"Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho"

Universidad Nacional Mayor de San Marcos

Universidad del Perú, Decana de América
Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Sistemas

Asunto: Proyecto - Informe Final

Profesor: Luis Ángel Guerra

Integrantes del grupo N° 11:

Celestino Rojas, Angela Wini

Echevarria Narrea, Marko

Mendoza Aymara, Luis Ernesto

Ramos Garriazo, Francis Alberto

Benites Pardavé, Eder Gustavo

24 Junio 2024

1. Introduccion

Antes de que se llevarán a cabo investigaciones sobre la complejidad de los algoritmos, se realizaron varios estudios que sirvieron de base para su avance. Con el estudio de la complejidad algorítmica, el cual se define como la cantidad de recursos para resolver una tarea, la ciencia de la computación ha avanzando, permitiendo el análisis de algoritmos eficientes.

Uno de los problemas más estudiados y difíciles en el campo de la optimización combinatoria es el problema del agente viajero (TSP, por sus siglas en inglés, Problema del agente viajero). Este busca la ruta más corta que permite a un vendedor visitar un conjunto de ciudades una vez y regresar a su ciudad de origen. El TSP es un problema NP-difícil, lo que significa que no existe un algoritmo eficiente que resuelva todos los casos en un tiempo polinomial, a pesar de su enunciado aparentemente simple.

El TSP tiene más usos que la planificación de rutas y la logística. Su solución afecta muchos campos, como la robótica, el diseño de circuitos y la bioinformática. Debido a la complejidad del TSP, se han creado varios enfoques, incluidos algoritmos precisos, heurísticas y metaheurísticas.

Este estudio analiza e implementa un algoritmo genético en Python para resolver el problema del agente viajero. Se discutirán las bases teóricas de los algoritmos genéticos, cómo se adaptaron al TSP y los resultados experimentales que demuestran su eficacia. El objetivo es proporcionar una comprensión profunda de cómo los AG pueden utilizarse para resolver el TSP y evaluar su desempeño en comparación con otros métodos tradicionales y modernos, tomando en cuenta el tiempo de complejidad.

2. Generalidades

2.1. Planteamiento de problema

2.1.1. Problema general

Determinar el algoritmo genético para la solución del problema del agente viajero, considerando su complejidad computacional.

2.1.2. Problemas especificos

- ¿Cómo adaptar un algoritmo genético para que aborde eficientemente el TSP?
- ¿Cuáles son los parámetros y operadores genéticos más adecuados para optimizar la solución del TSP?
- ¿Qué tan efectiva es la implementación del algoritmo genético en términos de calidad de la solución y tiempo de ejecución en comparación con otros métodos heurísticos y exactos.

2.2. Objetivos

2.2.1. Objetivos generales

Analizar e implementar un algoritmo genético para resolver el problema del agente viajero (TSP) y evaluar su rendimiento en comparación con otros enfoques heurísticos y exactos.

2.2.2. Objetivos especificos

- Desarrollar una adaptación de un algoritmo genético para el TSP.
- Determinar los parámetros óptimos del algoritmo genético.
- Evaluar el rendimiento del algoritmo genético.

2.3. Contribuciones a la investigacion

Esta investigación proporciona una adaptación efectiva de algoritmos genéticos para resolver el problema del agente viajero (TSP), que incluye un esquema de codificación específico y operadores genéticos optimizados (selección, cruce y mutación). Se han identificado y optimizado los parámetros cruciales, se ha mejorado la calidad de las soluciones y se ha evaluado la eficacia de una variedad de técnicas de selección. Además, la investigación analiza el rendimiento del algoritmo genético, lo que demuestra su eficacia en comparación con otras técnicas heurísticas y exactas. Además, se documentan los beneficios y desventajas del uso de algoritmos genéticos para el TSP, lo que sirve como base para futuras mejoras. Finalmente, se proporciona un marco teórico y práctico para la aplicación de algoritmos genéticos en optimización combinatoria, con resultados experimentales y código fuente disponible para futuras investigaciones.

