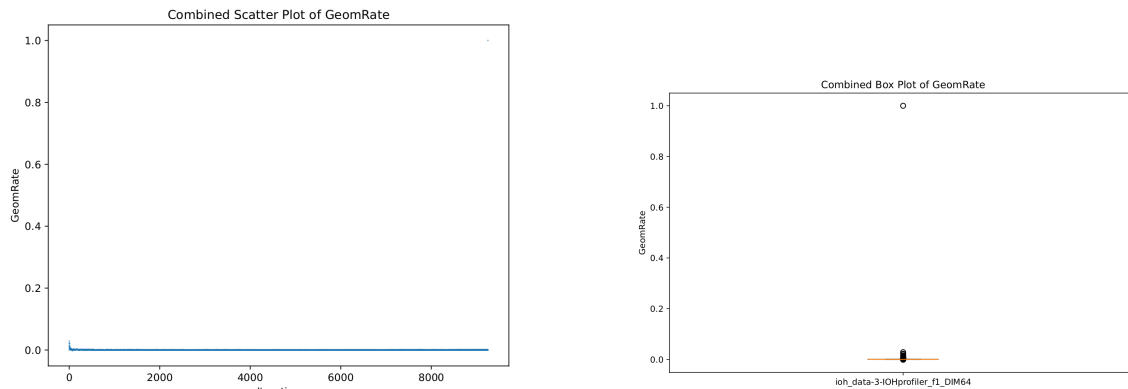


Figura 18: Gráfico de dispersión y *boxplot* de ratio de convergencia geométrico por iteración

Fuente: Elaboración propia

3.2.6. Diversidad

Se implementa la diversidad del marco conceptual, ya que permite obtener información respecto al nivel de exploración que genera tanto el algoritmo como los operadores. La diversidad ayuda a balancear la exploración y la explotación. Un nivel adecuado de diversidad asegura que el algoritmo no se quede atrapado en óptimos locales y siga buscando posibles mejores soluciones en todo el espacio. Sin una diversidad adecuada, el algoritmo podría converger prematuramente a una solución subóptima. La métrica de diversidad permite detectar y corregir situaciones donde el algoritmo comienza a perder su capacidad de explorar, evitando estancamientos.

Como se presentó en el capítulo anterior, la diversidad calcula un promedio de cambio de las variables de las soluciones candidatas, respecto el valor y el promedio de cada una de las dimensiones.

$$\bar{x}_j = \frac{1}{m} \sum_{i=1}^m x_{ij}$$

$$Div_j = \frac{1}{m} \sum_{i=1}^m |x_{ij} - \bar{x}_j|$$

$$Div = \frac{1}{n} \sum_{j=1}^b Div_j \quad (9)$$

Donde j es la dimensión o posición del vector solución, i es el vector de solución de la iteración o generación i y m es la cantidad de iteraciones totales del algoritmo. Por otro lado, $\bar{x}_j$ es el promedio de todos los valores en la dimensión j para cada una de las soluciones de las distintas iteraciones, y n la cantidad de dimensiones que tiene el problema, es decir, la cantidad de variables.

En la figura 19, se puede apreciar como a medida que el algoritmo avanza en su ejecución, se logra estabilizar la diversidad del algoritmo, alcanzando un valor estable que trata de explicar que mientras más iteraciones exista en la ejecución, la capacidad de lograr soluciones nuevas es más baja. Esto se debe a que a medida que se aumentan las iteraciones, resulta más complejo lograr diferenciarse de la solución promedio, es decir, la métrica tiende a estabilizarse rápidamente y forma un patrón más continuo a lo largo del tiempo.

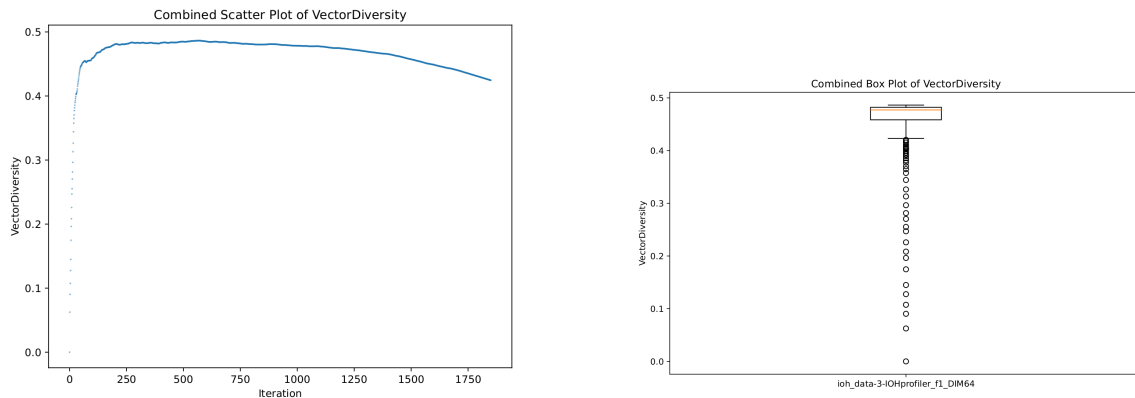


Figura 19: Gráfico de dispersión y *boxplot* de diversidad por iteración
Fuente: Elaboración propia

3.2.7. Diversidad basada en distancia al centro

Se propone una métrica de diversidad que se enfoque en analizar los cambios en la estructura de las soluciones, es decir, en los valores asignados a las variables. El objetivo de esta métrica es observar la variación de dichas estructuras a lo largo de la ejecución del algoritmo.

Para ello, se toma como referencia una solución inicial vacía u origen, donde todas las variables tienen valor cero ($x_1 = 0, x_2 = 0, \dots, x_n = 0$), y se calcula la distancia euclidiana de cada solución generada sobre este origen. La distancia $d(x)$ de una solución x se calcula utilizando la norma 2:

$$d(x) = \|x\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_m^2}$$

Además, en esta implementación se permite dividir la recta en particiones que permiten crear segmentos donde se pueda asignar un conjunto de distancias similares. Estos segmentos dividen el espacio entre 0 y la máxima distancia posible, denotada como r_R , en R segmentos del mismo tamaño. La distancia entre cada segmento se calcula como:

$$r_j = \frac{j \cdot r_{\max}}{R}, \quad \text{para } j = 1, 2, \dots, R$$

donde $r_1 = \frac{r_{\max}}{R}$ es el primer segmento y $r_R = r_{\max}$ es el segmento final. La idea de esto es poder agrupar distancias en representaciones que generen distancias muy altas, y termina siendo complejo de analizar grandes números, además de darle la posibilidad al usuario de la librería de poder ajustar dicho parámetro.

Cada segmento r_j define una partición correspondiente según:

$$\mathcal{P}_j = \{\mathbf{x} \in \mathbb{R}^n \mid r_j \leq d(\mathbf{x}) < r_{j+1}\}$$

La métrica se calcula identificando la partición $\mathcal{P}_j$ en la que se encuentra la solución x . Esto se determina encontrando el máximo segmento r_j tal que $r_j \leq d(x)$. En otras palabras, se busca partición $\mathcal{P}_j$ asociada al segmento más grande r_j que sea menor o igual a la distancia $d(x)$. La expresión para identificar esta partición es la siguiente:

$$ParS = \mathcal{P}_j \quad \text{donde} \quad j = \arg \max_i \{r_i \mid r_i \leq d(x)\}$$

De esta manera, $ParS$ representa una partición en la que se encuentra la solución x , lo cual es útil para evaluar cómo se distribuyen las soluciones en el espacio de búsqueda, además de analizar la exploración y explotación del algoritmo.

Como se puede ver en la figura 20, se considera un escenario con tres segmentos definidos: $r_1 = 3$, $r_2 = 6$ y $r_3 = 9$. Estos segmentos dividen el espacio de búsqueda en tres particiones distintas: Partición 1, Partición 2 y Partición 3. Un algoritmo A comienza con la solución inicial x_1 que comienza en el Partición 1, pero luego, cuando se realiza un movimiento mediante el algoritmo, la solución x_2 en la siguiente iteración pasa a la Partición 2, lo que significa que explora un área distinta a la iteración anterior. Luego, en la siguiente iteración, x_3 se mantiene en la misma área, y finalmente en la última iteración x_4 pasa a la Partición 3. En este ejemplo, se logra revisar todas las particiones, pero mientras más de estas se generan, resulta más complejo para el algoritmo recorrer todas ellas, es por esto que resulta interesante medir que particiones y con qué patrón se visitaron cada una en la ejecución del algoritmo.

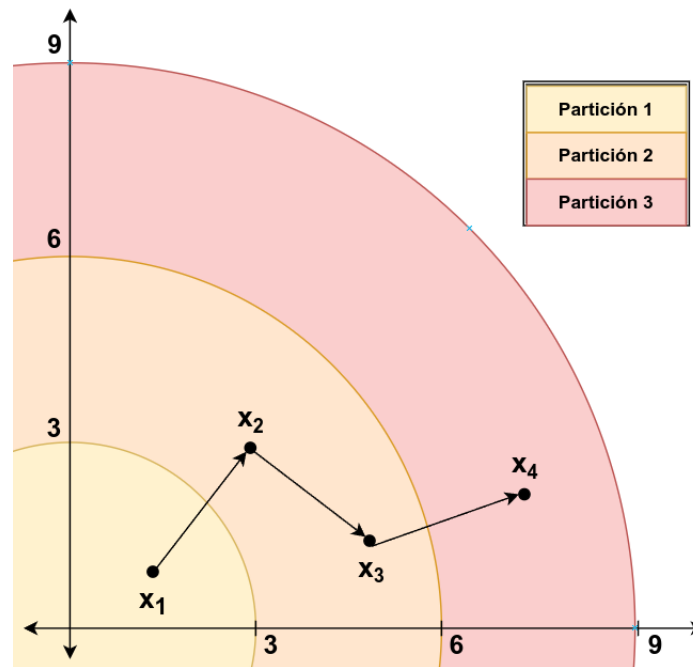


Figura 20: División de regiones en particiones

Fuente: Elaboración propia

En la figura 21, al utilizar segmentos $r_j = j + 1$ $j = 1, 2, \dots, 100$, se puede ver como las soluciones candidatas recorren distintas distancias, que comienzan desde el segmento 22 hasta el segmento 64, siendo esta última donde se encuentra el óptimo de la instancia del problema. Además de conocer el espectro de distancias que recorre, se puede revisar la distribución que logra el algoritmo en múltiples ejecuciones, permitiendo modificar parámetros para lograr visitar zonas distintas en cada ejecución.

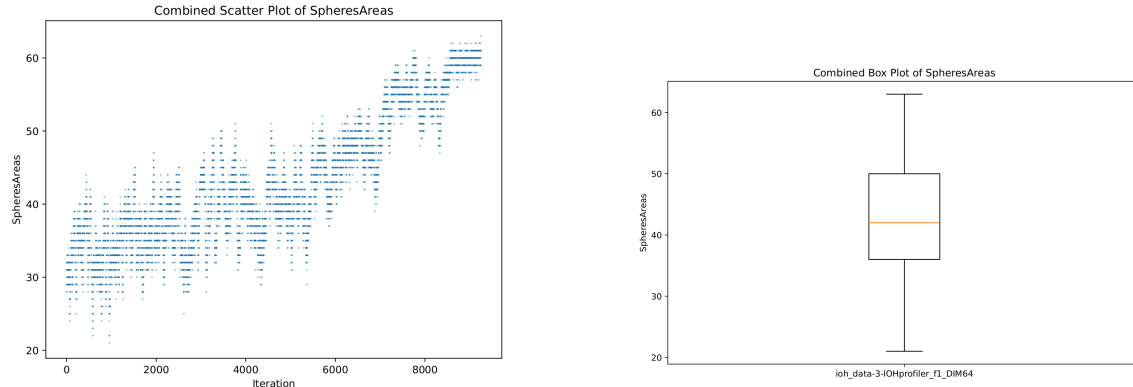


Figura 21: Gráfico de dispersión y *boxplot* de diversidad basada en distancia al centro por iteración

Fuente: Elaboración propia

3.2.8. Distancia de Hamming

La distancia de Hamming es una métrica útil para complementar con la diversidad, ya que permite medir el porcentaje de cambios entre dos soluciones, respecto a los valores de sus variables. En esta métrica, se calcula la diferencia que contiene una cadena de valores, ya sean binarios, numéricos o caracteres, y computa el porcentaje de valores distintos que contiene la cadena, es decir, se normaliza la distancia de Hamming según la cantidad de dimensiones del problema.

$$d_H(x, y) = \frac{\sum_{i=1}^n \begin{cases} 0 & \text{si } x_i = y_i \\ 1 & \text{si } x_i \neq y_i \end{cases}}{n} \quad (10)$$

Por ejemplo, si se tienen dos secuencias binarias x y y de longitud $n = 6$:

$$x = [1, 0, 1, 1, 0, 1]$$

$$y = [1, 1, 0, 1, 0, 0]$$

la Distancia de Hamming normalizada entre x y y se calcula de la siguiente manera:

$$\begin{array}{lll} x_1 = 1, & y_1 = 1 & \Rightarrow 0 \text{ (son iguales)} \\ x_2 = 0, & y_2 = 1 & \Rightarrow 1 \text{ (son diferentes)} \\ x_3 = 1, & y_3 = 0 & \Rightarrow 1 \text{ (son diferentes)} \\ x_4 = 1, & y_4 = 1 & \Rightarrow 0 \text{ (son iguales)} \\ x_5 = 0, & y_5 = 0 & \Rightarrow 0 \text{ (son iguales)} \\ x_6 = 1, & y_6 = 0 & \Rightarrow 1 \text{ (son diferentes)} \end{array}$$

$$\sum_{i=1}^6 \begin{cases} 0 & \text{si } x_i = y_i \\ 1 & \text{si } x_i \neq y_i \end{cases} = 0 + 1 + 1 + 0 + 0 + 1 = 3$$

$$d_H(x, y) = \frac{3}{6} = 0,5$$

Con esta métrica, se puede calcular 3 variantes de la misma:

1. **Distancia de Hamming desde la solución inicial:** En esta primera variante, x es la solución inicial del algoritmo, mientras que y va cambiando entre la solución de cada iteración. El objetivo es medir que tanto logra diferenciar el algoritmo de la primera solución, ya sea rápido o lento y en cuanto porcentaje.

En la figura 22, se puede ver como se logra diferenciar bastante de la solución inicial, logrando un 50 % de diferencia en los valores iniciales que comenzó el algoritmo en la mayoría de la ejecución.

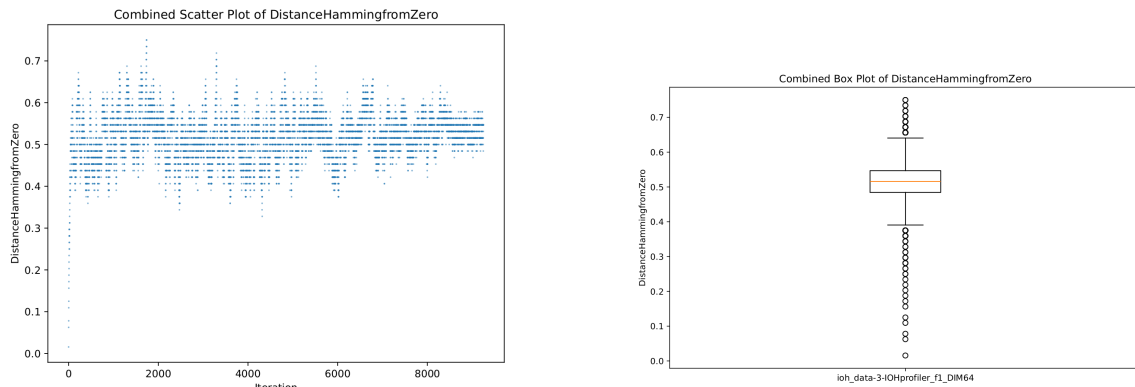


Figura 22: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la solución inicial por iteración

Fuente: Elaboración propia

2. **Distancia de Hamming entre iteraciones:** Para esta variante, x es la solución en la iteración i , mientras que y es la solución en la iteración $i + 1$. Para este caso, se busca obtener información respecto a cuanto modifica la solución el operador de búsqueda.

En la figura 23, se aprecia que el movimiento que contiene el algoritmo modifica levemente la solución, logrando alcanzar en general menos de un 10 % de diferencia con respecto a cada iteración.

