Nombre:

Darwin Domingo Mendez: **2023-0769**
Camily Yelissa Garcia De Los Santos: **2023-1467**
Roither Sanchez Sosa: **2022-2111**
Michael David Garcia: **2023-1256**

Materia:

Inteligencia Artificial Distribuida

Maestro:

Carlos Bienvenido Ogando Motas

Año

2025

PROYECTO – Compra Genética

Camily Yelissa Garcia De Los Santos, Darwin Domingo Méndez, Michael David Garcia Feliz y Roither Sánchez Sosa

Centro de Excelencia de Inteligencia Artificial/ITLA, Santo Domingo Este, Rep. Dom.

RESUMEN: ***El objetivo de este proyecto es diseñar un sistema de optimización de compras en supermercados que minimice el costo total de la compra, incluyendo tanto el precio de los productos como el gasto en combustible. Utilizando algoritmos genéticos, los estudiantes desarrollarán una solución que determine la ruta más eficiente para adquirir todos los productos necesarios, considerando múltiples sucursales de supermercados, cada una con su propio catálogo de productos y distancias específicas desde la casa del comprador.***

PALABRAS CLAVES: *Algoritmos Genéticos, Costo Total, Gasto en Combustible, Optimización de Compras, Producto Necesarios, Supermercado.*

ABSTRACT: ***The objective of this project is to design a supermarket shopping optimization system that minimizes the total cost of the purchase, including both the product prices and fuel expenses. Using genetic algorithms, students will develop a solution that determines the most efficient route to acquire all the necessary products, considering multiple supermarket branches, each with its own product catalog and specific distances from the buyer's home.***

KEYWORDS: *Fuel Expenses, Genetic Algorithms, Product Catalog, Shopping Optimization, Supermarket, Total Cost.*

1. INTRODUCCIÓN

Este proyecto busca desarrollar un sistema de inteligencia artificial utilizando algoritmos genéticos para encontrar la ruta de compra óptima en supermercados, minimizando el costo total que incluye los precios de los productos y el gasto en transporte. Se considerarán diferentes sucursales con sus precios y distancias, optimizando tanto los productos como el recorrido.

El trabajo se divide en tres partes: el diseño de la solución mediante algoritmos genéticos con demostración matemática, la implementación del sistema que reciba entradas como catálogo de productos y lista de compras, y la validación mediante un conjunto de datos para evaluar el rendimiento del algoritmo.

El objetivo es optimizar el proceso de compra, seleccionando las tiendas más convenientes y rutas eficientes, utilizando técnicas de crossover, mutación y selección adaptadas al problema.

2. DISEÑO METODOLÓGICO

El diseño metodológico de este proyecto se basa en el uso de algoritmos genéticos para optimizar la compra en supermercados, considerando tanto los precios de los productos como el gasto en transporte. La metodología se divide en tres fases:

1. **Modelado del problema**: Se definirá un grafo de sucursales de supermercados, donde cada nodo representa una tienda y las aristas las distancias entre ellas. Además, se tendrá un catálogo de productos con sus precios en cada sucursal.
2. **Algoritmos genéticos**: Se implementarán los componentes básicos de un algoritmo genético: selección, crossover, mutación y funciones de fitness. Las funciones de fitness se adaptarán para evaluar la combinación más eficiente de productos y rutas en términos de costo total.
3. **Validación**: Se probará el sistema utilizando un conjunto de datos con problemas de optimización predefinidos. El rendimiento del algoritmo se evaluará con base en la capacidad para minimizar el costo total de compra y transporte, ajustando los hiper parámetros del algoritmo según sea necesario.

3. DESCRIPCIÒN

3.1 Sintàxis de Genes

La sintaxis de los genes que hemos utilizado en nuestro proyecto está representada por un diccionario donde las claves son los nombres de los supermercados o sucursales, como 'Supermercado Nacional' y 'Jumbo'. Dentro de cada sucursal, tenemos otro diccionario con los productos que se venden en esa sucursal y sus respectivos precios.

