Trabajo de Fin de Máster

Máster en Modelización e Investigación Matemática, Estadística y Computación

TSP y Orthogonal Arrays: Una Posible Conexión

TSP & Orthogonal Arrays: A Possible Connection

Daniel Nuez Doreste

La Laguna, 5 de julio de 2024

Agradecimientos

A papá, mamá y hermanita, quiénes, cuando apenas se divisaba una atisbo de luz, fueron faro y apoyo incondicional para no perder el rumbo y seguir adelante.

A mis abuelos, quiénes siempre velan por mí y se alegran por todos los logros que voy consiguiendo, porque son el motor de mi vida.

A mi grupo de amigos; los de la infancia, quiénes son el reflejo de todo lo que conseguimos juntos; y los que he descubierto este en año en Tenerife, quiénes han sido mi hogar durante toda esta andadura.

En general, me gustaría expresar mi más profundo agradecimiento a todos aquellos que han hecho posible este trabajo y que, de una forma u otra, me han aportado nuevas experiencias y sentimientos enriquecedores, útiles para seguir mejorando día a día.

Resumen

El propósito de este trabajo es estudiar, programar y analizar métodos para la creación de los *orthogonal arrays* (matrices ortogonales), así como exponer una posible relación existente entre los métodos de obtención de dichos elementos con aquellos destinados a la resolución del *Travelling Salesman Problem* (Problema del Viajante de Comercio).

No obstante, se indagará en muchas de las propiedades que guardan dichos elementos, comprendiendo su naturaleza, la importancia que tienen en diversos ámbitos de trabajo y el origen de los mismos. Asimismo, se expondrán distintos algoritmos que buscarán poder hallar *orthogonal arrays* de forma eficiente y novedosa, donde se estudiará la capacidad de efectividad de cada uno de ellos y las limitaciones que presentan.

Este trabajo tiene dos vertientes claramente diferencias; en una primera instancia, se caracteriza por ser un apoyo teórico de ambos temas mencionados anteriormente (OA y TSP), exponiendo nociones necesarias y fundamentales para poder comprender lo que resta de documento; por otro lado, se incluye la parte práctica, donde se formularán métodos de resolución de los *orthogonal arrays*, justificando los diversos procedimientos, extrayendo conclusiones de cada uno de ellos y generando comparaciones con el fin de determinar en qué situaciones es óptimo optar por uno o por otro.

Todos los resultados han sido programados y obtenidos mediante el lenguaje de programación *Phyton*, suponiendo un apoyo indiscutible para la justificación de las conclusiones expuestas en las últimas páginas de este documento.

Palabras Clave: *Orthogonal Arrays, Matrices Ortogonales, TSP, Optimización Lineal, Métodos Heurísticos, Métodos Meta-heurísticos, Phyton, Algoritmos Genéticos, Gurobi*

Abstract

The purpose of this work is to study, program, and analyze methods for the creation of *orthogonal arrays*, as well as to explore a possible relationship between the methods for obtaining these elements and those aimed at solving the *Travelling Salesman Problem* (TSP).

Nevertheless many of the properties of these elements will be investigated, including understanding their nature, their importance in various fields of work, and their origin. Additionally, various algorithms will be presented that aim to find *orthogonal arrays* efficiently and innovatively, with an examination of the effectiveness and limitations of each algorithm.

This work has two clearly differentiated aspects. Initially, it serves as a theoretical support for both topics mentioned above (OA and TSP), presenting necessary and fundamental notions to understand the rest of the document. Subsequently, it includes the practical part, formulating methods for solving *orthogonal arrays*, justifying the various procedures, drawing conclusions from each, and making comparisons to determine in which situations it is optimal to choose one method over another.

All results have been programmed and obtained using the Python programming language, providing undeniable support for the justification of the conclusions presented in the final pages of this document.

Keywords: Orthogonal Arrays, TSP, Linear Optimization, Heuristic Methods, Metaheuristic Methods, Python, Genetic Algorithms, Gurobi

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En el vasto campo de la investigación operativa y la estadística, el Problema del Viajante (TSP) y los Orthogonal Arrays (OA) emergen como dos áreas de estudio de gran importancia y relevancia. Por un lado, se presenta un problema clásico en la teoría de grafos y la optimización combinatoria, que plantea el desafío de encontrar el trayecto más corto que permita visitar un conjunto de ciudades exactamente una vez cada una y regresar al punto de partida (TSP). Por otro lado, se exponen estructuras matemáticas, fundamentales en el diseño de experimentos, desde un enfoque sistemático para explorar y optimizar la relación entre múltiples variables (Orthogonal Arrays). A primera vista, estas dos áreas pueden parecer divergentes en sus aplicaciones y enfoques, sin embargo, existe un potencial fascinante para explorar posibles conexiones y sinergias entre ellas.

El propósito de este trabajo es investigar la relación entre el Problema del Viajante y los Orthogonal Arrays, explorando cómo los principios y técnicas del TSP pueden ser aplicados para abordar y analizar la generación de orthogonal arrays de manera novedosa y efectiva. A través de esta investigación, se busca, no solo ampliar la comprensión teórica de ambos problemas, sino también identificar posibles aplicaciones prácticas que puedan beneficiar campos tan diversos como la logística, la planificación de rutas, la optimización de procesos, entre otras. En esta introducción, se sientan las bases teóricas necesarias para comprender tanto el TSP como los OA, indicando los desafíos y oportunidades que presenta su estudio conjunto.

En las siguientes secciones, se explorará en detalle los fundamentos teóricos del TSP y los OA. Posteriormente, se presentará la metodología de investigación y los resultados obtenidos, antes de discutir las implicaciones y conclusiones del estudio.

1.1. Contextualización del Problema del Viajante y de los Orthogonal Arrays

El Problema del Viajante **(TSP)** se posiciona como un desafío emblemático en el campo de la optimización combinatoria y la teoría de grafos. Surgiendo en diversas disciplinas, desde la logística hasta la planificación de rutas y la ingeniería de redes, el TSP plantea la pregunta fundamental de encontrar la ruta más corta que permita a un viajero visitar un conjunto de ciudades exactamente una vez antes de regresar al punto de partida. A pesar de su aparente simplicidad, el TSP presenta una complejidad computacional considerable, especialmente a medida que aumenta el número de ciudades involucradas, convirtiéndose en un problema de optimización NP-completo de gran interés.

Por otro lado, los Orthogonal Arrays (***OA***) emergen como una herramienta esencial en el diseño de experimentos y la optimización de procesos. Estas estructuras matemáticas, desarrolladas en el ámbito de la estadística experimental, permiten una exploración sistemática y eficiente de las relaciones entre múltiples variables en un experimento. Los OA se caracterizan por su capacidad para garantizar un muestreo equilibrado y una reducción significativa en el número de combinaciones necesarias para investigar el efecto de varios factores sobre un resultado dado. Esta propiedad los convierte en una herramienta valiosa en campos

como la ingeniería, la manufacturación y la investigación operativa, donde la eficiencia en la experimentación es crucial para el éxito y la optimización de los procesos.

A primera vista, el TSP y los OA pueden parecer conceptos aislados, cada uno perteneciente a dominios separados de la matemática y la estadística. Sin embargo, existe un potencial intrigante para explorar posibles conexiones entre ambos problemas. Esta investigación busca precisamente explorar la intersección entre el TSP y los OA, investigando cómo los principios y técnicas del TSP pueden aplicarse para abordar los OA de manera innovadora y efectiva. Al hacerlo, no solo se aspira a ampliar la comprensión teórica de ambos problemas, sino también a identificar nuevas aplicaciones prácticas que puedan beneficiar una amplia gama de industrias y campos de estudio.

1.2. Declaración del problema y objetivo de la investigación

La intersección entre el Problema del Viajante (TSP) y los Orthogonal Arrays (OA) ofrece una oportunidad única para explorar y desarrollar nuevas metodologías en el ámbito de la optimización y el diseño experimental.

Los Orthogonal Arrays (OA) son estructuras matemáticas esenciales en el diseño de experimentos y la optimización de procesos. Desarrollados en la estadística experimental, los OA permiten una exploración sistemática y eficiente de las relaciones entre múltiples variables dentro de un experimento. Su capacidad para garantizar un muestreo equilibrado y reducir significativamente el número de combinaciones necesarias para investigar el efecto de varios factores los hace indispensables en campos como la ingeniería, la manufactura y la investigación operativa.

El objetivo principal de esta investigación es investigar la relación entre el TSP y los OA, y explorar cómo las técnicas y principios del TSP pueden ser aplicados para abordar la generación de OA de manera innovadora y eficiente. La investigación se centra en ampliar la comprensión teórica de ambos problemas, al tiempo que identifica aplicaciones prácticas que puedan beneficiar a una variedad de campos, tales como la logística, la planificación de rutas y la optimización de procesos. Al examinar las posibles sinergias entre estos dos problemas, esta tesis se propone descubrir métodos novedosos para la generación de $OA's$, utilizando enfoques inspirados en la resolución del TSP. No obstante, también se aportarán algoritmos de generación de orthogonal arrays basados en la búsqueda loca y la genética, justificando cuándo son eficientes y comparándolos con aquel obtenido mediante la analogía con el TSP.

2. Marco Teórico

En este apartado, se exploran los fundamentos teóricos que sustentan tanto el Problema del Viajante (TSP) como los Orthogonal Arrays (OA), abordando sus definiciones, características claves y aplicaciones relevantes en sus respectivos campos. Se parte de un análisis general para comprender la importancia y la complejidad inherente de ambos problemas, sentando las bases para una comprensión más profunda de sus implicaciones teóricas y prácticas. Este apartado servirá como fundamento conceptual para la posterior discusión de la relación entre el TSP y los OA, destacando la relevancia de esta intersección en el contexto de la modelización matemática, la optimización combinatoria y la estadística experimental.

2.1. Problema del Viajante

El Problema del Viajante de Comercio, conocido por **TSP** por sus siglas en inglés *"Travelling Salesman Problem"* consiste en dar respuesta a la necesidad de buscar una ruta que pase por **n** ciudades, empezando y terminando en la misma y que esta minimice la distancia total recorrida.

Bajo este contexto, una de las técnicas usadas para su resolución es la programación lineal, herramienta fundamental en la optimización de dicho problema. De la mano de George Dantzig, en el siglo XX, emerge dicha rama matemática que, mediante un sistema de ecuaciones lineales e inecuaciones con variables no negativas, trata de optimizar una función lineal restringida a un poliedro.

La primera aparición del término "Problema del Viajante" en los círculos matemáticos se remonta aproximadamente a 1931-32. Aunque no se puede determinar con certeza quién introdujo inicialmente el concepto, Merrill Flood es ampliamente reconocido como la figura clave en su difusión dentro de la comunidad. Flood tuvo su primer encuentro con el TSP a través de A.W. Tucker en 1937, mientras abordaba el desafío de una ruta de autobús escolar en New Jersey. Según Flood, Tucker mencionó haber oído hablar del problema por Hassler Whitney en la Universidad de Princeton, alrededor del mismo período de tiempo. Esto sugeriría que Whitney fue pionero en el TSP, pero él mismo negó cualquier vínculo entre él y el problema.

Gracias a la fuerte conexión de este problema con los nuevos temas en problemas de combinatoria que emergían de la mano de la programación lineal, y su alta complejidad y dificultad de resolución, se convirtió en objeto de estudio.

2.1.1. Conceptos Básicos

Definición 1. Se denomina **grafo** a un par $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, donde $\mathcal{V}$ es un conjunto finito de elementos denominados **vértices** y $\mathcal{A}$ es el conjunto de pares de vértices que denotarán las **aristas** del grafo.

Observación 1. Nótese que, si un grafo está compuesto por la unión de todos sus vértices, se tiene que $|A| = |V|^2$. En este caso se denomina **grafo completo**.

Definición 2. Una sucesión de vértices $v_1, v_2, \cdots, v_m \in \mathcal{V}$ se denomina **ciclo** si se satisface:

- $v_1, v_2, \cdots, v_{m-1}$ son todos distintos.
- $v_1 = v_m$
- $(v_i, v_{i+1}) \in \mathcal{A}, i = 1, \cdots, m-1$

Si el ciclo contiene a todos los vértices de $\mathcal{V}$, es conocido como **ciclo hamiltoniano**.

Observación 2. Con estas primeras definiciones, el TSP trata de encontrar aquel ciclo hamiltoniano del grafo cuya distancia sea la mínima entre todos ellos. Del mismo modo, el hecho de querer encontrar un ciclo hamiltoniano de algún grafo en particular, en realidad sería un caso particular del TSP.

2.1.2. Proceso de Construcción del Algoritmo

En la década de los 50, se consiguió dar respuesta a dicho problema en un conjunto reducido de 42 ciudades localizadas en los Estados Unidos de América, suponiendo uno de los principales hitos en la optimización combinatoria. Para tratar de entender el procedimiento que se llevó a cabo, es necesario recorrer el mundo de la programación lineal y descubrir cierto material que se empleará en él.

- **El Problema de Asignación**

También conocido como el problema de emparejamiento o el problema de asignación de tareas, es un problema clásico en la teoría de grafos y la optimización combinatoria. Consiste en asignar un conjunto de recursos (p.e. personas, máquinas o trabajos) a un conjunto de tareas (p.e. proyectos, asignaciones o trabajos) de manera óptima, minimizando o maximizando alguna métrica específica, como el costo total, el tiempo total o la satisfacción total.

En relación al TSP, se dispone de una matriz cuadrada $C = (c_{ij})$ de tamaño n cuyas entradas proporcionan el coste de viajar de la ciudad i a la ciudad j. Con ello, se busca encontrar aquella combinación de elementos de C cuya suma sea mínima y estén dispuestos en distintas columnas y filas. Así, una manera de plantearlo en el ámbito de la programación lineal es considerar el poliedro P en el espacio n^2 generado por todas las matrices $X = x_{ij}$. De esta forma, se define el problema como:

$$\begin{aligned} \min \quad & \sum_{i}^{n} \sum_{j}^{m} c_{ij} x_{ij} \\ \text{s.a.} \quad & \sum_{i}^{n} x_{ij} = 1, \forall j \\ & \sum_{j}^{n} x_{ij} = 1, \forall i \\ & x_{ij} \geqslant 0, \forall i, j \end{aligned}$$

Sin embargo, esta interpretación puede dar lugar a soluciones que sean un conjunto de subcircuitos entre las ciudades. Como en el TSP se pretende visitar cada ciudad exactamente una sola vez, es necesario que no se generen dichos subcircuitos o "subtours" desconectados.

La forma de imponer ciertas restricciones que los prohiban pasa por generar 2^{n-1} inecuaciones y añadirlas a las ya descritas.

Además de ello, se observa que los vértices de P son precisamente todas las matrices $X = (x_{ij})$ en las que aparece exactamente un 1 en cada fila y columna, siendo 0 el resto de entradas.

- **El Problema de Transporte**

El problema de transporte es otro problema clásico en la teoría de redes y la optimización, que tiene una relación interesante con el Problema del Viajante. El problema de transporte se centra en determinar la forma más eficiente de transportar bienes desde una serie de orígenes a una serie de destinos, teniendo en cuenta las capacidades de transporte disponibles, las demandas de los destinos y los costos asociados.

$$\begin{aligned} \min \quad & \sum_{i}^{n}\sum_{j}^{m} c_{ij}x_{ij} \\ \text{s.a.} \quad & \sum_{i}^{n} x_{ij} = b_j, \forall j \\ & \sum_{j}^{n} x_{ij} = a_i, \forall i \\ & x_{ij} \geqslant 0, \forall i, j \end{aligned}$$

En este caso, los enteros no negativos a_i y b_j hacen alusión a a la mercancía disponible en i y la demanda del punto j, respectivamente. Así, se debe cumplir que $\sum_i a_i = \sum_j b_j$. Además, las componentes c_{ij} hacen referencia al coste unitario que supone llevar mercancía de i a j. A diferencia de la construcción del poliedro P en el problema de asignación, las matrices resultantes en el problema de transporte pueden contener entradas distintas a 0 y 1, por lo que es necesario imponer que solo puedan tomar dichos valores.

Esto produce dificultades en la construcción del modelo; por un lado, ya no se tienen $2n$ ecuaciones en variables no negativas, sino que este número es mucho más grande. Y, por otro lado, siendo más complejo este obstáculo, el hecho de restringir las variables a unas binarias, lo convierten en un problema de programación lineal entera.

En 1953, ya estaban disponibles códigos efectivos que implementaban el método simplex, tanto en su forma general como adaptada para problemas específicos de transporte y asignación. Dantzig, Fulkerson y Johnson presentaron un enfoque para resolver un desafío de planificación relacionado con un petrolero. Aunque este método fue eventualmente reemplazado, su utilidad persistió, ya que proporcionaba un marco para explorar aspectos fundamentales de la teoría combinatoria. Por otro lado, Ford y Fulkerson publicaron un informe inicial sobre flujos en redes, estableciendo un tema crucial en el campo. Además, Selmer Johnson logró un avance significativo en la programación de trabajos en entornos de producción en masa, presentando el primer teorema importante en combinatoria aplicado al estudio de la programación de máquinas.

- **El Problema del Viajante de Comercio**

Así, relacionándose con estos dos últimos problemas, nace el TSP, que será objeto fundamental para el desarrollo de este documento.

Aunque el TSP es NP-difícil, se pueden formular y resolver modelos de optimización lineal entera para obtener soluciones exactas en casos de tamaño moderado.

Es decir, como se debe describir una curva poligonal cerrada que describa un recorrido por todas las ciudades, una forma de afrontarlo es mediante la evaluación de todas las posibles rutas que se pueden crear. Esta sencilla idea se vuelve tediosa con un número de ciudades no muy elevado, dado que, el número de posibles rutas crece de forma factorial al añadir más rutas:

- Posbiles rutas con 5 ciudades: $5! = 120$ rutas.
- Posibles rutas con 10 ciudades: $10! = 3628800$ rutas.
- Posibles rutas con 30 ciudades: $30! = 2.6525285981219 \cdot 10^{32}$ rutas.

Se observa que es inviable afrontar este problema mediante prueba y error cuando el número de ciudades tampoco es que sea muy excesivo. De esta forma, se opta por utilizar herramientas de optimización para buscar soluciones factibles.

Supóngase que se tienen n ciudades, identificadas con un número del conjunto $N = \{1, \cdots, n\}$ y una matriz $C = (c_{ij})_{i,j=1}^{n}$ que guarda la información del coste de viajar de la ciudad i a la j. Obsérvese que esta matriz puede ser presentada como una triangular superior o inferior y su traza siempre será un vector de $0's$.

Como se debe encontrar una ruta que pase por todas las ciudades, se definirá el siguiente conjunto de variables:

$$x_{ij} = \begin{cases} 1, & \text{si la ruta recorre el camino de la ciudad } i \text{ a } j \\ 0, & \text{en caso contrario} \end{cases}, \quad i, j \in \{1, \cdots, n\}, i \neq j \tag{2.1}$$

Gracias a esta notación, encontrar la ruta que minimiza el coste, se traduce en encontrar la solución para la siguiente expresión:

$$\text{mín} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.2}$$

No obstante, de no imponer ninguna restricción, la solución resulta ser la trivial donde todas las variables valen 0, dado que son binarias. Por ello, es necesario imponer que, como se debe visitar una sola vez cada ciudad, en la entrada y salida de cada una de ellas debe existir un único camino señalizado. Esto se traduce en las siguientes dos restricciones:

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = \{1, \cdots, n\} \tag{2.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = \{1, \cdots, n\} \tag{2.4}$$

Lo que determinan las ecuaciones (2.3) y (2.4), es que cada ciudad deber tener un único camino de entrada y salida, respectivamente. No obstante, aun con todo ello, se podrían encontrar soluciones que generen recorridos cíclicos y disjuntos por todas las ciudades. Hace falta imponer una nueva restricción para evitar estos casos.

La idea reside en controlar el orden de las ciudades y evitar que, si se viaja de las ciudad i a la j, no pueda darse que la ciudad i esté ubicada en una posición superior a la j en el orden de la visita. Si se define u_i como la posición que toma la ciudad i en el recorrido creado, se definen las denominadas *restricciones de Miller-Tucker-Zemlin (MTZ)*:

$$u_i - u_j + n \cdot x_{ij} \leqslant n - 1 \tag{2.5}$$

Esta ecuación, en el caso de que las ciudades no estén conectadas por ningún arco, no impone ninguna restricción en la enumeración de las mismas. Por otro lado, si las ciudades sí está conectadas, la ecuación es equivalente a la relación $u_j \geqslant u_i + 1$, que imposibilita que se vuelva a visitar i posteriormente. Con esto, el modelo de optimización lineal entera resulta ser:

$$\begin{aligned}
\text{mín} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
s.a \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = \{1, \cdots, n\} \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = \{1, \cdots, n\} \\
& u_i - u_j + n \cdot x_{ij} \leqslant n - 1 \\
& x_{ij} = \{0, 1\}, u_i = 2, \cdots, n
\end{aligned}$$

2.2. Orthogonal Arrays

En la década de los 40, el matemático C. R. Rao introduce ciertas herramientas de combinatoria con aplicación directa a la rama de la estadística. En una primera instancia, consideró únicamente una parte reducida de este importante estudio, sin embargo, rápidamente se generó un campo completo que se describe bajo el nombre de **orthogonals arrays** (matrices ortogonales). Gracias a este nuevo concepto de comprender y manipular los problemas de optimización, muchos investigadores encontraron grandes fuentes de inspiración al adentrarse en este nuevo mundo matemático.

2.2.1. Conceptos básicos

Definición 3. Una matriz A de dimensión $N \times k$ con elementos pertenecientes a un conjunto S se dice que es un **orthogonal array** con s niveles, fortaleza t e índice λ $(0 \leqslant t \leqslant k)$ si, para toda submatriz $N \times t$ de A, esta contiene exactamente λ veces t-tuplas de elementos de S colocados en filas.

Observación 3. En el caso particular de que $\lambda = 1$, se dice que el orthogonal array tiene índice único.

Notación: Los enteros N, k, s, t y λ se especifican como los parámetros del orthogonal array, indicándose como $OA(N, k, s, t)$. No es necesario indicar λ en esta notación ya que es determinado por la siguiente expresión:

$$\lambda = \frac{N}{s^t}$$

Otras expresiones que se pueden encontrar en la literatura son: $L_N, L_N(\lambda \times s^k), L_{\lambda \times s^t}(k), s - OA(t, k, \lambda)$, $OA(N, k, s, t) : \lambda, OA(N, s^k, t), OA_\lambda(y, k, s)$ y $OA(s, k, \lambda)$.

Ejemplo 1. Supóngase que se tiene un conjunto S con dos niveles $\{0, 1\}$. Un orthogonal array de elementos en S, con fortaleza tres, índice único, 8 ejecuciones y 4 factores, es decir, $OA(8, 4, 2, 3)$, es el siguiente:

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Al escoger cualquier trío de entre las 4 columnas de la matriz, se obtienen tríos de elementos que no se repiten, por ello es de índice único.

Ejemplo 2. Toda matriz $N \times k$ tiene fortaleza 0 por definición. Un orthogonal array con fortaleza 1 puede ser construido de manera trivial creando tantas filas como elementos contenga el conjunto S y donde en cada una de ellas no exista más de un elemento de este. Por ejemplo, $OA(2, 12, 2, 1)$.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

Observación 4. En el campo de las matemáticas, el adjetivo ortogonal se suele aplicar en el estudio de vectores o campos vectoriales, donde su significado viene a reflejar la particular posición que estos adquieren entre sí. Es decir, si se tienen vectores $u = (u_1, \cdots, u_n)$ y $v = (v_1, \cdots, v_n)$, se dicen que son vectores ortogonales si su producto interior es nulo.

$$u \cdot v^T = u_1 v_1 + \cdots + u_n v_n = 0$$

Sin embargo, un *orthogonal array* no adquiere dicho adjetivo por la definición usual de la que goza. Tanto la ortogonalidad en vectores, como la de estas particulares matrices, buscan asegurar una forma de independencia o balance. En la primera, la independencia lineal se manifiesta en la ortogonalidad. En los $OA's$, el balance se manifiesta en la

distribución uniforme de combinaciones.

En resumen, aunque la ortogonalidad en vectores y orthogonal arrays se aplica en contextos matemáticos distintos, ambas buscan estructurar información de manera equilibrada y optimizada. La ortogonalidad de vectores se centra en relaciones geométricas y algebraicas, mientras que en los orthogonal arrays se enfoca en la distribución uniforme y balanceada de combinaciones de símbolos en matrices

2.2.2. Propiedades de los Orthogonal Arrays

A continuación se muestran algunas de las propiedades fundamentales de los orthogonal arrays, descritas por Hedayat et. al (1999), y cuyas demostraciones son de elaboración propia del autor de este trabajo de fin de máster.

1 Los parámetros de cualquier orthogonal array satisfacen la siguiente identidad.

$$\lambda = \frac{N}{s^t} \longleftrightarrow N = \lambda s^t$$

Demostración:

Sea $OA(N, k, s, t)$ un orthogonal array compuesto por el conjunto de símbolos (niveles) $S = \{1, \ldots, s\}$. La matriz asociada debe hacer uso de cada uno de los niveles y, dado que la fortaleza indica las t-tuplas de símbolos que se repiten, es necesario que se den todas las posibles combinaciones de t símbolos del conjunto S, es decir, las s^t posibles combinaciones.

Asimismo, el índice del OA indica cuántas veces se repiten las combinaciones. Por tanto, habrá tantas filas como repeticiones de las t-tuplas de elementos de S haya. Es decir, $N = \lambda s^t$

□

2 Cualquier orthogonal array de fortaleza t es también un orthogonal array de fortaleza t' con $0 \leqslant t' < t$. Asimismo, el índice de este nuevo nivel de fortaleza es $\lambda s^{t-t'}$, donde λ es el índice con la fortaleza original.

