# Proyecto Final.
# Irreverentes.

Guzmán Cabrera Álvaro, Garcia Parra Jair Omar
*CENTRO UNIVERSITARIO DE CIENCIAS*
*EXACTAS E INGENIERÍAS, (CUCEI, UDG)*
alvaro.guzman3930@alumnos.udg.mx
jair.garcia5340@alumnos.udg.mx
moises.sotelo5881@alumnos.udg.mx

***Abstract*— El problema del enrutamiento de vehículos es un problema que a través de los años ha tenido distintos tipos de enfoque hasta llegar a lo que es hoy, básicamente consiste en determinar la mejor ruta posible a seguir por uno o varios vehículos, cubriendo todos los puntos y minimizando los costos. Fue solucionado usando python mediante el algoritmo genético el cual es considerado un algoritmo metaheurístico para de este modo cumplir con los lineamientos propuestos por el profesor. El resultado de esta implementación nos da como resultado una cantidad de rutas dependiendo de los datos usados la cual es la más optima posible de entre las propuestas durante el proceso, pero no siendo siempre está la mejor solución al problema. Como conclusión general al equipo nos pareció un proyecto algo complicado en un principio, pero al ir avanzando en este mismo pudimos cumplir el objetivo y comprender de que se trataba en general tanto el problema como el tipo de solución orientada a la metaheurística.**

***Palabras claves* – Metaheurística, Optimización, Grafos, Ruta, Logística, restricciones, problema de ruteo de vehículos, problema del agente viajero, optimización combinatoria, heurístico.**

**Repositorio de código: https://github.com/jairgg/Proyecto-Analisis-de-Algoritmos**
**Versión actual del código: Primera versión**

## I. Introducción

Problema del enrutamiento de vehículo (VRP), es un problema de optimización combinatoria, este problema consiste en encontrar la ruta más corta posible la cual recorra una cantidad x de puntos, teniendo uno de partida y regresando a este mismo al finalizar. Se abordará este mismo con un enfoque pensado en los procesos de la metaheurística, para así evaluar el rendimiento de este tipo de solución.

Algoritmo genético: El algoritmo que se usó en la elaboración del codigo fue el “algoritmo genético” el cual consiste en una serie de soluciones genéticas como selección, cruce y mutación para generar una cantidad de soluciones posibles y evolucionarlas para encontrar una solución óptima.

El funcionamiento de este tipo de algoritmos consiste en:

- Inicialización: Donde se genera una población de soluciones de un problema.
- Evaluación: Se evalúa cada uno de los individuos de esta población utilizando una función de aptitud que determine qué tan buena es la solución con relación al problema que se esté resolviendo.
- Selección: Se seleccionan los individuos más aptos para reproducirse y crear la siguiente generación.
- Crossover: Donde se combinan pares de individuos (padres) para producir nuevos individuos (hijos).
- Mutación: Se introducen pequeñas variaciones en los cromosomas de algunos individuos. Esto se hace cambiando algunos bits aleatoriamente, para mantener la diversidad genética y evitar la convergencia prematura.
- Reemplazo: Se forma una nueva generación de individuos, ya sea reemplazando completamente a la anterior o combinando parte de la antigua con los nuevos.
- Iteración: Los pasos vistos de evaluación, selección, cruzamiento y mutación se repiten durante muchas generaciones hasta que se alcanza un criterio de parada (puede ser, un número máximo de generaciones o una solución suficientemente buena)

Algunas ventajas que trae la utilización de los algoritmos genéticos pueden ser:

- Optimización global: Los algoritmos genéticos son buenos para explorar grandes espacios de búsqueda y pueden evitar quedarse atrapados en óptimos locales.
- Flexibilidad: Pueden aplicarse a una amplia variedad de problemas, tanto lineales como no lineales, y con funciones de aptitud que no necesitan ser derivables.
- Adaptabilidad: Pueden encontrar soluciones en problemas dinámicos y cambiantes.
- Robustez: Funcionan bien en problemas donde la información es incompleta o ruidosa.