Por ejemplo, nuestra representación es algo así:

```
{
    'Supermercado Nacional': {
        'Alfombras': 1274.99,
        'Purificadores de aire': 4381.99
    },
    'Jumbo': {
        'Pasta de dientes': 2897.99,
        'Aceite de girasol': 2689.99
    }
}
```

Descripción de la Sintaxis de los Genes:

1. Estructura Principal: La estructura principal de nuestros genes es un diccionario, donde las claves son los nombres de las sucursales o supermercados. Esto nos permite identificar fácilmente en qué supermercado encontramos cada producto.
2. Productos por Sucursal: Dentro de cada sucursal, tenemos un diccionario con los productos disponibles y sus precios. Cada producto tiene como clave el nombre del artículo (por ejemplo, 'Alfombras' o 'Pasta de dientes') y como valor el precio correspondiente.
3. Codificación de la Ruta de Compra: En el contexto de un algoritmo genético, cada uno de estos productos puede ser parte de la "solución" que estamos representando con un individuo (un gen). Es decir, el gen de cada individuo podría ser una secuencia de compras en distintos supermercados, donde cada producto de la lista de compras se compra en una sucursal específica.
4. Posibles Operaciones sobre los Genes:
    - Crossover: Podemos hacer un cruce entre las compras de diferentes individuos, combinando productos de distintos supermercados para generar nuevas combinaciones de compras.
    - Mutación: Podemos alterar la sucursal de un producto para explorar si cambiar de supermercado puede resultar en una mejor solución, ya sea por un precio más bajo o por una mejor ruta de transporte.
    - Selección: Seleccionamos los individuos que tengan los costos más bajos o las rutas más eficientes para que sean parte de la siguiente generación.

Las librerías que importamos fueron:

1. **os**: La usamos para interactuar con el sistema operativo y gestionar archivos y directorios.
2. **pandas**: La usamos para manipular y analizar datos en estructuras como DataFrame.
3. **json**: La usamos para manejar datos en formato JSON, tanto para leer como escribir archivos.
4. **matplotlib.pyplot**: La usamos para crear gráficos y visualizar datos con el estilo 'ggplot'.
5. **random**: La usamos para generar números aleatorios en nuestras simulaciones.
6. **numpy**: La usamos para realizar cálculos numéricos y manejar arrays grandes.
7. **pprint**: La usamos para imprimir estructuras de datos de manera legible.
8. **copy**: La usamos para hacer copias de objetos en memoria.
9. **logging**: La usamos para registrar eventos y errores durante la ejecución del código.
10. **ast**: La usamos para trabajar con árboles sintácticos abstractos y analizar código Python.
11. **time**: La usamos para medir el tiempo de ejecución y hacer pausas cuando es necesario.
12. **psutil**: La usamos para obtener información sobre el uso de recursos del sistema, como CPU y memoria.
13. **Counter**: La usamos para contar la frecuencia de elementos en colecciones de datos.
14. **google.colab.drive**: La usamos para integrar Google Drive y gestionar archivos dentro de Colab.

3.2 FUNCIONES

Función read_catalogo_grafo_fuel

Propósito:

Esta función se encarga de leer un archivo Excel basado en un índice proporcionado y extraer tres elementos clave: el catálogo de productos, el grafo de distancias y el precio del combustible.

Entrada: Recibe un índice índice que se usa para acceder al archivo específico dentro de un conjunto de datos.

Salida: Devuelve el catálogo, el grafo y el fuel_price correspondientes al índice dado.

Método __init__

Propósito:
Este es el método constructor de la clase GAOptimization. Inicializa los parámetros del algoritmo genético, configura el logger para registrar la ejecución del algoritmo y carga los datos necesarios para la optimización (catálogo, grafo, precios, etc.).

Método read_catalogo_grafo_fuel

Propósito:
Lee un archivo Excel basado en un índice dado y extrae los datos necesarios: catálogo de productos, gráfico de distancias y precio del combustible.

Método get_distance_matrix

Propósito:
Genera una matriz de distancias a partir del grafo, representando las distancias entre nodos de forma tabular y rellenando valores nulos con infinito (np.inf).