2.4. Antecedentes

Análisis e implementación del algoritmo genético de Chu Beasley para resolver el problema del agente viajero (TSP) y su variante, el problema de rutas de vehículo (VRP)

En este trabajo se lleva a cabo la implementación del algoritmo genético de Chu Beasley (AGCB), el cual presenta un proceso evolutivo altamente eficiente, que se desarrolla de forma clásica llevando a cabo los procesos de evolución natural como la selección, en el cual se determinan los padres aptos para pasar a la siguiente generación; la recombinación, en donde los cromosomas de los padres se comparten para generar nuevos individuos candidatos y así determinar cuál de los nuevos individuos sigue en el proceso, teniendo en cuenta que posea la mejor función objetivo; y finalmente la mutación en donde se controla la factibilidad nuevamente y se emplea una tasa que es ajustable dentro de los parámetros generales del algoritmo con el fin de alterar alguno de los alelos de forma aleatoria.

https://repositorio.utp.edu.co/server/api/core/bitstreams/7a17390a-b63d-4c44-a316-e7002e7d9c93/content

Implementación de un Algoritmo Genético modificado para la solución al Problema del Agente Viajero

En el presente trabajo se explora una alternativa de solución al problema planteado, un algoritmo genético modificado. Se presenta la implementación

de un algoritmo genético modificado para la solución al problema del agente viajero mediante la modificación de la función de cruce. Se realiza una comparativa con la implementación del algoritmo de optimización por colonia de hormigas, con la finalidad de explorar las fortalezas de cada implementación, así como la naturaleza de sus soluciones. La comparativa se presenta en términos de tiempo de ejecución y distancia entregada en la solución proporcionada por cada implementación.

https://repositorio.cetys.mx/bitstream/60000/1426/1/171-Texto.pdf

3. Marco teórico

3.1. Algoritmos genéticos

Los algoritmos genéticos (GA) se presentan como herramientas inspiradas en la selección natural y los principios de la genética. Introducidos por John Holland en la década de 1970, estos algoritmos han demostrado ser extremadamente eficaces para resolver problemas complejos en diversas disciplinas, incluidas la ingeniería, la economía y la biología. En el contexto del problema del viajero (TSP), los AG proporcionan un enfoque sólido para encontrar soluciones casi óptimas en plazos razonables.

3.2. Algoritmos genéticos ¿Cómo trabajan?

3.2.1. Principios básicos de los algoritmos genéticos

- **Codificación (representación del individuo):** Los individuos dentro de una población en un AG están representados mediante codificación genética, generalmente en forma de cadenas de bits. Cada individuo simboliza una posible solución al problema.
- **Población inicial:** Se genera aleatoriamente una población inicial de individuos. El tamaño de la población es variable pero debe ser lo suficientemente grande como para mantener la diversidad genética y garantizar una exploración eficiente del espacio de búsqueda.
- **Función de fitness:** Esta función evalúa la calidad de una solución particular. Para el TSP, la función de aptitud podría ser el recíproco de la longitud total del camino tomado, por lo que las soluciones con caminos más cortos tienen una mejor aptitud.
- **Selección:** La selección es el proceso mediante el cual se seleccionan individuos para pasar a la siguiente generación. La idea es que los

individuos con mejor aptitud física tengan mayores posibilidades de ser seleccionados para la reproducción.