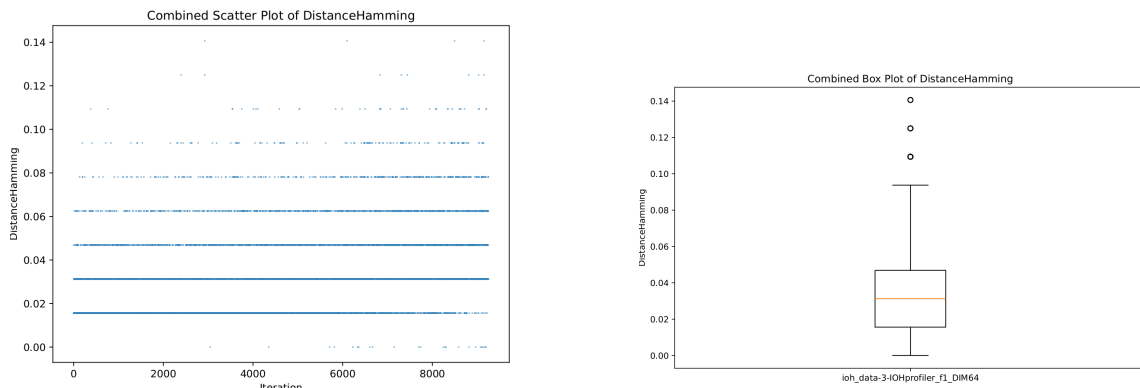


Figura 23: Gráfico de dispersión y *boxplot* de distancia de Hamming entre iteraciones

Fuente: Elaboración propia

3. **Distancia de Hamming desde *Current Best*:** Por último, para esta variante x es la solución en la iteración i , mientras que y es el *Current Best* alcanzado hasta la iteración i . Con esto, se puede obtener información respecto a cuanto porcentaje de la última solución se ha modificado.

En la figura 24, se aprecia que a medida que en las primeras iteraciones logra diferenciarse bastante de la mejor solución obtenida, pero a medida que avanzan las iteraciones esa distancia se va reduciendo debido a que principalmente el algoritmo logra mejoras de forma más constantes hasta alcanzar el óptimo.

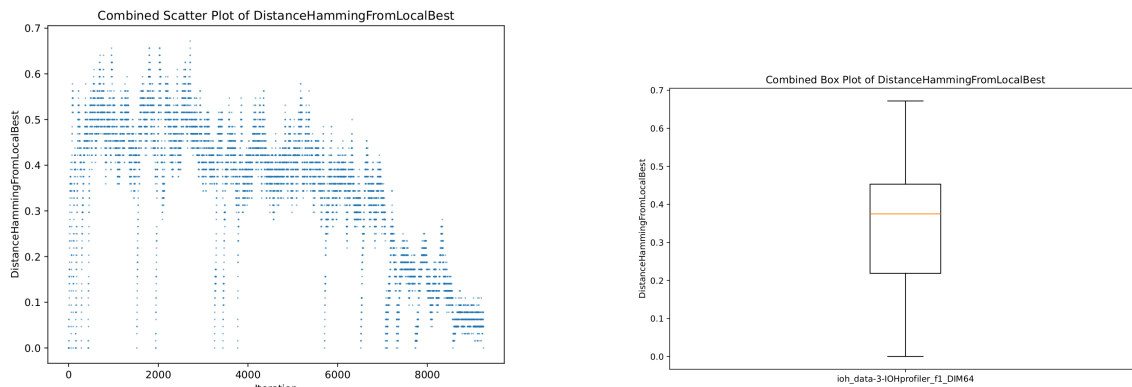


Figura 24: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la mejor solución actual por iteración

Fuente: Elaboración propia

3.2.9. Suma acumulada de proceso de intensificación y diversificación

En los algoritmos existe un concepto clave que se busca medir, el cual indica en qué momentos del algoritmo se intensifica la búsqueda, y en cuáles el algoritmo diversifica. El objetivo final de esto es analizar si existe un equilibrio entre ambos y que tanto puede afectar el sesgo hacia un extremo en la búsqueda de óptimos en el espacio.

Para esta métrica, se utiliza la siguiente idea: Para cada una de las iteraciones, se revisa si existe una mejora con respecto a la iteración anterior o no. En caso de que exista una mejora, se suma 1, en caso contrario, suma -1 a la métrica. Con esto se obtiene un gráfico que varía a medida que pasan iteraciones del algoritmo, y se puede revisar un patrón respecto a las etapas de intensificación y diversificación.

$$S_{ID} = \sum_i \begin{cases} 1 & \text{si } f_i > f_{i-1} \\ 0 & \text{si } f_i = f_{i-1} \\ -1 & \text{si } f_i < f_{i-1} \end{cases} \quad (11)$$

En el caso de la figura 25, se aprecia que principalmente diversifica más de lo que intensifica, logrando generar una diferencia de más de 2500 iteraciones entre ambos procesos, es decir, se realiza 2500 iteraciones más de diversificación que intensificación. Esto puede dar una evidencia de porque logró alcanzar el óptimo global del problema, el cual se puede complementar con el resto de métricas de este capítulo para validar dicha hipótesis.

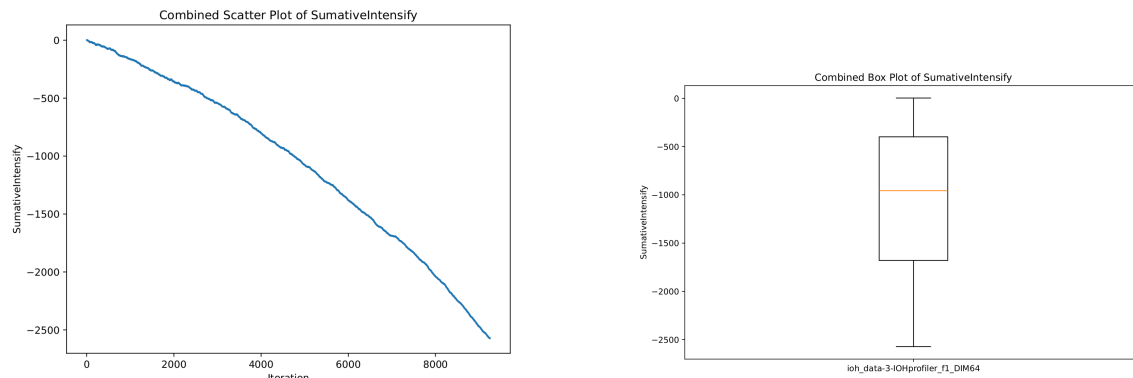


Figura 25: Gráfico de dispersión y *boxplot* de suma acumulada de proceso de intensificación y diversificación por iteración

Fuente: Elaboración propia

Con estas métricas, a continuación, en el siguiente capítulo se buscará explorar la capacidad de análisis que se puede lograr al tener mayor información debido a esta librería, en un escenario donde múltiples algoritmos tratan de resolver el mismo conjunto de problemas, y así lograr el objetivo de mejorar el proceso de diseño de cada uno de ellos mediante la experimentación.

3.2.10. Distribución de mejora o empeoramiento

Además de obtener con la métrica anterior, un espectro general de en qué momentos del algoritmo se intensifica la búsqueda, y en cuáles el algoritmo diversifica, es interesante

tener una distribución respecto a cuál es el *delta* de mejora o empeoramiento respecto a la calidad de las soluciones candidatas entre iteraciones. El objetivo de esto es poder determinar cuanto y cuando se producen tanto las mejoras, como los empeoramientos, dentro de la ejecución del algoritmo.

Para esta métrica, se define la diferencia de calidad entre la solución candidata s_i en la iteración i y la solución candidata s_{i-1} en la iteración $i - 1$ se define como:

$$\Delta Q_i = Q(s_i) - Q(s_{i-1})$$

Luego se definen los conjuntos de mejoras M y empeoramientos E como:

$$M = \{\Delta Q_i \mid \Delta Q_i > 0\}$$

$$E = \{\Delta Q_i \mid \Delta Q_i < 0\}$$

Si la diferencia es positiva ($\Delta Q_i > 0$), se considera una mejora y se almacena en el conjunto de mejoras. Si la diferencia es negativa ($\Delta Q_i < 0$), se considera un empeoramiento y se almacena en el conjunto de empeoramientos. Si la diferencia es cero ($\Delta Q_i = 0$), no se almacena en ninguno de los arreglos para evitar sesgos en la distribución.

En la figura 26, se puede ver un ejemplo donde la ejecución del algoritmo genera que el 50 % de las mejoras se distribuyan entre los 8 y las 30 unidades, con varios *outliers* que superan las 60 unidades. Mientras que el 50 % de los empeoramientos se distribuyen de una manera similar, entre los 8 y las 25 unidades. Con esto se puede ver un equilibrio respecto al cambio entre mejoras y empeoramientos que existe con respecto a los operadores que utiliza este algoritmo.

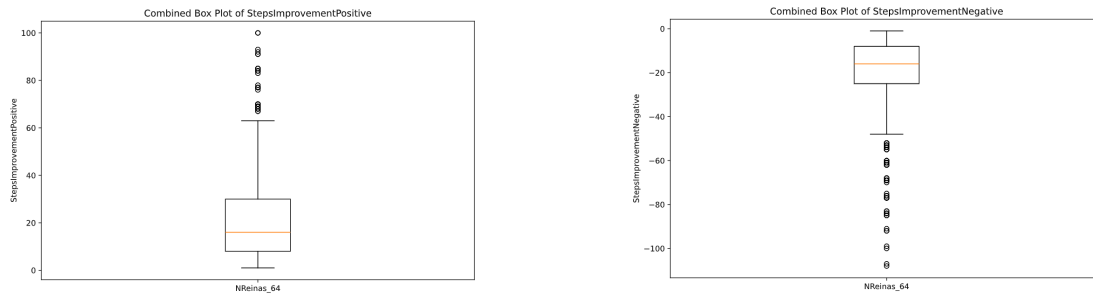


Figura 26: *Boxplot* de distribución de mejora y empeoramiento en ejemplo de ejecución
Fuente: Elaboración propia

Con estas métricas, en el siguiente capítulo se buscará explorar la capacidad de análisis que se puede lograr al tener mayor información debido a esta librería, en un escenario donde múltiples algoritmos tratan de resolver el mismo conjunto de problemas, y así lograr el objetivo de mejorar el proceso de diseño de cada uno de ellos mediante la experimentación.

3.2.11. Resumen de métricas implementadas

A continuación, se presenta la tabla resumen 1, con las 12 métricas explicadas en este capítulo, donde se encuentra el nombre de la métrica, la fórmula que se utiliza para calcular sus valores y por último la clasificación o familia de métricas que representa: Eficiencia, métricas que miden la velocidad con la cual es el algoritmo logra mejorar las soluciones; Exploración, para determinar la capacidad del algoritmo de explorar distintas estructuras de solución; Exploración de operadores, donde busca determinar el nivel de cambio que generan los operadores para perturbar la solución; y por último, Exploración e Intensificación, donde se mide cuanto tiempo transcurre entre cada etapa del proceso y el porcentaje de éxito que logran en cada proceso.

Nombre	Fórmula	Clasificación
Curva de convergencia	$y = f_i$	Eficiencia
<i>Current Best</i>	$cb_{\max} = \max_{j=1,2,\dots,i} f(x_j)$ $cb_{\min} = \min_{j=1,2,\dots,i} f(x_j)$	Eficiencia
Ratio de Error	$ER_{\max} = 1 - \frac{f_i}{f_{\text{best}}}$ $ER_{\min} = 1 - \frac{f_{\text{best}}}{f_i}$	Eficiencia
Ratio de Convergencia	$CR = \frac{ f_{i+1} - f_i }{ f_i - f_{i-1} }$	Eficiencia
Ratio de Convergencia Geométrico	$GCR = 1 - \left(\frac{ f_{\text{opt}} - f_i }{ f_{\text{opt}} - f_0 } \right)^{\frac{1}{i}}$	Eficiencia
Diversidad	$Div = \frac{1}{n} \sum_{j=1}^b Div_j$	Exploración
Diversidad basada en distancia al centro	$ParS = \mathcal{A}_j$ donde $j = \arg \max_i \{r_i \mid r_i \leq d(x)\}$	Exploración
Distancia de Hamming desde la solución inicial	$d_H(f_0, f_i)$	Exploración de operadores
Distancia de Hamming entre iteraciones	$d_H(f_i - 1, f_i)$	Exploración de operadores
Distancia de Hamming desde <i>Current Best</i>	$d_H(cb_i, f_i)$	Exploración de operadores
Suma acumulada de proceso de intensificación y diversificación	$S_{ID} = \sum_i \begin{cases} 1 & \text{si } f_i > f_{i-1} \\ 0 & \text{si } f_i = f_{i-1} \\ -1 & \text{si } f_i < f_{i-1} \end{cases}$	Exploración e Intensificación
Distribución de mejora o empeoramiento	$\Delta Q_i = Q(s_i) - Q(s_{i-1})$	Exploración e Intensificación

Tabla 1: Resumen de métricas implementadas en MHTMetLib

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Para la validación de la propuesta, se trabajará con una *suite* de problemas integrados en *IOHprofiler*, conocido dentro de la literatura como PBO (*Pseudo-Boolean Optimization*), los cuales son 25 problemas de optimización booleana, que permitirá analizar la ejecución de los distintos algoritmos. A continuación se presentará un resumen de los 25 problemas que se presentan en esta *suite*.

4.1. PBO

4.1.1. *OneMax*

La función *OneMax* es una función simple y una de las más estudiadas al momento de realizar *benchmark* a algoritmos de optimización. Formalmente, dado un vector binario x de longitud n , la función *OneMax* se define como la suma de los elementos del vector x que son iguales a 1, es decir:

$$f_1 : \{0, 1\} \rightarrow [0..n], x \mapsto \sum_{i=1}^n x_i \quad (12)$$

En cuanto a su dificultad, la función *OneMax* en sí misma no presenta desafíos significativos desde el punto de vista computacional. Sin embargo, su uso está más relacionado con la capacidad de los algoritmos para encontrar la solución óptima en un espacio de búsqueda combinatorio, lo cual puede ser un problema complejo dependiendo del tamaño y la naturaleza del problema en cuestión.

4.1.2. *LeadingOnes*

La función *LeadingOnes* es otra función comúnmente utilizada en el campo de los algoritmos de optimización combinatoria, especialmente en el contexto de algoritmos genéticos

y evolutivos. Dado un vector binario x de longitud n , la función *LeadingOnes* cuenta la cantidad de unos consecutivos desde el inicio del vector hasta el primer cero encontrado, es decir:

$$f_2 : \{0, 1\} \rightarrow [0..n], x \mapsto \max\{i \in [0..n] \mid \forall j \leq i : x_j = 1\} = \sum_{i=1}^n \prod_{j=1}^i x_j \quad (13)$$

La función *LeadingOnes* proporciona un tipo diferente de información sobre la estructura de la solución en comparación con la función *OneMax*. Mientras que *OneMax* se enfoca en la cantidad total de unos en el vector, *LeadingOnes* se centra en la longitud de la secuencia de unos consecutivos al principio del vector, lo que puede ser relevante en problemas donde la estructura de la solución es similar a *LeadingOnes*.

4.1.3. *Harmonic weights*

Esta función es un punto medio entre *OneMax*, que fue explicada anteriormente, y *BV* que es similar a *OneMax* pero con pesos exponenciales dados por $\sum_{i=1}^n 2^{n-i} x_i$. Dado que esta última puede ser demasiado extrema y no representa la mayoría de problemas que tienen pesos asociados a su solución, nace *Harmonic weights* que distribuye los pesos de forma lineal en toda la solución, es decir:

$$f_3 : \{0, 1\} \rightarrow \mathbb{R}, x \mapsto \sum_{i=1}^n i \cdot x_i \quad (14)$$

4.1.4. *W-model-transformed OneMax/LeadingOnes*

Existe una colección de formas diferentes de perturbar problemas para obtener nuevas funciones con dificultades escalables y distintos espacios de búsqueda, conocidas como el modelo W. Estas transformaciones del modelo W pueden combinarse arbitrariamente, dando lugar a un conjunto enorme de problemas posibles. Además, estas transformaciones pueden superponerse a cualquier problema base, proporcionando así otro grado de libertad. Las 4 transformaciones que posee este modelo son:

1. **Reducción de variables *dummy*** $W(m, *, *, *)$: Se reduce las variables al transformar cada variable de la solución $(x_1, \dots, x_n)$ hacia un *sub-string* $(x_{i_1}, \dots, x_{i_m})$ elegidos al azar, de forma diferente según $i_1, \dots, i_m \in [n]$. Esta modificación modela una situación en que algunas variables de decisión no tienen o tienen solo un impacto insignificante en la función de evaluación.

Ejemplo: Sí $n = 10$, $m = 5$, $i_1 = 1$, $i_2 = 2$, $i_3 = 4$, $i_4 = 7$, $i_5 = 10$, la solución $x = (1, 0, 1, 0, 1, 0, 1, 0, 1, 0)$ se reduce a $x' = (1, 0, 0, 1, 0)$.

2. **Neutralidad** $W(*, \mu, *, *)$: La solución $(x_1, \dots, x_n)$ se reduce a $(y_1, \dots, y_n)$, con $m = n/\mu$, donde μ es el parámetro de transformación. Por cada $i \in [m]$ el valor de y_i corresponde a la mayoría de *bits* que tiene el *sub-string* $(x_{(i-1)\mu+1}, x_{(i-1)\mu+2}, \dots, x_{i\mu})$. Es decir, si el *sub-string* tiene al menos la mitad de unos, $y_i = 1$, en caso contrario $y_i = 0$. Cuando $n/\mu \notin \mathbb{N}$, es decir, no se divide la solución en partes iguales de tamaño μ , los últimos $n - \mu \lfloor n/\mu \rfloor$ *bits* que no entran en un *sub-string*, son simplemente eliminados y no se utilizan para reducir la solución.