Demostración:

Supóngase que se tiene un $OA(N, k, s, t)$ con índice λ, esto quiere decir que, para toda submatriz $N \times t$ de dicho orthogonal array, las filas se repiten un número λ de veces. Si se escoge una fortaleza positiva t' menor que t, todo selección de t' columnas será un subconjunto de algunas t originales.

Como se tienen s símbolos, en cada una de las t' columnas hay $s^{t'}$ posibles combinaciones de los correspondientes símbolos y, debido a que el OA tiene fortaleza orginal t, cualquier subconjunto de t' columnas seleccionadas de las t columnas debe tener cada

combinación de símbolos de manera uniforme.

Asimismo, para cada combinación de t' símbolos en las t' columnas seleccionadas, existen $s^{t-t'}$ posibles combinaciones de completar tantas filas hasta la fortaleza original. Por ende, el número de veces que cada combinación de t' columnas aparece en cualquier conjunto de t' columnas es *lambda* multiplicado por $s^{t-t'}$.

□

3 Para cada $\boldsymbol{A_i}$, $\boldsymbol{i = 1, \cdots, r}$ sea $\boldsymbol{A_i = OA(N_i, k, s, t_i)}$, entonces la matriz $\boldsymbol{A}$ resultante de colocar las $\boldsymbol{r}$ matrices en una columna:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} \\ \vdots \\ \boldsymbol{A_r} \end{bmatrix}$$

es un $\boldsymbol{OA(N, k, s, t)}$, donde $\boldsymbol{N = N_1 + \cdots + N_r}$ y la fortaleza es $\boldsymbol{t}$ para algún entero $\boldsymbol{t \geqslant \text{mín}\{t_1, \cdots, t_r\}}$. Además, cuando $\boldsymbol{r = s}$ y cada $\boldsymbol{A_i}$ es un $\boldsymbol{OA(N, k, s, t)}$, tras añadir un $\boldsymbol{i}$ en cada fila de $\boldsymbol{A_i}$, $\boldsymbol{i = 1, \cdots, r}$, se obtiene un $\boldsymbol{OA(sN, k + 1, s, t)}$.

Demostración:

El número de filas del orthogonal array resultante de colocar todos los A_i en columna es trivial que resulta ser la suma de todas las filas de cada uno de ellos $N = N_1 + \cdots + N_r$. Del mismo modo, dado que todos los A_i tienen el mismo número de columnas k, la dimensión de A es $N \times k$.

Por otra parte, haciéndose uso del apartado anterior, se sabe que si un OA es de fortaleza t, también lo es de fortaleza t' con $0 \leqslant t' < t$. Por tanto, si se escogiese la menor fortaleza en cada uno de los OA descritos por A_i, todos ellos aceptarían dicha fortaleza, cumpliendo que la fortaleza del OA descrito por A debe ser, como mínimo, la menor de todas las de los orthogonal arrays que la componen. Es importante notar que esto solo se satisface si se describe la matriz como composición de matrices filas.

Por último, en el caso particular de que se tengan s matrices que describen OA's con parámetros semejantes, las dimensiones de A, en este caso, será $sN \times k$. Si además, a la primera matriz se le añade una columna de $1's$, a la segunda una de $2's$ y así sucesivamente con las s matrices. Resulta un OA de parámetros $OA(sN, k + 1, s, t)$, dado que si en cada una de las submatrices se escogen $t - 1$ columnas y la columna final, como dicha columna no contiene símbolos diferentes, no perjudica la fortaleza t original.

□

4 Una permutación de cualquier fila o factor de un orthogonal array genera un orthogonal array con los mismos parámetros.

Demostración:

De formar lógica, el hecho de permutar filas no interfiere en el análisis de las $t-tuplas$ del OA. Del mismo modo, permutar columnas es insignificante a la hora de seleccionar t columnas para estudiar las repeticiones de sus elementos.

□

5 Una permutación de los niveles de cualquier factor en un orthogonal array genera un orthogonal array con los mismos parámetros.

Demostración:

Permutar los niveles de un factor (columna) no altera las combinaciones de niveles en las demás columnas. Supóngase que, originalmente, se tiene un $OA(N, k, s, t)$, SPDG, si se permutasen los factores de la columna j, con $1 \leqslant j \leqslant k$, cualquier subconjunto de t columnas que no contenga a dicha columna sigue satisfaciendo la definición del OA. Por otro lado, si se escogiese dicha columna para formar las t-tuplas, del mismo se satisface, ya que cada uno de los niveles (símbolos) es intercambiado por otro, manteniendo las repeticiones originales.

□

6 Cualquier submatriz $\boldsymbol{N \times k'}$ de un $\boldsymbol{OA(N, k, s, t)}$ es un $\boldsymbol{OA(N, k', s, t')}$, donde $\boldsymbol{t' = \text{mín}\{k', t\}}$.

Demostración:

Si se escoge cualquier combinación de k' columnas, puede ocurrir que el número de columnas exceda a la fortaleza del OA, en ese caso, la fortaleza del nuevo orthogonal array es idéntica al del original.

Sin embargo, si el número de fila es inferior a t, por la propiedad 2, la matriz que describe el OA original, también goza de fortaleza k, luego, la submatriz será un $OA(N, k', s, k')$.

□

7 Tomar las ejecuciones de un $\boldsymbol{OA(N, k, s, t)}$ que empiecen por algún factor en particular y omitir la primera columna, genera un $\boldsymbol{OA(\frac{N}{s}, k-1, s, t-1)}$.

Demostración:

Por la uniformidad de la que goza la distribución de niveles en las columnas de la matriz, se observa que, en cada una de ellas, cada nivel se repite un número de $\frac{N}{s}$ veces. Por tanto, al elegir las filas que empiecen por un nivel en particular, se estarán seleccionando $\frac{N}{s}$ filas.

Esto genera una submatriz $\frac{N}{s} \times k$ y, al eliminar la primera columna (aquella con todos los elementos del mismo nivel), se obtiene una matriz $\frac{N}{s} \times (k-1)$.

Como se ha construido la matriz de forma que la primera columna contenga el mismo elemento en todas sus entradas, obsérvese que, las combinaciones de elementos de la primera columna y $t-1$ elementos de columnas distintas a esta, están repetidos un número de λ veces, por la propiedad del OA original.

Por ello, si se realizan combinaciones de $t-1$ columnas diferentes, sin seleccionar la primera, los elementos estarán repetidos un número de veces λ de igual forma, pues la primera columna no genera ninguna diferencia en las combinaciones al tener sus elementos iguales.

Así, el orthogonal array resultante es un $OA(\frac{N}{s}, k, s, t-1)$ y su índice coincide con el del original:

$$\lambda' = \frac{\frac{N}{s}}{s^{t-1}} = \frac{N}{s^t} = \lambda \tag{2.6}$$

8 Sea $\mathcal{C}$ el conjunto de todas las posibles ejecuciones que se pueden dar en un orthogonal array en particular A y, para cada $c \in \mathcal{C}$, sea f_c la frecuencia de c en A. Sea f el máximo de las f_c. Entonces, la matriz que contienen ejecuciones c con frecuencia $f - f_c$, $\forall c \in \mathcal{C}$, se denomina el *conjunto teórico complementario*, o simplemente el *complementario*, de A. Con ello, el complementario de $OA(N, k, s, t)$ es $OA(fs^k - N, k, s, t)$.

Demostración:

Sea $A = OA(N, k, s, t)$ y C el conjunto de todas las ejecuciones c que pueden aparecer en A, en otras palabras, todas las posibles combinaciones de k símbolos del conjunto $S = \{0, \cdots, s\}$, es decir, $|C| = s^k$.

Si se denota como f_c la frecuencia de las apariciones de la ejecución $c \in C$ en el orthogonal array, siendo f el máximo de estas frecuencias, y se construye la matriz que almacena $f - f_c$ ejecuciones $c \in C$, se observa, en primer lugar, que el número de filas debe ser $fs^k - N$. El número total de filas, debe ser el número total de frecuencias de cada una de las ejecuciones:

$$\sum_{c \in C}(f - f_c) = \sum_{c \in C} f - \sum_{c \in C} f_c = f|C| - N = fs^k - N \tag{2.7}$$

Esto implica que el complementario A' tiene $fs^k - N$ filas y k columnas, dado que está compuesto por ejecuciones de C que son listas de k elementos de S.

Por otro lado, para comprobar que la fortaleza no varía, obsérvese que cada submatriz

de t columnas de A' tiene repetidas sus filas un número de $fs^{k-t} - \lambda$ veces; esto es de haber creado las submatrices de t, dejando fuera $k-t$ posibles elecciones del conjunto S y, debido a la construcción de A', se invierten las frecuencias en las apariciones, es por ello que se resta la componente λ. Así, mediante la relación ente el índice y los parámetros del OA, descrita en la propiedad **(1)**, se puede afirmar que la fortaleza es t:

$$\lambda' = fs^{k-t} - \lambda = fs^{k-t} - \frac{N}{s^t} = \frac{fs^k - N}{s^t} \longrightarrow \textbf{fortaleza} = \boldsymbol{t} \tag{2.8}$$

Con todo ello, A' es un $OA(fs^k - N, k, s, t)$.

□

9 Sea $\boldsymbol{A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}}$ **un** $\boldsymbol{OA(N, k, s, t)}$**, donde** $\boldsymbol{A_1}$ **es, en sí misma, un** $\boldsymbol{OA(N_1, k, s, t_1)}$**. Entonces,** $\boldsymbol{A_2}$ **es un** $\boldsymbol{OA(N - N_1, k, s, t_2)}$**, con** $\boldsymbol{t_2 \geqslant \text{mín}\{t, t_1\}}$**.**

Demostración:

Es trivial observar que las dimensiones de los orthogonal arrays deben ser las descritas para que pueda encajar bien a la hora de generar el OA resultante de juntar los más pequeños. Así, se debe demostrar que la fortaleza de A_2 es $t_2 \geqslant \text{mín}\{t, t_1\}$.

Supóngase que t_2 es menor que estas fortalezas, entonces, haciéndose uso de las relación fundamental descrita en la propiedad **(1)**:

$$\lambda_2 = \frac{N_2}{s^{t_2}} = \frac{N - N_1}{s^{t_2}} = \frac{N}{s^{t_2}} - \frac{N_1}{s^{t_2}} \tag{2.9}$$

Se observa que, como mínimo, se puede tomar t_2 como el mínimo entre t y t_1 ya que, dado que A y A_1 son orthogonal arrays por definición, se tiene:

$$s^t \mid N \ \wedge \ s^{t_1} \mid N_1 \quad \Longrightarrow \quad s^{t^*} \mid N \ \wedge \ s^{t^*} \mid N_1, \quad t^* = \text{mín}\{t, t_1\} \tag{2.10}$$

Es decir, como mínimo, se puede tomar como fortaleza t_2 el valor t^*, dado que es divisor de N y N_1 □

2.2.3. Resultados Importantes

Tras estas propiedades, se exponen algunos de los resultados más interesantes de los orthogonal arrays, donde se busca comprender, de forma más profunda, la relación existente entre los parámetros de estas particulares matrices, sirviendo para justificar resultados obtenidos en la parte práctica. Todos estos resultados son extraídos de Hedayat et. al (1999).

Definición 4. Dos orthogonal arrays se dicen **isomorfos** si se puede obtener uno de ellos tras generar una serie de permutaciones de columnas, filas y niveles de cada factor del otro.

Teorema 5. *Los parámetros de un $OA(N, k, s, t)$, deben satisfacer alguna de las siguientes*

condiciones, dependiendo de la paridad de la fortaleza:

$$\begin{aligned} N &\geqslant \sum_{i=0}^{u} \left(\begin{array}{c} k \\ i \end{array} \right) (s-1)^i, \quad si\ t = 2u \\ N &\geqslant \sum_{i=0}^{u} \left(\begin{array}{c} k \\ i \end{array} \right) (s-1)^i + \left(\begin{array}{c} k-1 \\ u \end{array} \right) (s-1)^{u+1}, \quad si\ t = 2u+1 \end{aligned} \tag{2.11}$$

Demostración:

Sean $A = (a_{ij}), a_{ij} \in S, i = 1, \cdots, N,\ j = 1, \cdots, k$ y un $OA(N, k, s, t)$. Se considera $S = \{0, 1, \cdot, s-1\}$ como un subconjunto de los números reales. Por simplificar la notación, sea M la expresión que queda a la derecha de la desigualdad (2.11), dependiendo de si t es par o impar. Se usará el conjunto A para construir una matriz M, de dimensiones $N \times M$, cuyo rango sea M. Esto implicará que $N \geqslant M$, siendo demostrado el teorema.

La matriz H que se construirá, no solo será de rango M, pero tendrá la propiedad que $H^T H$ es una matriz diagonal de dimensiones $M \times M$. Para empezar, sea B una matriz $(s-1) \times s$ cuyas filas son ortogonales dos a dos y, además, todas ellas son ortogonales al vector 1_s^T (vector de dimensión s cuyas entradas son todo 1). Será convenienet denotar a las filas de la matriz B por $1, 2, cdots, s-1$, y sus columnas por $0, 1, \cdots, s-1$. Así, si $b(i, j)$ denota una entrada de B en la posición (i, j), se tienen las siguientes igualdades:

$$\begin{aligned} \sum_{j=0}^{s-1} b(i,j)b(i',j) = 0, \quad i \neq i' \\ \sum_{j=0}^{s-1} b(i,j) = 0 \end{aligned} \tag{2.12}$$

Caso 1: $t = 2u$. Para cada $m \in \{1, 2, \cdots, u\}$ y cada m-tupla ordenada $\{i_1, i_2, \cdots, i_m)$, donde $i_j \in \{1, 2, \cdots, s-1\}$, se define un matriz $H(i_1, i_2, \cdots, i_m)$ de dimensiones $N \times \binom{k}{m}$. Se etiquetarán las columnas por los $\binom{k}{m}$ m-conjuntos de $\{1, 2, \cdots, k\}$, y las filas por $1, 2, \cdots, N$.

Cualquier entrada de $H(i_1, i_2, \cdots, i_m)$ será denotada por $h^{(i_1, i_2, \cdots, i_m)}_{j,(l_1, l_2, \cdots, l_m)}$, donde los subíndices hacen referencia a las correspondientes filas y columnas que interactúan, respectivamente. Para evitar cualquier ambigüedad, se asume que las $l_i's$ están ordenadas. Las entradas de $H(i_1, i_2, \cdots, i_m)$ serán ahora definidas mediante el uso del orthogonal array A y la matriz B como sigue:

$$h^{(i_1, i_2, \cdots, i_m)}_{j,(l_1, l_2, \cdots, l_m)} = b(i_1, a_{jl_1}) \cdots b(i_m, a_{jl_m}) \tag{2.13}$$

Finalmente, la matriz H está dada por:

$$H = [1_N, H(1), \cdots, H(s-1), H(1,1), H(1,2), \cdots, H(s-1, s-1, \cdots, s-1)] \tag{2.14}$$

Tras la columna de $1's$, se construye la matriz por bloques con las matrices $H(i_1, i_2 m \cdots, i_m)$, con $m \in \{1, 2, \cdots, u\}$ y $i_j \in \{1, 2, \cdots, s-1\}$. Claramente, H tiene $1 + \binom{k}{1}(s-1) + \cdots + \binom{k}{u}(s-1)^u = M$ columnas y, por ende, es de dimensión $N \times M$.

Caso 2: $t = 2u + 1$. Se comienza con la matriz H como se construyó en el caso anterior. Además, para cada $(u+1)$-tupla $(i_1, \cdots, i_{u+1})$ de elementos de $\{1, \cdots, s-1\}$ se define una matriz $\bar{H}(i_1, i_2, \cdots, i_{u+1})$ de dimensiones $N \times \binom{k-1}{u}$, cuyas columnas se etiquetan como $\{1, l_2, \cdots, l_{u+1}\}$, $1 < l_2 < \cdots < l_{u+1} < k$, sus filas como $1, \cdots, N$ y sus entradas se definen como:

$$h^{(i_1,i_2,\cdots,i_{u+1})}_{j,(l_1,l_2,\cdots,l_{u+1})} = b(i_1, a_{j1})b(i_2, a_{jl_2}) \cdots b(i_{u+1}, a_{jl_{u+1}}) \tag{2.15}$$

Para obtener un matriz $N \times M$ para este caso, se añaden las columnas de estas $(s-1)^{u+1}$ matrices a la matriz construida H del caso 1. Todas las columnas de estas $N \times M$ matrices son no nulas. Falta ver que son ortogonales. Claramente, para $m \leqslant u + 1$:

$$\begin{aligned}
\sum_{j=1}^{N} h^{(i_1,\cdots,i_m)}_{j,(l_1,\cdots,l_m)} &= \sum_{j=1}^{N} (b(i_1, a_{jl_1}) \cdots b(i_m, a_{jl_m}) \\
&= \frac{N}{s^m} \left(\sum_{j_1=0}^{s-1} \cdots \sum_{j_m=0}^{s-1} (b(i_1, j_1) \cdots b(i_m, j_m) \right) \\
&= \frac{N}{s^m} \left(\sum_{j_1=0}^{s-1} \cdots \sum_{j_{m-1}=0}^{s-1} (b(i_1, j_1) \cdots b(i_{m-1}, j_{m-1}) \left(\sum_{j_m=0}^{s-1} b(i_m, j_m) \right) \right) \\
&= 0
\end{aligned} \tag{2.16}$$

donde se ha usado que cualquier colección de m columnas del orthogonal array A,$m \leqslant t$, forman un orthogonal array de fortaleza m e índice N/s^m (propiedad 2 de la sección 2.2.2). Esto demuestra que las columnas de $H(i_1, \cdots, i_m)$ y $\bar{H}(i_1, \cdots, i_{u+1})$ son ortogonales a 1_N. El producto interior de cualesquiera dos columnas es de la forma:

$$\sum_{j=1}^{N} (b(i_1, a_{jl_1}) \cdots b(i_m, a_{jl_m}) b(f_1, a_{jk_1}) \cdots b(f_n, a_{jk_n}) \tag{2.17}$$

El argumento básico por el que se anula la expresión en (2.16) es la misma que en el caso anterior. De la construcción de H, las columnas $l_1, \cdots, l_m$ y $k_1, \cdots, k_n$ en A, contienen, como mucho, t columnas distintas. Si hubiese c columnas iguales en ambas secuencias, dígase $l_{p_1} = k_{r_1}, \cdots, l_{p_c} = p_{r_c}$, entonces, con $l_{p_{r+1}}, \cdots, p_m$ y $k_{r_{c+1}}, \cdots, k_n$ como las columna restantes, se tiene que el producto interior se reduce a:

$$C \prod_{v=1}^{c} \left(\sum_{j=0}^{s-1} b(i_{p_v}, j) b(f_{r_v}, j) \right) \prod_{v=c+1}^{m} \left(\sum_{j=0}^{s-1} b(i_{p_v}, j) \right) \prod_{v=c+1}^{n} \left(\sum_{j=0}^{s-1} b(f_{r_v}, j) \right) \tag{2.18}$$

donde $C = N/s^{m+n-c}$.

Las propiedades de la matriz B garantizan que las expresión se anulan, a no ser que $m = n = c$ y $i_1 = f_1, \cdots, i_c = f_c$. Pero esto no puede ocurrir desde que se consideren dos columnas distintas de H.

□

Teorema 6. *Los parámetros de un $OA(N, k, s, 4)$ deben pertenecer a uno de lo siguientes*

casos:

(a) $N = 16, \quad k = 5, \quad s = 2$

(b) $N = 243, \quad k = 11, \quad s = 3$

(c) $N = 29m^2(9m^2 - 1), \quad k = (9m^2 + 1)/5, \quad s = 6$ *donde* m *debe satisfacer:*

- $m \equiv 0 \ (mod\ 3)$
- $m \equiv \pm 1 \ (mod\ 5)$
- $m \equiv 5 \ (mod\ 16)$

2.2.4. Aplicaciones Directas

Los orthogonal arrays (OA) son estructuras matemáticas con aplicaciones diversas en múltiples campos debido a sus propiedades únicas de balance y ortogonalidad. Estas aplicaciones abarcan desde el diseño de experimentos hasta la criptografía, pasando por la teoría de códigos y la ingeniería de software. A continuación, se exploran en detalle algunas de las aplicaciones más significativas de los OA en distintos ámbitos de trabajo.

- **Diseño de Experimentos**
  En el campo del diseño de experimentos, los OA son una herramienta fundamental. Estos arrays permiten diseñar experimentos de manera eficiente, asegurando que todos los factores y sus niveles se prueben de manera equilibrada y sistemática. Esto es particularmente útil en situaciones donde realizar todos los experimentos posibles sería costoso o impracticable. Por ejemplo, en la industria manufacturera, se pueden usar OA para diseñar experimentos que optimicen procesos de producción, minimizando la variabilidad y mejorando la calidad del producto final. Además, en el campo de la biología y la medicina, los OA ayudan a diseñar experimentos que investigan el efecto de varios factores en los resultados biológicos, permitiendo una interpretación más precisa y confiable de los datos experimentales.

- **Teoría de Códigos**
  En la teoría de códigos, los OA juegan un papel crucial en la construcción de códigos de corrección de errores. Los códigos de corrección de errores son esenciales para la transmisión de información a través de canales ruidosos, como los utilizados en las comunicaciones digitales y el almacenamiento de datos. Los OA se utilizan para diseñar códigos con buenas propiedades de detección y corrección de errores, lo que aumenta la fiabilidad de la transmisión y el almacenamiento de datos. Estos códigos encuentran aplicaciones en diversas áreas, incluyendo las telecomunicaciones, la transmisión de datos por satélite y las tecnologías de almacenamiento de datos como los discos duros y las memorias flash.

- **Criptografía**
  En el ámbito de la criptografía, los OA se utilizan para diseñar esquemas criptográficos seguros. Las propiedades de balance y ortogonalidad de los OA son aprovechadas para crear cifrados y protocolos que resistan varios tipos de ataques criptográficos. Por

ejemplo, los OA pueden ser utilizados en el diseño de funciones *hash* y generadores de números pseudoaleatorios que son primitivas criptográficas. Además, se emplean en la construcción de esquemas de compartición de secretos y en la autenticación de usuarios, contribuyendo a la seguridad de sistemas de información y comunicaciones.

- **Ingeniería de Software**
  Los OA también tienen aplicaciones importantes en la ingeniería de software, específicamente en la prueba de software. Las pruebas combinatorias, que se basan en OA, son utilizadas para generar casos de prueba que cubren todas las combinaciones posibles de parámetros de entrada hasta un cierto nivel de interacción. Esto asegura que las pruebas sean exhaustivas y que se detecten errores que podrían no ser descubiertos con métodos de prueba menos sistemáticos. En particular, las pruebas combinatorias son útiles en el desarrollo de software crítico, donde la fiabilidad y la robustez son esenciales, como en el software utilizado en la industria aeroespacial, la medicina y los sistemas financieros.

- **Optimización y Simulación**
  En el ámbito de la optimización y la simulación, los OA son utilizados para realizar análisis de sensibilidad y para optimizar funciones objetivo en presencia de múltiples variables. Los OA permiten explorar el espacio de soluciones de manera eficiente, reduciendo el número de simulaciones o evaluaciones necesarias. Esto es particularmente útil en la optimización de diseño de sistemas complejos, como redes de telecomunicaciones, sistemas de energía y procesos industriales. Además, se utilizan en *simulaciones de Monte Carlo* para mejorar la precisión de las estimaciones mediante la reducción de la varianza.

- **Educación y Formación**
  Los OA tienen aplicaciones en el campo de la educación y la formación, especialmente en la enseñanza de la estadística y el diseño de experimentos, proporcionando un marco práctico y accesible para enseñar conceptos avanzados de manera intuitiva. Por ejemplo, se pueden utilizar para diseñar actividades de laboratorio y proyectos de clase que involucren la recopilación y el análisis de datos. Esto no solo mejora la comprensión teórica de los estudiantes, sino que también les proporciona habilidades prácticas en el diseño y la interpretación de experimentos.

- **Agricultura y Ciencias Ambientales**
  En la agricultura y las ciencias ambientales, los OA son utilizados para diseñar estudios que investigan los efectos de múltiples factores ambientales en el crecimiento y rendimiento de cultivos, así como en la salud de los ecosistemas. Estos estudios son esenciales para desarrollar prácticas agrícolas sostenibles y para entender las interacciones complejas entre factores ambientales. Por ejemplo, pueden ser utilizados para diseñar experimentos que evalúen el impacto de diferentes combinaciones de fertilizantes, riego y variedades de cultivos en el rendimiento agrícola.

3. Algoritmos Genéticos y de Búsqueda Local para OA

En esta sección se estudiarán y explicarán ciertos algoritmos de creación propia, diseñados para poder elaborar orthogonal arrays. Se hará un recorrido por distintos algoritmos, comenzando por algoritmos muy básicos, basados en técnicas genéticas, hasta algoritmos más elaborados.

Adicionalmente, también se expondrá ciertos algoritmos que guardan relación con el TSP, ilustrándose la conexión que tienen ambos problemas.

3.1. Algoritmo 1: Genético Básico

En una primera instancia, generar un orthogonal array puede parecer una tarea fácil. Sin embargo, cuando el conjunto de símbolos es pequeño o la matriz es de dimensiones no es manejable, se puede intuir el gran grado de dificultad que presenta.

Recuperándose lo descrito en la sección 2.2.1, donde se demuestran ciertas propiedades de los arrays, se sabe que todos los elementos del conjunto de símbolos S deben estar uniformemente distribuidos en cada una de las columnas de la matriz. Es decir, si se desease buscar un $OA(N, k, s, t)$, cada elemento del conjunto S ($|S| = s$) debe aparecer un número de $\frac{N}{s}$ veces por columna.

Con esta propiedad, se puede plantear un algoritmo que, mediante permutaciones en las filas de una matriz que cumpla con la distribución uniforme por filas, seleccione aquella que reduzca el "coste" o "error" generado.

3.1.1. Descripción del Algoritmo 1

Paso 1. Se genera una matriz de tamaño $N \times k$ (dimensiones del orthogonal array objetivo) cuyas entradas sean elementos del conjunto S.