Método get_tienda_dict

Propósito:
Convierte el catálogo de productos en un diccionario organizado por tienda, donde la clave es la tienda y el valor es otro diccionario con los productos y sus precios.

Método generar_gen

Propósito:
Genera un gen aleatorio en forma de diccionario que representa una tienda y los productos comprados en ella, asegurándose de que los productos seleccionados no se repitan.

Entrada:

productos_seleccionados (set): Un conjunto de productos que ya han sido seleccionados en el cromosoma.

Salida:

Devuelve un diccionario con una tienda y los productos seleccionados de esa tienda, o None si no hay productos disponibles.

Método generar_cromosoma

Propósito:
Este método genera un cromosoma completo para el algoritmo genético, representado como un diccionario en el que las claves son las tiendas y los valores son diccionarios con los productos comprados y sus respectivos precios. El método asegura que un mismo producto no se compre en más de una tienda.

Entrada:
No recibe parámetros de entrada.

Proceso:

- Inicializa un conjunto para realizar el seguimiento de los productos seleccionados y un diccionario para almacenar el cromosoma.
- Calcula el conjunto total de productos únicos en el catálogo.
- En un ciclo, genera un gen aleatorio con productos no seleccionados y lo agrega al cromosoma.
- Si el gen no es válido (no quedan productos disponibles), asigna los productos faltantes a tiendas al azar.
- Se asegura de que los productos no se repitan entre tiendas.

Salida:
Devuelve el cromosoma completo validado, en el formato {Tienda: {Producto: Precio}}.

Método generar_poblacion_inicial

Propósito:
Este método genera una población inicial de cromosomas para el algoritmo genético, asegurando que todos los cromosomas en la población sean únicos.

Entrada:

- n (int): Número de cromosomas que se desean generar para la población.

Proceso:

- Inicializa una lista vacía para almacenar los cromosomas y un conjunto para realizar el seguimiento de cromosomas únicos.
- En un ciclo, genera cromosomas aleatorios hasta que se haya alcanzado el número deseado (n), asegurando que no se repitan utilizando un conjunto de cromosomas vistos.

Salida:

Devuelve una lista de cromosomas únicos generados.

Método generar_poblacion_inicial

Propósito:

Este método genera una población inicial de cromosomas para el algoritmo genético, asegurando que todos los cromosomas en la población sean únicos.

Entrada:

n (int): Número de cromosomas que se desean generar para la población.

Proceso:

- Inicializa una lista vacía para almacenar los cromosomas y un conjunto para realizar el seguimiento de cromosomas únicos.
- En un ciclo, genera cromosomas aleatorios hasta que se haya alcanzado el número deseado (n), asegurando que no se repitan utilizando un conjunto de cromosomas vistos.

Salida:

Devuelve una lista de cromosomas únicos generados.

Método evaluar_fitness

Propósito:

Este método calcula el "fitness" de un cromosoma, es decir, evalúa su calidad según dos componentes principales: el costo de los productos comprados y el costo de los viajes entre las tiendas (y la casa). Además, penaliza cualquier tramo desconectado (cuando no hay una ruta válida entre tiendas) aplicando una penalidad configurable.

Entrada:

cromosoma (dict): Un diccionario que representa el cromosoma con tiendas como claves y productos con sus precios como valores. Ejemplo: {tienda1: {producto1: precio1, producto2: precio2}, tienda2: {producto3: precio3}}.

Proceso:

- **Costo de los productos:** Se suma el costo de todos los productos comprados en el cromosoma.
- **Construcción de la ruta:** Se forma un recorrido en el que se parte de "Casa", se visitan las tiendas (según el cromosoma) y luego se regresa a "Casa".
- **Cálculo de la penalidad:** Se calcula la penalidad en caso de que haya distancias infinitas (es decir, tramos desconectados) en la ruta.
- **Costo de viaje:** Se calcula el costo de viajar entre las tiendas, aplicando la penalidad si se encuentran tramos desconectados.

Salida:
Devuelve el "fitness" del cromosoma, que es la suma del costo total de los productos y el costo del viaje (con penalizaciones si es necesario). Valores altos de fitness indican rutas inválidas o costosas.