3.2.2. Operadores genéticos

- **Cruce:** este operador combina dos padres para producir uno o más descendientes. En TSP, el cruce debe ocurrir de tal manera que las permutaciones resultantes sean válidas (es decir, cada ciudad aparece exactamente una vez en la ruta). Ejemplos de métodos de cruce son el cruce en bicicleta (CX) y el cruce ordenado (OX).
- **Mutación:** una mutación produce variaciones aleatorias entre individuos que ayudan a mantener la diversidad genética de la población y previenen la convergencia prematura. En TSP, una mutación típica podría ser el intercambio de dos ciudades en la ruta.
- Sustitución: Tras seleccionar y aplicar operadores genéticos, se forma una nueva generación de individuos. El reemplazo puede ser completo (la nueva generación reemplaza completamente a la vieja) o parcial (quedan algunos individuos de la vieja generación).
- **Criterios de finalización:** El algoritmo se ejecuta de forma iterativa hasta que se cumple un criterio de terminación. Esto puede ser un número fijo de generaciones, una mejora mínima en la fitness durante un cierto número de generaciones o hasta que se pueda encontrar una solución satisfactoria.

3.2.3. Algoritmos Genéticos ¿Por qué funcionan?

Los algoritmos genéticos (AG) funcionan porque usan mecanismos basados en evolución natural para buscar soluciones. Las razones de su eficiencia son:

- Diversidad genética: la generación aleatoria y mutación continua conducen a una diversidad poblacional, la cual permite a los AG explorar diferentes regiones del espacio de búsqueda, evitando así decantarse al inicio por una solución que podría no ser la más óptima.
- Selección natural: los AG imitan la selección natural favoreciendo las soluciones más adecuadas para la reproducción. Esto asegura que las mejores características individuales se transmitan a la siguiente generación, mejorando las soluciones con el paso de las generaciones

- Cruce: el proceso de cruce combina las propiedades de dos soluciones principales para producir una o más soluciones derivadas. Este intercambio de información genética permite combinar las mejores características de ambas soluciones, creando potencialmente mejores soluciones.
- Mutación: la mutación conduce a variaciones aleatorias en las soluciones, lo que permite explorar nuevas áreas del espacio de búsqueda y evita que la población se estanque en los óptimos locales.
- Adaptabilidad: los GA son flexibles y pueden adaptarse a diferentes tipos de problemas sin requerir un conocimiento extenso del espacio de búsqueda. Esto los hace aplicables a una variedad de problemas de optimización.
- Equilibrio entre exploración y explotación: Los AG equilibran bien la exploración (búsqueda de nuevas soluciones) y la explotación (refinamiento de las mejores soluciones). Este equilibrio es crucial para encontrar soluciones casi óptimas a problemas complejos.

3.3. Modelo TPS

El problema del agente viajero (TSP) se encuentra dentro de la categoría de problemas de optimización combinatoria conocidos como NP-completos. En su formulación, se parte de la definición de la estructura de una cadena de suministro, que se representa gráficamente por medio de nodos y líneas. Cada nodo representa un destino o ciudad, y las líneas conectan dos puntos, representando los posibles recorridos entre ciudades.

Cada recorrido entre ciudades tiene asociado un costo, que generalmente se basa en la distancia entre ellas. Un recorrido completo que pasa por todas las ciudades una vez y regresa al punto de origen se denomina ciclo hamiltoniano. La suma de todas las distancias en este recorrido representa la longitud del ciclo.

Cuando se aborda el problema desde la perspectiva de la ubicación de instalaciones, el objetivo es encontrar la mejor ubicación que minimice tanto el costo como la distancia asociada con los diferentes puntos de venta o distribución identificados para esta instalación. En otras palabras, se busca encontrar la disposición óptima de instalaciones que minimice los costos y las distancias entre ellas, considerando la red de puntos de venta o distribución establecidos.