Ej: Generar $OA(8, 4, 2, 2)$. *Se crea una matriz aleatoria* 8×4 *con elementos de* $S = \{0, 1\}$.

$$\begin{matrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$$

Paso 2 Mediante las funciones *check_proportions* y *unif_proportions*, se examina la propiedad de la uniformidad en la distribución por columnas y se corrige, en caso de que sea necesario.

Se corrigen las filas

$$\begin{matrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix} \Longrightarrow \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$$

Paso 3 Se calcula el "coste" o "error" de la matriz ortogonal. Para ello, se estudia el número de repeticiones de las combinaciones de elementos generadas al escoger cada una de las colecciones posibles de t columnas.

Obsérvese que, los parámetros del orthogonal array, como se discutió en secciones anteriores, facilitan el número de veces que se deben repetir las *t-tuplas*, conocido como el **índice** y denotado por $lambda = \frac{N}{s^t}$.

Por ende, se deberá sumar al coste, la diferencia generada entre el número de repeticiones de cada colección de elementos de la matriz y el índice. Para ello, se hará uso de la distancia euclídea.

En el ejemplo, $\lambda = \frac{8}{2^2} = 2$. Se estudian los errores cometidos por columnas; si una dupla de elementos no está repetida exactamente 2 veces, se suma la diferencia a dicho índice.

$$\begin{matrix} 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \end{matrix}$$

$$\text{Err} = 0 \qquad \text{Err} = 0 \qquad \text{Err} = 3 \qquad \text{Err} = 0 \qquad \text{Err} = 0 \qquad \text{Err} = 4$$

$$\text{Error Total Cometido} = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$

Paso 4. Si el error generado es distinto de 0, se seleccionan aleatoriamente dos filas y una columna de la matriz y se genera un intercambio en los índices correspondientes a dichas entradas, con la restricción de que las entradas seleccionadas tengan elementos distintos.

Paso 5. Se itera todo el procedimiento anterior hasta que resulte un error nulo o se

excedan las iteraciones máximas, establecidas previamente.

Observación 5. Véase que este algoritmo es poco eficiente, dado que lo único en lo que se basa es en la prueba y error de la evaluación de matrices ortogonales tras permutar dos de sus entradas.

Asimismo, no guarda la información de la permutación anterior, pudiendo entrar en un bucle de permutaciones, iterando el proceso sin ninguna mejora. Estas afirmaciones son reflejadas en las siguientes simulaciones, donde se observa que, para dimensiones de la matriz pequeñas y un conjunto reducido de elementos, el algoritmo parece funcionar correctamente.

3.1.2. Conclusiones Particulares

Tras realizar varias ejecuciones del algoritmo con distintos ejemplos, modificándose todos los posibles parámetros de los orthogonal arrays, se observa que este primer algoritmo funciona eficientemente cuando las dimensiones de la matriz son pequeñas, el conjunto de símbolos no excede de 3 elementos y la fortaleza es, como mucho, 2.

Además, se ha observado que experimenta un buen comportamiento con orthogonal arrays binarias, es decir, construida a base de $0's$ y $1's$, con dimensiones relativamente grandes, excediendo las tres cifras. Esto puede ser significativo para aplicar dicho algoritmo en tareas donde se pretenda distribuir equitativamente decisiones lógicas de presencia o ausencia de alguna característica u objeto.

Asimismo, se puede contemplar como, una simple variación en la fortaleza del orthogonal array, añade mucha dificultad para encontrar la solución en unas iteraciones consideradas con dicho algoritmo, como se puede observar en los ejemplos 2 y 6 de la sección 3.1.3. Ambos ejemplos, que difieren solamente en una unidad de la fortaleza, cumplen con las condiciones necesarias para que se genere un orthogonal array, ya que satisfacen la propiedad 1 de la sección 2.2.2. Sin embargo, al ejecutar el algoritmo, se muestra una elevada tasa de error en el ejemplo 6, donde alrededor del 70 % de las ejecuciones no encuentran una solución, mientras que, en el otro caso, las iteraciones nunca exceden a la mitad de las permitidas en ninguna de las ejecuciones, generando siempre soluciones

Cabe destacar que, como se expone en los ejemplos 4 y 5, aumentar el número de elementos en el conjunto de símbolos, también dificulta en gran medida la construcción del orthogonal array, viéndose reforzada esta afirmación en las gráficas adjuntas, donde se observa un aumento del tiempo empleado y un número de iteraciones que prácticamente consigue llegar a las máximas establecidas.

3.1.3. Ejecuciones en Phyton del Algoritmo 1

- **Ejemplo 1:** $OA(8, 4, 2, 3)$

$$
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1
\end{array}
$$

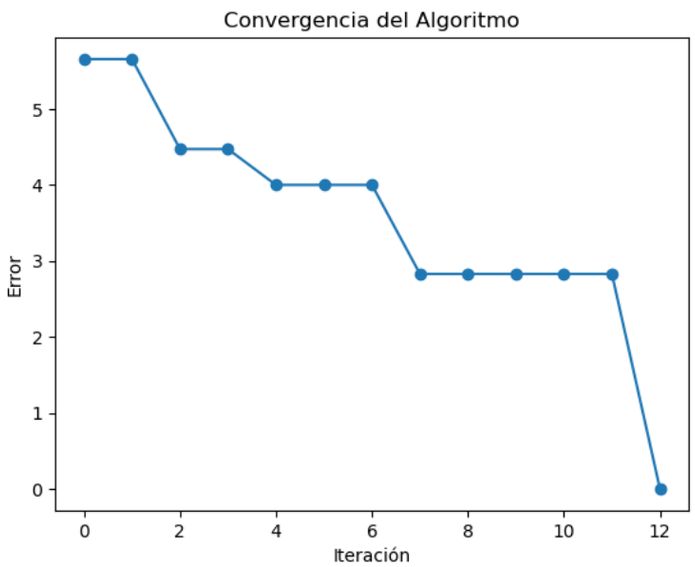


Figura 1: Error cometido en cada una de las iteraciones del algoritmo para hallar un orthogonal array de dimensiones 8×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza igual a 2.

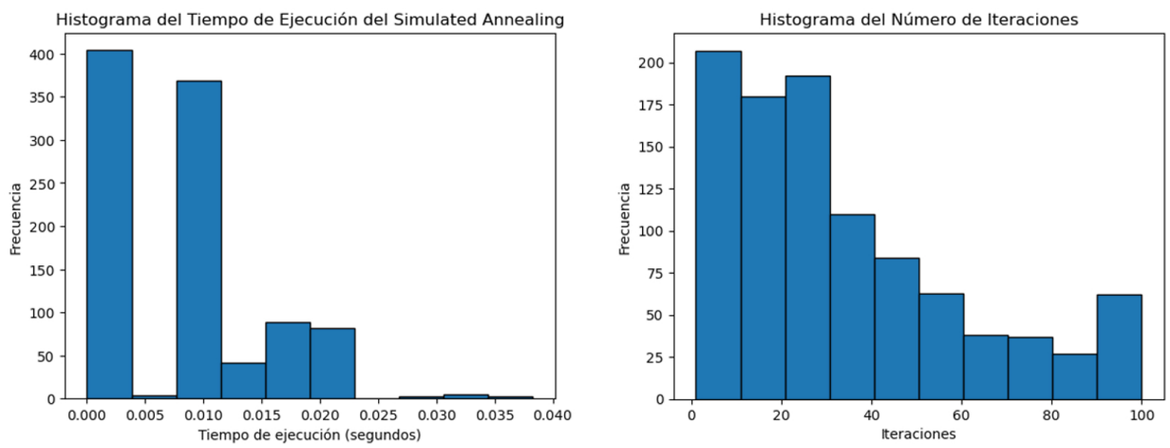


Figura 2: Tiempo de ejecución y número de iteraciones realizadas en 100 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 8×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza igual a 2.

- **Ejemplo 2:** $OA(16, 5, 2, 2)$ (Descrito en forma horizontal)

$$\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}$$

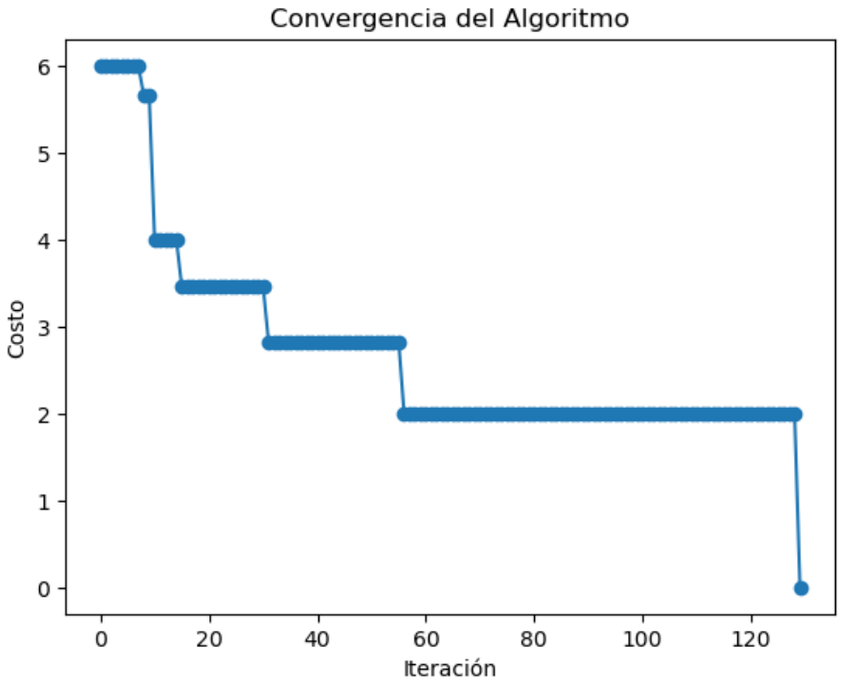


Figura 3: Error cometido en cada una de las iteraciones del algoritmo para hallar un orthogonal array de dimensiones 16×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza igual a 2.

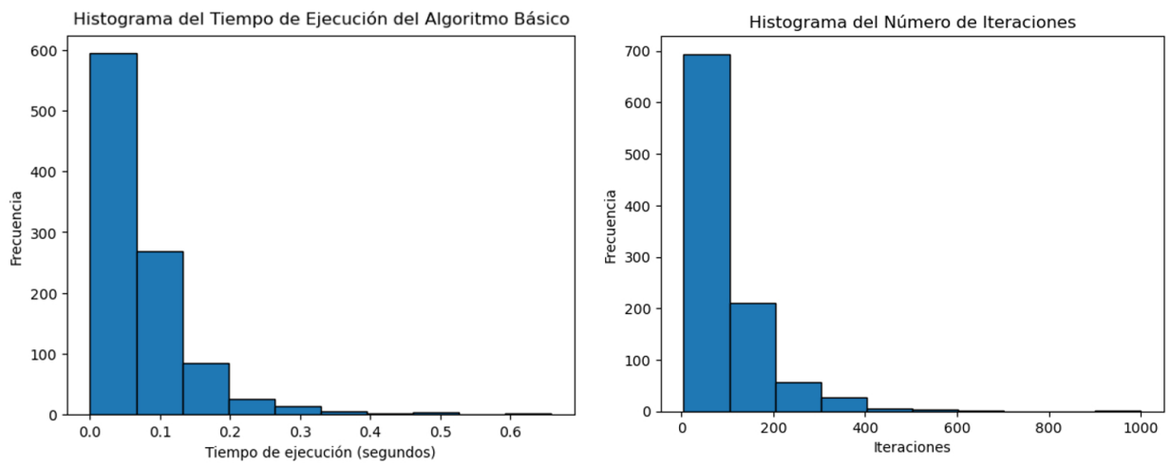


Figura 4: Tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza igual a 2.

- **Ejemplo 3:** $OA(60, 5, 2, 2)$ (Descrito en forma horizontal en dos bloques)

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

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

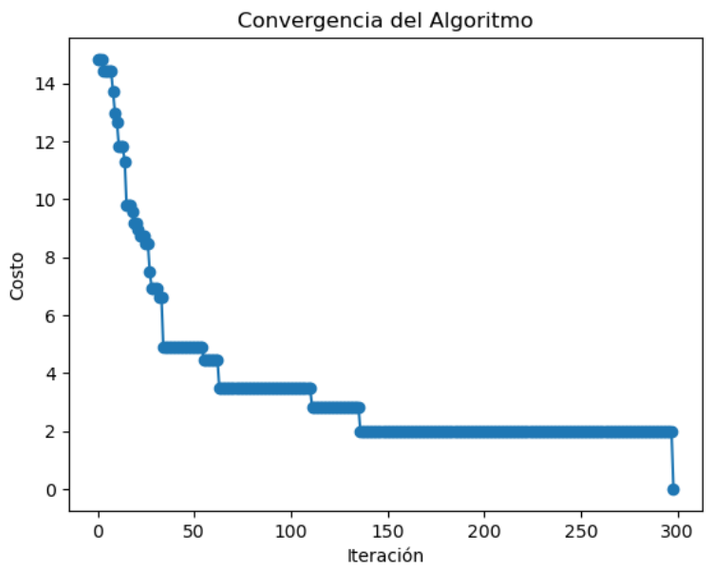


Figura 5: Error cometido en cada una de las iteraciones del algoritmo para hallar un orthogonal array de dimensiones 60×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza igual a 2.

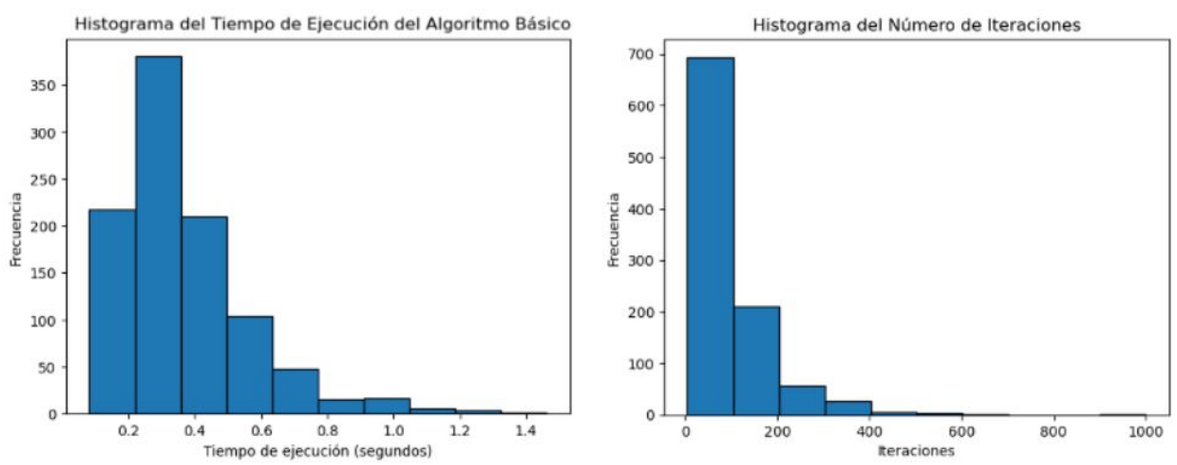


Figura 6: Tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 60×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 2.

- **Ejemplo 4:** $OA(64, 5, 4, 2)$ (Inviable hallarlo por este algoritmo)

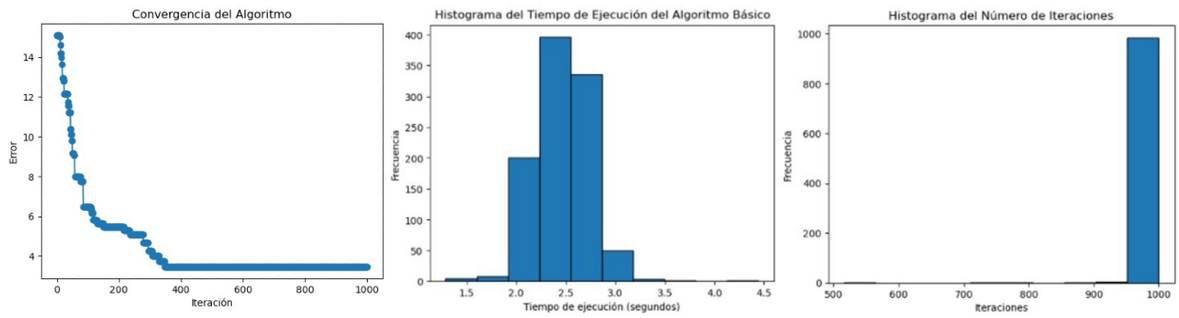


Figura 7: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 64×5, con elementos pertenecientes a $S = \{0, 1, 2, 3\}$ y de fortaleza 2.

- **Ejemplo 5:** $OA(16, 5, 4, 2)$ (Inviable hallarlo por este algoritmo)

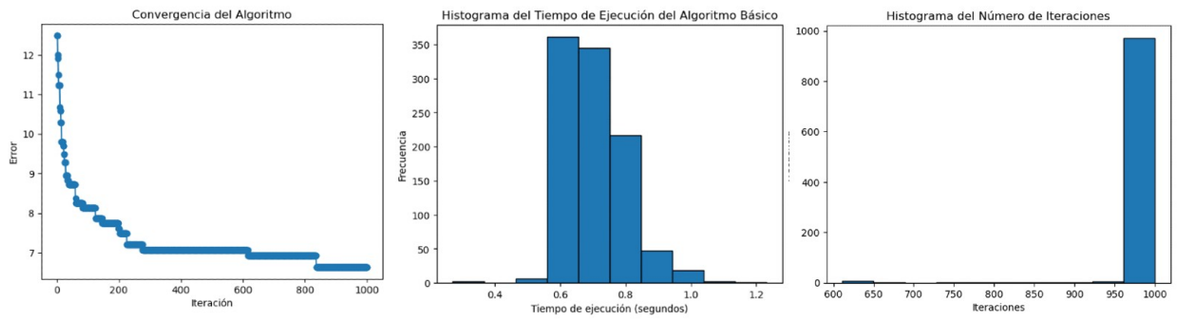


Figura 8: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×5, con elementos pertenecientes a $S = \{0, 1, 2, 3\}$ y de fortaleza 2.

- **Ejemplo 6:** $OA(16, 5, 2, 3)$ (La mayoría de ocasiones no halla ninguna solución)

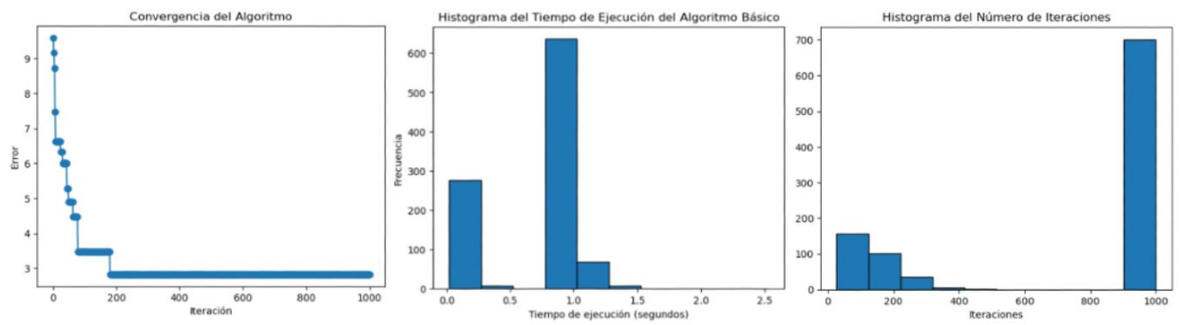


Figura 9: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 3.

3.2. Algoritmo 2: Búsqueda Local

Con la importante propiedad de la distribución uniforme por columnas que presentan los $OA's$, se pretende crear un nuevo algoritmo que también se base en ello.

A diferencia del primero, este tratará de permutar los elementos de muchas columnas al mismo tiempo e irá generando una solución tras evaluar posibles cambios en una matriz que satisfaga la propiedad de distribución uniforme de la columnas. Es decir, del mismo modo que el algoritmo 1 (3.1), se partirá de una matriz aleatoria de elementos del conjunto S que satisfaga la propiedad indicada y, mediante diferentes evaluaciones de permutaciones de las columnas de la matriz, condicionadas por fijar un cierto número de ellas, se escogerá aquella que minimice el error.

3.2.1. Descripción del Algoritmo 2

Paso 1. Se genera una matriz de tamaño $N \times k$ (dimensiones del orthogonal array objetivo) cuyas entradas sean elementos del conjunto S.

Ej: Generar $OA(8, 4, 2, 2)$. *Se crea una matriz aleatoria* 4×4 *de elementos de* $\{0, 1\}$.

$$\begin{matrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$$

Paso 2 Se examina la propiedad de la uniformidad en la distribución por columnas y se corrige, en caso de que sea necesario.

Se corrigen las filas.

$$\begin{matrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix} \quad \Longrightarrow \quad \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$$

Paso 3 Se calcula el "coste" o "error" de la matriz ortogonal. Para ello, se estudia el número de repeticiones de las combinaciones de elementos generadas al escoger cada una de las colecciones posibles de t columnas.

En el ejemplo, $\lambda = \frac{8}{2^2} = 2$, *por lo que cada dupla de elementos, fruto de seleccionar*

dos columnas de la matriz, debe a aparecer una única vez.

$$\begin{array}{cccc|cccc|cccc|cccc|cccc|cccc}
0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0\\
0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0\\
1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1\\
1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0\\
0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1\\
1&1&0&1&1&1&0&1&1&1&0&1&1&1&0&1&1&1&0&1&1&1&0&1
\end{array}$$

$$\text{Err} = 0 \qquad \text{Err} = 0 \qquad \text{Err} = 3 \qquad \text{Err} = 0 \qquad \text{Err} = 0 \qquad \text{Err} = 4$$

$$\text{Error Total Cometido} = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$

Paso 4. Si el error generado es distinto de 0, se evalúan los errores de k matrices que serán construidas tras fijar cada una de las k diferentes columnas y permutar aleatoriamente los elementos de las restantes.

$$\begin{array}{cccc|cccc|cccc|cccc}
0&1&1&1&1&1&1&0&1&0&1&0&0&0&0&0\\
1&1&1&0&1&1&0&1&1&1&1&1&1&1&1&1\\
0&1&0&0&0&0&1&1&1&0&1&0&0&0&0&0\\
0&1&1&0&0&1&1&0&0&0&0&1&1&0&1&0\\
1&0&0&1&1&0&0&1&1&1&0&1&0&1&1&1\\
1&0&0&1&0&0&0&0&0&0&1&0&1&1&1&0\\
0&0&0&1&1&0&0&0&0&1&0&1&1&1&0&1\\
1&0&1&0&0&1&1&1&0&1&0&0&0&0&0&1
\end{array}$$

$$\text{Err} = 8 \qquad \text{Err} = 4\sqrt{2} \qquad \text{Err} = 8 \qquad \text{Err} = 8$$

Paso 5. Se escoge aquella que minimice el error (si hay varias se escoge una cualquiera) y se guarda la columna que se ha fijado correspondientemente. Se realiza una segunda iteración pero esta vez se fijan dos columnas: la del paso anterior y una distinta:

$$\begin{array}{cccc|cccc|cccc}
1&1&0&0&0&1&1&1&0&1&0&0\\
1&1&1&0&1&1&0&0&0&1&0&1\\
0&0&1&0&0&0&1&1&0&0&0&1\\
0&1&0&1&1&1&1&0&1&1&1&0\\
1&0&0&1&1&0&0&0&1&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&1&0&0&0&1&1&0&1&0\\
0&1&1&1&1&1&1&1&1&1&1&1
\end{array}$$

$$\text{Err} = 0 \qquad \text{Err} = 8 \qquad \text{Err} = 8$$

Paso 6 En el caso en el que ninguno de los errores resultase ser 0, se volvería a operar siguiendo la misma idea pero ahora con una nueva columna fijada. Se itera este procedimiento hasta obtener un error nulo o completar k columnas fijadas de la matriz, donde el algoritmo resultará ineficiente para dicho ejemplo.

3.2.2. Ejecuciones en Phyton del Algoritmo 2

Como se puede observar en los ejemplos que se muestran a continuación, se puede conjeturar que el algoritmo es poco eficiente y en numerosas ocasiones no alcanza una solución factible. La justificación de esto puede recaer en la aleatoriedad que supone este algoritmo y al poco margen de análisis de las matrices.

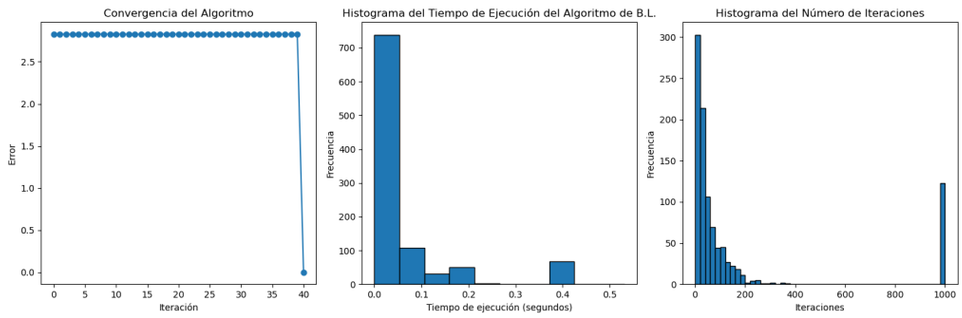


Figura 10: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 8×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 3.

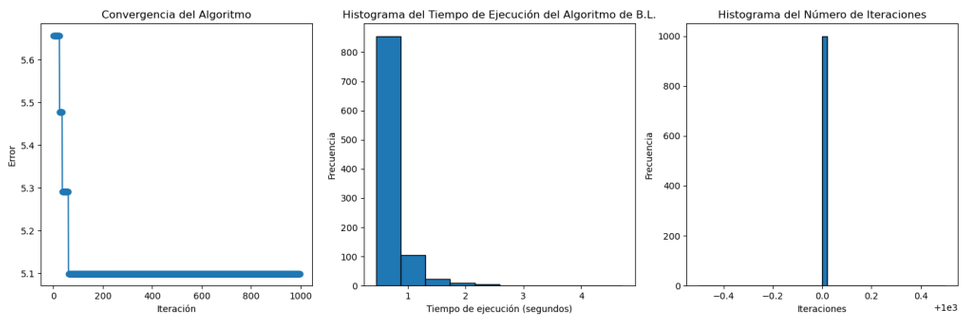


Figura 11: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 4.