Método selection

Propósito:
Este método selecciona cromosomas de una población para la reproducción, utilizando un método de selección basado en su "fitness" (aptitud). Existen dos métodos de selección disponibles: "soft" (suave) y "hard" (firme).

Entrada:

fitness_scores (list): Una lista de tuplas donde cada tupla contiene un valor de fitness y un cromosoma asociado, representado como (fitness, cromosoma).

Proceso:

Se calcula cuántos cromosomas deben ser seleccionados según la tasa de selección (selection_rate).

Según el método de selección:

Soft: Se asigna una probabilidad inversa al fitness para cada cromosoma. Los cromosomas con mejor fitness tienen más probabilidades de ser seleccionados.

Hard: Los cromosomas se ordenan según su fitness y se seleccionan los mejores.

Salida:

Devuelve una lista de cromosomas seleccionados para la reproducción.

Método cross_uniform

Propósito:

Este método aplica un cruce uniforme entre dos cromosomas padres, generando dos cromosomas hijos. Cada producto puede ser heredado de uno u otro padre con una probabilidad uniforme, respetando tasas de cruce y herencia para introducir diversidad en los hijos.

Entrada:

- parent1 (dict): Cromosoma padre 1, representado como un diccionario donde las claves son tiendas y los valores son diccionarios de productos y precios.
- parent2 (dict): Cromosoma padre 2 en el mismo formato.

Proceso:

- Si no se cumple la tasa de cruce (crossover_rate), los hijos son copias idénticas de los padres.
- Si se cumple la tasa de cruce, para cada tienda y producto se decide cuál de los dos padres heredará ese producto según las probabilidades de cruce y herencia.
- Luego, se eliminan tiendas vacías y se completan productos faltantes en los hijos.

Salida:

Devuelve dos cromosomas hijos válidos, respetando las restricciones de diversidad y herencia.

Método cross_sub_tree

Propósito:

Este método implementa un cruce de tipo "sub-tree" entre dos cromosomas padres. Intercambia la asignación completa de productos entre una tienda aleatoria de cada cromosoma, preservando la unicidad de los productos en cada cromosoma hijo.

Entrada:

- parent1 (dict): Cromosoma padre 1, representado como un diccionario donde las claves son tiendas y los valores son diccionarios de productos y precios.
- parent2 (dict): Cromosoma padre 2 en el mismo formato.

Proceso:

- Se realiza una copia profunda de los dos cromosomas padres.
- Si alguno de los cromosomas está vacío, se devuelve tal cual.
- Se selecciona aleatoriamente una tienda de cada padre.
- Se extrae la asignación de productos de cada tienda y se intercambia entre los dos cromosomas hijos, insertando la tienda en el cromosoma opuesto.

Se llama a una función cleanup para:

- Eliminar productos duplicados en las tiendas.
- Eliminar tiendas vacías.
- Reintegrar productos faltantes en tiendas válidas.

Salida:

Devuelve dos cromosomas hijos generados a partir del cruce "sub-tree" que están validados y listos para ser utilizados en las siguientes generaciones.

Método crossover

Propósito:

Este método despacha el proceso de cruce (crossover) entre dos cromosomas padres, eligiendo el tipo de cruce basado en la configuración self.crossover_params['method']. Los métodos de cruce disponibles son "uniforme" y "sub-tree", y se aplican dependiendo de la configuración.

Entrada:

- parent1 (dict): El primer cromosoma padre, representado como un diccionario donde las claves son tiendas y los valores son diccionarios de productos y precios.
- parent2 (dict): El segundo cromosoma padre en el mismo formato.

Proceso:

- El método de cruce se obtiene de la configuración self.crossover_params["method"].
- Si el método es "uniform", se llama al método cross_uniform para aplicar el cruce uniforme.
- Si el método es "sub_tree", se llama al método cross_sub_tree para aplicar el cruce tipo sub-tree.
- Si el método no es ninguno de estos dos, se lanza un error ValueError.

Salida:

- Devuelve una tupla de dos cromosomas hijos generados según el método de cruce seleccionado. Cada cromosoma hijo es un diccionario representando las tiendas y los productos adquiridos en cada tienda.