Ejemplo: Sí $n = 10$, $\mu = 3$, la solución $x = (1, 1, 1, 0, 1, 0, 1, 1, 1, 0)$, se generan 3 *sub-strings* que son: $x'_1 = (1, 1, 1)$, $x'_2 = (0, 1, 0)$ y $x'_3 = (1, 1, 1)$. Con esto $y_1 = 1$, $y_2 = 0$ y $y_3 = 1$, lo cual reduce la solución a $x' = (1, 0, 1)$.

3. **Epistasis** $W(*, *, v, *)$: La solución $(x_1, \dots, x_n)$ se divide en bloques subsecuentes de tamaño v . Utilizando una función de perturbación $e_v : \{0, 1\}^v \rightarrow \{0, 1\}^v$, cada *sub-string* es transformado en otro tal que $(y_{(i-1)v+1}, \dots, y_{iv}) = e_v((x_{(i-1)v+1}, \dots, x_{iv}))$. La perturbación $e_v(x)$, con x como el *sub-string* a modificar, se genera de la siguiente forma:

- En el caso de que x tenga 0 en su primer *bit*, se realiza un *shift* binario hacia la izquierda, para luego calcular cada nuevo *bit*, se realiza la operación XOR en todos los demás de x , excepto en el *bit* de la posición i que se está calculando.

Ejemplo: Para calcular $e_4(0101)$, se realiza el *shift* hacia la izquierda, generando el nuevo string $x' = 1010$. Luego se calcula cada nuevo *bit* con la operación $\oplus$ explicada anteriormente:

- $x_1 = x'_2 \oplus x'_3 \oplus x'_4 = 0 \oplus 1 \oplus 0 = 1$
- $x_2 = x'_1 \oplus x'_3 \oplus x'_4 = 1 \oplus 1 \oplus 0 = 0$
- $x_3 = x'_1 \oplus x'_2 \oplus x'_4 = 1 \oplus 0 \oplus 0 = 1$
- $x_4 = x'_1 \oplus x'_2 \oplus x'_3 = 1 \oplus 0 \oplus 1 = 0$

Por lo tanto, la permutación $e_4(0101) = 1010$.

- De lo contrario, si x tiene 1 en su primer *bit*, se debe cambiar ese primer bit a 0, para luego calcular la versión negada de $e_v(x)$, es decir, aplicar el mismo algoritmo del punto anterior, y cambiando todos los *bits* por su valor contrario.

Ejemplo: Para calcular $e_4(1011)$, se cambia el primer *bit* dejando $x = 0011$, se realiza el *shift* hacia la izquierda, generando el nuevo string $x' = 0110$. Luego se calcula cada nuevo *bit* con la operación XOR:

- $x_1 = x'_2 \oplus x'_3 \oplus x'_4 = 0 \oplus 1 \oplus 1 = 0$
- $x_2 = x'_1 \oplus x'_3 \oplus x'_4 = 0 \oplus 1 \oplus 0 = 1$
- $x_3 = x'_1 \oplus x'_2 \oplus x'_4 = 0 \oplus 1 \oplus 0 = 1$
- $x_4 = x'_1 \oplus x'_2 \oplus x'_3 = 0 \oplus 1 \oplus 1 = 0$

Ahora, tenemos que $e_4(0110) = 0110$, por lo que calculamos la versión negada que nos da como resultado $e_4(0110) = e_4(1011) = 1001$.

Con esta definición, se genera cada bloque de tamaño v con la función e_v . En el caso de que n/v no sea una división exacta, los *bits* que sobran al final se calculan con la función $e_{n-v\lfloor n/v \rfloor}$.

Ejemplo: Con $n = 10$, $v = 4$, y con el string $x = (1, 1, 1, 1, 0, 1, 1, 1, 0, 1)$, se generan los bloques $x'_1 = (1, 1, 1, 1)$, $x'_2 = (0, 1, 1, 1)$ y $x'_3 = (0, 1)$, y se calcula e_4 para cada bloque. Por lo tanto, $e_4(1111) = 1110$, $e_4(0111) = 0001$ y $e_2(01) = 10$, y genera el nuevo string $x' = (1, 1, 1, 0, 0, 0, 0, 1, 1, 0)$

4. **Perturbación de la calidad $W(*, *, *, r)$:** Con estas transformaciones, se puede determinar la rugosidad y la capacidad de engaño de una función. A diferencia de las transformaciones anteriores, esta perturbación opera sobre los valores de la función, no sobre las cadenas de *bits*. Se define una función $r : \{f(x) | x \in \{0, 1\}^n\} =: V \rightarrow V$, con esto, la función de evaluación se transforma en $r(f(x))$, es decir, $r \circ f$.

Para facilitar el análisis, se requiere que el óptimo global de la función no cambie al momento de aplicar $r(f(x))$, es decir, si v_{max} es el óptimo de la función $f(x)$, entonces $r(v_{max}) = v_{max}$.

Para la generación de nuevas funciones con este modelo, se utilizan las siguientes 3 funciones de rugosidad:

- $r_1 : [0..s] \rightarrow [0..\lceil s/2 \rceil + 1]$, con $r_1(s) = \lceil s/2 \rceil + 1$, $r_1(i) = \lfloor i/2 \rfloor + 1$ cuando $i < s$ y i es par, y $r_1(i) = \lceil i/2 \rceil + 1$ en caso de que $i < s$ y i sea impar.
- $r_2 : [0..s] \rightarrow [0..s]$, con $r_2(s) = s$, $r_2(i) = i + 1$ cuando $i \equiv s \pmod{2}$ y $i < s$, y $r_2(i) = \max\{i - 1, 0\}$ en cualquier otro caso.

- $r_3 : [0..s] \rightarrow [-5..s]$, con $r_3(s) = s$, $r_3(s - 5j + k) = s - 5j + (4 - k)$ para todo $j \in [s/5]$ y $k \in [0,4]$, y $r_3(k) = s - (5\lfloor s/5 \rfloor - 1) - k$ para todo $k \in [0..s - 5\lfloor s/5 \rfloor - 1]$.

Vemos que la función r_1 mantiene el orden de los valores de la función, pero introduce pequeñas mesetas de igual valor de función. En contraste con r_1 , la función r_2 es una permutación de los posibles valores de la función. Divide el conjunto de posibles valores de la función no óptimos $[0..s-1]$ en bloques de tamaño dos (comenzando en $s-1$ y dirigiéndose en dirección inversa) e intercambia los dos valores en cada bloque. Cuando s es impar, el valor 0 forma su propio bloque con $r_1(0) = 0$. De manera similar, r_3 divide el conjunto de posibles valores de la función en bloques de tamaño cinco (comenzando en $s-1$ y yendo en dirección inversa), e invierte el orden de los valores de la función en cada bloque.

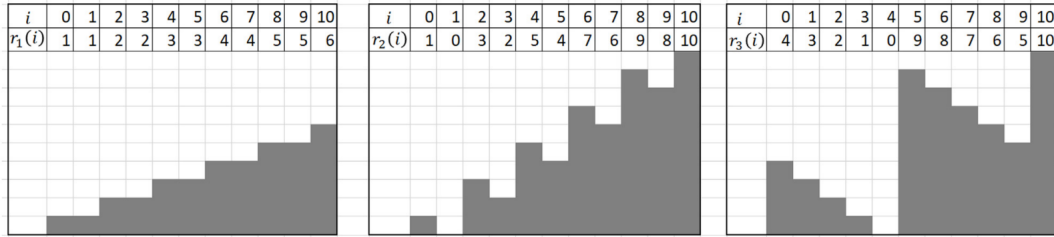


Figura 27: Funciones de rugosidad r_1 , r_2 y r_3

Fuente: *Benchmarking discrete optimization heuristics with IOHprofiler*
[Doerr et al., 2020]

Con este modelo W , se generan las funciones desde la f_4 hasta la f_{17} , donde se utiliza las funciones *OneMax* y *LeadingOnes* como bases para transformar. Las nuevas funciones se generan con los siguientes parámetros:

- $f_4 : \text{OneMax} + W([n/2], 1, 1, \text{id})$
- $f_5 : \text{OneMax} + W([0,9n], 1, 1, \text{id})$
- $f_6 : \text{OneMax} + W([n], 3, 1, \text{id})$
- $f_7 : \text{OneMax} + W([n], 1, 4, \text{id})$
- $f_8 : \text{OneMax} + W([n], 1, 1, r_1)$
- $f_9 : \text{OneMax} + W([n], 1, 1, r_2)$
- $f_{10} : \text{OneMax} + W([n], 1, 1, r_3)$
- $f_{11} : \text{LeadingOnes} + W([n/2], 1, 1, \text{id})$
- $f_{12} : \text{LeadingOnes} + W([0,9n], 1, 1, \text{id})$
- $f_{13} : \text{LeadingOnes} + W([n], 3, 1, \text{id})$

- $f_{14} : \text{LeadingOnes} + W([n], 1, 4, \text{id})$
- $f_{15} : \text{LeadingOnes} + W([n], 1, 1, r_1)$
- $f_{16} : \text{LeadingOnes} + W([n], 1, 1, r_2)$
- $f_{17} : \text{LeadingOnes} + W([n], 1, 1, r_3)$

4.1.5. Low-Autocorrelation Binary Sequence

Obtener secuencias binarias con un alto factor de mérito, también conocido como el problema de Secuencias Binarias de Baja Autocorrelación (LABS), constituye un gran desafío combinatorio con aplicaciones prácticas en la ingeniería de radares y mediciones. También plantea varias preguntas abiertas sobre su naturaleza matemática. Dada una secuencia de longitud n , $S = (s_1, \dots, s_n)$ con $s_i \in \{-1, +1\}$, el factor de mérito es proporcional al recíproco de la autocorrelación de la secuencia. Como una función pseudo-boolean sobre $\{0, 1\}^n$, puede reescribirse de la siguiente manera:

$$f_{18}(\vec{x}) = \frac{n^2}{2 \sum_{k=1}^{n-1} \left(\sum_{i=1}^{n-k} x'_i \cdot x'_{i+k} \right)^2} \text{ donde } x'_i = 2x_i - 1 \quad (15)$$

4.1.6. The Ising model

El modelo de *Ising Spin Glass* surgió de un concepto físico, con el objetivo de describir interacciones simples dentro de sistemas con muchas partículas. El modelo clásico de *Ising* considera un conjunto de espines colocados en una red regular, donde cada arista $\langle i, j \rangle$ está asociada con una fuerza de interacción $J_{i,j}$. En esencia, una instancia del problema se define al establecer la matriz de acoplamiento $\{J_{i,j}\}$. Cada espín se orienta hacia arriba o hacia abajo, asociado con un valor de ± 1 , y un conjunto de n espines se dice que forma una configuración, denotada como $S = (s_1, \dots, s_n) \in \{-1, +1\}^n$. La función de energía de la configuración está descrita por el sistema Hamiltoniano, como una función cuadrática de esas n variables de espín:

$$- \sum_{i < j} J_{i,j} s_i s_j - \sum_{i=1}^n h_i s_i$$

donde h_i es un campo magnético externo. El problema de optimización de interés es el estudio de las configuraciones de energía mínima, que se denominan estados fundamentales, en una red final. Este es claramente un problema de optimización combinatoria de-

safiante, que se sabe que es NP-Hard y tiene conexiones con todos los demás problemas NP.

Para definir formalmente las funciones objetivo de *Ising*, se adopta una perspectiva estricta de grafos, donde $G = (V, E)$ es un grafo no dirigido y $V = [n]$. Se aplica una transformación afín $\{-1, +1\}^n \Rightarrow \{0, 1\}^n$, donde los n espines se convierten en variables de decisión binarias (esto podría interpretarse, por ejemplo, como un problema de coloreo de grafos). Ahora se obtiene una forma generalizada y compacta para la función objetivo cuadrática:

$$F_{\text{Ising}}(\vec{x}) = \sum_{\{u,v\} \in E} [x_u x_v - (1 - x_u)(1 - x_v)] \quad (16)$$

De este modo, la definición de la instancia se mantiene dentro del grafo G . Con esta función objetivo, se generan 3 instancias del modelo *Ising*, modificando el grafo G para utilizar distintas estructuras.

- f_{19} El anillo (1D): Este modelo básico de Ising se define sobre una red unidimensional. La función objetivo sigue el siguiente grafo:

$$G_{\text{Is1D}} : e_{ij} = 1 \Leftrightarrow j = i + 1 \quad \forall i \in \{1, \dots, n - 1\} \\ \vee j = n, i = 1$$

- f_{20} El torus (2D): Esta instancia se define en una red bidimensional de tamaño N , utilizando en total $n = N^2$ vértices, denotados como (i, j) , $0 \leq i, j \leq N - 1$. Se define el siguiente grafo:

$$G_{\text{Is2D}} : e_{(i,j)(k,\ell)} = 1 \Leftrightarrow [k = (i + 1) \text{ mód } N \wedge \ell = j \vee i, j \in \{0, \dots, N - 1\}] \\ \vee [k = (i - 1) \text{ mód } N \wedge \ell = j \vee i, j \in \{0, \dots, N - 1\}] \\ \vee [\ell = (j + 1) \text{ mód } N \wedge k = i \vee j, i \in \{0, \dots, N - 1\}] \\ \vee [\ell = (j - 1) \text{ mód } N \wedge k = i \vee j, i \in \{0, \dots, N - 1\}]$$

- f_{21} Triangular (2D): Esta instancia también se define en una red bidimensional, pero construida en una rejilla isométrica (también conocida como rejilla triangular). Los vértices se colocan en valores enteros bidimensionales $n = N^2$ vértices, denotados como (i, j) , $0 \leq i, j \leq N-1$, resultando en un gráfico regular cuyos vértices tienen exactamente seis vecinos cada uno.

$$\begin{aligned}
 G_{\text{ISTR}} : e_{(i,j)(k,\ell)} = 1 \Leftrightarrow & [k = (i+1) \bmod N \wedge \ell = j \vee i, j \in \{0, \dots, N-1\}] \\
 & \vee [k = (i-1) \bmod N \wedge \ell = j \vee i, j \in \{0, \dots, N-1\}] \\
 & \vee [\ell = (j+1) \bmod N \wedge k = i \vee j, i \in \{0, \dots, N-1\}] \\
 & \vee [\ell = (j-1) \bmod N \wedge k = i \vee j, i \in \{0, \dots, N-1\}] \\
 & \vee [\ell = (j+1) \bmod N \wedge k = (i+1) \bmod N \vee j, i \in \{0, \dots, N-1\}] \\
 & \vee [\ell = (j-1) \bmod N \wedge k = (i-1) \bmod N \vee j, i \in \{0, \dots, N-1\}]
 \end{aligned}$$

4.1.7. Maximum independent vertex set

Dado un grafo $G = ([n], E)$, un conjunto independiente de vértices es un subconjunto de vértices donde no hay dos vértices que sean vecinos directos. MIVS (por sus siglas en inglés) se define como un subconjunto independiente $V' \subset [n]$ con el tamaño más grande posible. Usando la codificación binaria estándar $V' = \{i \in [n] \mid x_i = 1\}$, el MIVS se puede formular como la maximización de la función:

$$f_{22}(x) = \sum_i x_i - n \cdot \sum_{i,j} x_i x_j e_{i,j}, \quad (17)$$

donde $e_{i,j} = 1$ si $\{i, j\} \in E$ y $e_{i,j} = 0$ en caso contrario.

En particular, se considera una instancia específica y escalable del problema, definiendo su grafo booleano de la siguiente manera:

$$\begin{aligned}
 e_{ij} = 1 \Leftrightarrow & j = i + 1, \forall i \in \{1, \dots, n-1\} - \{\frac{n}{2}\} \\
 & \vee j = i + \frac{n}{2} + 1, \forall i \in \{1, \dots, \frac{n}{2} - 1\} \\
 & \vee j = i + \frac{n}{2} - 1, \forall i \in \{2, \dots, \frac{n}{2}\}
 \end{aligned}$$

4.1.8. N reinas

El problema de las N reinas es un clásico problema de ajedrez y programación que consiste en colocar N reinas en un tablero de ajedrez de $N \times N$, de manera que ninguna reina pueda atacar a otra. Es decir, no puede haber dos reinas en la misma fila, columna o diagonal.