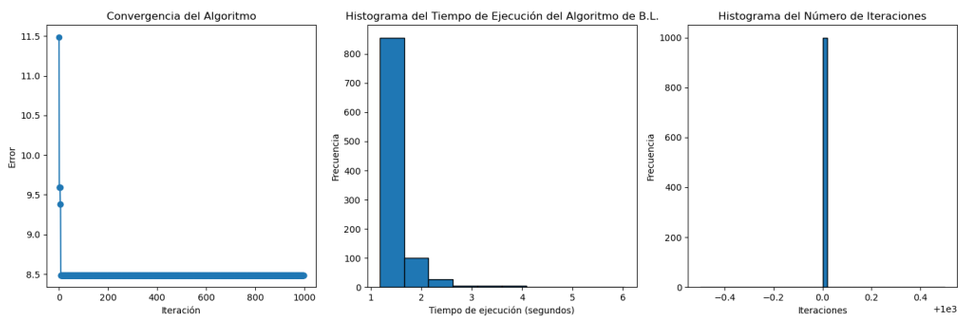


Figura 12: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 32×5, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 3.

3.3. Algoritmo 3: Genético con Mutación

Por último, antes de proseguir con el modelo de optimización lineal, se ha creado un algoritmo genético que profundice mejor en el campo de soluciones de las matrices.

Bajo este contexto, se ha desarrollado un algoritmo genético específico que genera una población inicial de matrices con elementos distribuidos uniformemente y, a través de generaciones sucesivas, busca optimizar la ortogonalidad de las matrices. La idea central es simular el proceso evolutivo para encontrar una configuración de niveles que cumpla con las propiedades de un $OA(N, k, s, t)$.

3.3.1. Descripción del Algoritmo 3

- **Paso 1.** El proceso comienza con la generación de una población inicial de matrices de tamaño $N \times k$ con símbolos en el conjunto usual que se ha empleado S. Cada matriz se ajusta para asegurar que cada columna contenga los elementos distribuidos uniformemente, es decir, cada símbolo aparece N/s veces por columna. Este ajuste inicial es crucial para garantizar una distribución de niveles balanceada en la población inicial.

- **Paso 2.** A continuación, el algoritmo entra en un ciclo iterativo donde, en cada generación, se crean nuevos individuos a partir de la combinación de columnas de dos matrices progenitoras seleccionadas aleatoriamente.

  El cruce se realiza dividiendo las columnas en dos partes de forma aleatoria y combinando las columnas de las dos matrices progenitoras para generar dos matrices hijas. Además de ello, se implementa una **probabilidad de mutación**, que se traduce en que las columnas, correspondientes a la segundas partes de cada uno de los hijos, permutan sus elementos.

- **Paso 3.** Una vez generadas las nuevas matrices, se evalúa su calidad utilizando una función de error que mide la desviación de la ortogonalidad deseada. Tras ordenar todas las matrices (generación anterior y actual) de menor a mayor, se seleccionan todas aquellas que estén en la primera mitad del grupo, definiendo la generación que sobrevive.

- **Paso 4.** Este proceso se repite hasta que se encuentra una matriz con un error de ortogonalidad igual a cero, indicando que se ha encontrado un OA válido, o hasta que se cumple un criterio de parada predefinido.

Este enfoque basado en algoritmos genéticos ofrece varias ventajas. En primer lugar, permite explorar un espacio de soluciones amplio y diverso, aumentando las probabilidades de encontrar configuraciones óptimas. En segundo lugar, la combinación de columnas de diferentes progenitores introduce variabilidad genética en la población, lo que ayuda a evitar el estancamiento en óptimos locales. Finalmente, la capacidad de ajustar y evaluar iterativamente las soluciones facilita la convergencia hacia $OA's$ que cumplen con las propiedades deseadas.

3.3.2. Ejecuciones en Phyton del Algoritmo 3

En este caso, a diferencia de los algoritmos anteriores, la convergencia a una solución factible es mucho más rápida, en cuanto a número de iteraciones, fruto de la gran cantidad de distintas matrices que se generan en cada una de las generaciones.

No obstante, tras varias ejecuciones se ha observado que, al igual que ocurre con los generados anteriormente, el algoritmo carece de eficiencia a medida que aumentan los parámetros del orthogonal array.

A continuación se muestran algunos de los ejemplos que se han ejecutado gracias al software de Phyton.

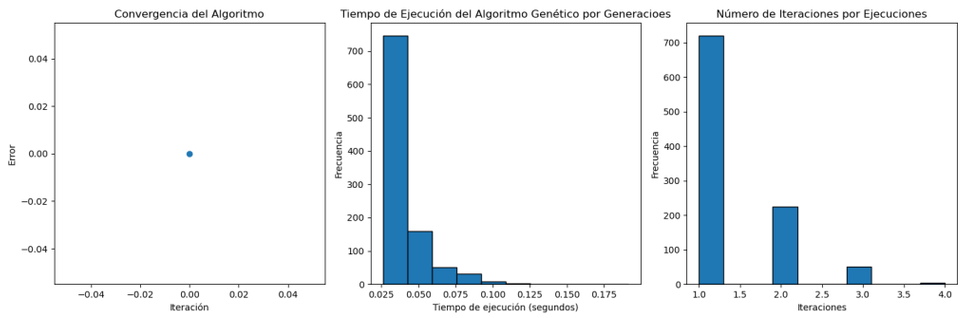


Figura 13: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 8×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 2.

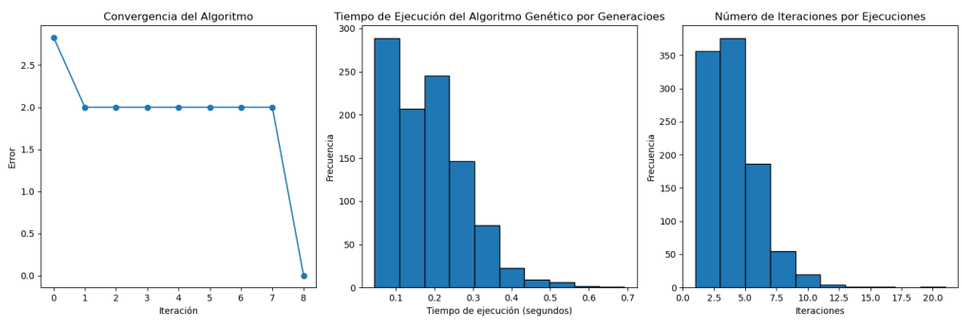


Figura 14: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 2.

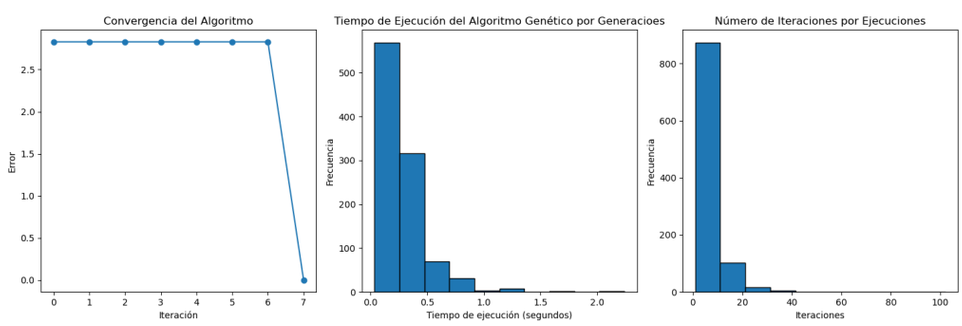


Figura 15: Convergencia en un aplicación del algoritmo, tiempo de ejecución y número de iteraciones realizadas en 1000 ejecuciones del algoritmo para hallar un orthogonal array de dimensiones 16×4, con elementos pertenecientes a $S = \{0, 1\}$ y de fortaleza 3.

4. Optimización Lineal Entera para Orthogonal Arrays

Dejando de lado los algoritmos heurísticos y meta-heurísticos, se pretenden abordar el problema de la generación de los orthogonal arrays mediante la programación lineal entera.

Del mismo modo que se construye el modelo de optimización entera para el problema del viajante de comercio en la sección 2.1.2, se puede desarrollar un modelo similar para orthogonal arrays. Ambos problemas comparten la necesidad de encontrar una configuración óptima que satisfaga un conjunto de restricciones complejas. Mientras que el TSP busca la ruta más corta que visita cada ciudad exactamente una vez, el problema de los OAs busca una disposición de niveles que garantice la distribución uniforme y la ortogonalidad. A través de una formulación adecuada, es posible adaptar técnicas y herramientas del TSP para resolver el problema de construcción de OAs.

Las variables no significarán lo mismo y las restricciones adquieren un mayor grado de complejidad, al igual que la función objetivo. Sin embargo, tras exponer el modelo, se podrán analizar que, en su naturaleza, ambos modelos están diseñados de la misma manera.

El modelo de optimización lineal se basa en la definición de variables de decisión que indican la asignación de niveles en cada celda de la matriz. Además, se introducen variables auxiliares para facilitar la representación de las combinaciones de niveles en las columnas seleccionadas. Las restricciones del modelo aseguran que cada nivel aparezca el número adecuado de veces en cada columna y que todas las combinaciones de niveles aparezcan con la frecuencia deseada en las selecciones de columnas especificadas por la fortaleza t

4.1. Construcción del Modelo

La idea es formular el problema de encontrar una matriz A que sea un $OA(N, k, s, t)$ como un problema de optimización lineal entera, donde se busca permutar y ajustar las filas y columnas de dicha matriz para cumplir con las propiedades ortogonales. El primer paso es definir las variables de decisión:

$$x_{ij}^{l} = \begin{cases} 1, & \text{si el elemento } a_{ij} \text{ de } A \text{ es igual a } l \\ 0, & \text{en caso contrario} \end{cases} \tag{4.1}$$

Obsérvese que $i = 1, \cdots, N$, $j = 1, \cdots, k$ y $l = 1, \cdots, s$. Con estas variables, se puede pensar en una función objetivo que minimice la diferencia del número de repeticiones en el que aparecen las $t-tuplas$ de cada una de las submatrices de t columnas del orthogonal array, con respecto al índice teórico $\lambda = \frac{N}{s^t}$. Supóngase que se tienen todas las posibles combinaciones de t-filas (esto es t-tuplas de elementos de $K = \{1, \cdots, k\}$ sin importar el orden) almacenadas en C, y todas las posibles t-tuplas de elementos de S en L $(l_1 l_2 \cdots, l_t,\ l_i \in S,\ \forall i)$:

$$\text{mín} \sum_{c \in C} \sum_{l \in L} \left| \sum_{i=1}^{N} x_{ij_1}^{l_1} \cdot x_{ij_2}^{l_2} \cdots x_{ij_t}^{l_t} - \lambda \right| \tag{4.2}$$

Dicha expresión refleja lo siguiente; se escoge una colección de t columnas y t elementos de S (pueden repetirse los símbolos) y se observa cuántas veces aparece por filas recorriendo

todos los posibles valores que puede tomar i. Tras esto, se calcula la diferencia con respecto a λ. Esta función, al optimizarse, debe resultar 0, dado que las ejecuciones en cada submatriz de t columnas, deben estar repetidas un número de λ veces.

Una vez fijada la función objetivo, al igual que con el TSP, es necesario imponer las respectivas restricciones del modelo para que se satisfagan las propiedades características de los orthogonal arrays.

- **Restricción 1. Único elemento en cada posición.**

Dado que, para cada posición (i, j) de la matriz, existen s posibles variables que pueden tomar el valor 1 e indicar el elemento que ocupa dicho lugar, es importante determinar que solo una de ellas puede ser distinta de 0. Con ello, la restricción que impone la unicidad de elementos en cada entrada de la matriz es:

$$\sum_{l=1}^{s} x_{ij}^{l} = 1, \quad i = 1, \cdots, N \; j = 1, \cdots, k \tag{4.3}$$

- **Restricción 2. Uniformidad en la distribución por columnas.**

Tras lo expuesto en las secciones teóricas del documento, se puede rescatar la condición de distribución uniforme de los símbolos en las columnas de la matriz; los elementos de S deben aparecer un número de N/s veces en cada una de las columnas. De este modo, la restricción que impone la uniformidad de las distribuciones de los símbolos es:

$$\sum_{i=1}^{N} x_{ij}^{l}, \quad j = 1, \cdots, k \; l = 1, \cdots, s \tag{4.4}$$

- **Restricción 3. Ortogonalidad de las *t*-tuplas.**

Aún con todo ello, es necesario imponer la peculiar característica de los orthogonal arrays; la repetición de las t-tuplas.

Bajo estas notaciones, una forma de poder expresar la necesidad de la ortogonalidad de las t-tuplas es mediante un producto de las variables de decisión:

$$\sum_{i=1}^{N} x_{i_{j_1}}^{l_1} \cdot x_{i_{j_2}}^{l_2} \cdots x_{i_{j_t}}^{l_t} = \lambda, \quad (j_1, \cdots, j_t) \in C, (l_1, \cdots, l_t) \in L \tag{4.5}$$

Es decir, fijándose una combinación de t columnas distintas de la matriz y una colección de t símbolos de S (permitiendo repeticiones), se evalúa el producto de las variables que indican si el símbolo l_i está en la columna j_i, en cada una de las filas; si este producto es 1, la combinación existe en la fila correspondiente de dicha submatriz; en caso contrario, al menos un elemento no coincide, imposibilitando esa combinación. Además, impone que se deben dar tantas combinaciones, en la submatriz generada, como indique el índice λ.

4.1.1. Modelo Preliminar.

El modelo que recoge toda la información explicada es el siguiente:

$$
\begin{aligned}
\text{mín} \quad & \sum_{c \in C} \sum_{l \in L} \left| \sum_{i=1}^{N} x_{ij_1}^{l_1} \cdot x_{ij_2}^{l_2} \cdots x_{ij_t}^{l_t} - \lambda \right| \\
s.a. \quad & \sum_{l=1}^{s} x_{ij}^{l} = 1, && i = 1, \cdots, N \quad j = 1, \cdots, k \\
& \sum_{i=1}^{N} x_{ij}^{l} = \frac{N}{s}, && j = 1, \cdots, k \quad l = 1, \cdots, s \\
& \sum_{i=1}^{N} x_{i_{j_1}}^{l_1} \cdot x_{i_{j_2}}^{l_2} \cdots x_{i_{j_t}}^{l_t} = \lambda, && (j_1, \cdots, j_t) \in C, \quad (l_1, \cdots, l_t) \in L \\
& x_{ij}^{l} = \{0, 1\}, && i = 1, \cdots, N \quad j = 1, \cdots, k, \quad l = 1, \cdots, s
\end{aligned}
\tag{4.6}
$$

En una primera instancia, puede parecer asequible abordar el problema con esta notación, sin embargo, obsérvese que no es un modelo correcto para la optimización lineal, dado que las restricciones no cumplen con las propiedades lineales.

Un objetivo previo a este estudio era el buscar cómo modelizar, mediante optimización, la generación de orthogonal arrays, consiguiéndose en esta sección. El nuevo objetivo es ahora encontrar el modelo de optimización lineal asociado.

Se observa que, las restricciones descritas en (4.3) y (4.4), que imponen la unicidad de elementos en cada posición de la matriz y la uniformidad en las distribuciones de los símbolos por columnas, respectivamente, son ambas lineales. Por tanto, estas restricciones no se verán obligadas a modificarse.

Por otro lado, la que genera más controversia en el planteamiento del modelo lineal es la tercera restricción (4.5), debido a que se sustenta en el sumatorio del **producto** de t variables de decisión. Una forma de modificar esta restricción, para transformarla en una lineal, pasa por declarar nuevas variables auxiliares que servirán para su linealización:

$$
z_{i,c}^{l} = \begin{cases} 1, & \text{si } l \in L \text{ está en la fila } i \text{ de la submatriz de columnas } c \in C \\ 0, & \text{en caso contrario} \end{cases}
\tag{4.7}
$$

Es decir, esta variable, al igual que las de decisión descritas al principio de esta sección, es de carácter binaria, indicando la presencia o ausencia de las t-tuplas de elementos en cada una de las filas de las posibles combinaciones de t columnas de la matriz que generará el orthogonal array.

Bajo esta hipótesis, se puede observar que dichas variables guardan una estrecha relación con las x_{ij}^{l}, del modo que:

$$
z_{ic}^{l} = 1 \Longleftrightarrow x_{ij_1}^{l_1} = x_{ij_2}^{l_2} = \cdots = x_{ij_t}^{l_t} = 1, \quad c = (j_1, \cdots, j_t) \in C, \ l = (l_1, \cdots, l_t) \in L \tag{4.8}
$$

Como es una variable binaria que solo se puede activar cuando todas las variables que indican la posición de los elementos seleccionados, en las columnas elegidas, adquieren el valor 1, o, dicho de otro modo, tomará el valor 0, cuando alguna de estas valga 0, se define el siguiente conjunto de restricciones:

$$z_{i,j_1,\cdots,j_t}^{l_1,\cdots,l_t} \leqslant x_{ij_r}^{l_r}, \quad r = 1,\cdots,t, \quad i = 1,\cdots,N, \quad (j_1,\cdots,j_t) \in C, \quad (l_1,\cdots,l_t) \in L \tag{4.9}$$

Por simplificar la notación, se usará la siguiente restricción que reduce el número de índices empleados:

$$z_{i,c}^{l} \leqslant x_{ij_r}^{l_r}, \quad r = 1,\cdots,t, \quad i = 1,\cdots,N, \quad c \in C, \quad l \in L \tag{4.9*}$$

No obstante, hace falta imponer además que, si resulta que todas las variables que declaran la presencia de los elementos en una fila toman el valor 1, la variable z_{ic}^{l}, asociada, deberá valer 1 y no 0. Para ello, se define la siguiente restricción:

$$z_{i,j_1,\cdots,j_t}^{l_1,\cdots,l_t} \geqslant \sum_{m=1}^{t} x_{ij_m}^{lm} - (t-1), \quad i = 1,\cdots,N, \quad (j_1,\cdots,j_t) \in C, \quad (l_1,\cdots,l_t) \in L \tag{4.10}$$

Nuevamente, se reescribirá esta restricción para acortar los índices utilizados:

$$z_{i,c}^{l} \geqslant \sum_{m=1}^{t} x_{ij_m}^{lm} - (t-1), \quad i = 1,\cdots,N, \quad c \in C, \quad l \in L \tag{4.10*}$$

De esta forma, la restricción (4.5) puede ser esxpresada, bajo la notación de estas nuevas variables auxiliares, como:

$$\sum_{i=1}^{N} z_{i,j_1,\cdots,j_t}^{l_1,\cdots,l_t} = \lambda, \quad (j_1,\cdots,j_t) \in C, \quad (l_1,\cdots,l_t) \in L \tag{4.11}$$

$$\sum_{i=1}^{N} z_{i,c}^{l} = \lambda, \quad c \in C, \quad l \in L \tag{4.11*}$$

Además, gracias a esta nueva notación, la función objetivo descrita en (4.2), se puede reescribir en función de estas nuevas variables auxiliares, suavizando su carácter lineal y convirtiéndola en una de mejor manejo:

$$\text{mín} \quad \sum_{c \in C} \sum_{l \in L} \left| \sum_{i=1}^{N} z_{i,c}^{l} - \lambda \right| \tag{4.12}$$

4.1.2. Modelo Definitivo

De esta forma el modelo completo de optimización lineal entera, que busca generar orthogonal arrays, se describe como se expone a continuación:

$$
\begin{aligned}
\text{mín} \quad & \sum_{c \in C} \sum_{l \in L} \left| \sum_{i=1}^{N} z_{i,c}^{l} - \lambda \right| & & \\
s.a. \quad & \sum_{l=1}^{s} x_{ij}^{l} = 1, & & i = 1, \cdots, N \quad j = 1, \cdots, k \\
& \sum_{i=1}^{N} x_{ij}^{l} = \frac{N}{s}, & & j = 1, \cdots, k \quad l = 1, \cdots, s \\
& z_{i,c}^{l} \leqslant x_{ij_r}^{l_r}, & & r = 1, \cdots, t \quad i = 1, \cdots, N \quad c \in C \quad l \in L \\
& z_{i,c}^{l} \geqslant \sum_{m=1}^{t} x_{ij_m}^{lm} - (t-1), & & i = 1, \cdots, N \quad c \in C, \quad l \in L \\
& \sum_{i=1}^{N} z_{i,c}^{l} = \lambda, & & c \in C \quad l \in L \\
& x_{ij}^{l} = \{0, 1\}, & & i = 1, \cdots, N \quad j = 1, \cdots, k \quad l = 1, \cdots, s \\
& z_{i,c}^{l} = \{0, 1\}, & & i = 1, \cdots, N \quad c \in C, \quad l \in L
\end{aligned}
\tag{4.13}
$$

Consideraciones del Modelo:

- La complejidad del modelo es muy elevada, dado el gran número de restricciones que se deben imponer, que aumentan conforme lo hacen los parámetros del OA. En general, se debe formalizar un número de restricciones igual a:

$$
\begin{aligned}
\text{Nº Restr.} &= n \cdot k + k \cdot s + t \cdot N \cdot |C| \cdot |L| + N \cdot |C| \cdot |L| + |C| \cdot |L| + N \cdot k \cdot s + N \cdot |C| \cdot |L| \\
&= (n + (1+s)N) \cdot k + (1 + (2+t)N) \cdot |C||L| \\
&= (n + (1+s)N) \cdot k + (1 + (2+t)N) \cdot \binom{k}{t} s^t
\end{aligned}
$$

- En busca de reducir la complejidad del algoritmo, se puede considerar la función objetivo que minimice la suma de las variables $z_{i,c}^{l}$:

$$
\text{mín} \quad \sum_{i=1}^{N} \sum_{c \in C} \sum_{l \in L} z_{ic}^{l} \tag{4.14}
$$

  De esta forma, ya que se está minimizando, se establecerán todas las z_{ic}^{l} iguala a 0 y, al añadir las restricciones, algunas estarán forzadas a valer 1. También es válido imponer una función objetivo nula y encontrar las soluciones mediante las restricciones.