Y algunas desventajas que pueden surgir en la utilización de esta clase de algoritmos incluyen:

- Tiempo computacional: Este tipo de algoritmos pueden ser computacionalmente intensivos, especialmente si la evaluación de la función de aptitud es costosa
- Parámetros sensibles: El rendimiento de esta clase de algoritmos depende de la elección de parámetros como tasas de mutación, cruzamiento, tamaño de la población, etc.
- Convergencia lenta: Pueden ser necesarias muchas generaciones para encontrar una solución óptima, y la velocidad de convergencia no siempre es predecible.
- Complejidad de implementación: En algunos casos, diseñar una buena representación del "cromosoma" y una función de aptitud adecuada puede ser complicado.

## II. Trabajos Relacionados

A partir de esta sección, se desarrollan los contenidos del proyecto modular, de una forma ordenada y secuencial. Nótese que la sección debe ir organizada usando títulos como el anterior para cada tema nuevo incluido. Aparte, se incluyen subtítulos como el siguiente.

Algunos trabajos relacionados que pudimos encontrar en nuestra búsqueda para recabar información acerca del problema de ruteo y la metaheurística:

"El empleo de modelos metaheurísticos en la logística industrial. El caso del enrutamiento de vehículos": El cual propone algunas maneras de solucionar este problema como lo es por optimización, complejidad computacional, la heurística y la metaheurística. Además, nos pareció un trabajo muy completo al incluir ciertos tipos de restricciones y algunos agregados mas que nos sumaron nociones acerca de el tratamiento que se le suele dar al problema. (Boza, 2012) [1]

"Metaheurísticas para el problema de ruteo de vehículos con ventanas de tiempo (VRP-TW)": Este trabajo nos pareció uno de los mas completos ya que explica de lo más básico hasta lo mas complejo que puede surgir en la resolución de este problema y aunque algunas cosas eran más de lo que estábamos buscando encontrar otras nos sirvieron bastante como la inclusión de varias técnicas de solución. (Edwin, 2017) [2]

"Resolución del problema de enrutamiento de vehículo con limitaciones de capacidad utilizando un procedimiento metaheurístico de dos fases": Nos pareció un trabajo que seguía el formato propuesto para nuestro proyecto además incluía mucho material visual lo que nos ayudo a entender mejor su funcionamiento y explicación. (Julio, 2009) [3]

## III. Descripción del Desarrollo del Proyecto Modular

### *A) Problema de Enrutamiento de vehículos (VRP)*

El problema de enrutamiento de vehículos como ya se mencionó anteriormente consiste en determinar la mejor manera de asignar un conjunto de vehículos a un conjunto de clientes, con el objetivo de minimizar los costos totales.

Este problema fue propuesto por primera vez por George Dantzig y John Ramser, quienes propusieron una aproximación algorítmica que fue aplicada para la entrega de gasolina en estaciones de servicio.

Hay tres diferentes aproximaciones principales a la modelización el VRP:

- Formulaciones de flujo del vehículo - Esto utiliza variables de entero asociado con cada arco que cuenta el número de veces que la arista es recorrida por un vehículo. Es generalmente utilizado para el básico VRPs. Esto es bueno para casos donde el coste de la solución puede ser expresado como la suma de los costes asociado con los arcos. Pero aun asi, puede no ser usado en muchas aplicaciones prácticas.
- Formulaciones de flujo de la mercancía - Variables de entero adicional están asociadas con los arcos o aristas qué representan el flujo de las mercancías a lo largo de los caminos recorrido por los vehículos. Esto sólo ha sido utilizado recientemente encontrar una solución exacta.
- Particionar el problema en conjuntos - Tienen un número exponencial de variables binarias qué es asociado con una ruta factible diferente. El VRP es entonces formulado como un problema qué pregunta cuales la colección de rutas con coste mínimo que satisface las restricciones del VRP.

### *B) Lenguaje de programación*

El lenguaje de programación que se utilizará para la resolución de este problema será Python siguiendo con la dinámica que se ha tenido a lo largo del semestre de utilizarlo como lenguaje principal para las tareas propuestas.

### *C) Desarrollo del código*

Para resolver el problema Vehicle Routing Problem (VRP) con un algoritmo metaheurístico como nos fue designado, primero necesitamos aprender un poco más sobre estos y que opciones teníamos disponibles para implementar, siendo finalmente escogido el algoritmo genético como nuestro algoritmo metaheurístico para la resolución de este problema.