Para esta *suite* de problemas, se utiliza $n = N^2$ variables de decisión binarias x_{ij} , que están asociadas con las coordenadas del tablero de ajedrez, teniendo un origen (1, 1) en la esquina superior izquierda. Asignar un valor binario de 1 implica la colocación de una única reina en esa celda. Debido a las simetrías del tablero de ajedrez, el problema de las N reinas (NQP) posee una multiplicidad de soluciones óptimas. Su atractivo, sin embargo, radica en su dificultad. En términos de una función objetivo de caja negra, se formula el NQP como la maximización de la siguiente función:

$$\begin{aligned}
 f_{23}(x) = & \sum_{i=1}^N \sum_{j=1}^N x_{ij} - N \cdot \left(\sum_{i=1}^N \max \left\{ 0, -1 + \sum_{j=1}^N x_{ij} \right\} \right. \\
 & + \sum_{j=1}^N \max \left\{ 0, -1 + \sum_{i=1}^N x_{ij} \right\} \\
 & + \sum_{k=-N+2}^{N-2} \max \left\{ 0, -1 + \sum_{\substack{i,j \in \{1,2,\dots,N\} \\ j-i=k}} x_{ij} \right\} \\
 & \left. + 2 \sum_{\ell=3}^{2N-1} \max \left\{ 0, -1 + \sum_{\substack{i,j \in \{1,2,\dots,N\} \\ j+i=\ell}} x_{ij} \right\} \right) \quad (18)
 \end{aligned}$$

4.1.9. Concatenated Trap

Concatenated Trap (CT) se define particionando una cadena de bits en segmentos de longitud k y concatenando $m = \frac{n}{k}$ funciones trampa que toman cada segmento como entrada. La función trampa se define de la siguiente manera:

$$f_{\text{trap},k}(u) = \begin{cases} 1 & \text{si } u = k \\ k - 1 - u & \text{en caso contrario} \end{cases}$$

4.1.10. *NK landscapes*

Los valores de la función se definen como el promedio de n sub-funciones $F_i : [0, 2^k + 1 - 1] \rightarrow \mathbb{R}, i \in [1..n]$, donde cada componente F_i solo toma como entrada un conjunto de $k \in [0..n - 1]$ *bits* que son especificados por una matriz de vecindario. En esta *suite*, k se establece en 1 y las entradas de la matriz de vecindario se eligen uniformemente al azar en $[1..n]$. Los valores de la función de los F_i se muestrean independientemente de una distribución uniforme en $(0, 1)$.

4.2. Escenario experimental

La etapa de experimentación para esta *suite* de problemas, se llevó a cabo en un computador con procesador AMD Ryzen 3500X, 16 GB de memoria RAM y un total de 200 GB disponibles en disco, esto último debido a la gran cantidad de espacio que necesitan los *logs* de ejecución de los distintos algoritmos.

Se seleccionaron los 3 algoritmos trayectoriales que presenta *IOHAlgorithm*, los cuales son: *Greedy Hill Climbing* (gHC), *Randomized Local Search* (RLS o RS) y *Simulated Annealing* (SA). Se utilizaron sus implementaciones sin modificar, es decir, todos poseen el movimiento *bit-flip*, pero cada algoritmo tiene una toma de decisiones distinta, que los hace diferenciarse a la hora de escoger sus soluciones candidatas.

Se ejecutaron los 25 problemas de la *suite* de PBO, donde para cada uno de ellos, se ejecutaron instancias del problema en las dimensiones de 100, 625, 5625 y 10000 variables. Para cada instancia, se realizaron 5 ejecuciones independientes con cada uno de los algoritmos, utilizando el mismo conjunto de semillas para generar números aleatorios (*seed*) y con máximo de 10000 iteraciones para cada ejecución.

4.3. Resultados

Al ejecutar la *suite*, tomó un total aproximado de 20 horas para obtener los resultados de los 3 algoritmos en los 25 problemas. Luego, para obtener las distintas métricas, el tiempo total utilizado fue de 15 horas, donde para cada instancia tomo un total aproximado de 5 minutos por instancia, donde para las instancias de dimensión 10000 logran tiempos límites de 15 minutos, donde la mayor parte del tiempo se realiza la lectura del archivo de *logs*.

Luego de obtener los resultados para cada conjunto de instancias, el algoritmo que resuelve de forma más eficiente es gHC, donde en cada una de las dimensiones, logro superar al resto de algoritmos en la mayoría de problemas presentes en la *suite* de PBO.

Las tablas 2 y 3, contienen una fila para cada uno de los 25 problemas, donde para cada algoritmo representando en las distintas columnas, se encuentra la mejor solución encontrada en el conjunto de ejecuciones. Las casillas destacadas representan las mejores soluciones entre los 3 algoritmos utilizados, para facilitar la visualización de los datos.

En la tabla 2, para la dimensión de 100 variables, SA es el algoritmo con mejores resultados, donde logra superar a gHC en varios problemas de mayor complejidad computacional, en donde se encuentran problemas como *Ising model* (f_{19}) o N-Reinas (f_{23}), debido a que son problemas definidos como NP, donde encontrar la solución puede ser difícil o incluso infactible en tiempo razonable [Doerr *et al.*, 2020]. Por otro lado, RLS en estas dimensiones no logra obtener resultados óptimos, donde es superado por ambos algoritmos en todos los problemas, exceptuando casos como el problema *LeadingOnes + Epistasis* (f_{14}), donde logra obtener una superioridad, esto debido a que es un problema con demasiados óptimos locales de baja calidad, que terminan funcionando como puntos trampa para algoritmos como gHC y SA, siendo un problema muy difícil de resolver.

Luego, para la dimensión de 625 variables, SA baja considerablemente la cantidad de problemas con mejor resolución, llegando a solo 6, del total de 19 de la dimensión anterior. Por otro lado, gHC y RLS mantienen la misma tendencia de algoritmos en los cuales resultan eficientes, es decir, obtienen las mejores soluciones entre todos los algoritmos, en una misma cantidad de iteraciones límite.

Luego, para la tabla 3, la cual sigue el mismo formato de la tabla anterior, pero donde ahora se encuentran las dimensiones de 5625 y 10000 variables, nuevamente gHC logra superar al resto, obteniendo 18 y 19 problemas con mejor rendimiento. Por otro lado, la brecha entre RLS y SA disminuye, igualando la cantidad de problemas con mejor rendimiento en

la última dimensión.

DIM-100				DIM-625			
	gHC	RS	SA		gHC	RS	SA
f1	100	70	100	f1	625	366	592
f2	100	17	91	f2	625	14	25
f3	5050	3856	5050	f3	195625	115803	195625
f4	50	38	50	f4	312	193	300
f5	90	65	90	f5	562	328	537
f6	33	27	33	f6	208	134	201
f7	79	71	88	f7	498	367	511
f8	45	37	49	f8	270	183	269
f9	55	71	97	f9	316	370	534
f10	59	74	74	f10	339	369	404
f11	50	14	50	f11	312	16	30
f12	90	14	90	f12	562	19	25
f13	33	17	33	f13	208	14	50
f14	6	17	18	f14	7	18	15
f15	51	8	37	f15	83	9	11
f16	100	15	77	f16	80	17	14
f17	4	14	24	f17	14	19	14
f18	4.27	2.24	2.48	f18	4.28	1.44	1.42
f19	56	70	98	f19	323	365	545
f20	200	128	200	f20	1250	700	1082
f21	300	190	300	f21	1875	1031	1643
f22	48	-1267	50	f22	276	-127652	267
f23	8	-632	10	f23	19	-21794	24
f24	16.60	11.40	16	f24	102.6	52.6	70.8
f25	-0.30	-0.41	-0.41	f25	-0.29	-0.45	-0.46

Tabla 2: Calidad mejor *run* en dimensiones de 100 y 625 variables

Fuente: Elaboración propia

DIM-5625				DIM-10000			
	gHC	RS	SA		gHC	RS	SA
f1	5625	2987	3815	f1	10000	5212	6197
f2	5625	17	6	f2	10000	18	5
f3	15823125	8443587	14456970	f3	50005000	26090627	41244231
f4	2812	1514	1972	f4	5000	2666	3195
f5	5062	2696	3457	f5	9000	4700	5586
f6	1875	1022	1398	f6	3333	1790	2228
f7	4401	2963	3878	f7	7818	5196	6467
f8	2353	1488	1768	f8	4201	2606	2931
f9	2840	2978	3734	f9	5075	5240	5823
f10	2864	2968	3089	f10	5064	5210	5239
f11	2812	19	8	f11	5000	18	4
f12	5062	14	4	f12	9000	14	7
f13	1875	15	9	f13	3333	14	7
f14	3	23	3	f14	3	16	6
f15	11	9	6	f15	4	8	4
f16	6	16	22	f16	5	15	9
f17	4	18	9	f17	4	19	4
f18	3.26	1.16	1.11	f18	2.77	1.10	1.06
f19	2865	2969	5857	f19	5076	5212	6278
f20	9148	5854	7296	f20	14196	10292	12180
f21	14879	8695	11103	f21	24138	15360	18054
f22	2343	-13989833	-425815	f22	2233	-45345188	-6017164
f23	52	-759872	-173688	f23	2	-1851511	-807029
f24	916	413	506.8	f24	1628.80	720.80	828.80
f25	-0.30	-0.48	-0.49	f25	-0.32	-0.48	-0.49

Tabla 3: Calidad mejor *run* en dimensiones de 5625 y 10000 variables

Fuente: Elaboración propia

En general, se puede apreciar como gHC logra mantener un rendimiento estable a medida que se aumenta la dimensionalidad del problema, al igual que RLS, mientras que SA sufre una caída a medida que se aumentan las dimensiones, pasando de 19 problemas con mejor rendimiento en la dimensión 100, a 3 problemas en la dimensión 10000. A continuación, se presentará un análisis por algoritmo, para obtener información respecto a la ejecución de esta etapa de experimentación y lograr encontrar puntos de mejora para cada algoritmo utilizado.

4.4. Análisis del algoritmo gHC

Los resultados muestran que gHC fue un algoritmo robusto a la hora de resolver la mayoría de problemas, obteniendo en varios casos el óptimo teórico de la instancia.

En esta sección se analizará el desempeño del algoritmo en los problemas *Low Autocorrelation Binary Sequences* (LABS) (f_{18}) y OneMax + Rugosidad r_3 (f_{10}). LABS resulta interesante de analizar debido a que es un problema catalogado como NP y el algoritmo logra ser el líder en las distintas dimensiones del problema, siendo un algoritmo con un funcionamiento *greedy* que por regla general no debiese tener buenos resultados en este tipo de problemas. Por otro lado, OneMax + Rugosidad r_3 es uno de los problemas con peor rendimiento utilizando gHC, donde es superado por el resto de algoritmos en todas las dimensiones, y en la revisión de las métricas se puede obtener conclusiones respecto a las causas de su rendimiento.

Para las figuras presentes en este capítulo, se presentan la métrica graficada tanto con un gráfico de dispersión como en un gráfico *boxplot* para poder analizar la distribución de la métrica. Se entrega un color para cada algoritmo en el gráfico de dispersión: azul para gHC, naranja para RLS y verde para SA, donde cada punto representa el valor de la métrica calculada según el eje Y, para cada una de las iteraciones representadas en el eje X.

Para el problema LABS (f_{18}), tanto en la curva de convergencia de la figura 28, como en el ratio de error de la figura 29, se visualiza como a medida que avanza la ejecución de gHC, se consigue una mejora constante entre iteraciones y disminuyendo el error relativo con el óptimo teórico de esta instancia. Por otro lado, al revisar la suma acumulada de proceso de intensificación y diversificación de la figura 30, se puede apreciar como gHC está en constante proceso de intensificación, mientras que en el caso de SA y RLS logran un equilibrio entre ambos procesos, pero no resultan ser óptimos para resolver este problema en particular.

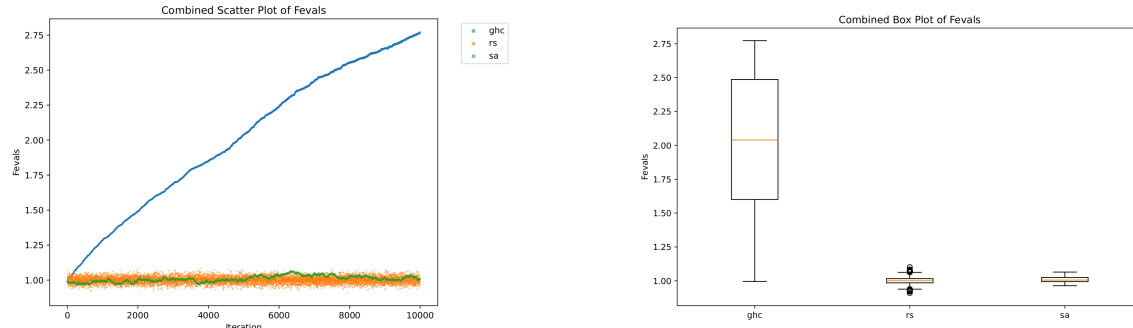


Figura 28: Gráfico de dispersión y *boxplot* de calidad de solución actual por iteración en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

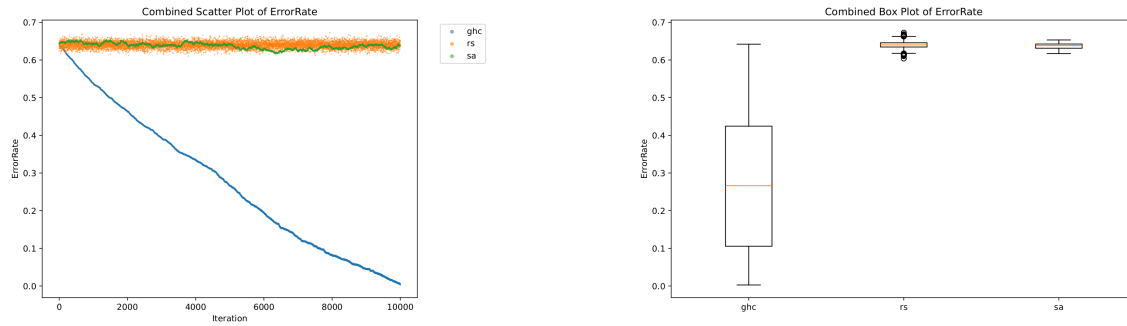


Figura 29: Gráfico de dispersión y *boxplot* de ratio de error por iteración en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

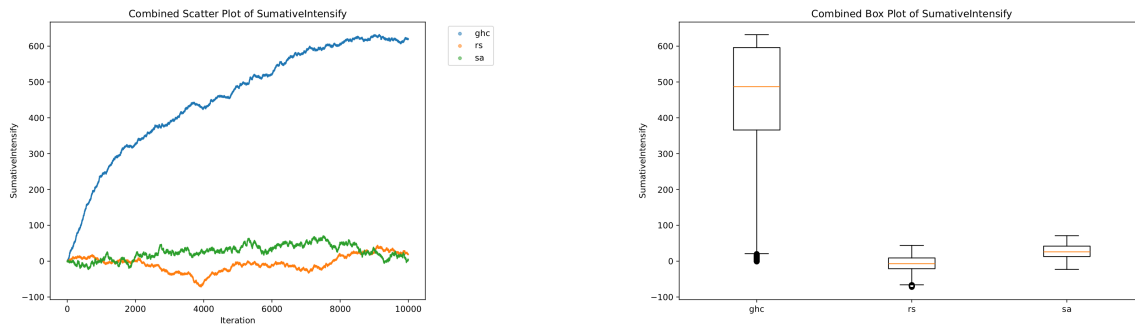


Figura 30: Gráfico de dispersión y *boxplot* de suma acumulada de proceso de intensificación y diversificación por iteración en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

Además, en este problema se puede observar que el algoritmo no diversifica de manera efectiva en comparación con los otros, pero aun así logra obtener los mejores resultados. Esto se evidencia al analizar la métrica de diversidad de la figura 31, donde gHC crece muy lentamente y no cambia significativamente en el tiempo. Esto sugiere que este algoritmo genera variaciones pequeñas en comparación con los otros, ya que SA logra aumentar la diversidad entre las iteraciones de mejor forma.

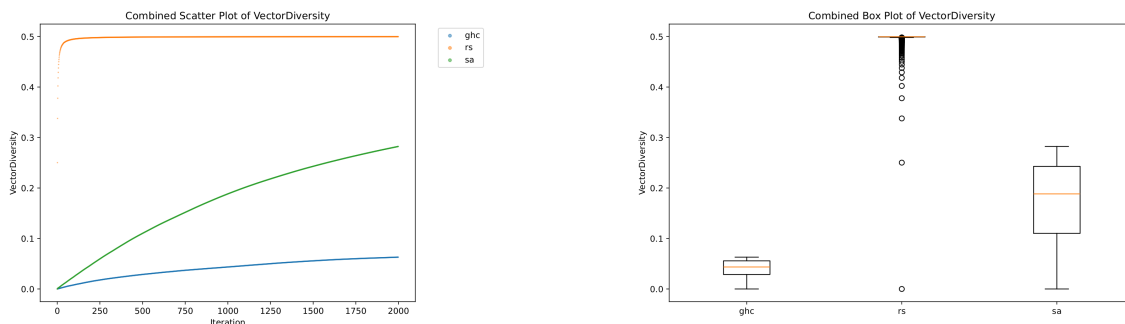


Figura 31: Gráfico de dispersión y *boxplot* de diversidad por iteración en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

Esto también se presenta en la distancia de Hamming en sus distintas variantes, tanto comparando con la primera solución candidata, entre iteraciones o con el *current best* de la ejecución. En cada de una de ellas, se puede apreciar como gHC es el algoritmo que me-

nos porcentaje de cambios al transcurrir la ejecución. En la primera variante de la figura 32, se aprecia como el cambio entre máximo entre la primera solución y la última es solo de un 20 %, mientras que en el resto de algoritmos es de un 50 %, esto evidencia más que gHC no es un algoritmo que explore de forma amplia el espacio de búsqueda, esto debido a que por iteración solo se puede cambiar como máximo una variable de la solución, existiendo casos donde podrían existir iteraciones con la misma solución que mantenía la iteración anterior, debido a que no genero mejoras en la función de evaluación. En la figura 33, que presenta la segunda variante, se demuestra que los cambios que genera entre iteraciones es mínimo, donde los cambios son mínimos, ya que como se define el algoritmo, solo cambia una variable por iteración, lo cual se demuestra en la distancia de Hamming entre iteraciones. Por último, en la variante con el *current best* de la figura 34, se ve el mismo patrón que en la variante anterior, ya que gHC al ir mejorando de forma constante el algoritmo, el *current best* va cambiando rápidamente y no se logra distanciar tanto las soluciones candidatas en las siguientes iteraciones.