- Si se impone la función objetivo descrita en 4.14, la restricción (4.9*) es innecesaria, dado que las variables auxiliares z_{ic}^{l} valdrán cero si no son forzadas a valer 1, cuya función recae en las restricciones sobre las variables x_{ij}^{l} y la que las relaciona con z_{ic}^{l} (4.10*)

4.2. Resultados con Phyton

Ejemplos con fortaleza $t = 2$

- $OA(25, 5, 5, 2)$ (Expuesto en forma horizontal)

$$\begin{array}{ccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 & 4 \\
0 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 4 \\
0 & 1 & 2 & 3 & 4 & 1 & 4 & 3 & 0 & 2 & 2 & 3 & 1 & 4 & 0 & 3 & 0 & 4 & 2 & 1 & 4 & 2 & 0 & 1 & 3 \\
0 & 1 & 2 & 3 & 4 & 2 & 3 & 1 & 4 & 0 & 3 & 0 & 4 & 2 & 1 & 4 & 2 & 0 & 1 & 3 & 1 & 4 & 3 & 0 & 2 \\
0 & 1 & 2 & 3 & 4 & 3 & 0 & 4 & 2 & 1 & 4 & 2 & 0 & 1 & 3 & 1 & 4 & 3 & 0 & 2 & 2 & 3 & 1 & 4 & 0
\end{array}$$

- $OA(81, 4, 3, 2)$ (Expuesto en forma horizontal y en tres bloques de 27 ejecuciones)

$$\begin{array}{ccccccccccccccccccccccccccc}
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

$$\begin{array}{ccccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

$$\begin{array}{ccccccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

- $OA(50, 5, 5, 2)$ (Expuesto en forma horizontal y en dos bloques de 25 ejecuciones)

$$\begin{array}{ccccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 2 & 0 & 0 & 1 & 1 & 1 & 3 & 1 & 2 & 1 & 1 & 1 & 1 & 2 & 2 \\
0 & 1 & 0 & 1 & 2 & 2 & 3 & 0 & 4 & 0 & 0 & 3 & 4 & 1 & 1 & 2 & 0 & 2 & 1 & 3 & 4 & 4 & 3 & 2 & 0 \\
0 & 0 & 3 & 4 & 2 & 1 & 4 & 2 & 1 & 4 & 0 & 3 & 2 & 3 & 0 & 1 & 4 & 2 & 1 & 3 & 4 & 0 & 1 & 4 & 3 \\
0 & 2 & 2 & 1 & 4 & 3 & 4 & 1 & 1 & 0 & 3 & 3 & 0 & 3 & 4 & 2 & 2 & 2 & 1 & 1 & 3 & 4 & 0 & 4 & 4 \\
0 & 1 & 3 & 3 & 2 & 4 & 4 & 4 & 1 & 2 & 3 & 2 & 0 & 0 & 4 & 3 & 0 & 2 & 2 & 0 & 1 & 3 & 1 & 0 & 1
\end{array}$$

$$\begin{array}{ccccccccccccccccccccccccc}
2 & 3 & 2 & 3 & 2 & 2 & 2 & 2 & 3 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
1 & 0 & 2 & 1 & 3 & 3 & 4 & 4 & 1 & 0 & 2 & 2 & 3 & 3 & 4 & 4 & 0 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\
4 & 1 & 3 & 3 & 2 & 0 & 1 & 2 & 2 & 1 & 0 & 4 & 2 & 1 & 3 & 0 & 2 & 2 & 0 & 0 & 3 & 1 & 3 & 4 & 4 \\
0 & 3 & 0 & 2 & 3 & 1 & 2 & 2 & 4 & 4 & 3 & 1 & 0 & 4 & 1 & 0 & 1 & 3 & 1 & 2 & 4 & 0 & 0 & 2 & 3 \\
4 & 4 & 1 & 2 & 3 & 2 & 0 & 4 & 1 & 2 & 1 & 3 & 3 & 0 & 4 & 2 & 1 & 0 & 0 & 4 & 3 & 3 & 4 & 1 & 2
\end{array}$$

Ejemplos con fortaleza $t = 3$

- $OA(16, 5, 2, 3)$ (Expuesto en forma horizontal)

$$\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1
\end{array}$$

- $OA(27, 4, 3, 3)$ (Expuesto en forma horizontal)

$$\begin{array}{ccccccccccccccccccccccccc}
1 & 2 & 2 & 1 & 1 & 1 & 2 & 0 & 0 & 1 & 2 & 2 & 1 & 2 & 0 & 0 & 2 & 0 & 0 & 1 & 1 & 0 & 0 & 2 & 0 & 2 & 1 \\
1 & 0 & 2 & 2 & 1 & 0 & 2 & 1 & 0 & 2 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 1 & 2 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 2 & 1 & 0 & 0 & 2 & 2 & 0 & 1 & 2 \\
2 & 2 & 2 & 1 & 1 & 0 & 1 & 1 & 2 & 0 & 0 & 2 & 0 & 1 & 0 & 1 & 0 & 0 & 2 & 2 & 2 & 1 & 0 & 0 & 2 & 1 & 1
\end{array}$$

- $OA(81, 4, 3, 3)$ (Expuesto en forma horizontal)

$$\begin{array}{ccccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 2 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 0 & 0 & 2 & 2 & 1 & 0 \\
0 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 0 & 1 & 2 & 2 & 2 & 2 & 1 & 0 & 2 & 1 \\
0 & 2 & 2 & 1 & 2 & 1 & 2 & 1 & 1 & 0 & 2 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 2 & 1 & 2 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$\begin{array}{ccccccccccccccccccccccccc}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 2 & 2 & 0 & 1 & 1 & 1 & 2 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 & 2 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 2 & 0 & 2 & 2 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 2 & 2 \\
1 & 2 & 1 & 0 & 1 & 2 & 1 & 0 & 0 & 1 & 1 & 0 & 2 & 0 & 0 & 2 & 1 & 0 & 2 & 0 & 1 & 0 & 2 & 0 & 1 & 2 & 2
\end{array}$$

$$\begin{array}{ccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 2 & 2 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 0 & 0 & 1 & 0 & 2 & 2 & 1 & 2 \\
0 & 2 & 0 & 1 & 2 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 2 & 1 & 0 & 0 & 1 & 0 & 1 & 2 \\
2 & 2 & 0 & 1 & 1 & 2 & 2 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 0 & 2 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 2 & 2
\end{array}$$

Ejemplos con fortaleza $t = 4$

- $OA(16, 4, 2, 4)$ (Expuesto en forma horizontal)

$$\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1
\end{array}$$

- $OA(81, 4, 3, 4)$ (Expuesto en forma horizontal)

$$
\begin{array}{cccccccccccccccccccccccc}
2 & 0 & 1 & 2 & 0 & 0 & 2 & 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 2 & 1 & 2 & 2 \\
2 & 1 & 1 & 1 & 2 & 0 & 1 & 2 & 2 & 2 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 2 & 2 & 2 & 1 & 1 \\
1 & 1 & 2 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 1 & 2 & 0 & 2 & 0 & 2 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 2 & 0 & 1 & 1 & 0 & 0 & 0 & 2 & 1 & 2 & 2 & 2 & 0 & 2 & 2 & 1 & 0 & 0 & 2 & 2 & 1 & 2 & 0 & 2
\end{array}
$$

$$
\begin{array}{cccccccccccccccccccccccc}
0 & 1 & 1 & 2 & 1 & 2 & 2 & 2 & 0 & 1 & 2 & 0 & 0 & 1 & 0 & 0 & 2 & 1 & 2 & 1 & 2 & 0 & 2 & 0 & 0 & 1 & 1 \\
2 & 1 & 2 & 2 & 2 & 0 & 0 & 0 & 1 & 0 & 2 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 2 & 1 & 2 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 2 & 0 & 2 & 1 & 2 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 2 & 0 & 0 & 1 & 1 & 1 & 2 & 0 & 2 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 2 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 & 2
\end{array}
$$

$$
\begin{array}{cccccccccccccccccccccccc}
2 & 1 & 1 & 2 & 0 & 2 & 0 & 1 & 1 & 2 & 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 0 & 1 & 2 & 2 & 1 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 2 & 1 & 2 & 0 & 2 & 1 & 2 & 1 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & 1 & 1 & 2 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 2 & 2 & 1 & 2 & 0 & 2 & 2 & 0 & 2 & 2 & 0 & 2 & 2 & 2 & 2 & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\
1 & 0 & 1 & 2 & 0 & 2 & 2 & 2 & 1 & 1 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 1 & 0 & 2 & 1 & 1 & 1 & 1 & 1 & 2
\end{array}
$$

Ejemplos con fortaleza $t = 5$

- $OA(16, 4, 2, 5)$ (Expuesto en forma horizontal)

$$
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

$$
\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1
\end{array}
$$

4.2.1. Conclusiones Particulares

Estos resultados muestran la mejoría en la capacidad de resolución que presenta el modelo de optimización lineal frente a los algoritmos heurísticos y meta-heurísticos explorados anteriormente, ya que se han generado diferentes matrices que satisfacen las condiciones de ser orthogonal arrays con los parámetros que se especifican en cada uno de los ejemplos.

Además de la precisión en la resolución de dicho problema, se ha observado una disminución del tiempo de ejecución en casi todos los ejemplos, siendo menos significativo en aquellos donde los parámetros del OA son más pequeños. No obstante, se han conseguido generar orthogonal arrays de dimensiones mayores en tiempos no muy elevados.

A continuación, se muestra una tabla donde se expone el alcance de dicho modelo lineal, indicando si genera un orthogonal array con los parámetros solicitados ("True" significa que es posible hallarlo mediante el modelo y, en caso contrario, se determina "False") y el tiempo que emplea en su búsqueda.

N	**k**	**s**	**t**	**Existencia**	**Tiempo**		**N**	**k**	**s**	**t**	**Existencia**	**Tiempo**
8	4	2	2	True	1.672 s		64	4	4	3	True	2.136 s
8	4	2	3	True	0.375 s		81	4	3	3	True	47.631 s
8	5	2	2	True	0.109 s		100	5	5	2	False	603.198 s
8	4	2	3	True	0.030 s		125	5	5	3	False	612.654 s
9	3	3	2	True	0.109 s		128	7	2	3	False	602.845 s
9	4	3	2	True	0.608 s		144	5	2	3	True	14.698 s
9	5	3	2	False	4.586 s		144	5	2	4	True	1.433 s
16	4	2	3	True	0.061 s		144	6	3	2	True	574.557 s
16	5	2	4	True	0.656 s		169	5	13	2	False	623.160 s
25	4	5	2	True	7.798 s		196	7	2	2	True	54.908 s
25	5	5	2	True	40.951 s		225	5	3	3	False	605.433 s
25	6	5	2	True	104.271 s		243	5	3	5	True	35.349 s
27	3	3	3	True	0.078 s		256	5	4	3	True	30.412 s
27	4	3	3	True	1.093 s		324	6	3	4	False	651.060 s
27	5	3	3	False	50.132 s		400	7	2	4	False	625.273 s
32	5	2	3	True	0.265 s		441	6	3	3	False	619.646 s
32	5	2	4	True	0.343 s		484	5	2	2	True	100.206 s
32	5	2	5	True	0.156 s		576	6	6	2	False	628.166 s
32	7	2	5	False	243.883 s		672	6	2	4	False	614.899 s

Cuadro 1: Efectividad del modelo para distintos parámetros de un orthogonal array y su tiempo de resolución (con cota máxima de 10 minutos)

Se puede observar cómo el modelo capta perfectamente un gran número de diversos orthogonal arrays, con una gran variedad en las dimensiones de la matriz, el tamaño del conjunto de símbolos y la fortaleza.

Además, el tiempo de resolución es bastante asequible en aquellos en los que se consigue dar una matriz factible. No obstante, cabe destacar que esta tabla solo muestra los resultados cuando el tiempo de ejecución es inferior a 10 minutos. Por dificultades técnicas, se ha optado por tener una cota de tiempo relativamente baja, pero no se descarta que, muchos de los casos donde se expone que no existe una matriz que cumpla con los parámetros asociados al orthogonal array, se pueda generar una solución si se precisa de un mejor sistema de computación.

5. Conclusiones y Trabajo a Futuro

En el presente trabajo de fin de máster, se han explorado y comparado tres enfoques diferentes para la generación de Orthogonal Arrays (OA): dos algoritmos genéticos, un algoritmo de búsqueda local, y un modelo de optimización lineal entera. Cada uno de estos métodos tiene sus propias fortalezas y debilidades, y se ha demostrado que el modelo de optimización lineal entera, programado en Python utilizando Gurobi, ofrece una mayor eficacia y un espectro más amplio de resultados comparado con los algoritmos heurísticos y metaheurísticos.

Los algoritmos genéticos, basados en la selección y combinación de individuos en una población para mejorar iterativamente la calidad de los OAs, han mostrado ser útiles para generar soluciones razonables en tiempos relativamente cortos. Sin embargo, su rendimiento decrece significativamente a medida que la complejidad del problema aumenta, especialmente cuando se requiere una mayor fortaleza del OA.

El modelo de optimización lineal entera, en contraste, ha mostrado una robustez y capacidad superiores para resolver problemas complejos de generación de OAs. Este modelo ha demostrado ser capaz de encontrar soluciones óptimas para una amplia gama de parámetros, incluyendo aquellos con altos niveles de fortaleza y números de símbolos elevados. Aunque el tiempo de cómputo puede ser considerable para instancias extremadamente grandes, la exactitud y la garantía de encontrar una solución óptima hacen que este enfoque sea el más recomendado.

A pesar de los éxitos alcanzados con el modelo de optimización lineal entera, existen múltiples vías para futuros trabajos que podrían ampliar y mejorar los resultados obtenidos en este TFM. Primero, la integración de técnicas híbridas que combinan la robustez de la optimización lineal con la flexibilidad y rapidez de los algoritmos genéticos y de búsqueda local podría ofrecer soluciones eficientes para problemas de muy gran escala que son actualmente inabordables mediante optimización lineal pura.

Además, explorar la paralelización y optimización del código para aprovechar completamente las arquitecturas de computación moderna podría reducir significativamente los tiempos de cómputo, haciendo que el modelo de optimización lineal sea aún más práctico para su uso en aplicaciones industriales y de investigación.

Finalmente, la aplicación de estos métodos en problemas reales en áreas como el diseño experimental, la criptografía, y la codificación de errores podría no solo validar los resultados teóricos obtenidos sino también proporcionar mejoras prácticas y directas en estos campos. La colaboración con expertos en estos dominios sería esencial para adaptar y refinar los algoritmos a las necesidades específicas de cada aplicación.

En resumen, este TFM ha sentado una base sólida en la generación de Orthogonal Arrays mediante varios enfoques algorítmicos, destacando la superioridad del modelo de optimización lineal. Sin embargo, existe un amplio campo para la mejora y expansión de este trabajo, asegurando así su relevancia y aplicabilidad futura.

6. Conclusions and Future Works

In this master's thesis, three different approaches for generating Orthogonal Arrays (OAs) were explored and compared: two genetic algorithms, a local search algorithm, and an integer linear optimization model. Each of these methods has its own strengths and weaknesses, and it was demonstrated that the integer linear optimization model, programmed in Python using Gurobi, offers greater efficiency and a broader spectrum of results compared to the heuristic and metaheuristic algorithms.

The genetic algorithms, which rely on the selection and combination of individuals in a population to iteratively improve the quality of the OAs, proved useful for generating reasonable solutions in relatively short times. However, their performance significantly decreases as the complexity of the problem increases, especially when higher strength OAs are required.

In contrast, the integer linear optimization model has shown superior robustness and capability in solving complex OA generation problems. This model has proven capable of finding optimal solutions for a wide range of parameters, including those with high strength levels and large numbers of symbols. Although computation time can be considerable for extremely large instances, the accuracy and guarantee of finding an optimal solution make this approach the most recommended. Despite the successes achieved with the integer linear optimization model, there are multiple avenues for future work that could extend and improve the results obtained in this thesis. First, integrating hybrid techniques that combine the robustness of linear optimization with the flexibility and speed of genetic and local search algorithms could offer efficient solutions for very large-scale problems that are currently unmanageable through pure linear optimization.

Furthermore, exploring the parallelization and optimization of the code to fully leverage modern computing architectures could significantly reduce computation times, making the linear optimization model even more practical for industrial and research applications.

Finally, applying these methods to real-world problems in areas such as experimental design, cryptography, and error-correcting codes could not only validate the theoretical results obtained but also provide practical and direct improvements in these fields. Collaboration with experts in these domains would be essential to adapt and refine the algorithms to the specific needs of each application.

In summary, this thesis has laid a solid foundation in the generation of Orthogonal Arrays through various algorithmic approaches, highlighting the superiority of the linear optimization model. However, there is ample room for improvement and expansion of this work, ensuring its future relevance and applicability.

7. Referencias Bibliográficas

1. Applegate, D. L. (2006). *The traveling salesman problem: a computational study* (Vol. 17). Princeton university press.

2. Bertsimas, D., & Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific.

3. Blum, C., & Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys (CSUR)*, 35(3), 268-308.

4. Chvátal, V. et. al (2010). Solution of a large-scale traveling-salesman problem. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, 7-28.

5. Conforti, M., et. al (2014). *Integer Programming*. Springer.

6. Glover, F., & Kochenberger, G. A. (2003). *Handbook of Metaheuristics*. Springer.

7. Gurobi Optimization, LLC. (2021). *Gurobi Optimizer Reference Manual*. Retrieved from https://www.gurobi.com/documentation/9.5/refman/index.html

8. Hedayat, A. S. et. al (1999). *Orthogonal Arrays: Theory and Applications*. Springer Science & Business Media.

9. Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press.

10. Michalewicz, Z., & Fogel, D. B. (2004). *How to Solve It: Modern Heuristics*. Springer.

11. Montgomery, D. C. (2012). *Design and Analysis of Experiments*. Wiley.

12. Mukerjee, R., & Wu, C. F. J. (2007). *A Modern Theory of Factorial Designs*. Springer.

13. Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and Combinatorial Optimization*. Wiley-Interscience.

14. Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall.

15. Python Software Foundation. (2021). *Python Language Reference, version 3.9*. Retrieved from https://www.python.org

16. R Core Team. (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org

17. Rao, C. R. (1947). Factorial Experiments Derivable from Combinatorial Arrangements of Arrays. *Journal of the Royal Statistical Society, Series B (Methodological)*, 9(1), 128-139.

18. Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer.

19. Schrijver, A. (1998). *Theory of Linear and Integer Programming.* Wiley-Interscience.

8. Apéndice 1. Script en Phyton

TRABAJO DE FIN DE MÁSTER - TSP & Orthogonals Arrays: A possible connection

Nuez Doreste, Daniel

07 de julio de 2024 **Librerías:**

[1]:
```
import numpy as np
import random
import time
import itertools
import pandas as pd
from itertools import combinations, product
from collections import Counter
import matplotlib.pyplot as plt
from gurobipy import Model, GRB, quicksum
```

Parámetros de Gurobi:

[ ]:
```
!pip install gurobipy
from gurobipy import *

params = {"WLSACCESSID" : '??????????-????-????-?????-??????????',␣
 ↪"WLSSECRET" : '?????????-????-????-????-????????', "LICENSEID" : ???????
 ↪???}
env = Env(params = params)
```

8.1. FUNCIONES GENERALES BÁSICAS

Función de Validación de Proporciones:

[2]:
```
def check_proportions(OA):
    N, k = OA.shape
    s = len(np.unique(OA))
    pp = N // s  #Proporción esperada de cada nivel en cada columna

    # Verificación de proporciones
    for j in range(k):
        col_counts = Counter(OA[:, j])
        for level in range(s):
            if col_counts[level] != pp :
                return False  #Si alguna proporción no se cumple, resulta␣
 ↪False
```

```
    return True  #Si todas las proporciones se cumplen, resulta True
```

Función de Uniformidad de Distribuciones:

[3]:
```
def unif_proportions(OA):
    N, k = OA.shape
    s = len(np.unique(OA))
    pp = N // s  #Proporción esperada de cada nivel en cada columna

    # Corrección de proporciones
    for j in range(k):
        col_counts = Counter(OA[:, j])

        # Listas para mantener el índice de filas con exceso y déficit de␣
↪cada nivel
        excess = {level: [] for level in range(s)}
        deficit = {level: [] for level in range(s)}

        #Se añaden los niveles en alguna de las listas, depediendo si␣
↪exceden o no llegan a la frecuencia pp
        for i in range(N):
            level = OA[i,j]
            if col_counts[level] > pp:  #Excede la proporcion
                excess[level].append(i)
            elif col_counts[level] < pp:  #No llegan a la proporción
                deficit[level].append(i)

        #Se realizan intercambios para equilibrar las proporciones
        for level in range(s):
            while col_counts[level] > pp:
                i_excess = excess[level].pop()

                # Encontrar un nivel con déficit para intercambiar
                for other_level in range(s):
                    if col_counts[other_level] < pp:
                        OA[i_excess, j] = other_level
                        col_counts[level] -= 1
                        col_counts[other_level] += 1
                        if col_counts[other_level] == pp:
                            break

    return OA
```

Función de Coste de Posibles OA:

[4]:
```
def Error(OA, t):
    N, k = OA.shape
    s = len(np.unique(OA))
    lambda_esp = N / s**t

    coste = 0

    #Se generan todas las combinaciones posibles de t columnas
    combinaciones = list(combinations(range(k), t))

    #Se iteran sobre todas las combinaciones posibles de t columnas
    for c in combinaciones:
        #Diccionario para contar las apariciones de cada combinación de␣
↪niveles
        count_dict = {tupla: 0 for tupla in product(range(s), repeat=t)}

        #Se cuentan las combinaciones observadas en las t columnas␣
↪seleccionadas
        for i in range(N):  # Iterar sobre todas las filas de la matriz
            tupla = tuple(OA[i, c])
            count_dict[tupla] += 1

        #Se calcula el costo u error basado en la desviación con el␣
↪índice lambda
        for count in count_dict.values():
            coste += (count - lambda_esp) ** 2  #Se usa la norma L2

    return coste**(1/2)
```

8.2. ALGORITMO 1: INTERCAMBIO DE ELEMENTOS

Función de Modificación por Vecino:

[5]:
```
def neighbor(OA):
    N, k = OA.shape
    new_OA = OA.copy()

    i1, i2 = random.sample(range(N), 2)    #Se seleccionan dos ejecuciones␣
↪distintas del OA
    j = random.choice(range(k))            #Se selecciona una columna␣
↪aleatoria
    while new_OA[i1,j] == new_OA[i2,j]:     #Si son elementos iguales, se␣
↪vuelven a seleccionar dos ejecucciones distintas
        i1, i2 = random.sample(range(N), 2)
```

```
    new_OA[i1, j], new_OA[i2, j] = new_OA[i2, j], new_OA[i1, j]    #Se␣
↪intercambian sus elementos

    return new_OA
```

Función para generar OA básica:

[6]:
```
def OA_basico(OA, t, nmax):    #Se modifica una matriz aleatoria "OA" para␣
↪que sea un orthogonal array con fortaleza "t"
    if not check_proportions(OA):    #Se distribuyen uniformemente los␣
↪valores por columnas
        OA = unif_proportions(OA)
    best_OA = OA.copy()               #Se guarda el OA como el mejor de␣
↪momento
    best_cost = Error(best_OA, t)     #Se guardan su error como el mejor␣
↪de momento
    cont = 0

    #Se genera un vector que guardará los errores o costes en cada␣
↪iteración
    costs = [best_cost]

    while best_cost > 0 and cont < nmax:    #Hasta que se alcancen las␣
↪iteraciones máximas o el error sea 0 se ejecuta
        new_OA = neighbor(best_OA)          #Se aplica la función␣
↪'neighbor'
        new_cost = Error(new_OA, t)         #Se calcula el nuevo error
        delta_E = new_cost - best_cost      #Se calcula la diferencia de␣
↪errores

        if delta_E < 0:              #Si el nuevo error es menor, se␣
↪ejecuta
            best_OA = new_OA         #Se actualiza el OA
            best_cost = new_cost     #Se actualiza el error
        costs.append(best_cost)    #Se añade el mejor error a 'costs' por␣
↪iteración
        cont += 1  #Se actualiza el contador de iteración

    return best_OA, best_cost, costs, cont
```

Funciones de Evaluación de Tiempo e Iteraciones:

[7]:
```
def tiempos_basico(OA,t,nmax,niter):   #Se genera un histograma con los␣
↪tiempos de ejecución de aplicarlo 'niter' veces
    execution_times = []
    for _ in range(niter):
        start_time = time.time()
        OA_basico(OA, t, nmax)
        end_time = time.time()
        execution_times.append(end_time - start_time)
    return execution_times
    #plt2.hist(execution_times, bins=10, edgecolor='black')
   #plt2.set_xlabel('Tiempo de ejecución (segundos)')
    #lt2.set_ylabel('Frecuencia')
    #lt2.set_title('Histograma del Tiempo de Ejecución del Algoritmo␣
↪Básico')
```

[8]:
```
def iteraciones_basico(OA,t,nmax,niter):   #Histograma de iteraciones␣
↪necesarias para hallar el OA al aplicarlo 'niter' veces
    execution_cont = []
    for _ in range(niter):
        cont = OA_basico(OA, t, nmax)[3]
        execution_cont.append(cont)
    return execution_cont
    #plt3.hist(execution_cont, bins=10, edgecolor='black')
    #plt3.set_xlabel('Iteraciones')
    #plt3.set_ylabel('Frecuencia')
    #plt3.set_title('Histograma del Número de Iteraciones')
```

[9]:
```
def Graficos(OA,t,nmax,niter):
    optimized_OA, optimized_cost, costs, cont = OA_basico(OA, t, nmax)
    fig, (plt1, plt2, plt3) = plt.subplots(1, 3, figsize=(15, 5))
    print(f"Optimized Cost: {optimized_cost}")
    print(f"Optimized OA: \n{optimized_OA}")
    plt1.plot(costs, '-o')
    plt1.set_xlabel('Iteración')
    plt1.set_ylabel('Error')
    plt1.set_title('Convergencia del Algoritmo')

    execution_times = tiempos_basico(OA,t,nmax,niter)
    plt2.hist(execution_times, bins=10, edgecolor='black')
    plt2.set_xlabel('Tiempo de ejecución (segundos)')
    plt2.set_ylabel('Frecuencia')
    plt2.set_title('Histograma del Tiempo de Ejecución del Algoritmo␣
↪Básico')
```

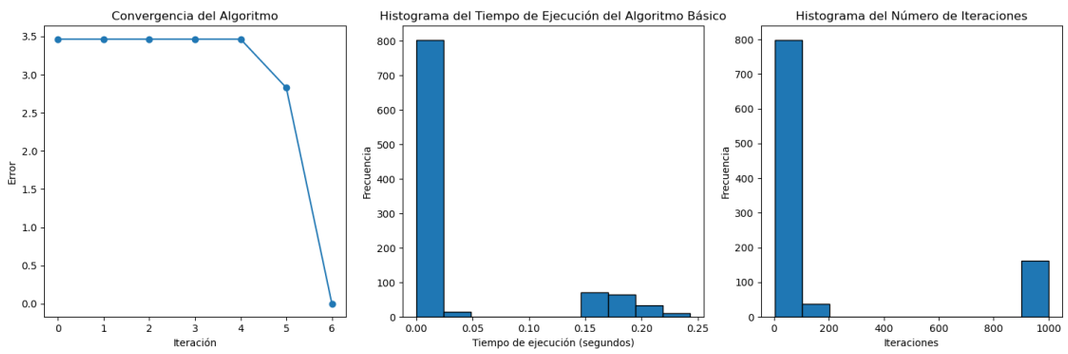
```
    execution_cont = iteraciones_basico(OA,t,nmax,niter)
    plt3.hist(execution_cont, bins=10, edgecolor='black')
    plt3.set_xlabel('Iteraciones')
    plt3.set_ylabel('Frecuencia')
    plt3.set_title('Histograma del Número de Iteraciones')

    plt.tight_layout()
    plt.show()
```

8.2.1. Ejemplos

- **Ejemplo 1:** $OA(8, 4, 2, 3)$

[57]:
```
N, k, s, t, nmax = 8, 4, 2, 3, 1000
OA1 = np.random.randint(0, s, size=(N, k))
Graficos(OA1,t,nmax,1000)
```

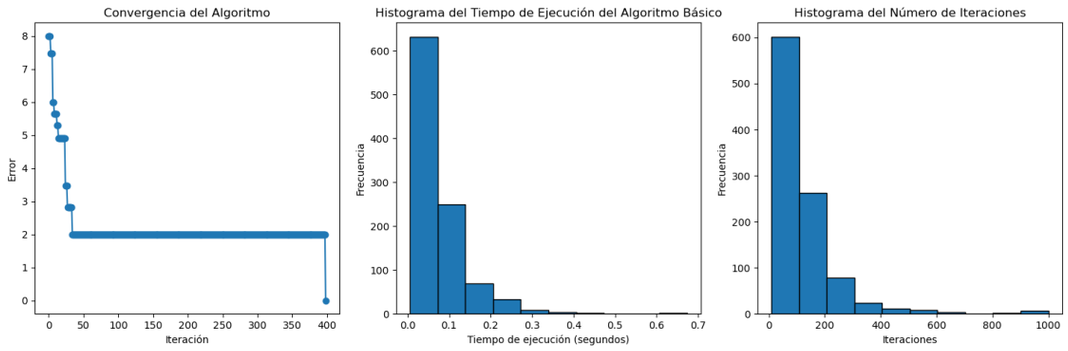
```
Optimized Cost: 0.0
Optimized OA:
[[1 0 1 0]
 [1 0 0 1]
 [0 1 1 0]
 [0 1 0 1]
 [1 1 1 1]
 [0 0 0 0]
 [1 1 0 0]
 [0 0 1 1]]
```