Inicialmente se agregaron las librerías que requeriríamos para el desarrollo de las funciones a implementar, siendo estas random, numpy, matplotlib e itertools.

```python
import random
import numpy as np
import matplotlib.pyplot as plt
import itertools
```

El problema, así como nuestro algoritmo genético requiere de distintos parámetros iniciales para funcionar, los cuales se dividen en propios del problema VRP (num_clientes, num_vehiculos y deposito) y propios del algoritmo genético (tamaño_poblacion, generaciones y tasa_mutacion).

```python
# Parámetros del problema
num_clientes = 5  # Número de clientes
num_vehiculos = 3  # Número de vehículos
deposito = 0  # Índice del depósito

tamano_poblacion = 20  # Tamaño de la población para el algoritmo genético
generaciones = 50  # Número de generaciones para el algoritmo genético
tasa_mutacion = 0.05  # Tasa de mutación para el algoritmo genético
```

Al no encontrar una correcta base de datos la cual nos otorgara ubicaciones de “clientes” para poder trabajar y probar el código, se opto por la creación de muestras aleatorias, las cuales se generarían automáticamente dependiendo del numero de clientes establecidos anteriormente.

```python
# Generar ubicaciones de clientes aleatorias
ubicaciones = np.random.rand(num_clientes + 1, 2)
```

Existen dos funciones que nuevamente son propias únicamente del problema y no tanto del algoritmo genético. La primera de ellas es la función “distancia”, que mediante cálculos obtiene la distancia euclidiana siendo la distancia "ordinaria" entre dos puntos de un espacio euclídeo, la cual se deduce a partir del teorema de Pitágoras.

```python
# Función de distancia euclidiana
def distancia(punto1, punto2):
    return np.sqrt(np.sum((punto1 - punto2) ** 2))
```

La segunda función llamada “calcular_aptitud” nos ayuda a ya teniendo una ruta dada y haciendo uso de la función “distancia”, obtener el recorrido o la distancia totales recorrida por ruta o rutas (en caso de ser varios lo vehículos).

```python
# Evaluación de aptitud (fitness)
def calcular_aptitud(rutas):
    distancia_total = 0
    for ruta in rutas:  # Para cada ruta
        if len(ruta) > 0:  # Si la ruta no está vacía
            distancia_total += distancia(ubicaciones[deposito], ubicaciones[ruta[0]])  # Dist
            for i in range(len(ruta) - 1):  # Para cada par de clientes en la ruta
                distancia_total += distancia(ubicaciones[ruta[i]], ubicaciones[ruta[i + 1]])
            distancia_total += distancia(ubicaciones[ruta[-1]], ubicaciones[deposito])  # Dis
    return distancia_total  # Devuelve la distancia total
```

Una vez teniendo las funciones y los parámetros básicos se requieren para satisfacer la raíz del problema, se desarrolló el algoritmo genético que seria utilizado para la resolución de este.

“crear_poblacion_inicial” como primera función del nuestro algoritmo genético (AG) es la encargada de crear de manera completamente al azar la ruta o rutas a seguir por los vehículos designados. Esta enumera las ubicaciones que se tienen enlistadas para posteriormente mezclarlas y partirlas (según el número de vehículos se requiera), generando de esta forma una ruta completamente nueva y al azar, la cual será agregada a una lista llamada “población” (llamada así y otras variables al seguir la lógica de la base genética).

```python
# Inicialización de la población
def crear_poblacion_inicial():
    poblacion = []
    for _ in range(tamano_poblacion):
        clientes = list(range(1, num_clientes + 1))  # Lista de clientes
        random.shuffle(clientes)  # Mezclar aleatoriamente los clientes
        puntos_division = sorted(random.sample(range(1, num_clientes), num_vehiculos - 1))  # Puntos de divi
        individuo = [clientes[i:j] for i, j in zip([0] + puntos_division, puntos_division + [num_clientes])]
        poblacion.append(individuo)  # Añadir el individuo a la población
    return poblacion  # Devolver la población
```

Enseguida se encuentra la función “selección_aleatoria”, la cual su único propósito es escoger de manera aleatoria una ruta de la lista población.

```python
# Selección aleatoria
def seleccion_aleatoria(poblacion):
    return random.choice(poblacion)
```

El algoritmo genético es llamado y diferenciado de tal forma por su acercamiento e inspiración es el proceso biológico de evolución para obtener el mejor resultado que le sea posible. Este concepto es importante para el desarrollo y entendimiento de las funciones restantes de nuestro AG.