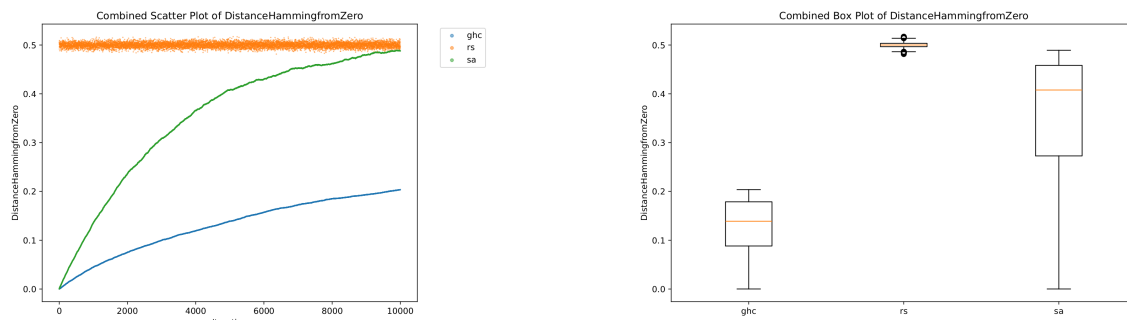


Figura 32: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la solución inicial por iteración en LABS (f_{18}) de dimensión 10000

Fuente: Elaboración propia

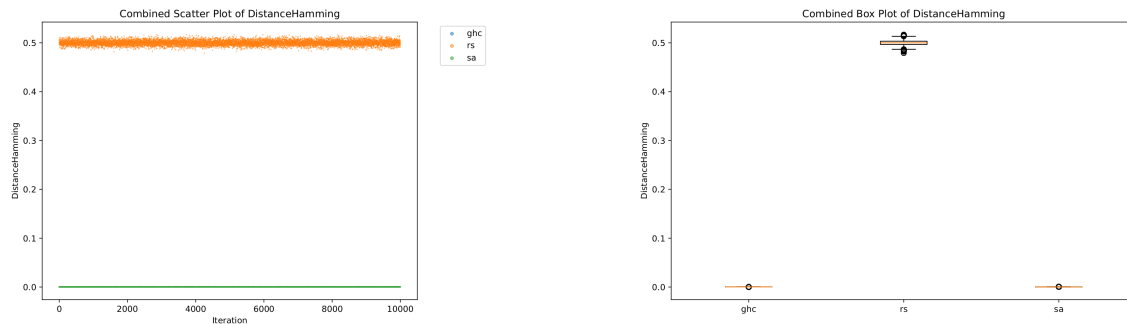


Figura 33: Gráfico de dispersión y *boxplot* de distancia de Hamming entre iteraciones en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

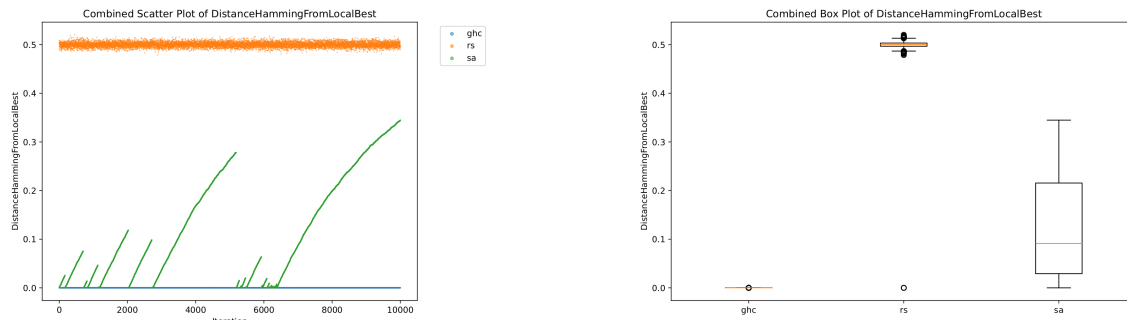


Figura 34: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la mejor solución actual por iteración en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

Por otro lado, las mejoras de gHC son de valores pequeños comparados con RLS. Esto se puede observado en la figura 35, donde se observa el cambio de calidad entre iteraciones al existir mejoras, donde gHC tiene mejoras distribuidas entre 0 y 0,005, mientras que en RLS tiene mejoras entre 0,01 y 0,04. Esto mismo ocurre al revisar los cambios de calidad en empeoramientos, como se puede ver en la figura 36, que tienen la misma distribución presentada en los cambios de mejora, por lo cual presenta un equilibrio entre ambos procesos.

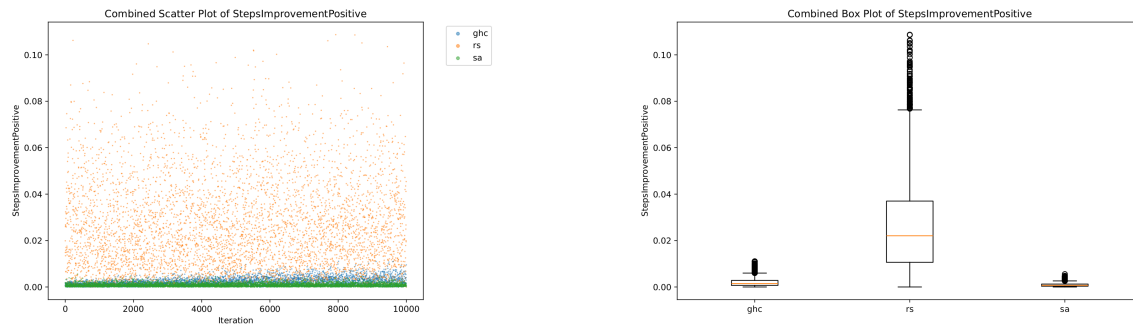


Figura 35: Gráfico de dispersión y *boxplot* de distribución de *delta* de mejora de ejecución en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

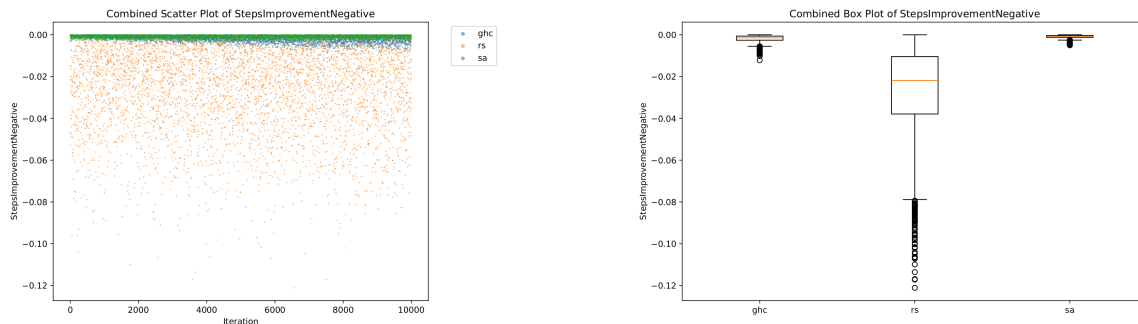


Figura 36: Gráfico de dispersión y *boxplot* de distribución de *delta* de empeoramiento en LABS (f_{18}) de dimensión 10000
Fuente: Elaboración propia

En otros problemas, donde se requiere una mayor exploración del espacio de búsqueda, gHC se ve superado por el resto de algoritmos, no logrando escapar de ciertos óptimos locales. En el caso de OneMax + Rugosidad r_3 (f_{10}), se observa que gHC es de los algoritmos con peor rendimiento. La curva de convergencia presente en la figura 37, se observa como gHC no logra mejorar en toda la ejecución del problema, y se mantiene constantemente en la misma calidad en la distribución de sus soluciones candidatas. Por otro lado, SA logra mejorar considerablemente en las últimas iteraciones, logrando obtener el mejor óptimo entre el resto de algoritmos.

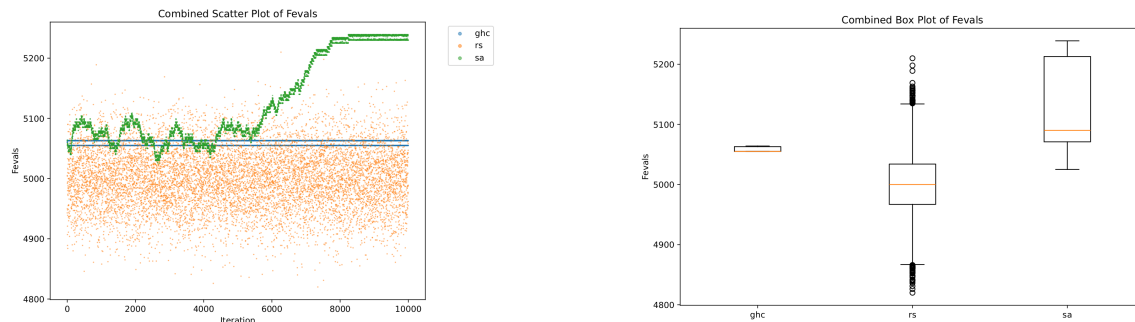


Figura 37: Gráfico de dispersión y *boxplot* de calidad de solución actual por iteración en OneMax + $r_3(f_{10})$ de dimensión 10000
Fuente: Elaboración propia

Al observar la diversificación del algoritmo, se puede apreciar que el algoritmo explora pocas estructuras de la solución, este fenómeno ocurre debido a que gHC solo cambia un *bit* de la solución por iteración, lo que genera que el algoritmo tenga un proceso lento de exploración, donde no se puede perturbar correctamente las soluciones candidatas.

Como se puede observar en la figura 38, donde se presentan las distancias recorridas que explora según la diversidad basada en distancia al centro. Para el caso de gHC, se exploran distancias cercanas a 5060 de distancia, mientras que para SA se exploran distancias entre 5070 y 5210. Por otro lado, al revisar la diversidad según la variación en las soluciones, se observa el mismo fenómeno que con la métrica anterior, gHC se estanca en su diversidad, no logra variar sus soluciones candidatas entre iteraciones, y se queda constantemente cercano al 0, mientras que SA logra variar considerablemente sus soluciones entre las iteraciones.

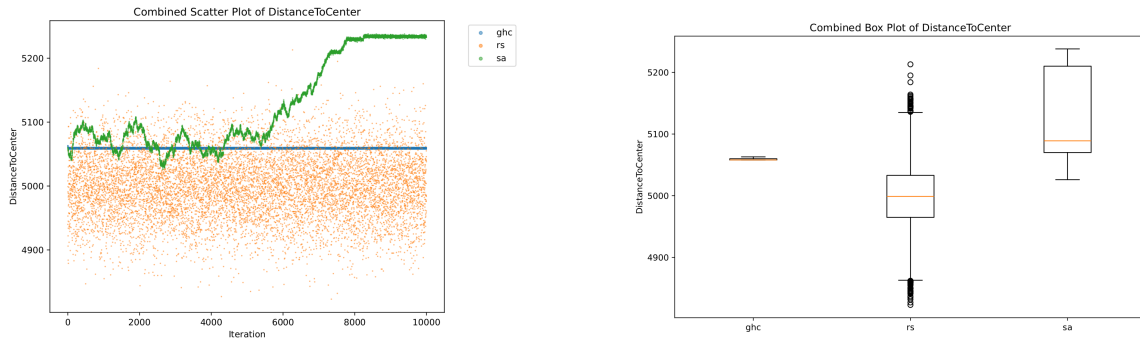


Figura 38: Gráfico de dispersión y *boxplot* de diversidad basada en distancia al centro por iteración en OneMax + r_3 (f_{10}) de dimensión 10000

Fuente: Elaboración propia

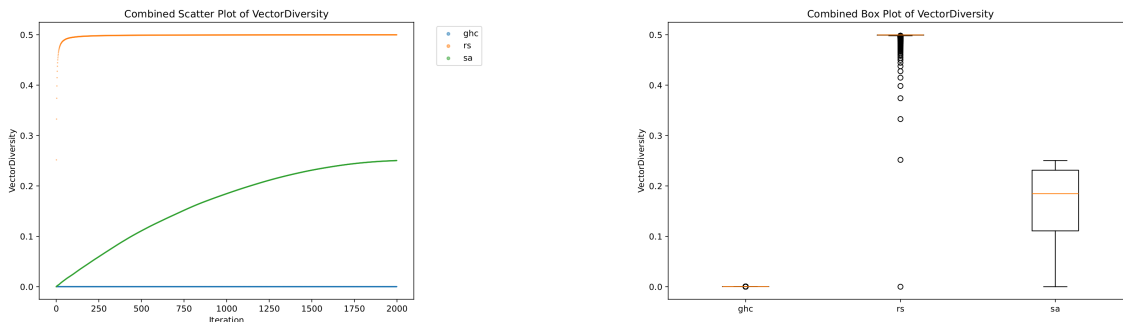


Figura 39: Gráfico de dispersión y *boxplot* de diversidad por iteración en OneMax + r_3 (f_{10}) de dimensión 10000

Fuente: Elaboración propia

4.5. Análisis del algoritmo RLS

Para Random Local Search (RLS o RS), se presentan resultados peores que el algoritmo anterior, ya que solo logra obtener resultados interesantes en el problema de *LeadingOne* + *Epistasis* (f_{14}). Según el autor de la *suite*, este problema tiene una alta dificultad debido a que el espacio de búsqueda contiene una gran cantidad de óptimos locales de baja calidad, lo que terminan funcionando como puntos trampas para algoritmos que traten de intensificar en esas zonas, es por esto que analizar las características de RLS en este problema resulta interesante.

En este problema, RLS logra superar al resto de algoritmos, obteniendo soluciones de calidad que triplican la calidad del resto de algoritmos. Al revisar la curva de convergencia de la figura 40, se observa como RLS no tiene una curva clara, sino más bien, va generando soluciones candidatas totalmente diferentes, donde la variación de calidad es muy alta entre iteraciones. Esto también se observa al revisar el ratio de convergencia entre iteraciones de la figura 41, donde RLS tiene una distribución de cambio entre 0 y 1, con varios *outliers* sobre 2, mientras que el resto de algoritmos no logran una distribución mejor, quedándose estancado cerca del 0.

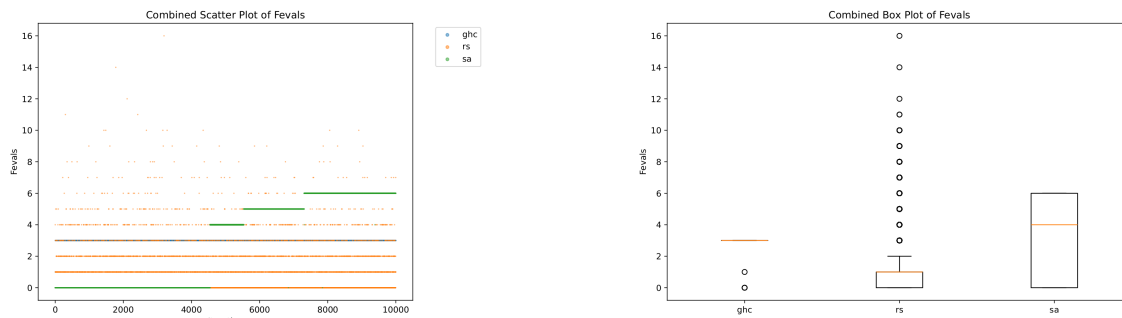


Figura 40: Gráfico de dispersión y *boxplot* de calidad de solución actual por iteración en *LeadingOne + Epistasis* (f_{14}) de dimensión 10000

Fuente: Elaboración propia

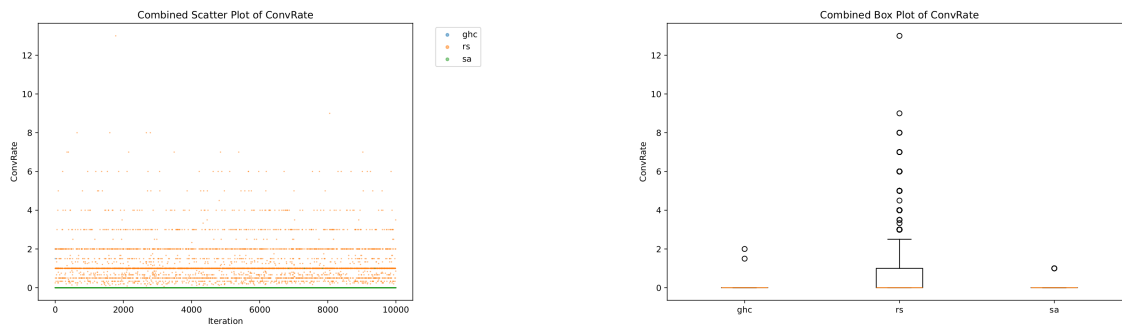


Figura 41: Gráfico de dispersión y *boxplot* de ratio de convergencia por iteración en *LeadingOne + Epistasis* (f_{14}) de dimensión 10000

Fuente: Elaboración propia

Este algoritmo se enfoca principalmente en lograr explorar al máximo el espacio de búsqueda, al generar soluciones aleatorias, alejadas constantemente entre cada iteración.