- **Ejemplo 2:** $OA(16, 5, 2, 2)$

[58]:
```
N, k, s, t, nmax = 16, 5, 2, 2, 1000
OA2 = np.random.randint(0, s, size=(N, k))
Graficos(OA2,t,nmax,1000)
```

```
Optimized Cost: 0.0
Optimized OA:
[[0 1 1 1 1]
 [1 1 0 0 0]
 [1 1 1 0 1]
 [1 0 1 1 0]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [1 1 1 1 1]
 [0 1 1 0 0]
 [0 0 1 0 0]
 [0 1 0 1 1]
 [0 1 0 1 0]
 [1 0 0 1 1]
 [0 0 0 0 1]
 [1 0 1 0 1]
 [1 0 0 1 0]
 [0 0 1 1 0]]
```



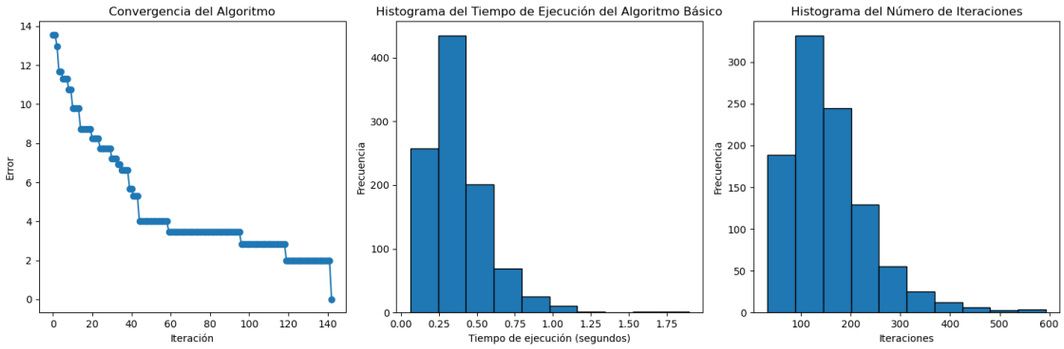
- **Ejemplo 3:** $OA(60, 5, 2, 2)$

[59]:
```
N, k, s, t = 60, 5, 2, 2
OA3 = np.random.randint(0, s, size=(N, k))
Graficos(OA3,t,nmax,1000)
```

```
Optimized Cost: 0.0
Optimized OA:
[[0 0 1 0 1]
 [0 1 1 1 0]
 [0 1 0 0 1]
 [0 0 0 1 0]
 [1 1 0 0 1]
 [1 1 1 1 1]
 [1 0 0 0 1]
```

```
[0 1 1 0 0]
[1 1 1 0 0]
[0 0 0 0 1]
[1 0 1 0 0]
[0 1 0 0 1]
[1 1 1 1 0]
[1 0 1 0 1]
[0 1 0 1 1]
[1 0 1 1 1]
[1 0 1 1 0]
[1 1 1 0 0]
[1 0 0 1 1]
[0 0 1 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[1 0 1 0 1]
[1 0 0 0 0]
[1 1 1 1 1]
[1 0 1 0 0]
[1 0 0 1 0]
[0 1 1 1 1]
[1 0 1 1 0]
[1 1 0 0 1]
[1 1 1 1 0]
[0 0 0 1 0]
[1 1 1 1 1]
[0 0 0 1 1]
[1 1 0 0 1]
[1 1 0 0 0]
[1 1 0 1 1]
[0 1 1 1 1]
[1 0 0 0 1]
[0 0 1 0 1]
[0 0 1 0 1]
[0 1 0 1 0]
[0 0 0 1 0]
[1 0 0 1 1]
[0 0 0 0 0]
[1 0 1 0 0]
[1 0 0 1 0]
[0 1 1 1 1]
[1 1 0 1 1]
[0 0 0 1 0]
[0 0 1 1 1]
```

```
 [0 0 1 1 1]
 [0 1 1 0 1]
 [0 1 0 1 0]
 [0 1 1 0 0]
 [0 1 0 0 0]
 [0 1 1 0 0]
 [0 1 0 0 0]
 [1 1 0 0 0]
 [1 1 0 1 0]]
```



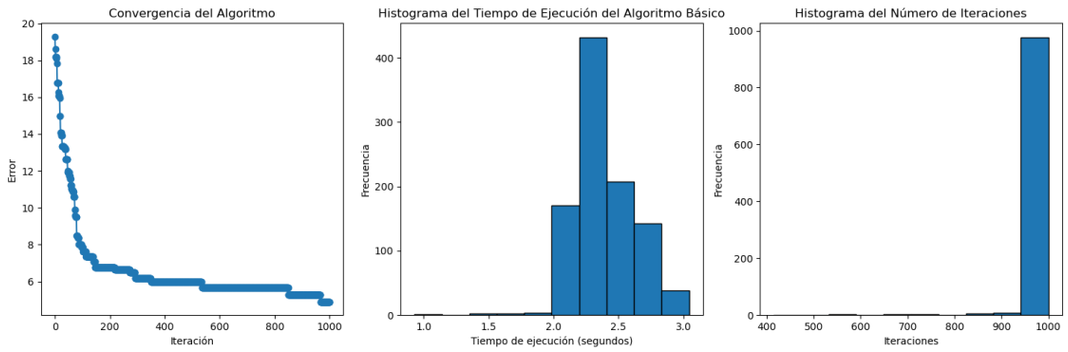
- **Ejemplo 4:** $OA(64, 5, 4, 2)$

[60]:
```
N, k, s, t = 63, 5, 3, 2
OA4 = np.random.randint(0, s, size=(N, k))
Graficos(OA4,t,nmax,1000)
```

```
Optimized Cost: 4.898979485566356
Optimized OA:
[[0 0 0 1 2]
 [0 1 0 2 2]
 [0 1 2 1 2]
 [2 2 2 0 2]
 [2 2 0 2 0]
 [0 1 2 0 1]
 [1 2 1 2 1]
 [0 1 0 2 1]
 [0 1 1 1 1]
 [0 1 2 0 0]
 [1 0 1 2 2]
 [0 2 0 0 1]
 [2 1 1 1 2]
 [1 1 0 1 0]
 [1 1 0 0 2]
 [2 1 2 1 1]
```

```
[0 2 0 1 2]
[0 2 1 1 0]
[0 2 0 2 0]
[0 1 1 2 1]
[1 2 2 0 2]
[2 2 1 0 1]
[2 2 0 1 1]
[2 0 1 1 1]
[0 2 0 0 2]
[0 0 1 0 0]
[2 2 2 2 0]
[1 2 1 0 1]
[0 1 1 0 0]
[1 2 2 1 0]
[2 1 1 0 2]
[0 0 2 2 1]
[0 2 2 1 1]
[2 1 0 2 0]
[2 0 0 1 1]
[1 1 1 2 1]
[0 0 1 1 0]
[1 1 0 1 1]
[1 2 2 2 2]
[0 2 2 1 2]
[2 1 1 0 2]
[2 0 2 0 1]
[2 0 0 2 0]
[2 1 2 2 0]
[2 0 1 1 2]
[2 1 0 0 0]
[1 2 1 1 0]
[0 0 1 2 2]
[1 1 2 2 2]
[0 0 0 0 2]
[0 2 1 2 0]
[1 0 0 0 0]
[2 2 1 2 2]
[1 0 0 2 1]
[0 0 2 2 1]
[0 0 2 0 0]
[1 0 1 0 0]
[2 0 2 2 2]
[2 2 0 0 1]
[1 1 2 1 0]
```

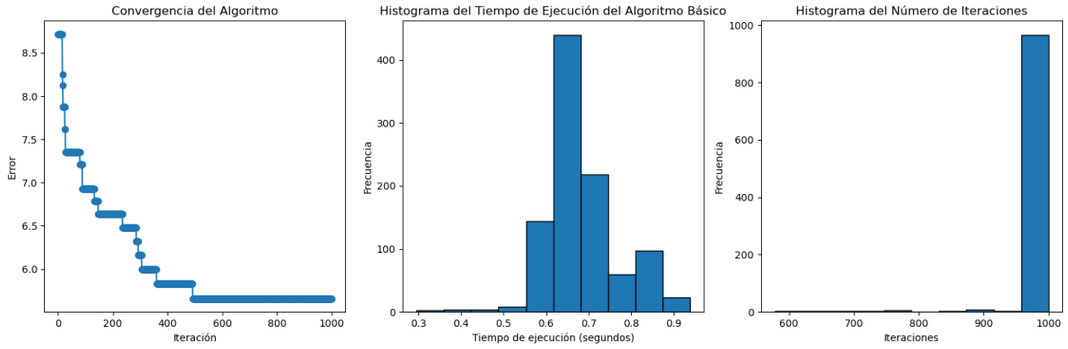
```
[2 0 2 1 0]
[1 0 2 0 1]
[1 0 0 1 2]]
```



- **Ejemplo 5:** $OA(16, 5, 4, 2)$

[61]:
```
N, k, s, t, nmax = 16, 5, 4, 2, 1000
OA5 = np.random.randint(0, s, size=(N, k))
Graficos(OA5,t,nmax,1000)
```

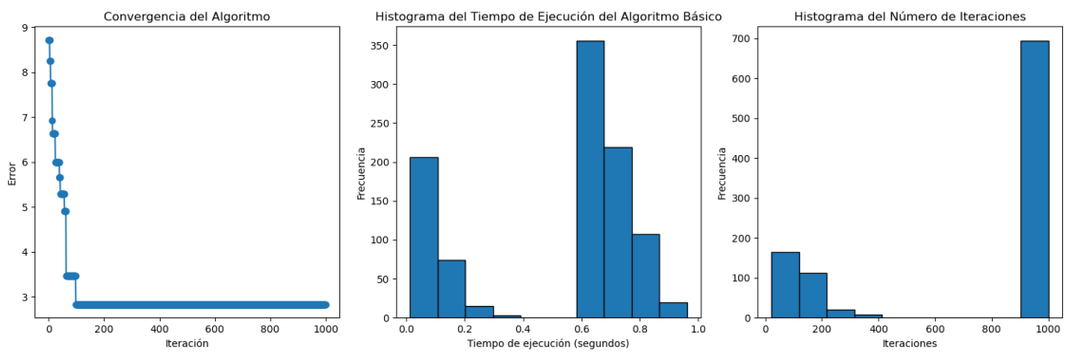
```
Optimized Cost: 5.656854249492381
Optimized OA:
[[3 2 3 0 1]
 [1 0 1 2 1]
 [3 0 1 3 0]
 [1 2 2 1 0]
 [1 1 3 2 0]
 [1 2 0 3 2]
 [0 2 0 2 3]
 [2 1 2 3 3]
 [2 1 1 0 2]
 [2 0 0 0 0]
 [1 3 1 0 3]
 [2 3 3 1 3]
 [0 0 3 1 2]
 [3 3 2 2 2]
 [0 3 2 3 1]
 [3 1 0 1 1]]
```



- **Ejemplo 6:** $OA(16, 5, 2, 3)$

[62]:
```
N, k, s, t, nmax = 16, 5, 2, 3, 1000
OA6 = np.random.randint(0, s, size=(N, k))
Graficos(OA6,t,nmax,1000)
```

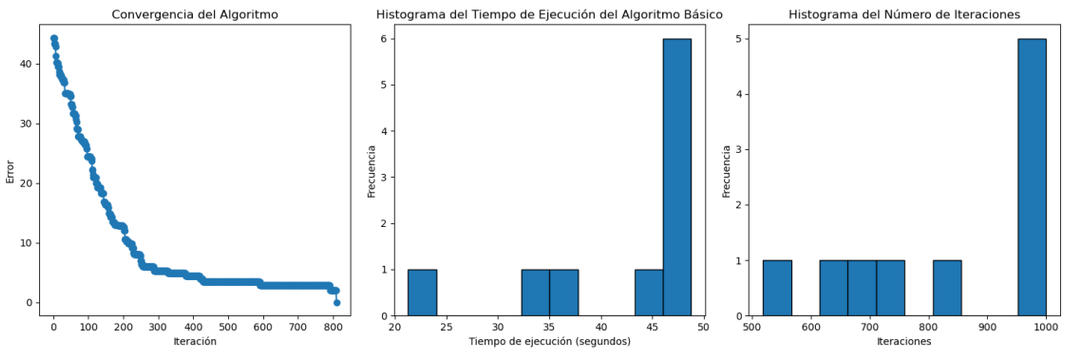
```
Optimized Cost: 2.8284271247461903
Optimized OA:
[[1 1 0 1 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]
 [0 0 1 1 1]
 [1 1 1 1 1]
 [0 1 1 1 0]
 [1 1 0 0 0]
 [0 1 1 0 1]
 [0 1 0 1 1]
 [1 0 1 0 0]
 [1 0 0 1 1]
 [0 1 0 0 0]
 [1 0 0 0 1]
 [1 1 1 0 1]
 [1 0 1 1 0]]
```



- **Ejemplo 7:** $OA(400, 7, 2, 2)$

[12]:
```
N, k, s, t, nmax= 400, 7, 2, 2, 1000
OA7 = np.random.randint(0, s, size=(N, k))
Graficos(OA7,t,nmax,10)
```

```
Optimized Cost: 0.0
Optimized OA:
[[1 1 0 ... 1 1 1]
 [1 1 1 ... 0 0 0]
 [0 0 0 ... 1 0 0]
 ...
 [0 1 1 ... 0 0 1]
 [0 1 1 ... 0 0 1]
 [0 1 1 ... 1 0 1]]
```



8.3. ALGORTIMO 2: BÚSQUEDA LOCAL

Función de Permutación de Columnas:

[20]:
```
def permute_columns(matrix, fixed_columns, k):
    N = matrix.shape[0]
    permuted_matrices = []
    for j in range(k):
        if j not in fixed_columns:
            new_matrix = matrix.copy()
            for col in range(k):
                if col not in fixed_columns and j!=col:
                    np.random.shuffle(new_matrix[:, col])
            N_matrix = [new_matrix,j]
            permuted_matrices.append(N_matrix)
    return permuted_matrices
```

Función de Búsqueda Local 1:

[21]:
```
def local_search_algorithm(N, k, s, t):
    OA = np.random.randint(0, s, size=(N, k))
    if not check_proportions(OA):   #Se distribuyen uniformemente los valores por columnas
        OA = unif_proportions(OA)
    error = Error(OA, t)
    cont = 0
    col = 0
    if error == 0:
        return OA
    fixed_columns = []

    while error > 0 and len(fixed_columns) < k:
        best_error = error
        best_matrix = OA
        cont+=1
        permuted_matrices = permute_columns(OA, fixed_columns, k)
        for perm_matrix in permuted_matrices:
            perm_error = Error(perm_matrix, t)
            if perm_error < best_error:
                best_error = perm_error
                best_matrix = perm_matrix
        error = best_error
        OA = best_matrix
        fixed_columns.append(np.argmin([Error(OA[:, [i]], t) for i in range(k)]))
    return OA,cont,error
```

[22]:
```
def LocalSearch1(N,k,s,t):
    best_oa,cont,err = local_search_algorithm(N, k, s, t)
    #Err = Error(best_oa[0],t)
    if err > 0:
        print("No solution found")
    else:
        print("Best Orthogonal Array found:")
        print(best_oa)
        print("Error:")
        print(err)
```

Función de Búsqueda Local 2:

[23]:
```
def local_search_algorithm2(N, k, s, t):
    OA = np.random.randint(0, s, size=(N, k))
```

```
    if not check_proportions(OA):   #Se distribuyen uniformemente los␣
↪valores por columnas
        OA = unif_proportions(OA)
    error = Error(OA, t)
    cont = 0
    col = 0
    costs =[]
    if error == 0:
        return OA
    fixed_columns = []

    while error > 0 and len(fixed_columns) < k:
        best_error = error
        prev_error = error
        best_matrix = OA
        cont+=1
        permuted_matrices = permute_columns(OA, fixed_columns, k)
        for perm_matrix in permuted_matrices:
            perm_error = Error(perm_matrix[0], t)
            if perm_error < best_error:
                best_error = perm_error
                best_matrix = perm_matrix[0]
                col_append = perm_matrix[1]
        if best_error < prev_error:
            fixed_columns.append(col_append)
            error = best_error
            OA = best_matrix
        if cont == 1000:
            return OA,cont,error,costs
        costs.append(error)
    return OA,cont,error,costs
```

[24]:
```
def LocalSearch2(N,k,s,t):
    best_oa,cont,err,costs = local_search_algorithm2(N, k, s, t)
    #Err = Error(best_oa[0],t)
    if err > 0:
        print("No solution found")
    else:
        print("Best Orthogonal Array found:")
        print(best_oa)
        print("Error:")
        print(err)
```

Función para Histograma del Tiempo de Ejecución:

[25]:
```
def tiempos_LS(N,k,s,t,niter):   #Se genera un histograma con los tiempos␣
↪de ejecución de aplicarlo 'niter' veces
    execution_times = []
    for _ in range(niter):
        start_time = time.time()
        local_search_algorithm2(N,k,s,t)
        end_time = time.time()
        execution_times.append(end_time - start_time)
    return execution_times
```

Función para Histograma del Número de Iteraciones:

[26]:
```
def iteraciones_LS(N,k,s,t,niter):   #Histograma de iteraciones␣
↪necesarias para hallar el OA al aplicarlo 'niter' veces
    execution_cont = []
    for _ in range(niter):
        cont = local_search_algorithm2(N,k,s,t)[1]
        execution_cont.append(cont)
    return execution_cont
```

Función de Representación de las Gráficas:

[27]:
```
def GraficosLS(N,k,s,t,niter):
    optimized_OA, cont, optimized_cost, costs,  =␣
↪local_search_algorithm2(N,k,s,t)
    fig, (plt1, plt2, plt3) = plt.subplots(1, 3, figsize=(15, 5))
    print(f"Optimized Cost: {optimized_cost}")
    print(f"Optimized OA: \n{optimized_OA}")
    plt1.plot(costs, '-o')
    plt1.set_xlabel('Iteración')
    plt1.set_ylabel('Error')
    plt1.set_title('Convergencia del Algoritmo')

    execution_times = tiempos_LS(N,k,s,t,niter)
    plt2.hist(execution_times, bins=10, edgecolor='black')
    plt2.set_xlabel('Tiempo de ejecución (segundos)')
    plt2.set_ylabel('Frecuencia')
    plt2.set_title('Histograma del Tiempo de Ejecución del Algoritmo de B.
↪L.')



    execution_cont = iteraciones_LS(N,k,s,t,niter)
    plt3.hist(execution_cont, bins=50, edgecolor='black')
    plt3.set_xlabel('Iteraciones')
    plt3.set_ylabel('Frecuencia')
```

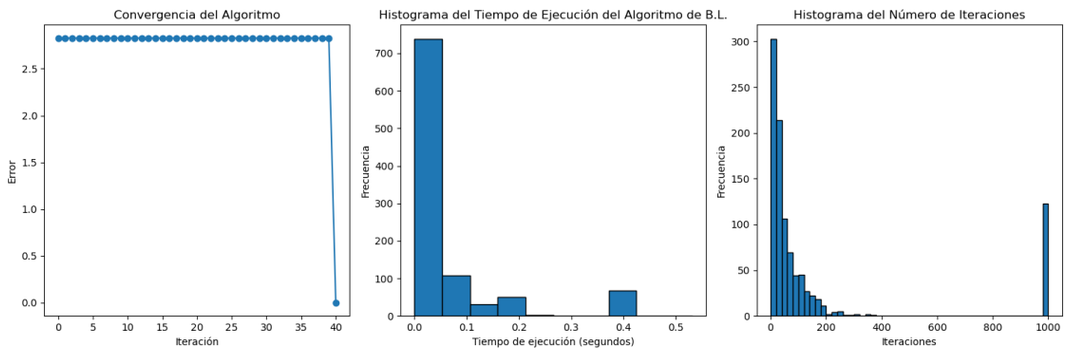
```
    plt3.set_title('Histograma del Número de Iteraciones')

    plt.tight_layout()
    plt.show()
```

8.3.1. Ejemplos

- **Ejemplo 1:** $OA(8, 4, 2, 3)$

[57]:
```
N,k,s,t = 8,4,2,3
GraficosLS(N,k,s,t,1000)
```

```
Optimized Cost: 0.0
Optimized OA:
[[1 1 1 0]
 [1 0 1 1]
 [1 0 0 0]
 [1 1 0 1]
 [0 0 1 0]
 [0 1 0 0]
 [0 1 1 1]
 [0 0 0 1]]
```

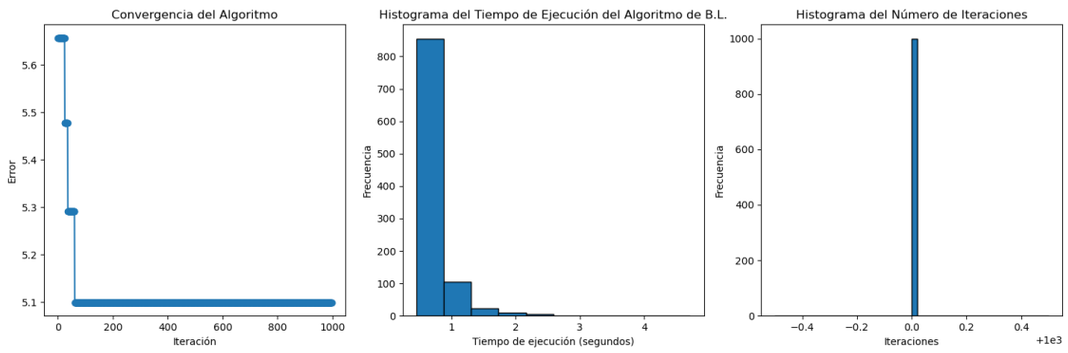


- **Ejemplo 2:** $OA(16, 5, 2, 4)$

[58]:
```
N,k,s,t = 16,5,2,4
GraficosLS(N,k,s,t,1000)
```

```
Optimized Cost: 5.0990195135927845
Optimized OA:
[[1 1 1 1 0]
 [0 1 0 1 1]
 [1 1 0 1 0]
 [0 1 0 0 0]
 [1 1 1 0 0]
```

```
 [0 1 1 0 1]
 [1 1 0 0 1]
 [0 1 1 1 0]
 [0 0 1 1 0]
 [1 0 1 1 1]
 [0 0 1 0 1]
 [1 0 0 1 0]
 [0 0 0 0 0]
 [0 0 0 0 1]
 [1 0 1 0 1]
 [1 0 0 1 1]]
```

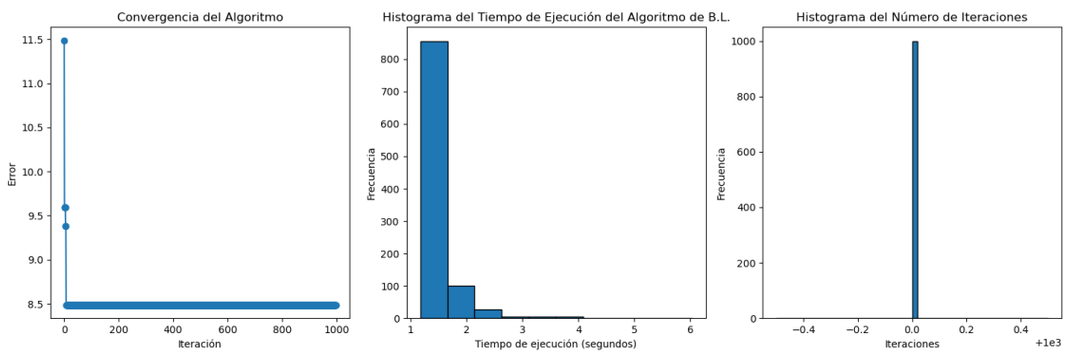


- **Ejemplo 3:** $OA(32, 5, 2, 3)$

[59]:
```
N,k,s,t = 32,5,2,3
GraficosLS(N,k,s,t,1000)
```

```
Optimized Cost: 8.48528137423857
Optimized OA:
[[0 0 1 0 1]
 [1 1 1 0 1]
 [1 0 0 1 0]
 [0 1 1 1 1]
 [0 1 1 1 0]
 [1 0 1 1 0]
 [1 1 1 1 1]
 [1 0 0 0 0]
 [1 0 0 0 1]
 [1 0 1 1 1]
 [1 0 0 0 1]
 [0 0 0 0 1]
 [0 1 1 1 0]
 [0 0 0 0 1]
 [0 0 1 1 0]
```

```
 [0 0 1 0 0]
 [1 0 0 0 1]
 [1 0 1 1 0]
 [0 1 0 1 1]
 [1 1 1 0 1]
 [1 1 1 1 0]
 [0 1 1 0 0]
 [1 1 1 0 0]
 [0 0 1 0 1]
 [1 1 0 0 0]
 [0 0 0 1 1]
 [0 1 0 0 0]
 [0 1 0 1 1]
 [0 0 0 1 0]
 [0 1 0 0 0]
 [1 1 0 1 0]
 [1 1 0 1 1]]
```