Comenzando por el primer paso, la función “cruzamiento” se encarga de obtener una parte o fracción de dos rutas diferentes, esto lo hace por un proceso en el se toma dos “padres” dos rutas de nuestra población, estas serán trabajadas por dos funciones internas de esta “cruzamiento”. “aplanar” deshará el formato de “rutas” convirtiendo la lista de listas en una única lista; “des_aplanar” reconstruirá una lista de listas para la creación del o los “hijos”.

Por último la función “crear_hijo” también interna, sera la encargada de crear dos nuevas rutas, esto mediante la combinación y sustitución de partes de un padre por el segmento inicialmente extraído del otro padre y viceversa.

```python
# Cruzamiento de orden corregido
def cruzamiento(padre1, padre2):
    tamano = sum(len(ruta) for ruta in padre1)  # Tamaño tot
    corte1, corte2 = sorted(random.sample(range(tamano), 2))
```

```python
def aplanar(padre):
    return [item for sublist in padre for item in sublist]
```

```python
def des_aplanar(lista_planar, longitudes):
    resultado = []
    idx = 0
    for longitud in longitudes:
        resultado.append(lista_planar[idx:idx + longitud])
        idx += longitud
    return resultado
```

```python
segmento1 = aplanar(padre1)[corte1:corte2]  # Segmento del primer padre
segmento2 = aplanar(padre2)[corte1:corte2]  # Segmento del segundo padre

def crear_hijo(segmento, padre):
    padre_planar = aplanar(padre)  # Aplanar el padre
    nuevo_planar = [item for item in padre_planar if item not in segmento]
    nuevo_planar[corte1:corte1] = segmento  # Insertar el segmento
    longitudes = [len(ruta) for ruta in padre]  # Longitudes de las rutas
    return des_aplanar(nuevo_planar, longitudes)  # Reconstruir el hijo

hijo1 = crear_hijo(segmento2, padre1)  # Crear primer hijo
hijo2 = crear_hijo(segmento1, padre2)  # Crear segundo hijo

return hijo1, hijo2  # Devolver los hijos
```

La mutación siendo la parte más importante del AG junto con el cruzamiento, es definida en la siguiente función llamada de igual forma “mutación”. Esta función trabaja con el parámetro inicial llamado “tasa_mutación” (el cual definimos nosotros). De manera aleatoria usando la función “random” se generará un número y en caso de ser este menor a la tasa de mutación establecida, se ejecutara la función, la cual consta en tomando un ruta dada, se tomara de forma aleatoria dos sub rutas de esta (ruta individual de vehículo) y en dos puntos completamente al azar los intercambiara entre sí, creando nuevamente una nueva variante.

```python
# Mutación sencilla sin duplicados
def mutacion(individuo):
    if random.random() < tasa_mutacion:  # Probabilidad de mutación
        ruta1, ruta2 = random.sample(individuo, 2)  # Seleccionar dos rutas al azar
        if ruta1 and ruta2:
            i, j = random.randint(0, len(ruta1) - 1), random.randint(0, len(ruta2) - 1)
            ruta1[i], ruta2[j] = ruta2[j], ruta1[i]  # Intercambiar elementos
```

Finalmente, nuestra última función del AG, “algoritmo_genetico” hará uso de todas las funciones anteriormente explicadas el número de generaciones epecificadas inicialmente, para así, haciendo uso de la función “calcular_aptitud” así como la función de mínimo, encontrar la ruta en la cual menor recorrido se haga de las generadas.

```python
# Algoritmo Genético
def algoritmo_genetico():
    poblacion = crear_poblacion_inicial()  # Crear la población
    for generacion in range(generaciones):  # Para cada generac
        nueva_poblacion = []
        for _ in range(tamano_poblacion // 2):
            padre1 = seleccion_aleatoria(poblacion)  # Seleccion
            padre2 = seleccion_aleatoria(poblacion)  # Seleccion
            hijo1, hijo2 = cruzamiento(padre1, padre2)  # Cruzar
            mutacion(hijo1)  # Mutar el primer hijo
            mutacion(hijo2)  # Mutar el segundo hijo
            nueva_poblacion.extend([hijo1, hijo2])  # Añadir hi
        poblacion = nueva_poblacion  # Reemplazar la población
        mejor_individuo = min(poblacion, key=calcular_aptitud)
        #print(f"Generación {generacion}, Mejor Aptitud: {calcul
    return mejor_individuo  # Devolver el mejor individuo
```

Siendo todo esto lo que es parte del algoritmo genético para la resolución del VRP, se comenzó con la realización de otras dos formas de resolver este problema utilizando paradigmas mas “tradicionales” como lo son la fuerza bruta y los algoritmos voraces.