De acuerdo con De los Cobos et al. (2010), se define el problema del agente viajero, dado un entero $n > 0$ y las distancias entre cada par de las n ciudades, estas distancias se dan por medio de la matriz (d_{ij}) de dimensión $n \times n$, donde d_{ij} es un entero mayor o igual a cero. Un recorrido es una trayectoria que visita todas las ciudades exactamente una vez. El problema consiste en encontrar un recorrido con longitud total mínima. Una permutación π cíclica representa un recorrido, si se interpreta $\pi(j)$ como la ciudad después de la ciudad j, $j = 1, 2, \ldots, n$. Así, el costo del recorrido es:

$$\sum_{j=1}^{n} d_{j,\pi(j)}$$

3.3.1. Aplicaciones práctica del TSP

- **Logística y Distribución:** En la industria de la logística y distribución, el TSP se utiliza para planificar las rutas más eficientes para la entrega de bienes y servicios, minimizando los costos operativos y el tiempo de viaje. Esto se aplica a empresas de mensajería, transporte de mercancías y gestión de flotas de vehículos.
- **Planificación de Rutas Turísticas:** En el sector turístico, el TSP se utiliza para planificar itinerarios óptimos para viajes turísticos, visitas guiadas y excursiones, maximizando la experiencia del viajero al visitar múltiples lugares de interés en un período de tiempo limitado. Esto se aplica tanto a viajes individuales como a tours grupales.
- **Robótica y Automatización:** En robótica y automatización, el TSP se utiliza para planificar movimientos eficientes de robots y sistemas automatizados en entornos industriales y logísticos. Esto incluye aplicaciones en fabricación, ensamblaje, almacenamiento y manipulación de materiales.
- **Secuenciación de ADN y Bioinformática:** En bioinformática, el TSP se utiliza para optimizar la secuenciación de ADN y la alineación de secuencias genéticas, ayudando a los investigadores a entender mejor la estructura y función de los genomas. Esto es fundamental en el desarrollo de medicina personalizada, biotecnología y estudios de evolución molecular.

3.3.2. Métodos de resolución

Los algoritmos genéticos son métodos matemáticos que emulan a la naturaleza y buscan solucionar problemas complejos utilizando el concepto de evolución. El algoritmo realiza una búsqueda simultánea en varias áreas del espacio factible, intensifica en algunas de ellas y explora otros subespacios mediante el intercambio de información entre configuraciones. Utiliza tres mecanismos fundamentales: selección, crossover y mutación:

- **Selección:** Es el operador genético que selecciona las configuraciones de la población actual que participarán en la generación de la nueva población (nueva generación). Este operador concluye tras determinar la cantidad de descendientes que debe tener cada configuración de la población actual.
- **Crossover o recombinación:** Es el mecanismo que permite transmitir información genética de un par de cromosomas originales a sus descendientes, lo que significa que se puede pasar de un espacio de búsqueda a otro, generando de esta manera diversidad genética en la población.
- **Mutación:** Permite realizar la intensificación en un espacio específico moviéndose a través de vecinos. Esto implica intercambiar el valor de un gen de un cromosoma en una población. De manera aleatoria, se selecciona un cromosoma como candidato, se genera un número aleatorio y, si es menor que la tasa de mutación ($\rho < \rho_m$), se lleva a cabo la mutación. La tasa de mutación se elige dentro del rango $[0{,}001, 0{,}05]$.

3.3.3. Colonia de hormigas (aco)

La optimización por colonia de hormigas se basa en el comportamiento de las hormigas reales para resolver problemas de optimización, como el del cartero viajante. Las "hormigas.ªrtificiales imitan este comportamiento, dejando rastros de feromonas mientras exploran posibles soluciones. Estos rastros guían a otras hormigas hacia caminos más prometedores, creando un proceso iterativo de mejora continua de las soluciones. En el problema del cartero viajante, las hormigas artificiales eligen sus destinos según la cantidad de feromonas presentes y una función de distancia heurística. Después de cada iteración, se actualizan los rastros de feromonas, fortaleciendo los caminos más cortos. Este enfoque combina la exploración aleatoria con la explotación de información previamente encontrada, lo que lleva a la convergencia hacia

soluciones óptimas o cercanas a óptimas.