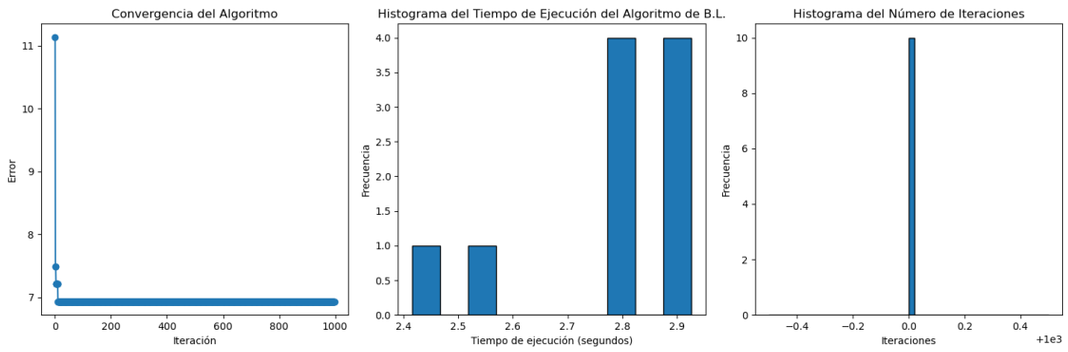
- **Ejemplo 4:** $OA(64, 5, 2, 2)$

[28]:
```
N,k,s,t = 64,5,2,2
GraficosLS(N,k,s,t,10)
```

```
Optimized Cost: 6.928203230275509
Optimized OA:
[[1 1 1 0 1]
 [1 1 1 0 0]
 [0 0 1 0 0]
 [0 1 0 1 0]
 [0 1 0 0 1]
 [1 1 0 0 1]
 [1 1 0 1 0]
 [1 1 1 1 1]
 [0 0 1 1 0]
```

```
[0 0 1 0 1]
[0 0 0 1 1]
[1 1 0 1 0]
[0 1 1 0 1]
[0 0 1 1 0]
[0 0 0 1 1]
[0 0 0 1 1]
[0 1 1 1 0]
[1 0 1 0 1]
[0 1 0 1 1]
[0 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 0]
[0 1 1 0 1]
[0 0 0 1 1]
[1 1 1 1 0]
[1 1 0 1 1]
[1 1 1 0 0]
[1 1 0 1 1]
[0 1 1 1 0]
[1 0 1 0 1]
[0 1 0 0 1]
[1 0 0 1 1]
[0 1 1 1 0]
[0 0 1 1 1]
[0 1 0 0 0]
[0 1 1 0 0]
[0 1 1 0 0]
[1 0 0 1 0]
[0 0 1 1 0]
[0 1 0 1 0]
[0 1 1 1 1]
[0 0 0 0 1]
[1 0 1 1 0]
[1 0 1 1 0]
[1 0 0 0 1]
[0 1 1 0 1]
[1 0 0 1 1]
[1 1 0 0 0]
[1 1 0 0 0]
[1 1 1 1 0]
[0 0 0 1 0]
[1 1 0 1 1]
[0 0 0 0 0]
```

```
[0 0 1 0 1]
[1 1 0 0 1]
[1 0 1 0 0]
[1 0 1 0 0]
[1 0 1 1 1]
[1 0 0 0 1]
[1 1 1 0 0]
[1 0 0 0 1]
[1 1 1 0 1]
[1 0 1 1 1]
[1 0 0 0 0]]
```



8.4. ALGORITMO 3: GENÉTICO CON MUTACIÓN

Función de Cruce:

[6]:
```python
def crossover(parent1, parent2):
    N, k = parent1.shape
    cross_point = random.randint(0, k-1)
    child1 = np.hstack((parent1[:, :cross_point], parent2[:, cross_point:
↪]))
    child2 = np.hstack((parent2[:, :cross_point], parent1[:, cross_point:
↪]))
    alpha = random.random()
    if alpha > 0.5:
        for j in range(cross_point,k):
            np.random.shuffle(child1[:,j])
            np.random.shuffle(child2[:,j])
    return child1, child2
```

Función del Algoritmo:

[7]:
```python
def genetic_algorithm_OA(N, k, s, t, n_ind, max_iter):
    population = []
```

```
    for _ in range(n_ind):
        individual = np.random.randint(0, s, size=(N, k))
        individual = unif_proportions(individual)
        population.append(individual)
    best_error = float('inf')
    best_OA = None
    itera = 0
    errors =[]
    while itera < max_iter and best_error != 0:
        new_population = []
        for _ in range(n_ind):
            parent1, parent2 = random.sample(population, 2)
            child1, child2 = crossover(parent1, parent2)
            new_population.append(child1)
            new_population.append(child2)

        combined_population = population + new_population
        combined_population.sort(key=lambda OA: Error(OA, t))
        population = combined_population[:n_ind]

        best_error = Error(population[0], t)
        if best_error == 0:
            best_OA = population[0]
        itera+= 1
        errors.append(best_error)
    return best_OA,itera,best_error,errors
```

Función de Histogramas de Iteraciones y Tiempo:

[11]:
```
def tiempos_iter_GG(N,k,s,t,n_iter, n_ind, max_iter):   #Se genera un␣
↪histograma con los tiempos de ejecución de aplicarlo 'niter' veces
    execution_times = []
    execution_cont = []
    for _ in range(n_iter):
        start_time = time.time()
        cont = genetic_algorithm_OA(N, k, s, t, n_ind, max_iter)[1]
        end_time = time.time()
        execution_cont.append(cont)
        execution_times.append(end_time - start_time)
    return execution_times,execution_cont
```

[12]:
```
def GraficosGG(N,k,s,t,n_iter, n_ind, max_iter):
    optimized_OA, cont, optimized_cost, costs,  = genetic_algorithm_OA(N,␣
↪k, s, t, n_ind, max_iter)
```

```
    fig, (plt1, plt2, plt3) = plt.subplots(1, 3, figsize=(15, 5))
    print(f"Optimized Cost: {optimized_cost}")
    print(f"Optimized OA: \n{optimized_OA}")
    plt1.plot(costs, '-o')
    plt1.set_xlabel('Iteración')
    plt1.set_ylabel('Error')
    plt1.set_title('Convergencia del Algoritmo')

    execution_times = tiempos_iter_GG(N,k,s,t,n_iter, n_ind, max_iter)[0]
    plt2.hist(execution_times, bins=10, edgecolor='black')
    plt2.set_xlabel('Tiempo de ejecución (segundos)')
    plt2.set_ylabel('Frecuencia')
    plt2.set_title('Tiempo de Ejecución del Algoritmo Genético por␣
↪Generacioes')


    execution_cont = tiempos_iter_GG(N,k,s,t,n_iter, n_ind, max_iter)[1]
    plt3.hist(execution_cont, bins=10, edgecolor='black')
    plt3.set_xlabel('Iteraciones')
    plt3.set_ylabel('Frecuencia')
    plt3.set_title('Número de Iteraciones por Ejecuciones')

    plt.tight_layout()
    plt.show()
```

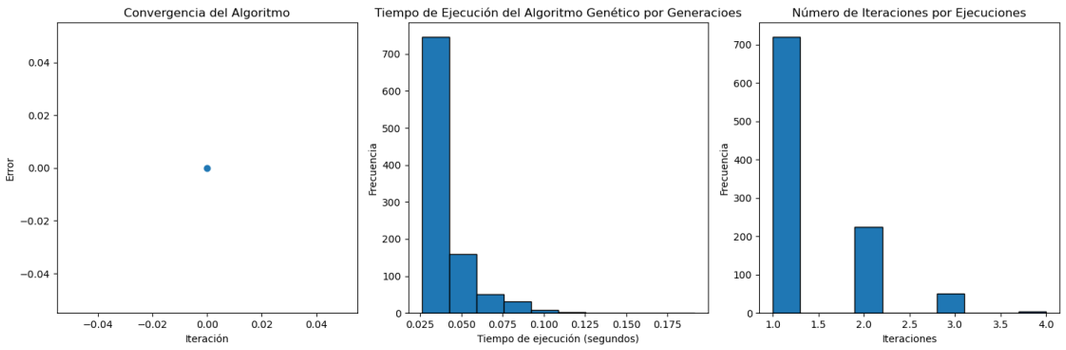
8.4.1. Ejemplos

:

- **Ejemplo 1:** $OA(8, 4, 2, 2)$

[34]:
```
GraficosGG(8,4,2,2,1000, 40, 1000)
```

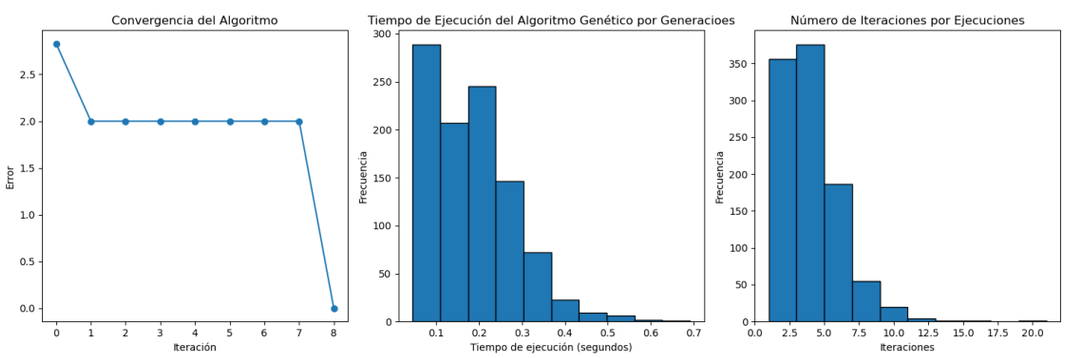
```
Optimized Cost: 0.0
Optimized OA:
[[0 1 0 0]
 [0 1 1 1]
 [1 1 1 0]
 [1 0 0 1]
 [1 1 0 1]
 [0 0 0 0]
 [1 0 1 0]
 [0 0 1 1]]
```



- **Ejemplo 2:** $OA(16, 4, 2, 2)$

[35]:
```
GraficosGG(16,4,2,2,1000, 40, 1000)
```

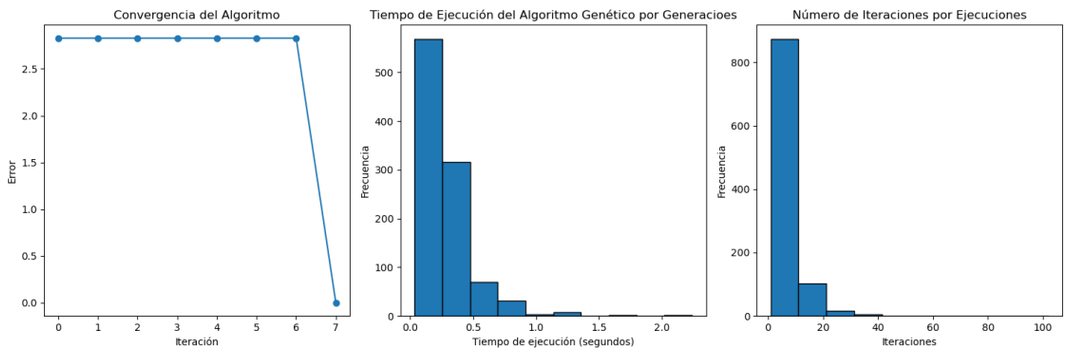
```
Optimized Cost: 0.0
Optimized OA:
[[1 0 1 0]
 [1 1 0 0]
 [1 1 1 1]
 [0 0 0 0]
 [1 1 0 1]
 [1 1 1 0]
 [1 0 0 1]
 [1 0 1 1]
 [0 1 0 0]
 [1 0 0 0]
 [0 1 1 0]
 [0 1 1 1]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]
 [0 1 0 1]]
```



- **Ejemplo 3:** $OA(16, 4, 2, 3)$

[36]:
```
GraficosGG(16,4,2,3,1000, 40, 1000)
```

```
Optimized Cost: 0.0
Optimized OA:
[[0 1 1 0]
 [0 1 0 0]
 [0 0 1 0]
 [1 1 0 0]
 [0 0 1 1]
 [1 1 0 1]
 [0 1 0 1]
 [0 0 0 0]
 [0 1 1 1]
 [0 0 0 1]
 [1 0 0 0]
 [1 1 1 1]
 [1 1 1 0]
 [1 0 1 0]
 [1 0 0 1]
 [1 0 1 1]]
```

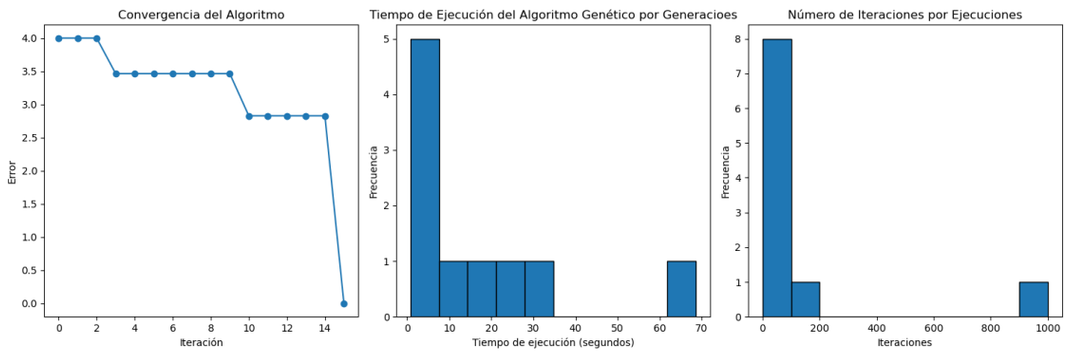


- **Ejemplo 4:** $OA(32, 4, 2, 3)$

[14]:
```
GraficosGG(32,4,2,3,10, 40, 1000)
```

```
Optimized Cost: 0.0
Optimized OA:
[[1 0 1 1]
 [1 0 1 0]
 [0 0 1 1]
 [0 1 1 0]
 [1 1 1 0]
 [1 1 0 0]
```

```
[1 0 1 1]
[1 1 1 1]
[1 0 0 1]
[1 0 0 0]
[1 0 0 0]
[1 0 1 0]
[0 1 1 1]
[0 0 0 0]
[1 1 0 0]
[0 1 1 1]
[0 1 0 0]
[0 0 1 1]
[1 1 1 0]
[0 0 0 0]
[0 1 0 1]
[0 0 1 0]
[1 1 1 1]
[1 1 0 1]
[1 1 0 1]
[1 0 0 1]
[0 0 1 0]
[0 0 0 1]
[0 1 0 0]
[0 0 0 1]
[0 1 1 0]
[0 1 0 1]]
```



8.5. PROBLEMA DEL VIAJANTE DE COMERCIO (Travelling Salesman Problem)

Datos:

$$c_{ij} := \text{coste de viajar de la ciudad } i \text{ a la } j. \tag{8.1}$$

Variables de Decisión:

$$x_{ij} = \begin{cases} 1, & \text{si la ruta recorre el camino de la ciudad } i \text{ a } j \\ 0, & \text{en caso contrario} \end{cases}, \quad i, j \in \{1, \cdots, n\}, i \neq j \tag{8.2}$$

$u_i = \{2, \cdots, n\}$: posición que toma la ciudad i en el tour.

Modelo de Optimización:

$$\begin{aligned} \text{mín} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ s.a \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = \{1, \cdots, n\} \\ & \sum_{j=1}^{n} x_{ij} = 1, \quad i = \{1, \cdots, n\} \\ & u_i - u_j + n \cdot x_{ij} \leqslant n - 1 \\ & x_{ij} = \{0, 1\}, u_i = 2, \cdots, n \end{aligned}$$

Función de Generación Aleatoria de Ciudades:

[25]:
```
#Función para calcular la distancia entre dos puntos
def calculate_dist(city1, city2):
    return np.linalg.norm(np.array(city1) - np.array(city2))

#Generar coordenadas aleatorias para las ciudades
def rng_coordinates(n_cities, xlim=(0, 100), ylim=(0, 100)):
    return [(random.uniform(*xlim), random.uniform(*ylim)) for _ in␣
 ↪range(n_cities)]

#Calcular la matriz de distancias entre todas las ciudades
def calculate_dist_matrix(coord):
```

```
    num_cities = len(coord)
    dist_matrix = np.zeros((n_cities, n_cities))
    for i in range(n_cities):
        for j in range(i, n_cities):
            dist = calculate_dist(coord[i], coord[j])
            dist_matrix[i, j] = dist
            dist_matrix[j, i] = dist
    return dist_matrix
```

Función de Optimización del Modelo TSP:

[3]:
```
def solve_tsp(dist_matrix):
    n = len(dist_matrix)
    model = Model(env = env)
    model.setParam('OutputFlag', 0)

    #Variables de decisión
    x = model.addVars(n, n, vtype=GRB.BINARY, name="x")
    u = model.addVars(n, vtype=GRB.INTEGER, name="u")

    #Función objetivo
    model.setObjective(quicksum(dist_matrix[i][j] * x[i, j] for i in␣
↪range(n) for j in range(n)), GRB.MINIMIZE)

    #Restricciones
    model.addConstrs(quicksum(x[i, j] for j in range(n) if j != i) == 1␣
↪for i in range(n))
    model.addConstrs(quicksum(x[i, j] for i in range(n) if i != j) == 1␣
↪for j in range(n))

    #Restricciones de subtour
    model.addConstrs(u[i] - u[j] + n * x[i, j] <= n - 1 for i in range(1,␣
↪n) for j in range(1, n) if i != j)

    #Optimización
    model.optimize()

    #Solución
    solution = model.getAttr('x', x)
    tour = []
    current_city = 0
    while len(tour) < n:
        tour.append(current_city)
```

```
        next_city = [j for j in range(n) if solution[current_city, j] > 0.
↪5][0]
        current_city = next_city

    return tour, model.objVal
```

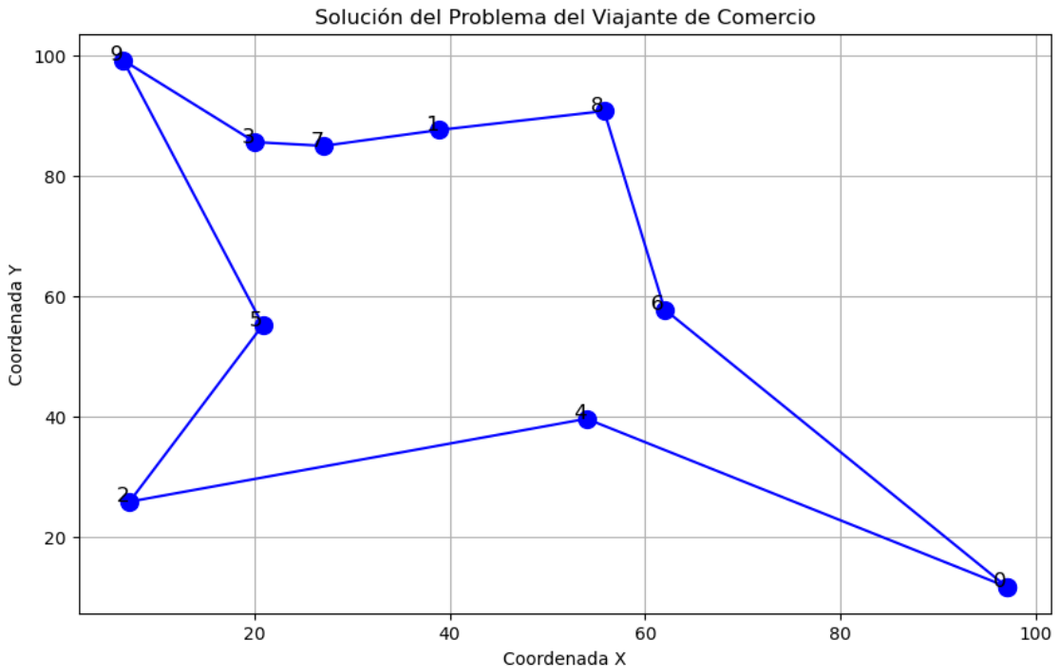
Función de Representación de Soluciones:

[23]:
```
def plot_tsp_sol(coordinates, solution):
    tour_x = [coordinates[i][0] for i in solution] +␣
↪[coordinates[solution[0]][0]]
    tour_y = [coordinates[i][1] for i in solution] +␣
↪[coordinates[solution[0]][1]]

    plt.figure(figsize=(10, 6))
    plt.plot(tour_x, tour_y, 'o-', color='blue', markersize=10)
    plt.title('Solución del Problema del Viajante de Comercio')
    plt.xlabel('Coordenada X')
    plt.ylabel('Coordenada Y')

    for idx, (x, y) in enumerate(coordinates):
        plt.text(x, y, str(idx), fontsize=12, ha='right')

    plt.grid(True)
    plt.show()
```

8.5.1. Ejemplos

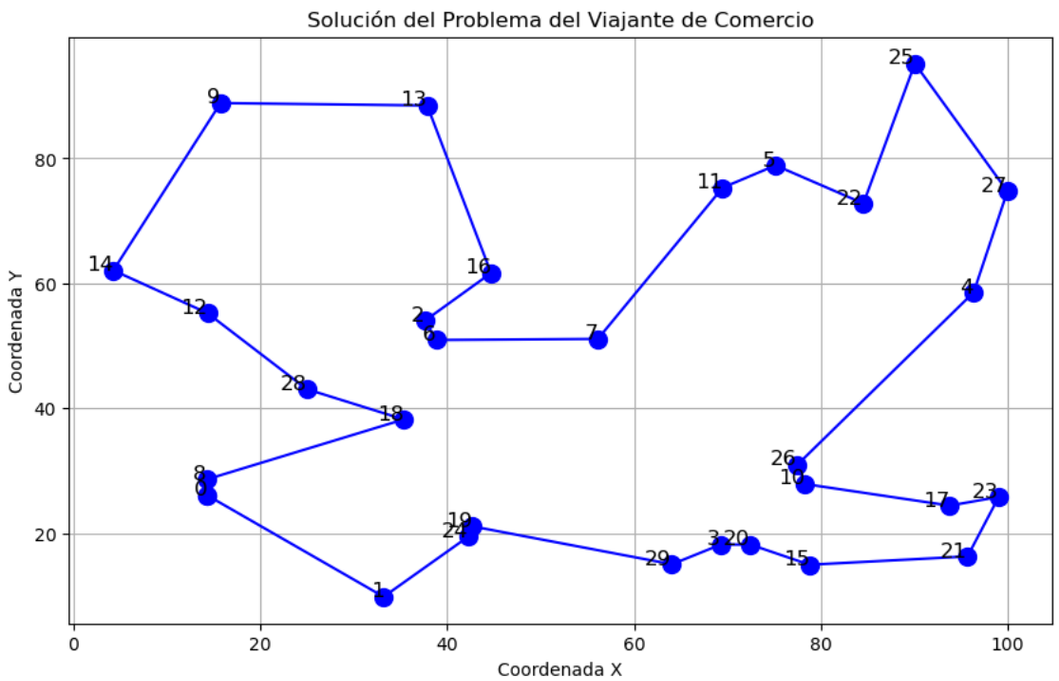
Número de Ciudades = 10

[27]:
```
n_cities = 10
coord = rng_coordinates(n_cities)
dist_matrix = calculate_dist_matrix(coord)

plot_tsp_sol(coord,solve_tsp(dist_matrix)[0])
```



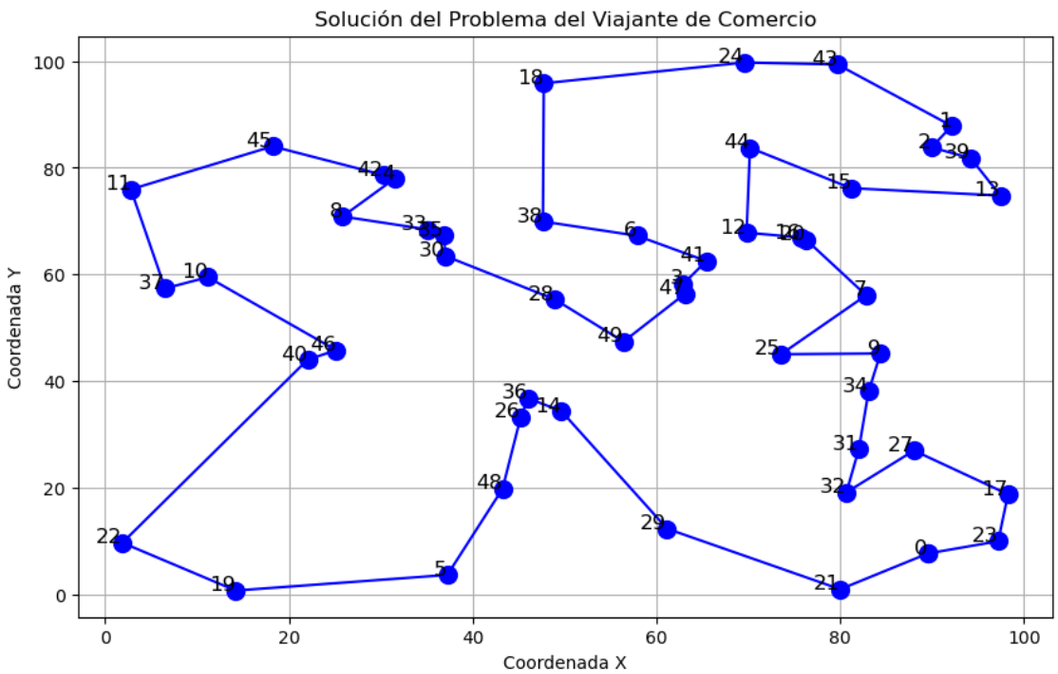
Número de Ciudades = 30

[8]:
```
n_cities = 30
coord = rng_coordinates(n_cities)
dist_matrix = calculate_dist_matrix(coord)

plot_tsp_sol(coord,solve_tsp(dist_matrix)[0])
```



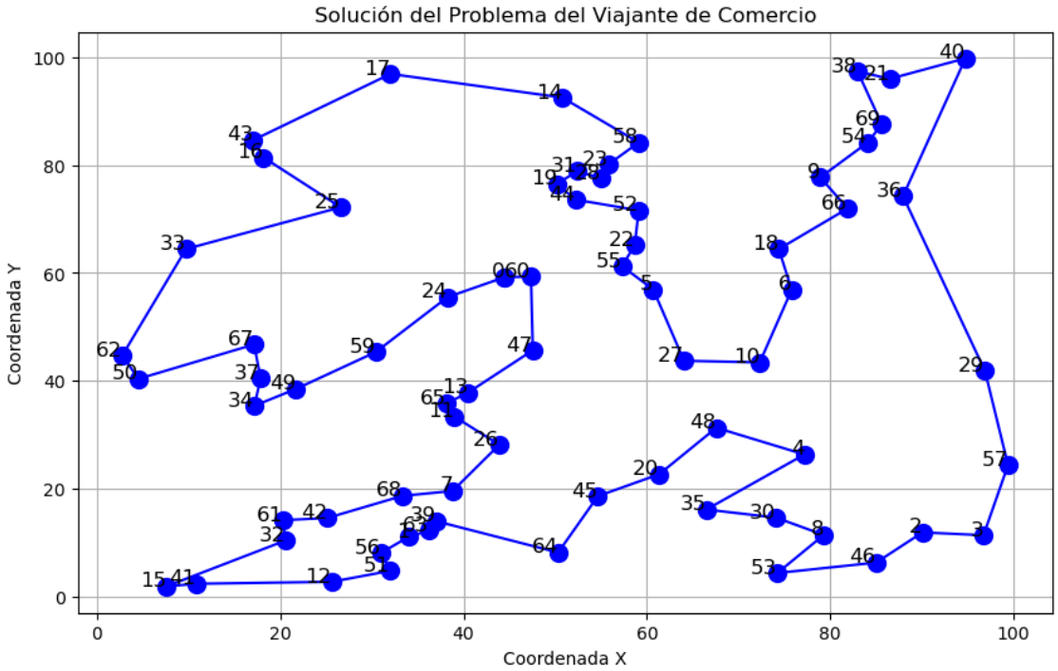
Número de Ciudades = 50

[9]:
```
n_cities = 50
coord = rng_coordinates(n_cities)
dist_matrix = calculate_dist_matrix(coord)

plot_tsp_sol(coord,solve_tsp(dist_matrix)[0])
```



Número de Ciudades = 70

[28]:
```
n_cities = 70
coord = rng_coordinates(n_cities)
dist_matrix = calculate_dist_matrix(coord)

plot_tsp_sol(coord,solve_tsp(dist_matrix)[0])
```



8.6. MODELO DE OPTIMIZACIÓN DE ORTHOGONAL ARRAYS

Conjuntos Necesarios:

- S:= $\{0, 1, \cdots, s\}$ Conjunto de símbolos del orthogonal array
- I:= $\{1, \cdots, N\}$ Conjunto de Filas
- J:= $\{1, \cdots, k\}$ Conjunto de Columnas
- A:= (a_{ij}), $i \in I$, $j \in J$ Matriz a completar para el orthogonal array
- C:= Conjunto de todas las posibles colecciones de t elementos diferentes de J sin importar el orden.
- L:= Conjunto de todas las posibles t-tuplas que se pueden fortmar con elementos de S, con repetición.