Esto se puede apreciar al observar la distancia de Hamming en cada una de sus variantes, en las figuras 42, 43 y 44, donde RLS tiene un valor constante de cambio del 50 %, el cual demuestra que el algoritmo está en constante proceso de diversificación, ya que no presenta etapas claras donde trate de intensificar en la búsqueda de óptimos en ciertas zonas y disminuya el porcentaje de cambios para lograr mejoras más claras.

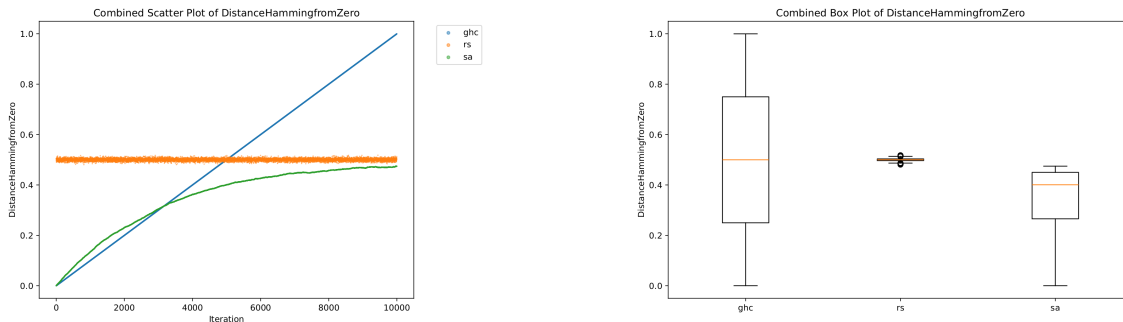


Figura 42: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la solución inicial por iteración en *LeadingOne* + *Epistasis* (f_{14}) de dimensión 10000
Fuente: Elaboración propia

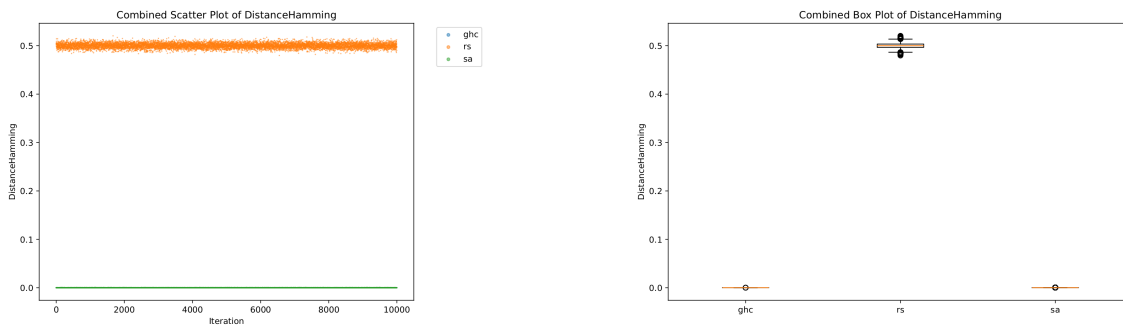


Figura 43: Gráfico de dispersión y *boxplot* de distancia de Hamming entre iteraciones en *LeadingOne* + *Epistasis* (f_{14}) de dimensión 10000
Fuente: Elaboración propia

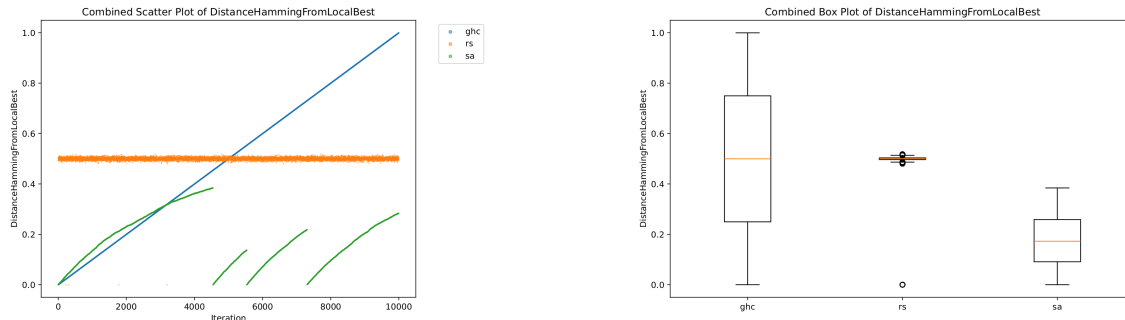


Figura 44: Gráfico de dispersión y *boxplot* de distancia de Hamming sobre la mejor solución actual por iteración en *LeadingOne + Epistasis* (f_{14}) de dimensión 10000
Fuente: Elaboración propia

Al revisar la métrica de diversidad, se puede observar una tendencia extraña con RLS, la cual es que un algoritmo que explora mucho y produce soluciones muy dispersas, termina generando que la métrica de diversidad se estabilice en un valor constante. En este caso, como se utiliza una representación binaria, cuando los valores se distribuyen equitativamente entre ceros y unos en cada dimensión del problema, la diversidad en cada dimensión será aproximadamente 0,5, como se aprecia en la figura 45. Al promediar sobre todas las dimensiones, el valor total de la diversidad será también cercano a 0,5, lo cual termina generando esta tendencia constante, que trata de suponer que el algoritmo no logra explorar de buena forma, lo cual es incorrecto. Esta idea no ocurre con el resto de algoritmos, ya que la tendencia va cambiando hacia el alza al avanzar las iteraciones, indicando que los algoritmos sí están explorando nuevas zonas del espacio de búsqueda.

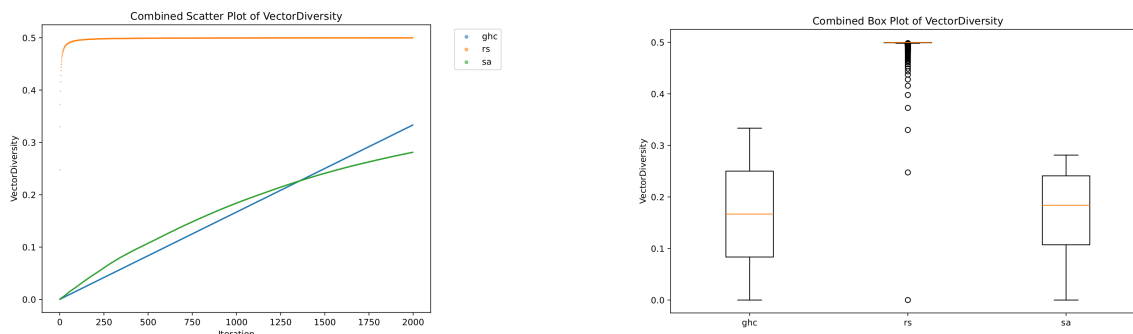


Figura 45: Gráfico de dispersión y *boxplot* de diversidad por iteración en *LeadingOne + Epistasis* (f_{14}) de dimensión 10000
Fuente: Elaboración propia

Para evitar estas falsas conclusiones, la métrica de diversidad basada en distancia al centro permite obtener una aproximación más clara respecto al nivel de diversificación que logra el algoritmo al explorar el espacio de búsqueda. En este caso, en la figura 46 se puede observar que RLS explora un rango mayor, aproximadamente entre 4965 y 5032, un resultado más amplio que el resto de algoritmos, que exploran un rango menor de distancias.

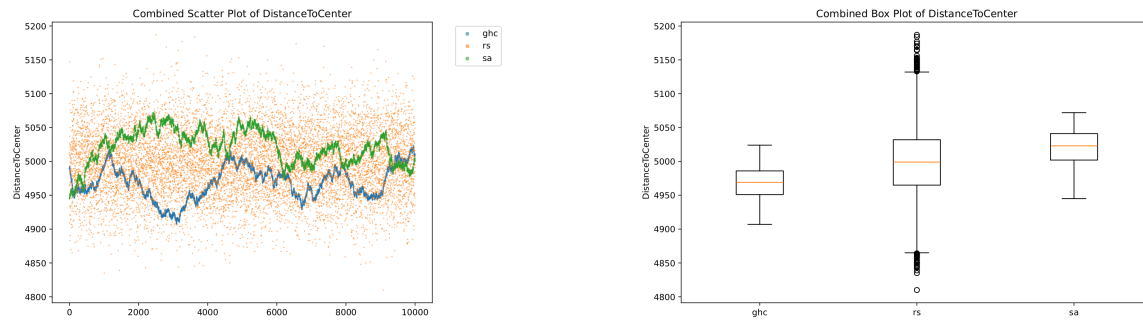


Figura 46: Gráfico de dispersión y *boxplot* de diversidad basada en distancia al centro en *LeadingOne + Epistasis* (f_{14}) de dimensión 10000

Fuente: Elaboración propia

4.6. Análisis del algoritmo SA

Para el caso de SA, este algoritmo logra una dominancia en diversos problemas en la dimensión 100, pero a medida que aumenta la dimensionalidad de los problemas, este algoritmo disminuye drásticamente su rendimiento. Para este análisis se escogió analizar los problemas *IsingRing* (f_{19}) y *N reinas* (f_{23}), el primero se escoge porque es uno de los pocos problemas donde SA logra mantener su rendimiento al aumentar la dimensionalidad, mientras que para el segundo, SA pierde rendimiento al aumentar la dimensionalidad de las instancias, por lo que resulta clave analizar las características que tiene este algoritmo en ambos problemas para determinar causas respecto a la caída de rendimiento en ciertos problemas.

Para *IsingRing*, se puede ver que en la figura 47 de curva de convergencia, todos los algoritmos se mantienen estables hasta la iteración 6000, donde SA comienza una etapa de intensificación final, que le permite mejorar considerablemente sus soluciones candidatas en las últimas iteraciones.

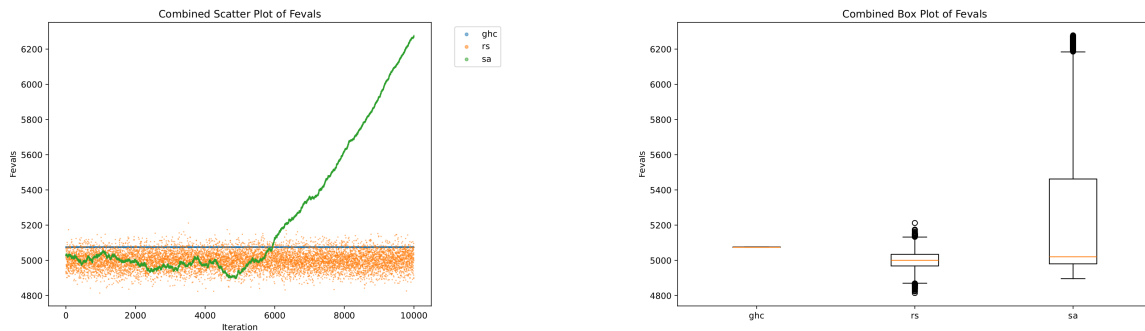


Figura 47: Gráfico de dispersión y *boxplot* de curva de convergencia basado en áreas por iteración en *IsingRing* (f_{19}) de dimensión 10000
Fuente: Elaboración propia

Al revisar la métrica de suma acumulada de proceso de intensificación y diversificación de la figura 48, se observa la misma tendencia de la curva de convergencia, donde SA se encuentra en un proceso equilibrado de intensificación y diversificación hasta la iteración 6000, y luego comienza una etapa clara de intensificación hasta finalizar la ejecución. En este caso, también se puede apreciar como RLS tiene una tendencia similar en los últimos algoritmos, pero más equilibrada que SA, donde luego de la iteración 4000, RLS va levemente intensificado en la búsqueda, mientras que gHC se estanca, no pudiendo lograr mejoras en toda la ejecución.

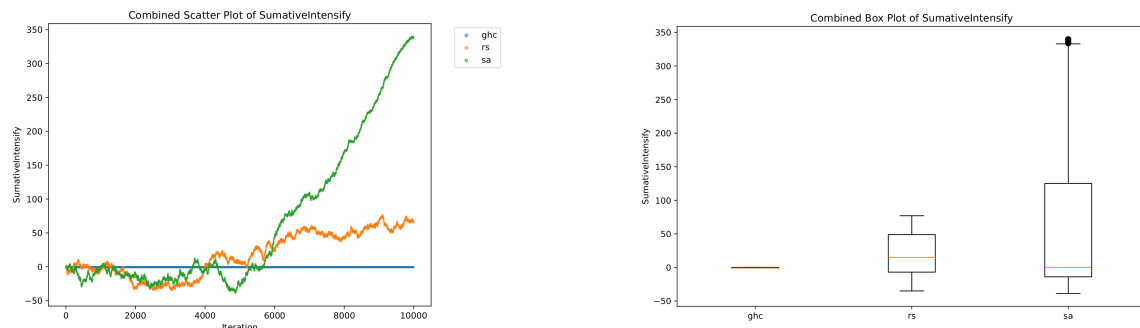


Figura 48: Gráfico de dispersión y *boxplot* de suma acumulada de proceso de intensificación y diversificación en *IsingRing* (f_{19}) de dimensión 10000
Fuente: Elaboración propia

En la métrica de diversidad basada en distancia al centro de la figura 49, SA tiene distri-

bución similar a RLS, solo que en este ultimo los datos están más dispersos en los valores extremos, pero no se observa una trayectoria clara como si la presenta SA, que explora un gran conjunto de distancias en las últimas 2000 iteraciones de la ejecución. Mientras que gHC nuevamente evidencia un estancamiento en la exploración de nuevas zonas del espacio de búsqueda.

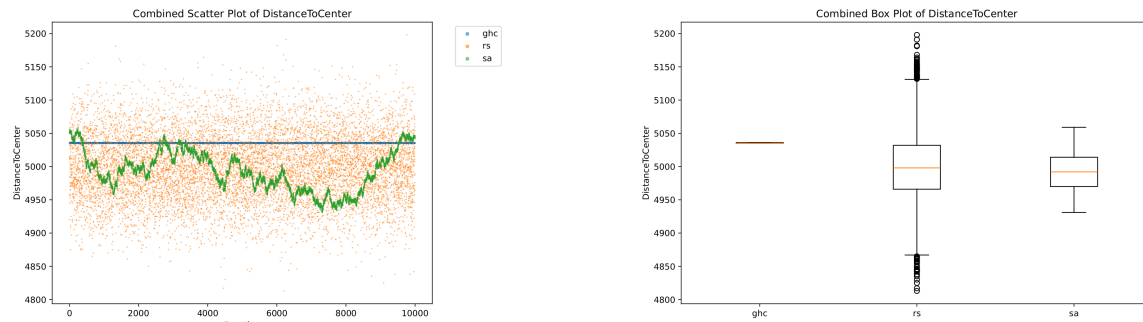


Figura 49: Gráfico de dispersión y *boxplot* de diversidad basada en distancia al centro en *IsingRing* (f_{19}) de dimensión 10000
Fuente: Elaboración propia

Por otro lado, SA presenta en su implementación una disminución de la probabilidad de cambio al aumentar dimensiones en los problemas. Este fenómeno ocurre en N reinas (f_{23}), donde se puede validar al revisar la distancia de Hamming en las distintas dimensiones. En este problema, SA logra buenos resultados en las primeras dimensiones, de 100 y 625 variables, pero al pasar a 5625 variables, el algoritmo no logra diversificar de buena forma y su rendimiento sufre una baja considerable en la calidad de las soluciones candidatas, como se ve en la figura 50.