Excepciones:

Si el método de cruce no es reconocido (es diferente de "uniform" o "sub_tree"), se lanza un ValueError.

Método _mutate_swap

Propósito:
Este método realiza una mutación de tipo "swap" sobre un cromosoma, es decir, intercambian productos entre tiendas o mueve productos a tiendas diferentes. Dependiendo del caso (una sola tienda o múltiples tiendas), el método maneja la mutación de manera distinta.

Entrada:

crom (dict): El cromosoma original representado como un diccionario {tienda: {producto: precio}}.

Proceso:

Se realiza una copia profunda del cromosoma para evitar modificar el original directamente.

Caso 1 (Una sola tienda en el cromosoma):

- Si el cromosoma tiene solo una tienda, se selecciona un producto aleatorio de esa tienda.
- Se busca una tienda diferente que también venda ese producto.
- Si existe una tienda válida, se mueve el producto a esa tienda, actualizando el precio.
- Si la tienda original queda vacía después de mover el producto, se elimina del cromosoma.

Caso 2 (Múltiples tiendas en el cromosoma):

- Si hay varias tiendas en el cromosoma, se realiza un intercambio (swap) de productos entre dos tiendas diferentes.
- El intercambio se realiza solo si ambas tiendas venden los productos del otro.
- Se actualizan los precios de los productos según la tienda destino.
- Si alguna tienda queda vacía después del intercambio, se elimina del cromosoma.

Salida:

Retorna un nuevo cromosoma mutado, donde se intercambian productos entre tiendas o se mueve un producto a otra tienda, eliminando las tiendas vacías si es necesario.

Método _mutate_scramble

Propósito:
Este método realiza una mutación de tipo "scramble" sobre un cromosoma. La idea es mover aleatoriamente un subconjunto de productos de su tienda original a otras tiendas que también los venden, manteniendo la unicidad de los productos en el cromosoma.

Entrada:

crom (dict): El cromosoma original representado como un diccionario {tienda: {producto: precio}}.

Proceso:

Se realiza una copia profunda del cromosoma para evitar modificar el original directamente.

Mapeo de productos a tiendas:

- Se construye un diccionario que mapea cada producto a la tienda en la que se encuentra actualmente (product_to_store).
- Se genera una lista de todos los productos en el cromosoma.

Selección aleatoria de productos:

- Se selecciona aleatoriamente un subconjunto de productos a mutar, con al menos un producto seleccionado.

Reasignación de productos:

Para cada producto seleccionado:

- Se busca una tienda diferente a la tienda original que también venda ese producto.
- Si se encuentra una tienda válida, el producto se mueve allí, y el precio se actualiza con el precio de esa tienda.
- Si la tienda original queda vacía tras mover el producto, se elimina del cromosoma.

Salida:

Retorna un nuevo cromosoma mutado, donde los productos seleccionados han sido movidos a nuevas tiendas, y las tiendas vacías han sido eliminadas.

Método mutation

Propósito:

Este método aplica una mutación sobre un cromosoma con una probabilidad definida en self.mutation_params['mutation_rate']. La mutación se puede realizar de dos maneras: "swap" (intercambiando productos entre tiendas) o "scramble" (moviendo productos entre tiendas). Además, cualquier producto faltante tras la mutación se reinserta en el cromosoma.

Entrada:

crom (dict): El cromosoma original representado como un diccionario {tienda: {producto: precio}}.

Proceso:

1. **Copia del cromosoma:**
   Se crea una copia profunda del cromosoma para evitar modificar el original directamente.
2. **Verificación de la probabilidad de mutación:**
   Si un número aleatorio es mayor que mutation_rate, el cromosoma original se devuelve sin cambios.
3. **Selección del método de mutación:**
   Si el valor de self.mutation_params['method'] es "swap", se llama al método _mutate_swap. Si es "scramble", se llama al método _mutate_scramble. Si el método no es válido, se lanza una excepción.
4. **Reinserción de productos faltantes:**
   Después de realizar la mutación, se verifica si algún producto está faltando en el cromosoma. Si es así, se reinserta el producto en una tienda válida que lo venda, actualizando el precio con el correspondiente de la tienda.