Variables de Decisión:

$$x_{ij}^{s} = \begin{cases} 1, & \text{si el elemento } a_{ij} \text{ de } A \text{ es igual a } s \\ 0, & \text{en caso contrario} \end{cases} \qquad i \in F, \quad j \in J,$$

$$z_{i,c}^{l} = \begin{cases} 1, & \text{si la combinación } l \in L \text{ está en la fila } i \text{ de la submatriz de columnas } c \in C \\ 0, & \text{en caso contrario} \end{cases} \qquad i \in I, \quad c \in C, \tag{8.3}$$

Modelo de Optimización:

$$
\begin{aligned}
\text{mín} \quad & \sum_{c \in C} \sum_{l \in L} \left| \sum_{i=1}^{N} z_{i,c}^{l} - \lambda \right| \\
s.a. \quad & \sum_{l=1}^{s} x_{ij}^{l} = 1, && i = 1, \cdots, N \quad j = 1, \cdots, k \\
& \sum_{i=1}^{N} x_{ij}^{l} = \frac{N}{s}, && j = 1, \cdots, k \quad l = 1, \cdots, s \\
& z_{i,c}^{l} \leqslant x_{ijr}^{l_r}, && r = 1, \cdots, t \quad i = 1, \cdots, N \quad c \in C \quad l \in L \\
& z_{i,c}^{l} \geqslant \sum_{m=1}^{t} x_{ij_m}^{lm} - (t-1), && i = 1, \cdots, N \quad c \in C, \quad l \in L \\
& \sum_{i=1}^{N} z_{i,c}^{l} = \lambda, && c \in C \quad l \in L \\
& x_{ij}^{l} = \{0, 1\}, && i = 1, \cdots, N \quad j = 1, \cdots, k \quad l = 1, \cdots, s \\
& z_{i,c}^{l} = \{0, 1\}, && i = 1, \cdots, N \quad c \in C, \quad l \in L
\end{aligned}
\tag{8.4}
$$

Implementación del modelo:

[3]:
```
def OA_Lineal(N,k,s,t):
    # Crear un nuevo modelo
    model = Model(env = env)
    model.setParam('OutputFlag', 0)
    lambda_value = N // (s ** t)

    #Se generan todas las combinaciones posibles de t columnas
    column_combinations = list(itertools.combinations(range(k), t))

    #Se generan todas las combinaciones posibles de niveles para cada␣
↪conjunto de t columnas
    level_combinations = list(itertools.product(range(s), repeat=t))

    #Variables de decisión
    x = model.addVars(N, k, s, vtype = "B", name="x")
    z = model.addVars(N, len(column_combinations),␣
↪len(level_combinations), vtype = "B", name="z")

    #Restricción 1: Un único nivel por celda
    for i in range(N):
        for j in range(k):
            model.addConstr(quicksum(x[i, j, l] for l in range(s)) == 1,␣
↪f"cell_{i}_{j}")

    #Restricción 2: Distribución uniforme de niveles en cada columna
    for j in range(k):
```

```
        for l in range(s):
            model.addConstr(quicksum(x[i, j, l] for i in range(N)) == N //
↪ s, f"column_{j}_level_{l}")

    #Restricción 3 y 4: Relación variables z y x
    for i in range(N):
        for comb_idx, comb in enumerate(column_combinations):
            for level_idx, level_comb in enumerate(level_combinations):
                for j in range(t):
                    model.addConstr(
                        z[i, comb_idx, level_idx] <= x[i, comb[j],␣
↪level_comb[j]])
                model.addConstr(z[i, comb_idx, level_idx] >=␣
↪quicksum(x[i, comb[j], level_comb[j]] for j in range(t)) - (t - 1))

    #Restricción 5: Frecuencia de las combinaciones de niveles para cada␣
↪conjunto de t columnas
    for comb_idx in range(len(column_combinations)):
        for level_idx in range(len(level_combinations)):
            model.addConstr(quicksum(z[i, comb_idx, level_idx] for i in␣
↪range(N)) == lambda_value)

    #Se impone la suma de variables z, para que, a priori, valgan todas 0
    model.setObjective(quicksum(z[i,j,l] for i in range(N) for j in␣
↪range(len(column_combinations))for l in␣
↪range(len(level_combinations))), sense=GRB.MINIMIZE)
    model.optimize()

    #Se muestran los resultados
    if model.status == GRB.OPTIMAL:
        print("Optimal solution found.")
        solution = np.zeros((N, k), dtype=int)
        for i in range(N):
            for j in range(k):
                for l in range(s):
                    if x[i, j, l].x > 0.5:
                        solution[i, j] = l
        print("Solution:")
        print(solution)
    else:
        print("No optimal solution found.")
```

Función para la Creación de la Tabla:

[17]:
```
def OA_Lineal_Tabla(N,k,s,t):
    # Crear un nuevo modelo
    model = Model(env = env)
    model.setParam('OutputFlag', 0)
    model.setParam('TimeLimit', 600)
    lambda_value = N // (s ** t)

    #Se generan todas las combinaciones posibles de t columnas
    column_combinations = list(itertools.combinations(range(k), t))

    #Se generan todas las combinaciones posibles de niveles para cada␣
↪conjunto de t columnas
    level_combinations = list(itertools.product(range(s), repeat=t))

    #Variables de decisión
    x = model.addVars(N, k, s, vtype = "B", name="x")
    z = model.addVars(N, len(column_combinations),␣
↪len(level_combinations), vtype = "B", name="z")

    #Restricción 1: Un único nivel por celda
    for i in range(N):
        for j in range(k):
            model.addConstr(quicksum(x[i, j, l] for l in range(s)) == 1,␣
↪f"cell_{i}_{j}")

    #Restricción 2: Distribución uniforme de niveles en cada columna
    for j in range(k):
        for l in range(s):
            model.addConstr(quicksum(x[i, j, l] for i in range(N)) == N //
↪ s, f"column_{j}_level_{l}")

    #Restricción 3 y 4: Relación variables z y x
    for i in range(N):
        for comb_idx, comb in enumerate(column_combinations):
            for level_idx, level_comb in enumerate(level_combinations):
                for j in range(t):
                    model.addConstr(
                        z[i, comb_idx, level_idx] <= x[i, comb[j],␣
↪level_comb[j]])
                model.addConstr(z[i, comb_idx, level_idx] >=␣
↪quicksum(x[i, comb[j], level_comb[j]] for j in range(t)) - (t - 1))
```

```
    #Restricción 5: Frecuencia de las combinaciones de niveles para cada␣
↪conjunto de t columnas
    for comb_idx in range(len(column_combinations)):
        for level_idx in range(len(level_combinations)):
            model.addConstr(quicksum(z[i, comb_idx, level_idx] for i in␣
↪range(N)) == lambda_value)

    #Se impone la suma de variables z, para que, a priori, valgan todas 0
    model.setObjective(quicksum(z[i,j,l] for i in range(N) for j in␣
↪range(len(column_combinations))for l in␣
↪range(len(level_combinations))), sense=GRB.MINIMIZE)
    model.optimize()

    #Se muestran los resultados
    if model.status == GRB.OPTIMAL:
        solution = np.zeros((N, k), dtype=int)
        for i in range(N):
            for j in range(k):
                for l in range(s):
                    if x[i, j, l].x > 0.5:
                        solution[i, j] = l
        return True
    else:
        return False
```

Ejemplos:

8.6.1. Fortaleza $t = 2$

- $OA(25, 5, 5, 2)$

[4]:
```
OA_Lineal(25,5,5,2)
```

```
Optimal solution found.
Solution:
[[0 0 0 0 0]
 [1 0 1 2 3]
 [0 1 1 1 1]
 [0 2 2 2 2]
 [0 3 3 3 3]
 [1 1 3 4 2]
 [0 4 4 4 4]
 [1 2 4 1 0]
 [1 3 2 0 4]
 [2 0 2 4 1]
```

```
 [1 4 0 3 1]
 [2 1 4 0 3]
 [2 2 1 3 4]
 [2 3 0 1 2]
 [3 0 4 3 2]
 [2 4 3 2 0]
 [3 1 0 2 4]
 [3 2 3 0 1]
 [4 0 3 1 4]
 [4 1 2 3 0]
 [3 3 1 4 0]
 [4 2 0 4 3]
 [3 4 2 1 3]
 [4 3 4 2 1]
 [4 4 1 0 2]]
```

- $OA(16, 5, 4, 2)$

[21]:
```
OA_Lineal(16,5,4,2)
```

```
Optimal solution found.
Solution:
[[1 3 2 2 0]
 [0 0 0 2 1]
 [3 3 0 0 2]
 [0 3 3 1 3]
 [2 1 0 1 0]
 [1 0 1 1 2]
 [1 1 3 0 1]
 [2 0 2 0 3]
 [0 2 1 0 0]
 [3 2 2 1 1]
 [3 0 3 3 0]
 [2 2 3 2 2]
 [3 1 1 2 3]
 [2 3 1 3 1]
 [1 2 0 3 3]
 [0 1 2 3 2]]
```

- $OA(64, 5, 4, 2)$

[33]:
```
OA_Lineal(64,5,4,2)
```

```
Optimal solution found.
Solution:
[[3 3 3 3 3]
 [3 3 3 3 3]
```

```
[3 3 3 3 3]
[3 3 3 3 3]
[3 2 2 2 2]
[3 2 2 2 2]
[3 2 2 2 2]
[3 2 2 2 2]
[3 1 1 1 1]
[3 1 1 1 1]
[3 1 1 1 1]
[3 1 1 1 1]
[3 0 0 0 0]
[3 0 0 0 0]
[3 0 0 0 0]
[3 0 0 0 0]
[2 3 2 1 0]
[2 3 2 1 0]
[2 3 2 1 0]
[2 3 2 1 0]
[2 2 3 0 1]
[2 2 3 0 1]
[2 2 3 0 1]
[2 2 3 0 1]
[2 1 0 3 2]
[2 1 0 3 2]
[2 1 0 3 2]
[2 1 0 3 2]
[2 0 1 2 3]
[2 0 1 2 3]
[2 0 1 2 3]
[2 0 1 2 3]
[1 3 1 0 2]
[1 3 1 0 2]
[1 3 1 0 2]
[1 3 1 0 2]
[1 2 0 1 3]
[1 2 0 1 3]
[1 2 0 1 3]
[1 2 0 1 3]
[1 1 3 2 0]
[1 1 3 2 0]
[1 1 3 2 0]
[1 1 3 2 0]
[1 0 2 3 1]
[1 0 2 3 1]
```

```
 [1 0 2 3 1]
 [1 0 2 3 1]
 [0 3 0 2 1]
 [0 3 0 2 1]
 [0 3 0 2 1]
 [0 3 0 2 1]
 [0 2 1 3 0]
 [0 2 1 3 0]
 [0 2 1 3 0]
 [0 2 1 3 0]
 [0 1 2 0 3]
 [0 1 2 0 3]
 [0 1 2 0 3]
 [0 1 2 0 3]
 [0 0 3 1 2]
 [0 0 3 1 2]
 [0 0 3 1 2]
 [0 0 3 1 2]]
```

- $OA(48, 4, 4, 2)$

[34]:
```
OA_Lineal(48,4,4,2)
```

```
Optimal solution found.
Solution:
[[3 3 3 3]
 [3 3 3 3]
 [3 3 3 3]
 [3 2 2 2]
 [3 2 2 2]
 [3 2 2 2]
 [3 1 1 1]
 [3 1 1 1]
 [3 1 1 1]
 [3 0 0 0]
 [3 0 0 0]
 [3 0 0 0]
 [2 3 2 1]
 [2 3 2 1]
 [2 3 2 1]
 [2 2 3 0]
 [2 2 3 0]
 [2 2 3 0]
 [2 1 0 3]
 [2 1 0 3]
```

```
 [2 1 0 3]
 [2 0 1 2]
 [2 0 1 2]
 [2 0 1 2]
 [1 3 1 0]
 [1 3 1 0]
 [1 3 1 0]
 [1 2 0 1]
 [1 2 0 1]
 [1 2 0 1]
 [1 1 3 2]
 [1 1 3 2]
 [1 1 3 2]
 [1 0 2 3]
 [1 0 2 3]
 [1 0 2 3]
 [0 3 0 2]
 [0 3 0 2]
 [0 3 0 2]
 [0 2 1 3]
 [0 2 1 3]
 [0 2 1 3]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 0 3 1]
 [0 0 3 1]
 [0 0 3 1]]
```

- $OA(81, 4, 3, 2)$

[35]:
```
OA_Lineal(81,4,3,2)
```

```
Optimal solution found.
Solution:
[[2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 1 1 1]
```

```
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 1 1 1]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[2 0 0 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 2 1 0]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 1 0 2]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
[1 0 2 1]
```

```
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 2 0 1]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 1 2 0]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]
 [0 0 1 2]]
```

[ ]:
```
OA_Lineal(49,5,7,2)
```

8.6.2. Fortaleza $t = 3$

- $OA(16, 5, 2, 3)$

[5]:
```
OA_Lineal(16,5,2,3)
```

```
Optimal solution found.
Solution:
[[1 1 1 0 0]
 [0 0 0 0 0]
 [0 0 0 1 1]
 [0 0 1 0 1]
 [0 1 0 0 0]
 [1 1 0 1 0]
 [0 1 1 0 1]
 [1 0 1 1 1]
```

```
 [0 0 1 1 0]
 [1 0 1 0 0]
 [1 1 0 0 1]
 [0 1 0 1 1]
 [1 0 0 1 0]
 [0 1 1 1 0]
 [1 1 1 1 1]
 [1 0 0 0 1]]
```

- $OA(27, 4, 3, 3)$

[38]:
```
OA_Lineal(27,4,3,3)
```

```
Optimal solution found.
Solution:
[[1 1 2 2]
 [2 0 2 2]
 [2 2 0 2]
 [1 2 0 1]
 [1 1 1 1]
 [1 0 1 0]
 [2 2 2 1]
 [0 1 2 1]
 [0 0 1 2]
 [1 2 2 0]
 [2 0 0 0]
 [2 1 1 2]
 [1 1 0 0]
 [2 1 0 1]
 [0 1 1 0]
 [0 2 1 1]
 [2 2 1 0]
 [0 2 0 0]
 [0 2 2 2]
 [1 2 1 2]
 [1 0 0 2]
 [0 0 0 1]
 [0 0 2 0]
 [2 1 2 0]
 [0 1 0 2]
 [2 0 1 1]
 [1 0 2 1]]
```

- $OA(81, 4, 3, 3)$

[39]:
```
OA_Lineal(81,4,3,3)
```

```
Optimal solution found.
Solution:
[[0 0 0 0]
 [0 0 0 2]
 [0 0 1 2]
 [0 0 2 1]
 [0 1 0 2]
 [0 0 1 1]
 [0 0 2 2]
 [0 1 0 1]
 [0 1 1 1]
 [1 2 2 0]
 [0 1 1 2]
 [0 1 2 1]
 [0 1 2 0]
 [0 2 0 1]
 [0 2 0 2]
 [1 2 1 2]
 [0 2 1 1]
 [1 2 0 2]
 [0 2 1 2]
 [0 2 2 1]
 [0 2 2 2]
 [1 0 2 0]
 [0 0 2 0]
 [0 2 1 0]
 [0 2 0 0]
 [2 1 2 1]
 [0 0 1 0]
 [1 0 0 1]
 [0 1 2 2]
 [0 0 0 1]
 [0 2 2 0]
 [1 2 2 1]
 [1 0 1 2]
 [1 1 0 1]
 [0 1 1 0]
 [0 1 0 0]
 [1 2 1 1]
 [1 0 1 1]
 [1 0 1 0]
 [2 0 0 2]
 [1 1 1 0]
 [2 0 0 0]
```

```
[1 1 1 2]
[2 0 1 1]
[1 1 2 0]
[2 0 1 2]
[1 2 1 0]
[1 2 0 1]
[1 1 0 0]
[1 1 0 2]
[1 0 0 0]
[1 1 2 1]
[1 0 2 2]
[1 1 2 2]
[1 0 0 2]
[1 2 2 2]
[1 2 0 0]
[1 1 1 1]
[1 0 2 1]
[2 0 2 2]
[2 1 0 2]
[2 1 1 1]
[2 1 2 0]
[2 2 0 1]
[2 2 1 2]
[2 2 2 1]
[2 0 2 0]
[2 1 0 1]
[2 1 1 0]
[2 1 2 2]
[2 2 0 0]
[2 2 1 1]
[2 2 2 0]
[2 0 2 1]
[2 0 1 0]
[2 1 0 0]
[2 0 0 1]
[2 2 1 0]
[2 2 0 2]
[2 1 1 2]
[2 2 2 2]]
```

8.6.3. Fortaleza $t = 4$

- $OA(81, 4, 3, 4)$

[40]:
```
OA_Lineal(81,4,3,4)
```

```
Optimal solution found.
Solution:
[[2 2 1 0]
 [0 1 1 1]
 [1 1 2 0]
 [2 1 0 2]
 [0 2 2 0]
 [0 0 0 1]
 [2 1 0 1]
 [0 2 0 0]
 [1 2 2 0]
 [1 2 0 0]
 [0 2 1 2]
 [0 0 2 1]
 [2 0 0 2]
 [0 0 2 2]
 [0 1 0 2]
 [0 0 2 0]
 [1 0 1 2]
 [0 0 0 2]
 [0 2 0 1]
 [1 0 0 0]
 [0 0 1 0]
 [1 1 0 2]
 [1 2 1 2]
 [2 2 0 1]
 [1 2 0 2]
 [2 1 2 0]
 [2 1 2 2]
 [0 2 1 0]
 [1 1 1 1]
 [1 2 1 0]
 [2 2 0 0]
 [1 2 1 1]
 [2 0 2 0]
 [2 0 0 0]
 [2 0 2 1]
 [0 1 1 2]
 [1 0 2 1]
 [2 2 1 1]
 [0 1 0 1]
 [0 1 1 0]
 [1 1 1 0]
 [0 0 0 0]
```

```
[0 0 1 1]
[2 1 1 1]
[1 1 2 1]
[2 1 0 0]
[1 0 0 1]
[2 0 1 0]
[0 2 1 1]
[2 1 1 0]
[0 2 2 1]
[0 1 0 0]
[1 1 2 2]
[1 1 1 2]
[2 0 0 1]
[1 0 1 0]
[1 0 1 1]
[2 2 0 2]
[0 1 2 0]
[2 2 2 2]
[0 0 1 2]
[1 2 2 2]
[1 1 0 1]
[2 2 2 1]
[0 1 2 2]
[0 2 0 2]
[0 2 2 2]
[1 0 2 0]
[1 0 0 2]
[1 0 2 2]
[2 2 2 0]
[2 0 2 2]
[0 1 2 1]
[1 1 0 0]
[2 0 1 2]
[2 2 1 2]
[1 2 0 1]
[2 0 1 1]
[1 2 2 1]
[2 1 2 1]
[2 1 1 2]]
```

- $OA(16, 4, 2, 4)$

[6]:
```
OA_Lineal(16,4,2,4)
```

```
Optimal solution found.
```

```
Solution:
[[0 0 0 0]
 [0 0 1 0]
 [0 1 0 1]
 [1 0 0 0]
 [1 0 1 1]
 [0 1 1 0]
 [0 1 1 1]
 [1 1 0 0]
 [1 1 1 1]
 [0 0 1 1]
 [1 0 1 0]
 [0 1 0 0]
 [1 0 0 1]
 [0 0 0 1]
 [1 1 1 0]
 [1 1 0 1]]
```

- $OA(243, 11, 3, 4)$ *(No soportado por el ordenador)*

[ ]:
```
OA_Lineal(243,11,3,4)
```

8.6.4. Fortaleza $t = 5$

- $OA(32, 5, 2, 5)$

[18]:
```
OA_Lineal(32,5,2,5)
```

```
Optimal solution found.
Solution:
[[1 1 1 1 0]
 [1 0 0 0 0]
 [1 0 1 1 0]
 [0 0 0 0 1]
 [0 1 1 1 0]
 [1 1 0 1 1]
 [0 1 1 0 1]
 [1 1 0 1 0]
 [1 0 0 0 1]
 [1 1 0 0 1]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [1 0 1 1 1]
 [0 1 1 1 1]
 [0 0 1 0 1]
 [1 0 1 0 0]
```

```
 [1 1 1 0 0]
 [0 1 0 0 1]
 [0 0 0 1 1]
 [0 1 0 1 1]
 [0 1 0 1 0]
 [1 1 1 0 1]
 [1 0 0 1 0]
 [0 0 1 1 1]
 [1 0 0 1 1]
 [1 1 1 1 1]
 [0 0 0 1 0]
 [0 0 1 1 0]
 [0 1 1 0 0]
 [0 0 0 0 0]
 [1 1 0 0 0]
 [1 0 1 0 1]]
```

- $OA(243, 5, 3, 5)$

[10]:
```
OA_Lineal(243,5,3,5)
```

```
Optimal solution found.
Solution:
[[2 0 0 0 2]
 [1 0 1 2 0]
 [0 1 1 2 2]
 ...
 [1 0 0 2 2]
 [0 1 0 1 1]
 [1 2 0 2 1]]
```

[18]:
```
def ejecutar_OALineal(N, k, s, t):
    start_time = time.time()
    if OA_Lineal_Tabla(N, k, s, t) == True:
        solucion_existe = True
    else:
        solucion_existe = False
    end_time = time.time()

    tiempo = end_time - start_time
    return solucion_existe, tiempo

# Lista para almacenar los resultados
resultados = []
```

```
# Lista de parámetros para probar
parametros = [
    (8, 4, 2, 2),(8, 4, 2, 3),(8, 5, 2, 2),(8, 4, 2,␣
 ↪3),(9,3,3,2),(9,4,3,2),(9,5,3,2),(16,4,2,3),(16,5,2,4),
    (25, 4, 5, 2),(25, 5, 5, 2),(25, 6, 5, 2),(27, 3, 3, 3),(27, 4, 3,␣
 ↪3),(27, 5, 3, 3),(32, 5, 2, 3),(32, 5, 2, 4),
    (32, 5, 2, 5), (32,7,2,5),(48,3,4,3),(48,5,4,3), (64, 4, 4, 3), (81,␣
 ↪4, 3, 3), (100, 5, 2, 4), (128, 7, 2, 3)
    (100, 5, 2, 2),(100, 5, 5, 2),(125,5,5,3), (144,5,2,3),(144,5,2,4),␣
 ↪(144, 6, 3, 2), (169, 5, 13, 2), (196, 7, 2, 2),
    (225, 5, 3, 3), (243,5,3,5),  (256, 5, 4, 3), (324, 6, 3, 4), (400,␣
 ↪7, 2, 4), (441, 6, 3, 3), (484, 5, 2, 2), (576, 6, 6, 2),
    (672, 6, 2, 4), (729, 9, 3, 4), (784, 14, 4, 2), (900, 10, 3,␣
 ↪2),(1024, 16, 4, 4), (1024, 16, 4, 5)
    ]

#Se ejecuta el modelo con cada conjunto de parámetros y se guardan los␣
 ↪resultados
for params in parametros:
    N, k, s, t = params
    existencia, tiempo = ejecutar_OALineal(N, k, s, t)
    resultados.append({
        'N': N,
        'k': k,
        's': s,
        't': t,
        'Existencia': existencia,
        'Tiempo': tiempo
    })

#Se convierte la lista de resultados en un DataFrame
resultados_df = pd.DataFrame(resultados)

#Se muestra la tabla de resultados
print(resultados_df)

#Se guarda la tabla de resultados en un archivo CSV
resultados_df.to_csv('resultados_OALineal.csv', index=False)
```