Incluido en el mismo código como una función se creó “fuerza_bruta_vrp” el cual es la resolución del problema utilizando este paradigma.

Haciendo uso de la librería itertools para hacer permutaciones, esta función hará e iterará sobre todas las permutaciones posibles de las ubicaciones de los clientes indicados, creando en cada iteración todas las formas de crear las sub rutas posibles según el número de vehículos indicados y midiendo la distancia total de recorrido de cada una de ellas, guardando siempre la mejor ruta con su respectiva distancia para finalmente retornarla al acabar todas las iteraciones posibles.

```python
# Fuerza Bruta para VRP
def fuerza_bruta_vrp(clientes, num_vehiculos):
    mejor_distancia = float('inf')
    mejor_solucion = None
    for perm in itertools.permutations(clientes):  # Todas las permutaciones de clientes
        for puntos_division in itertools.combinations(range(1, len(clientes)), num_vehiculos - 1):  # Todas las p
            rutas = [list(perm[i:j]) for i, j in zip((0,) + puntos_division, puntos_division + (len(clientes),))]
            distancia_total = calcular_aptitud(rutas)  # Calcular la aptitud de la solución
            if distancia_total < mejor_distancia:  # Si es la mejor distancia encontrada
                mejor_distancia = distancia_total  # Actualizar la mejor distancia
                mejor_solucion = rutas  # Actualizar la mejor solución
    return mejor_solucion, mejor_distancia  # Devolver la mejor solución y distancia
```

Siguiendo la misma dinámica que fuerza bruta, se incluyó la función “voraz_vrp”, la cual resuelve el problema utilizando el paradigma voraz. Esta funciona de la siguiente manera:

Creando una lista vacía para el número de vehículos a utilizar, un conjunto con los clientes a visitar y dando índices los vehículos:

- Se añadirá la ruta de cada vehículo como primera locación el depósito (de donde iniciaran y finalizaran la ruta).

-Mientras queden clientes por asignar (en el conjunto clientes) iterando sobre cada vehículo, se buscará la localización disponible más cercana (utilizando la función “distancia”) y agregándola a su ruta.

- Se eliminará dicha locación usada de clientes para evitar repeticiones en otras rutas.

- finalmente se volverá a añadir la localización del depósito para que vuelva.

```python
# Algoritmo Voraz para VRP
def voraz_vrp(clientes, num_vehiculos):
    rutas = [[] for _ in range(num_vehiculos)]  # Inicializar rutas vacías para cada vehículo
    clientes_restantes = set(clientes)  # Clientes que quedan por asignar
    indices_vehiculos = list(range(num_vehiculos))  # Índices de los vehículos
    # Inicializar rutas con el depósito
    for ruta in rutas:
        ruta.append(deposito)  # Añadir el depósito al inicio de cada ruta
    while clientes_restantes:  # Mientras queden clientes por asignar
        for vehiculo in indices_vehiculos:  # Para cada vehículo
            if not clientes_restantes:  # Si no quedan clientes, salir del bucle
                break
            ubicacion_actual = rutas[vehiculo][-1]  # Última ubicación en la ruta del vehículo
            cliente_mas_cercano = min(clientes_restantes, key=lambda x: distancia(ubicaciones[ubicacion_actual]
            rutas[vehiculo].append(cliente_mas_cercano)  # Añadir cliente a la ruta del vehículo
            clientes_restantes.remove(cliente_mas_cercano)  # Quitar cliente de los pendientes
    # Añadir el depósito al final de cada ruta
    for ruta in rutas:
        ruta.append(deposito)  # Añadir el depósito al final de la ruta
    # Quitar el depósito inicial de cada ruta para evitar duplicaciones
    for ruta in rutas:
        if ruta[0] == deposito:
            ruta.pop(0)  # Quitar el depósito inicial

    return rutas  # Devolver las rutas
```