Salida:

Retorna un nuevo cromosoma mutado. Si la mutación fue exitosa, contiene los cambios correspondientes y se eliminan tiendas vacías o productos faltantes.

Método sumar

Propósito:

Este método calcula el costo total de una lista de cromosomas. El costo total se calcula sumando el precio de los productos de cada cromosoma y añadiendo el costo de viaje entre las tiendas, considerando la matriz de distancias y el precio del combustible.

Entrada:

- serie_cromosoma (list): Lista de cromosomas, donde cada cromosoma es un diccionario en el formato {tienda: {producto: precio}}.
- distance_matrix (pd.DataFrame): Matriz de distancias que indica las distancias entre las ubicaciones de las tiendas.
- fuel_price (float): Precio del combustible por unidad de distancia.

Proceso:

- **Inicialización de la lista de costos:**
  Se crea una lista total para almacenar el costo total de cada cromosoma.
- **Cálculo del costo de los productos:**
  Para cada cromosoma, se recorren todas las tiendas y productos, sumando los precios de los productos.
- **Cálculo del costo de viaje:**
  Se construye la ruta de viaje, que incluye las tiendas y la "Casa" al principio y al final del recorrido. Luego, se calcula la distancia total entre las tiendas usando la matriz de distancias y se multiplica por el precio del combustible.
- **Suma de los costos:**
  Se suma el costo de los productos y el costo del viaje para obtener el costo total del cromosoma.

Salida:

Retorna una lista total que contiene el costo total de cada cromosoma en la lista de entrada.

Método _validate_cromosoma

Propósito:

Este método valida un cromosoma asegurando que todos los productos del catálogo estén asignados a alguna tienda. Si algún producto falta, se asigna a una tienda válida que lo venda.

Entrada:

crom (dict): El cromosoma a validar, representado como un diccionario de tiendas con productos y sus precios.

Proceso:

- **Identificación de productos faltantes:**
  Se compara la lista de productos presentes en el cromosoma con todos los productos disponibles en el catálogo. Si falta algún producto, se agrega aleatoriamente a una tienda que lo venda.
- **Reinserción de productos faltantes:**
  Para cada producto faltante, se selecciona aleatoriamente una tienda que lo ofrezca y se reintegra al cromosoma con el precio correspondiente.

Salida:

- Retorna el cromosoma validado con todos los productos presentes y asignados a tiendas válidas

Método run

Propósito:

Este es el ciclo principal de la evolución. En cada época, se evalúa la población, se seleccionan los mejores cromosomas, se aplica el cruce y la mutación para generar la siguiente generación, y se guarda el mejor cromosoma de cada generación.

Entrada:

Ninguna entrada directa, ya que utiliza los atributos de la clase.

Proceso:

- **Evaluación de la población:**
  Se evalúan los cromosomas de la población actual utilizando la función evaluar_fitness.
- **Selección de padres:**
  Se seleccionan los cromosomas con mejor fitness para la reproducción.
- **Generación de la siguiente generación:**
  Se genera la siguiente generación mediante cruce y mutación de los cromosomas seleccionados.

- **Elitismo:**
  Se conserva el mejor cromosoma de la generación anterior para asegurar que el mejor individuo no se pierda.
- **Actualización de la población:**
  La población se actualiza con los cromosomas de la nueva generación.

Salida:

Retorna el mejor cromosoma encontrado y su valor de fitness después de todas las épocas.