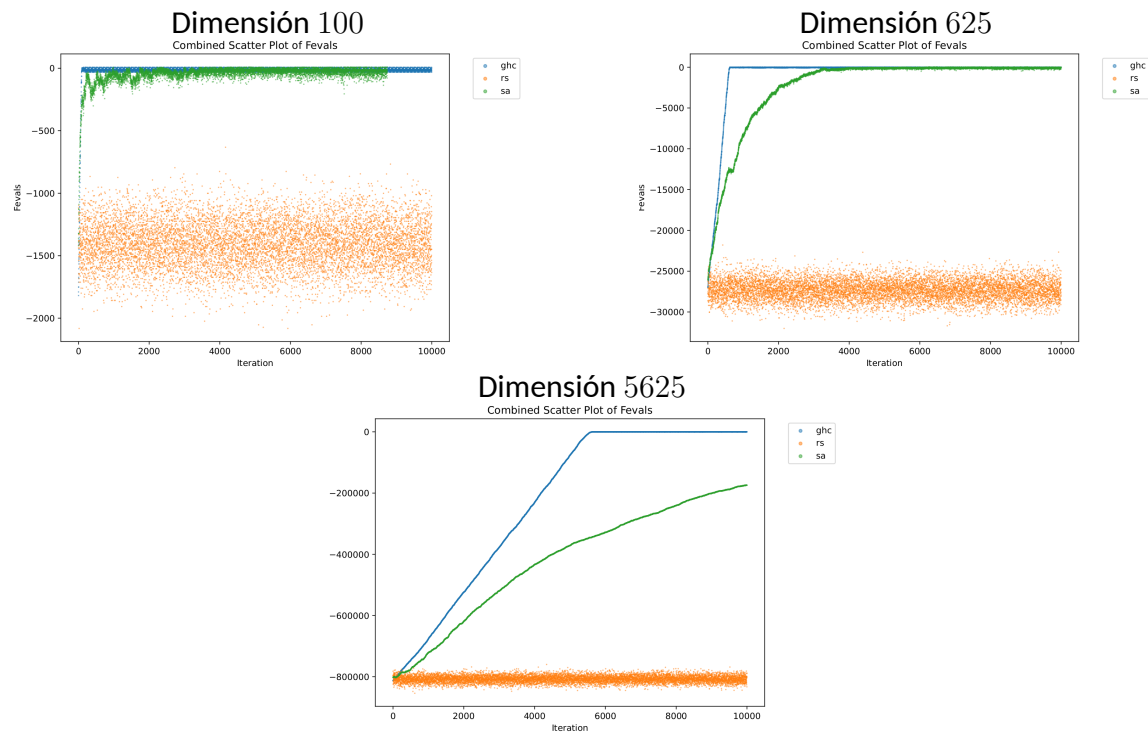


Figura 50: Gráfico de dispersión de curva de convergencia para N Reinas(f_{23}) de dimensión 100, 625 y 5625
Fuente: Elaboración propia

Al revisar la primera variante de distancia de Hamming, presente en la figura 51, se puede observar como SA en la primera instancia de dimensión 100 logra alcanzar hasta un 60 % de diferencia entre la solución inicial y las últimas soluciones candidatas, mientras que en la dimensión de 5625 apenas alcanza un 45 % de diferencia. Además, a medida que aumentan las dimensiones, el algoritmo necesita más iteraciones para aumentar el nivel de diferencia.

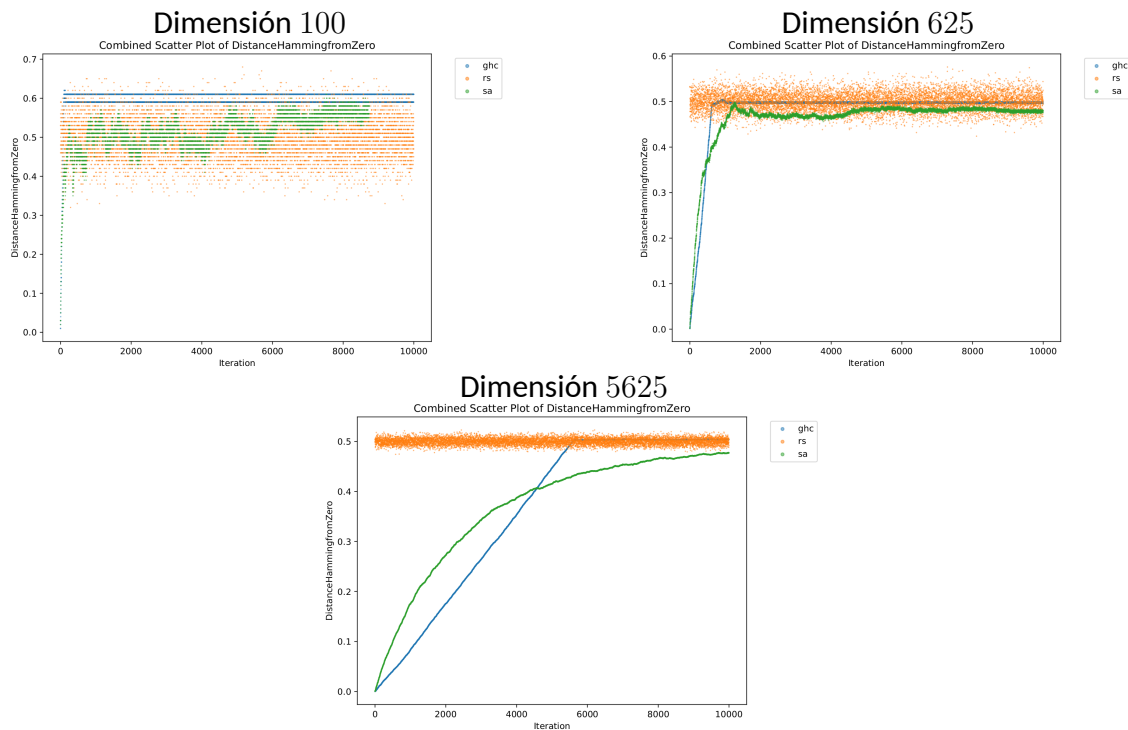


Figura 51: Gráfico de dispersión de distancia de Hamming sobre la solución inicial por iteración en N Reinas(f_{23}) de dimensión 100, 625 y 5625
Fuente: Elaboración propia

En la segunda variante entre iteraciones de la figura 52, se puede observar a medida que aumenta la dimensión del problema, la dispersión de los datos en el *boxplot* tiende a disminuir, lo que resulta en una distribución más concentrada o achatada alrededor del 0. En la primera dimensión de 100 variables, se alcanzan valores de 10 % de diferencia entre iteración, mientras que en la dimensión 5625 los valores apenas sobrepasan el 2 %. En esta métrica, se puede apreciar más claro que la implementación de SA, respecto a la probabilidad del movimiento *bit-flip* según la cantidad de variables del problema, afecta la exploración del espacio debido a la poca capacidad de cambio que logra el algoritmo en dimensiones altas, ya que la probabilidad de poder cambiar *bits* es menor.

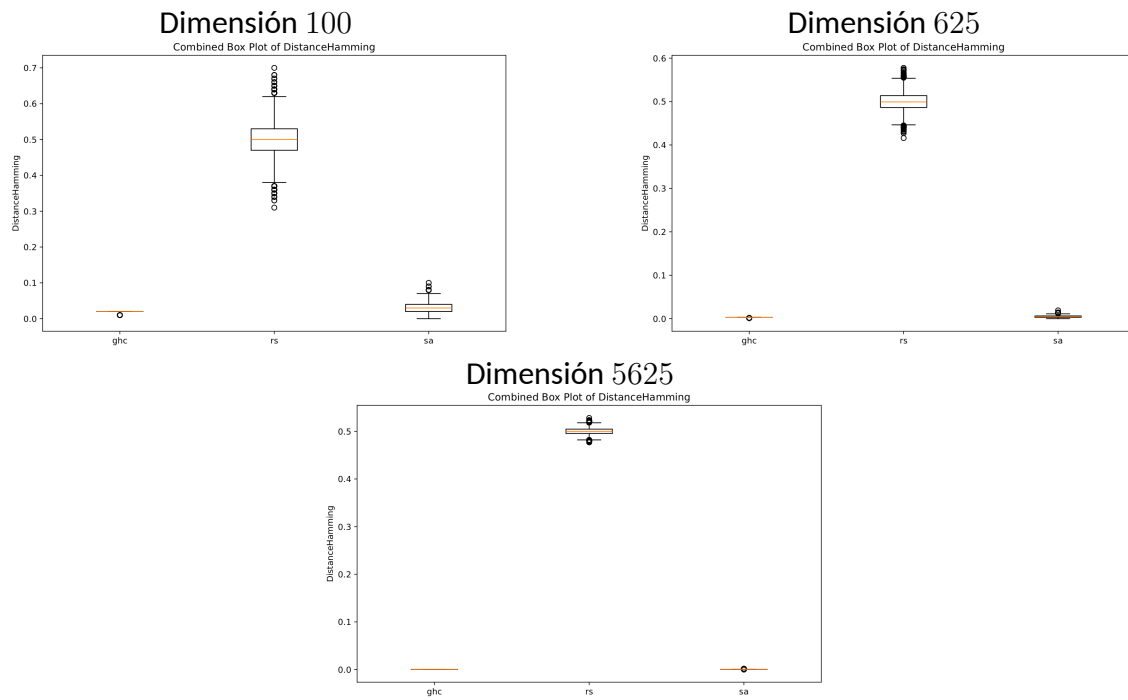


Figura 52: Gráfico de *boxplot* de distancia de Hamming entre iteración en N Reinas(f_{23}) de dimensión 100, 625 y 5625
Fuente: Elaboración propia

CAPÍTULO 5

CONCLUSIONES

A lo largo de este trabajo, se logra proponer, diseñar e implementar una librería de métricas de rendimiento, MHTMetLib, enfocada en algoritmos metaheurísticos trayectoriales. Este trabajo ha permitido cubrir una importante necesidad en el ámbito de los algoritmos de optimización, donde las herramientas existentes no proporcionaban suficientes detalles sobre el comportamiento de los algoritmos durante su ejecución. La integración con el *framework* IOHprofiler ha sido un paso fundamental para aprovechar las capacidades de experimentación y análisis de datos, permitiendo así una validación robusta de las métricas propuestas.

En el estudio del estado del arte, se logró recopilar un conjunto de métricas definidas por distintos autores, que fueron utilizadas para comparar y analizar propuestas realizadas en sus trabajos acerca de algoritmos metaheurísticos, la mayoría proveniente de la familia de algoritmos poblacionales.

Por otro lado, se diseñó un conjunto de métricas de rendimiento para algoritmos metaheurísticos trayectoriales, las cuales permiten capturar distintas características clave de su comportamiento, como la eficiencia en la exploración del espacio de soluciones, la capacidad de explotación de las soluciones cercanas, la tasa de convergencia hacia soluciones óptimas y la diversidad de las soluciones generadas a lo largo de las iteraciones. Estas métricas permiten realizar una evaluación más detallada y precisa del rendimiento de los algoritmos, facilitando su comparación y ajuste para mejorar su eficacia en distintos problemas de optimización.

Con el fin de proporcionar una herramienta flexible y adaptable, se implementó la librería de métricas de rendimiento en dos lenguajes de programación ampliamente utilizados en el ámbito de la optimización: *Python* y *C++*. La implementación en *Python* busca aprovechar su facilidad de uso y su ecosistema de bibliotecas científicas y de visualización, permitiendo visualizar los datos en distintos gráficos. Por otro lado, en *C++* se logró minimizar el tiempo que necesita la librería para calcular las distintas métricas, el cual en este caso fue de menos de 15 minutos para instancias con 10000 variables, permitiendo que la integración de este proceso dentro del flujo de los distintos algoritmos, no resulte un aumento en los tiempos de ejecución.

Al evaluar esta librería en el contexto de problemas binarios, incluidos en el *framework*

IOHprofiler, se obtienen resultados útiles para comparar un conjunto de algoritmos trayectoriales, los cuales tienen un mecanismo distinto para escoger las distintas soluciones candidatas en cada una de las iteraciones de ejecución.

Para la *suite* de problemas PBO, utilizando gHC, RS y SA como algoritmos de prueba, se obtiene que gHC es el algoritmo con mejor rendimiento, pero que a su vez tiene una gran cantidad de punto de mejora que se encuentran a la hora de revisar los resultados de sus métricas de rendimiento en los distintos problemas, al igual que RS y SA. Entre los resultados más destacados, se ha comprobado la capacidad de las métricas para capturar información detallada sobre la convergencia y la diversidad a lo largo de las iteraciones, lo que facilita una comprensión más profunda del rendimiento de los algoritmos. Para este caso, los algoritmos tienen diseños simples que no constan varios pasos, pero se logran ver reflejados esas tomas de decisiones al revisar las distintas métricas evaluadas.

Para el caso de gHC, es un algoritmo que no diversifica de buena forma, lo cual le puede ayudar a resolver problemas donde el espacio de búsqueda tiene suavidad o estabilidad que permite intensificar de forma sencilla hacia los óptimos en problemas como *OneMax* o *LeadingOnes*, mientras que en problemas como *IsingRing* el algoritmo no logra obtener buenas soluciones candidatas, debido a que se queda estacando en óptimos locales del problema de forma constante.

Para RLS, se ve una idea completamente distinta a gHC, donde se trata de diversificar de forma aleatoria en el espacio de búsqueda, lo cual genera que sea el algoritmo con mayor diversidad, distancia de cambio y etapas de diversificación. Aunque esto obtiene buenos resultados en problemas como *LeadingOnes + Epistasis*, no es una estrategia precisa y útil a la hora de resolver el resto de problemas de la *suite*, debido a su poca capacidad de convergencia y claridad para entrar en procesos de intensificación dentro de la ejecución.

Por último, en el algoritmo SA, se observa un equilibrio mejor entre las etapas de intensificación y diversificación, obtiene resultados sólidos en problemas como *IsingRing* y *N* Reinas en las primeras dimensiones. Pero al aumentar la dimensionalidad de los problemas, se observa un punto de diseño que empeora el porcentaje de cambios entre las distintas soluciones candidatas, lo que genera que el proceso de diversificación sea más lento, disminuyendo la eficiencia del algoritmo, al necesitar mayor cantidad de iteraciones para lograr mejorar sus resultados en sus ejecuciones.

Con esta propuesta, se puede avanzar en posibles mejoras que logren ampliar el alcance que este tiene en el área del diseño de algoritmos de optimización. Para este trabajo, solo se abarca el uso de algoritmos trayectoriales, pero se puede seguir ampliando la experi-

mentación con el resto de familias, como lo son los algoritmos poblacionales, permitiendo extender su uso al agregar nuevas métricas que permitan capturar la información de estas nuevas familias.

Además, es importante evaluar nuevas métricas para el uso en problemas multi-objetivos, es decir, problemas que tiene más de una función objetivo a resolver, lo cual permite modelar una mayor cantidad de problemas de la vida real. En la propuesta actual, solo se alcanza a analizar su uso en problemas mono-objetivo, debido a que se necesita una extensión mayor de este trabajo para lograr abarcar ambos tipos de problemas en un solo trabajo.

Por otro lado, se puede continuar con la idea que se comentó en el capítulo de definición del problema, el cual consiste en utilizar estas métricas de rendimiento para poder sintonizar los algoritmos, tanto ya sean parámetros iniciales como de control dinámico, permitiendo que los algoritmos puedan utilizar esta información para poder ajustar parámetros, con el objetivo que el algoritmo responda a las características cambiantes del problema, aumentando su eficiencia y efectividad en la resolución del mismo.

Finalmente, se debe incluir nuevos métodos y variables a capturar en la ejecución de algoritmos, esto debido a que actualmente la cantidad de variables de entrada que tiene la librería, puede resultar escasa para lograr calcular un nuevo conjunto de métricas. Mientras mayor información se logre capturar de la toma de decisiones del algoritmo, mejores métricas se pueden construir para lograr cuantificar características principales de los distintos algoritmos.