Para finalizar el código se realizo la función “graficar solución”, la cual graficara las locaciones de los clientes como puntos de color azul, el deposito como un punto de color rojo, y siguiendo las rutas ya generadas realizara la conexión de puntos por vehículo.

```python
# Función para graficar la mejor solución
def graficar_solucion(solucion, ubicaciones, titulo):
    plt.figure(figsize=(10, 6))
    ubicacion_deposito = ubicaciones[deposito]
    plt.scatter(ubicacion_deposito[0], ubicacion_deposito[1], c='red', label='Depósito'
    plt.scatter(ubicaciones[1:, 0], ubicaciones[1:, 1], c='blue', label='Clientes')  #

    for i, ruta in enumerate(solucion):  # Para cada ruta
        ubicaciones_ruta = [ubicaciones[deposito]] + [ubicaciones[cliente] for cliente
        plt.plot([ubicacion[0] for ubicacion in ubicaciones_ruta], [ubicacion[1] for ub

    plt.legend()
    plt.xlabel('Coordenada X')
    plt.ylabel('Coordenada Y')
    plt.title(titulo)
    plt.show()
```

### *D. Complejidad*

Realizado y resuelto el problema de VRP con tres diferentes paradigmas, lo siguiente por hacer era observar la complejidad de cada uno de ellos y así obtener cual es mejor en qué casos.

Para el AG se dedujo en complejidad de tiempo:

- La inicialización de la población conlleva una complejidad de O(P*N), donde P es el tamaño de la población y N el número de clientes.
- La evaluación de aptitud requiere O(P*M*$N^2$), M siendo el número de generaciones, O($N^2$) en el peor de los casos pues Cada evaluación de aptitud para un individuo implica recorrer todas las rutas de los vehículos.

Y siendo las demás funciones inferiores en complejidad, se concluye que la complejidad final del AG es: O(P*M*$N^2$)

Para el algoritmo de Fuerza Bruta:

- En la generación de permutaciones, una complejidad O(N!) es la presente debido a que hay N! permutaciones por posibles N clientes.
- Para cada permutación hay O($\frac{N-1}{V-1}$) combinaciones posibles de dividir los clientes entre vehículos (V)
- Para calcular la aptitud de cada conjunto de rutas se requiere O($N^2$).

Siendo finalmente la complejidad: O($N! * \left(\frac{N-1}{V-1}\right) * N^2$)

Por último, para el algoritmo Voraz:

- La asignación de clientes a cada vehículo requiere O(V*$N^2$) donde V son los vehículos y N el número de clientes.
- El añadido de los depósitos a cada vehículo representa O(V*N)

Siendo la complejidad final de: O(V*$N^2$)

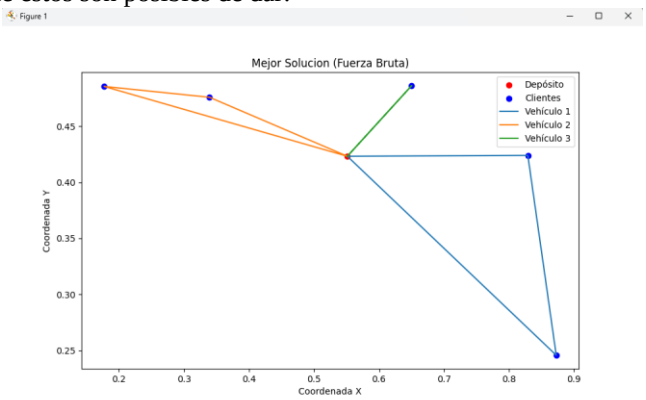
El algoritmo genético es eficiente en términos de encontrar soluciones buenas en grandes espacios de búsqueda, pero su complejidad depende en gran medida del tamaño de la población y el número de generaciones. Requiere más espacio para almacenar múltiples individuos en la población.

La fuerza bruta garantiza encontrar la solución óptima al probar todas las posibles permutaciones y divisiones de clientes. Sin embargo, es computacionalmente ineficiente y su complejidad crece factorialmente con el número de clientes, lo que lo hace impracticable para problemas grandes.