Funciones Principales:

1. **Inicialización (__init__)**:
   - Inicializa varios parámetros, incluidos el algoritmo genético (ga), el conjunto de problemas (problem_set), y los parámetros para los problemas pequeños, medianos y grandes.
2. **Configuración del Proyecto (conectar_proyecto)**:
   - Se asegura de que el directorio correcto esté siendo utilizado y crea la estructura de directorios necesaria para almacenar los resultados de los registros.
3. **Procesamiento de Consultas (make_query)**:
   - Convierte la mejor solución encontrada por el algoritmo genético en una secuencia de pares tienda-producto y calcula el costo total, que incluye tanto el costo de los productos como el costo de transporte entre tiendas.
4. **Flujos de Optimización**:
   - flujo_small, flujo_medium y flujo_big: Estas funciones configuran y ejecutan el algoritmo genético para problemas pequeños, medianos y grandes respectivamente, utilizando los parámetros pasados durante la inicialización.
   - Cada una de estas funciones devuelve el mejor puntaje y la mejor solución encontrada por el algoritmo genético.
5. **Flujo Principal (flujo_general)**:

- Esta función orquesta todo el proceso. Recorre las filas en el conjunto de problemas y ejecuta el flujo de optimización correspondiente (pequeño, mediano o grande) según el grupo de problemas.
- Realiza un seguimiento de diversas métricas (como la mejor solución, el puntaje de ajuste, el tiempo de ejecución, el uso de memoria y otros parámetros) y los almacena en un diccionario self.insights.
- Después de cada paso de optimización, almacena la mejor solución en el directorio de registros correspondiente.

6. **Cálculo de Costos**:
   - La función make_query toma una solución y calcula el costo total considerando tanto el costo del producto como el costo de transporte entre las tiendas.
7. **Registro de Resultados**:

- Después de ejecutar la optimización para cada grupo de problemas (pequeño, mediano o grande), la mejor solución se guarda en un archivo .json para su
- posterior análisis.

4. HALLAZGOS Y CONCLUSIONES A PARTIR DEL BENCHMARK

Comparación entre los nuestros

Después de comparar todas las métricas de error, queda claro que **hard_uniform_swap** es la mejor configuración para nuestro objetivo de minimizar el precio:

- ENP (≈ –3 160): subestima menos el precio real que cualquier otra combinación.

- MAE (≈ 2 563): tiene la menor desviación absoluta promedio, es decir, sus predicciones están más cerca del valor real.

- MSE (≈ 2.36×10^7): registra el menor valor, lo que indica que evita errores grandes mejor que el resto.

- Boxplots de error puntual y absoluto: muestran que su mediana está casi en cero y presenta la menor dispersión.

En resumen, hard_uniform_swap ofrece precios bajos y consistentes, cumpliendo perfectamente con el criterio de “menos es mejor”.