ANEXOS

func	gHC			RLS			SA			
	avg	std	error-opt	avg	std	error-opt	avg	std	error-opt	opt
f1	100.0	0.0	0.00 %	69.0	1.0	30.00 %	99.6	0.9	0.00 %	100
f2	100.0	0.0	0.00 %	14.0	2.1	83.00 %	75.8	11.5	9.00 %	100
f3	5050.0	0.0	0.00 %	3657.2	137.6	23.64 %	5050.0	0.0	0.00 %	5050
f4	50.0	0.0	0.00 %	37.6	0.5	24.00 %	50.0	0.0	0.00 %	50
f5	90.0	0.0	0.00 %	62.6	1.7	27.78 %	89.8	0.4	0.00 %	90
f6	33.0	0.0	0.00 %	26.8	0.4	18.18 %	33.0	0.0	0.00 %	33
f7	78.4	0.9	21.00 %	69.8	0.8	29.00 %	85.0	2.7	12.00 %	100
f8	43.2	1.5	11.76 %	35.2	1.1	27.45 %	46.6	1.3	3.92 %	51
f9	50.2	3.0	45.00 %	69.6	0.9	29.00 %	96.6	0.9	3.00 %	100
f10	51.0	5.7	41.00 %	70.0	2.2	26.00 %	73.0	2.2	26.00 %	100
f11	50.0	0.0	0.00 %	12.4	1.1	72.00 %	50.0	0.0	0.00 %	50
f12	90.0	0.0	0.00 %	12.2	1.3	84.44 %	79.2	9.7	0.00 %	90
f13	33.0	0.0	0.00 %	14.2	2.3	48.48 %	33.0	0.0	0.00 %	33
f14	2.8	1.8	94.00 %	14.6	1.8	83.00 %	14.0	3.0	82.00 %	100
f15	51.0	0.0	0.00 %	7.6	0.5	84.31 %	26.2	7.2	27.45 %	51
f16	100.0	0.0	0.00 %	14.2	1.1	85.00 %	65.8	11.3	23.00 %	100
f17	4.0	0.0	96.00 %	14.0	0.0	86.00 %	17.8	4.1	76.00 %	100
f18	3.9	0.3	-	2.2	0.1	-	2.3	0.1	-	-
f19	53.2	2.7	44.00 %	68.4	0.9	30.00 %	95.2	2.7	2.00 %	100
f20	194.0	13.4	0.00 %	127.2	1.8	36.00 %	195.2	10.7	0.00 %	200
f21	236.8	117.3	0.00 %	186.4	3.0	36.67 %	291.6	18.8	0.00 %	300
f22	45.6	1.7	4.00 %	-1467.4	199.3	2634.00 %	46.6	2.4	0.00 %	50
f23	7.4	0.5	20.00 %	-709.0	57.5	6420.00 %	9.2	0.4	0.00 %	10
f24	16.3	0.2	17.00 %	10.9	0.3	43.00 %	14.6	0.8	20.00 %	20
f25	-0.3	0.0	-	-0.4	0.0	-	-0.4	0.0	-	-

Tabla 4: Tabla de resultados para problemas de *suite* PBO de dimensión 100

Fuente: Elaboración propia

func	gHC			RLS			SA			opt
	avg	std	error-opt	avg	std	error-opt	avg	std	error-opt	
f1	625.0	0.0	0.00 %	360.2	4.4	41.44 %	582.6	5.6	5.28 %	625
f2	625.0	0.0	0.00 %	13.4	0.9	97.76 %	17.4	5.5	96.00 %	625
f3	195625.0	0.0	0.00 %	114548.2	1133.9	40.80 %	195607.0	28.6	0.00 %	195625
f4	312.0	0.0	0.00 %	189.8	3.8	38.14 %	297.8	1.9	3.85 %	312
f5	562.0	0.0	0.00 %	325.6	2.3	41.64 %	529.4	5.5	4.45 %	562
f6	208.0	0.0	0.00 %	132.6	1.3	35.58 %	198.2	3.0	3.37 %	208
f7	489.0	6.0	20.32 %	360.8	3.8	41.28 %	499.4	7.8	18.24 %	625
f8	264.2	5.5	14.01 %	181.8	1.3	41.72 %	258.2	6.8	14.33 %	314
f9	309.2	6.9	49.44 %	359.2	6.1	40.80 %	525.6	4.8	14.56 %	625
f10	317.0	15.2	45.76 %	360.8	4.6	40.96 %	393.0	11.4	35.36 %	625
f11	312.0	0.0	0.00 %	13.0	2.0	94.87 %	15.6	8.6	90.38 %	312
f12	562.0	0.0	0.00 %	15.2	3.2	96.62 %	18.6	4.5	95.55 %	562
f13	208.0	0.0	0.00 %	13.2	0.8	93.27 %	26.0	13.7	75.96 %	208
f14	3.4	2.1	98.88 %	14.6	2.1	97.12 %	9.0	3.5	97.60 %	625
f15	63.6	13.0	73.57 %	8.0	1.0	97.13 %	6.6	2.9	96.50 %	314
f16	68.0	10.5	87.20 %	14.6	1.9	97.28 %	10.2	3.6	97.76 %	625
f17	4.0	0.0	99.36 %	15.0	2.2	96.96 %	9.8	2.4	97.76 %	625
f18	3.9	0.3	-	1.4	0.0	-	1.4	0.0	-	-
f19	313.8	6.7	48.32 %	361.8	2.3	41.60 %	538.2	6.4	12.80 %	625
f20	1216.0	46.6	0.00 %	695.2	4.1	44.00 %	1071.6	9.5	13.44 %	1250
f21	1797.0	71.2	0.00 %	1022.6	5.9	45.01 %	1631.8	14.5	12.37 %	1875
f22	275.2	1.8	11.54 %	-134765.4	4358.3	41014 %	258.4	5.5	14.42 %	312
f23	18.2	0.4	24.00 %	-22437.2	487.6	87276 %	22.2	1.3	4.00 %	25
f24	101.5	0.7	17.92 %	52.3	0.3	57.92 %	70.2	0.6	43.36 %	125
f25	-0.3	0.0	-	-0.5	0.0	-	-0.5	0.0	-	-

Tabla 5: Tabla de resultados para problemas de suite PBO de dimensión 625

Fuente: Elaboración propia

func	gHC			RLS			SA			opt
	avg	std	error-opt	avg	std	error-opt	avg	std	error-opt	
f1	5625.0	0.0	0.00 %	2960.2	18.6	46.90 %	3788.6	16.3	32.18 %	5625
f2	5625.0	0.0	0.00 %	14.4	2.5	99.70 %	3.8	1.8	99.89 %	5625
f3	15823125.0	0.0	0.00 %	8375630.4	41536.4	46.64 %	14381591.4	48184.9	8.63 %	15823125
f4	2812.0	0.0	0.00 %	1507.2	5.0	46.16 %	1943.4	25.6	29.87 %	2812
f5	5062.0	0.0	0.00 %	2673.2	13.8	46.74 %	3449.0	5.9	31.71 %	5062
f6	1875.0	0.0	0.00 %	1018.6	2.9	45.49 %	1370.8	22.3	25.44 %	1875
f7	4388.2	10.5	21.76 %	2955.4	6.8	47.32 %	3844.0	25.3	31.06 %	5625
f8	2340.6	11.8	16.38 %	1479.4	5.5	47.12 %	1750.2	17.1	37.17 %	2814
f9	2791.6	50.5	49.51 %	2964.2	12.9	47.06 %	3531.6	114.0	33.62 %	5625
f10	2829.0	25.0	49.08 %	2951.6	9.8	47.24 %	3048.0	29.5	45.08 %	5625
f11	2812.0	0.0	0.00 %	14.8	3.0	99.32 %	4.8	2.6	99.72 %	2812
f12	5062.0	0.0	0.00 %	12.4	1.1	99.72 %	3.0	0.7	99.92 %	5062
f13	1875.0	0.0	0.00 %	12.8	1.3	99.20 %	3.6	3.3	99.52 %	1875
f14	2.2	0.4	99.95 %	15.6	4.2	99.59 %	2.4	0.5	99.95 %	5625
f15	8.2	2.6	99.61 %	7.8	0.8	99.68 %	3.4	1.7	99.79 %	2814
f16	5.2	1.1	99.89 %	14.4	1.7	99.72 %	6.6	8.9	99.61 %	5625
f17	4.0	0.0	99.93 %	14.8	1.8	99.68 %	6.0	2.7	99.84 %	5625
f18	3.2	0.1	-	1.1	0.0	-	1.1	0.0	-	-
f19	2833.0	26.3	49.07 %	2950.2	12.0	47.22 %	4218.6	916.1	-4.12 %	5625
f20	9092.0	40.1	18.68 %	5831.6	13.1	47.96 %	7259.2	26.4	35.15 %	11250
f21	14691.8	129.3	11.83 %	8676.6	21.0	48.47 %	10944.6	107.1	34.20 %	16875
f22	2335.2	10.8	16.68 %	-14120316.2	106958.8	497605 %	-505695.0	59853.2	15243 %	2812
f23	51.6	0.5	30.67 %	-764925.2	3018.6	1013263 %	-183209.8	5720.1	231684 %	75
f24	913.9	1.8	18.58 %	410.4	1.8	63.29 %	498.4	5.3	54.95 %	1125
f25	-0.3	0.0	-	-0.5	0.0	-	-0.5	0.0	-	-

Tabla 6: Tabla de resultados para problemas de suite PBO de dimensión 5625

Fuente: Elaboración propia

func	gHC			RLS			SA			opt
	avg	std	error-opt	avg	std	error-opt	avg	std	error-opt	
f1	9999.6	0.5	0.00 %	5191.2	13.1	47.88 %	6130.2	67.0	38.03 %	10000
f2	9999.6	0.5	0.00 %	13.8	2.8	99.82 %	4.0	1.2	99.95 %	10000
f3	50001000.0	5477.2	0.00 %	26066676.4	26512.8	47.82 %	41142232.0	88997.0	17.52 %	50005000
f4	5000.0	0.0	0.00 %	2641.2	15.5	46.68 %	3131.0	37.7	36.10 %	5000
f5	9000.0	0.0	0.00 %	4686.0	13.9	47.78 %	5554.6	25.1	37.93 %	9000
f6	3333.0	0.0	0.00 %	1782.0	5.1	46.29 %	2190.8	28.6	33.15 %	3333
f7	7804.8	14.9	21.82 %	5183.2	11.1	48.04 %	6401.4	55.4	35.33 %	10000
f8	4158.8	25.4	16.00 %	2594.2	8.0	47.89 %	2892.2	29.4	41.39 %	5001
f9	4987.0	68.8	49.25 %	5195.8	25.3	47.60 %	5763.8	36.9	41.77 %	10000
f10	4988.0	72.1	49.36 %	5199.6	9.1	47.90 %	5196.0	34.7	47.61 %	10000
f11	5000.0	0.0	0.00 %	15.6	2.6	99.64 %	2.0	1.4	99.92 %	5000
f12	9000.0	0.0	0.00 %	13.0	1.0	99.84 %	3.0	2.5	99.92 %	9000
f13	3333.0	0.0	0.00 %	12.8	1.1	99.58 %	4.8	2.9	99.79 %	3333
f14	2.4	0.5	99.97 %	12.8	1.8	99.84 %	3.4	1.7	99.94 %	10000
f15	2.4	1.3	99.92 %	7.8	0.4	99.84 %	2.2	1.3	99.92 %	5001
f16	3.0	2.0	99.95 %	14.0	1.0	99.85 %	4.6	3.3	99.91 %	10000
f17	4.0	0.0	99.96 %	16.0	2.7	99.81 %	4.0	0.0	99.96 %	10000
f18	2.8	0.0	-	1.1	0.0	-	1.1	0.0	-	-
f19	5017.8	43.5	49.24 %	5194.4	15.1	47.88 %	6251.2	26.0	37.22 %	10000
f20	14010.4	126.5	29.02 %	10271.6	15.3	48.54 %	12043.6	90.7	39.10 %	20000
f21	24074.4	62.5	19.54 %	15333.2	19.4	48.80 %	17915.2	88.2	39.82 %	30000
f22	2194.6	31.3	55.34 %	-45835181.8	295465.2	907004 %	-6205163.8	178228.7	120443 %	5000
f23	1.2	0.4	98.00 %	-1859052.0	7070.5	1851611 %	-819738.8	17088.8	807129 %	100
f24	1625.8	2.8	18.56 %	717.4	3.4	63.96 %	819.1	8.4	58.56 %	2000
f25	-0.3	0.0	-	-0.5	0.0	-	-0.5	0.0	-	-

Tabla 7: Tabla de resultados para problemas de *suite* PBO de dimensión 10000

Fuente: Elaboración propia

REFERENCIAS BIBLIOGRÁFICAS

- [Abdoun *et al.*,] Abdoun, O., Tajani, C., y Abouchabaka, J. Using multi-agents systems to resolve an np-complete problem: Travelling salesman problem. *Algorithms*, 1:3.
- [Abualigah *et al.*, 2022] Abualigah, L., Elaziz, M. A., Sumari, P., Khasawneh, A. M., Alshinwan, M., Mirjalili, S., Shehab, M., Abuaddous, H. Y., y Gandomi, A. H. (2022). Black hole algorithm: A comprehensive survey. *Applied Intelligence*, 52(10):11892–11915.
- [Barr *et al.*, 1995] Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G., y Stewart, W. R. (1995). Designing and reporting on computational experiments with heuristic methods. *Journal of heuristics*, 1:9–32.
- [Cheng *et al.*, 2014] Cheng, S., Shi, Y., Qin, Q., Zhang, Q., y Bai, R. (2014). Population diversity maintenance in brain storm optimization algorithm. *Journal of Artificial Intelligence and Soft Computing Research (JAISCR)*, 4:83–97.
- [Crescenzi *et al.*, 1995] Crescenzi, P., Kann, V., y Halldórsson, M. (1995). A compendium of np optimization problems.
- [Davis, 1991] Davis, L. (1991). Bit-climbing, representational bias, and test suite design. En *Proc. 4th Int'l Conf. on Genetic Algorithms*, pp. 18–23. Morgan Kaufmann.
- [Dhivyaprabha *et al.*, 2018] Dhivyaprabha, T., Subashini, P., y Krishnaveni, M. (2018). Synergistic fibroblast optimization: a novel nature-inspired computing algorithm. *Frontiers of Information Technology & Electronic Engineering*, 19:815–833.
- [Doerr *et al.*, 2018] Doerr, C., Wang, H., Ye, F., van Rijn, S., y Bäck, T. (2018). Iohprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv e-prints*:1810.05281.
- [Doerr *et al.*, 2020] Doerr, C., Ye, F., Horesh, N., Wang, H., Shir, O. M., y Bäck, T. (2020). Benchmarking discrete optimization heuristics with iohprofiler. *Applied Soft Computing*, 88:106027.
- [Dorigo *et al.*, 2006] Dorigo, M., Birattari, M., y Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39.
- [Giel y Wegener, 2003] Giel, O. y Wegener, I. (2003). Evolutionary algorithms and the maximum matching problem. En *Annual Symposium on Theoretical Aspects of Computer Science*, pp. 415–426. Springer.

- [Glover y Laguna, 1998] Glover, F. y Laguna, M. (1998). *Tabu search*. Springer.
- [Halim e Ismail, 2019] Halim, A. H. e Ismail, I. (2019). Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem. *Archives of Computational Methods in Engineering*, 26:367–380.
- [He, 2016] He, J. (2016). An analytic expression of relative approximation error for a class of evolutionary algorithms. En *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 4366–4373. IEEE.
- [Huang et al., 2019] Huang, C., Li, Y., y Yao, X. (2019). A survey of automatic parameter tuning methods for metaheuristics. *IEEE transactions on evolutionary computation*, 24(2):201–216.
- [Kennedy y Eberhart, 1995] Kennedy, J. y Eberhart, R. (1995). Particle swarm optimization. En *Proceedings of ICNN'95-international conference on neural networks*, volumen 4, pp. 1942–1948. IEEE.
- [Lambora et al., 2019] Lambora, A., Gupta, K., y Chopra, K. (2019). Genetic algorithm—a literature review. En *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pp. 380–384. IEEE.
- [Neri y Cotta, 2012] Neri, F. y Cotta, C. (2012). Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14.
- [Osaba et al., 2021] Osaba, E., Villar-Rodriguez, E., Del Ser, J., Nebro, A. J., Molina, D., LaTorre, A., Suganthan, P. N., Coello, C. A. C., y Herrera, F. (2021). A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems. *Swarm and Evolutionary Computation*, 64:100888.
- [Selman y Gomes, 2006] Selman, B. y Gomes, C. P. (2006). Hill-climbing search. *Encyclopedia of cognitive science*, 81:82.
- [Stegherr et al., 2022] Stegherr, H., Heider, M., y Hähner, J. (2022). Classifying metaheuristics: Towards a unified multi-level classification system. *Natural Computing*, 21(2):155–171.
- [Van Laarhoven et al., 1987] Van Laarhoven, P. J., Aarts, E. H., van Laarhoven, P. J., y Aarts, E. H. (1987). *Simulated annealing*. Springer.
- [Wang et al., 2019] Wang, G.-G., Gandomi, A. H., Alavi, A. H., y Gong, D. (2019). A comprehensive review of krill herd algorithm: variants, hybrids and applications. *Artificial Intelligence Review*, 51:119–148.

- [Wang *et al.*, 2022] Wang, H., Vermetten, D., Ye, F., Doerr, C., y Bäck, T. (2022). Iohanalyzer: Detailed performance analyses for iterative optimization heuristics. *ACM Trans. Evol. Learn. Optim.*, 2(1).
- [Yang, 2009] Yang, X.-S. (2009). Harmony search as a metaheuristic algorithm. *Music-inspired harmony search algorithm: theory and applications*, pp. 1–14.
- [Yang, 2010] Yang, X.-S. (2010). *Nature-inspired metaheuristic algorithms*. Luniver press.
- [Žerovnik, 2015] Žerovnik, J. (2015). Heuristics for np-hard optimization problems-simpler is better!? *Logistics, Supply Chain, Sustainability and Global Challenges*, 6(1):1–10.