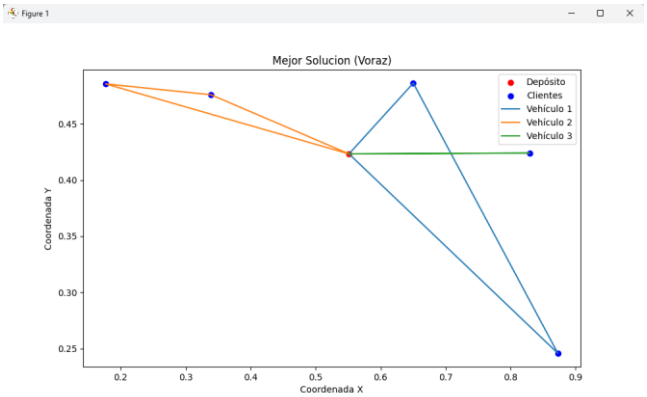
El algoritmo voraz es eficiente en términos de tiempo y espacio, proporcionando una solución rápida. Sin embargo, no garantiza encontrar la solución óptima, ya que siempre toma decisiones locales óptimas sin considerar el impacto global.

## IV. Resultados Obtenidos del Proyecto

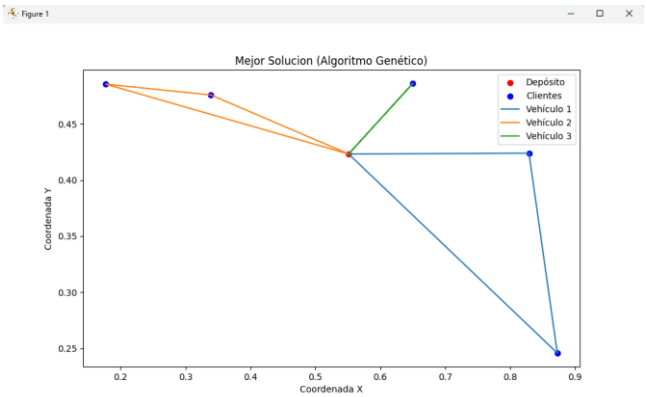
El desarrollo del código, así como la resolución del problema VRP fueron cumplidos de manera exitosa mediante los paradigmas utilizados, obteniendo los mejores resultados que estos son posibles de dar.




```
Mejor Solucion (Fuerza Bruta): [[1, 3], [2, 4], [5]]
Mejor Distancia (Fuerza Bruta): 1.8233590280463063
```




```
Mejor Solucion (Voraz): [[5, 1, 0], [4, 2, 0], [3, 0]]
Mejor Distancia (Voraz): 2.127981772958275
```




```
Mejor Solucion (Algoritmo Gen?tico): [[1, 3], [4, 2], [5]]
Mejor Distancia (Algoritmo Gen?tico): 1.8233590280463063
```

## V. Conclusiones y Trabajo a Futuro

Guzmán Cabrera Alvaro:

La solución de este problema fue una bastante difícil de realizar, sobre todo en cuestión al algoritmo metaheurístico, pues la investigación previa no daba la suficiente información para comenzar de manera rápida con el desarrollo de este. La implementación de los otros paradigmas fue bastante más sencilla una vez teniendo las bases del problema. Un buen proyecto complicado en cuestión al inicio del desarrollo que se va facilitando conforme este avanza.

Garcia Parra Jair Omar:

La solución del problema me pareció complicado ya que los algoritmos metaheurísticos eran algo nuevo para mi y no lograba entender del todo en que consistían realmente estos mismos, aunque al entender un poco de que se trata se va volviendo mas sencillo entender y avanzar con el desarrollo hasta que pudimos dar con una solución que nos pareció correcta y comenzar a dar conclusiones.

## Referencias


[1] Boza, O. C. (12 de junio de 2012). *redalyc*. Obtenido de https://www.redalyc.org/pdf/816/81624969008.pdf

[2] Edwin, O. M. (9 de mayo de 2017). *zaloamati*. Obtenido de http://zaloamati.azc.uam.mx/bitstream/handle/11191/5699/Metaheuristicas_problema_de_ruteo_de_vehiculos_2017_Montes_MOPT.pdf;jsessionid=CC2C336EF41D6C53C64A7AEE6F735738?sequence=1

[3] Julio, D. M. (diciembre de 2009). *scielo*. Obtenido de http://www.scielo.org.co/pdf/eia/n12/n12a03.pdf