Error_Og_ga_Vs_our_ga

Igual o mejor que Og
30%

Error_Og_qbf_Vs_our_ga

Igual o mejor que og
26%

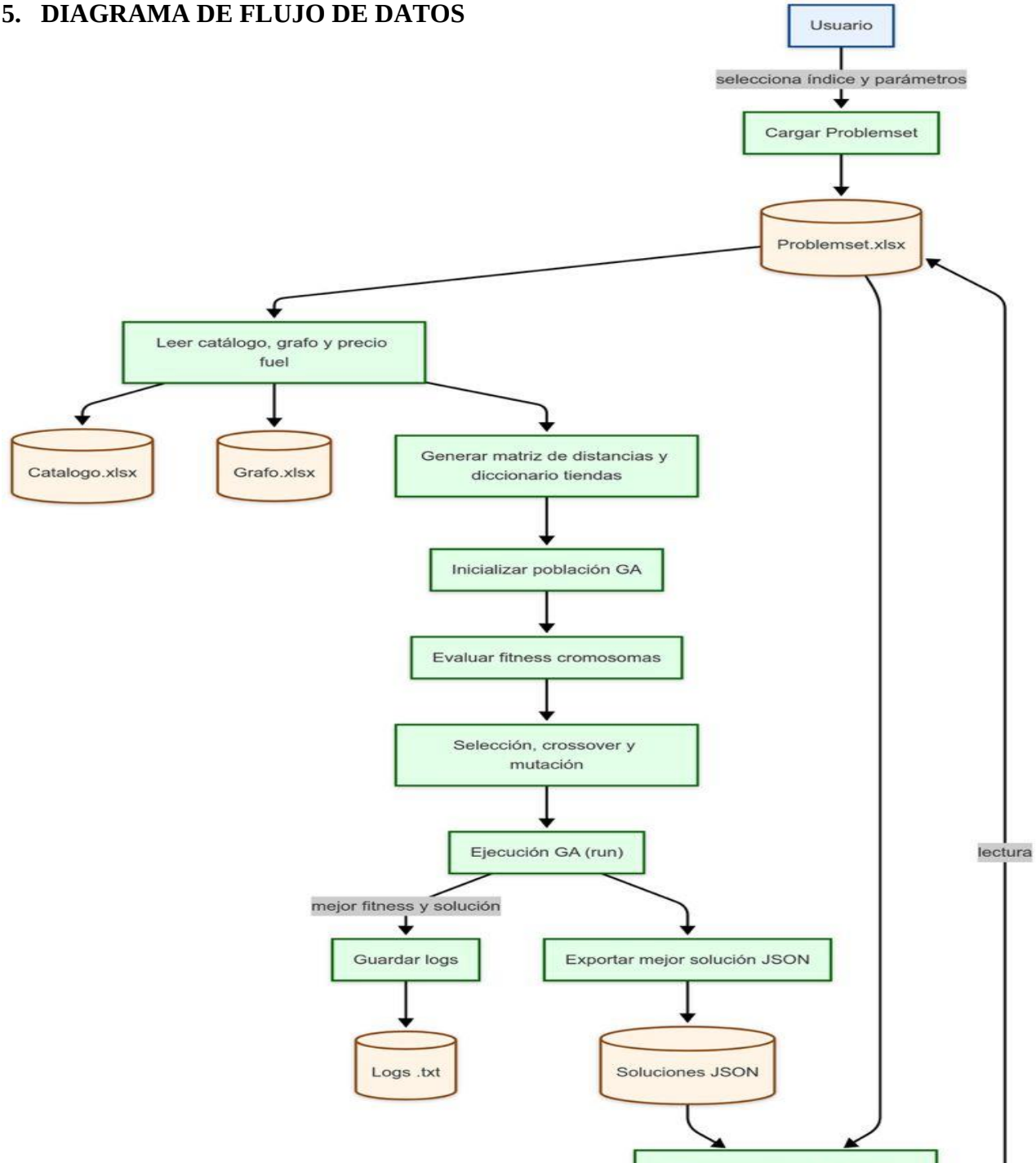
Error_Og_gc_Vs_our_ga

Igual o mejor que og
17%

Error_Og_gn_Vs_our_ga

Igual o mejor que og
88%

5. DIAGRAMA DE FLUJO DE DATOS



6. CONCLUSIÒN

Este proyecto ha demostrado cómo los algoritmos genéticos pueden ser una herramienta poderosa para optimizar la compra en supermercados, integrando eficientemente la selección de productos y la planificación de rutas de transporte. Mediante el diseño y la implementación de un sistema basado en algoritmos genéticos, hemos sido capaces de minimizar el costo total de las compras, considerando tanto los precios de los productos como los gastos de transporte entre las diferentes sucursales.

El proceso ha seguido una metodología clara de modelado del problema, aplicación de los algoritmos genéticos y validación de los resultados. Las librerías utilizadas han permitido gestionar y analizar los datos, realizar simulaciones eficientes y medir el rendimiento del sistema. En cuanto a los resultados, el algoritmo ha demostrado ser competitivo, con una clara mejora sobre soluciones convencionales al ofrecer un balance entre precios bajos y rutas eficientes.

La validación de las soluciones generadas, evaluadas a través de diversas métricas de error como ENP, MAE y MSE, ha mostrado que la configuración "hard_uniform_swap" ha logrado los mejores resultados, con menores desviaciones y precios más consistentes. Esto confirma la efectividad del algoritmo para lograr la optimización deseada.

En resumen, el proyecto ofrece una solución práctica y eficiente para la optimización del proceso de compra, aplicando técnicas avanzadas de inteligencia artificial para mejorar tanto el costo de los productos como el gasto en transporte, brindando un valor real a los consumidores y minoristas.

7. BIBLIOGRAFÌA

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm

https://www.nature.com/articles/s41598-023-39696-0