3.3.4. Búsqueda tabú (ts)

La búsqueda tabú (TS) es una técnica metaheurística utilizada para gestionar algoritmos heurísticos de búsqueda local y evitar que se queden atrapados en óptimos locales. Realiza una exploración del espacio de configuraciones, evitando los óptimos locales mediante la clasificación de movimientos recientes como "movimientos tabú", lo que impide revisitar configuraciones previamente exploradas.

Esta metodología emplea dos tipos de memoria: una de corto plazo, que registra eventos recientes, y otra de largo plazo, que almacena datos de frecuencia de eventos, siendo crucial para definir estrategias de diversificación y explorar nuevas regiones del espacio de soluciones.

Cuando un movimiento es marcado como tabú pero produce una mejora significativa en la función objetivo en comparación con una referencia preestablecida, se aplica la regla de aspiración, que levanta la prohibición y permite ejecutar el movimiento. Esta regla asegura que, si un movimiento prohibido conduce a una mejora notable en la solución, se aproveche esta oportunidad sin restricciones. En resumen, la búsqueda tabú utiliza una combinación de memoria a corto y largo plazo junto con reglas específicas, como la regla de aspiración, para evitar los óptimos locales y explorar de manera efectiva el espacio de soluciones en la búsqueda de soluciones óptimas o cercanas a óptimas.

3.3.5. Grasp

GRASP, una evolución de los algoritmos heurísticos constructivos, utiliza indicadores de sensibilidad para identificar atributos favorables en problemas de optimización. Este método combina elementos de Simulated Annealing y Búsqueda Tabú para la fase de exploración, demostrando eficacia en la resolución de problemas complejos.

El procedimiento iterativo de GRASP incluye una fase de preprocesamiento, seguida de una búsqueda constructiva y una exploratoria. Durante la fase constructiva, se define una vecindad de la solución y se busca una solución factible de mejor calidad. Este proceso puede demandar un alto esfuerzo computacional, especialmente en problemas que requieren resolver problemas de programación lineal para analizar cada vecino.

GRASP ha sido aplicado con éxito en el contexto del Problema del Agente Viajero (TSP). En esta aplicación, la fase constructiva de GRASP se utiliza para generar soluciones factibles al TSP, construyendo iterativamente rutas que visitan todas las ciudades una vez y regresan al punto de partida. La fase exploratoria de GRASP se encarga de mejorar estas soluciones, explorando vecindades de soluciones para encontrar rutas de menor longitud. Esta combinación de construcción y exploración permite a GRASP encontrar soluciones de alta calidad para instancias del TSP, ayudando a superar los desafíos inherentes a este problema NP-difícil.

3.4. Herramientas de software

3.4.1. Python

Creado a principios de los años 90 por Guido van Rossum, cuándo se encontraba trabajando en el sistema operativo Amoeba. Este lenguaje nace como sucesor del lenguaje ABC, principalmente para manejar excepciones e interfaces con el sistema operativo mencionado. El nombre "Python"proviene de la afición de Van Rossum por los humoristas británicos Monty Python.

Es un lenguaje de alto nivel, ampliamente utilizado en la actualidad en el área de la ciencia de datos e inteligencia artificial. Posee una sintaxis sencilla y utiliza un tipado dinámico. Posee una diversa y organizada bibliografía del lenguaje y documentación en línea. Ofrece una amplia gama de bibliotecas y herramientas que son muy útiles para implementar algoritmos genéticos. Alguna de las principales librerías son las siguientes:

- **NumPy:** Esta biblioteca ofrece una amplia gama de rutinas que permiten realizar operaciones sobre estos arreglos de manera rápida y eficiente. Incluye operaciones matemáticas, lógicas, de ordenamiento y selección, transformadas de Fourier, álgebra lineal, operaciones estadísticas básicas y generación de números aleatorios, entre otras funcionalidades esenciales.
- **MatplotLib:** Biblioteca de visualización de datos en 2D que produce figuras de calidad en una variedad de formatos y en diferentes entornos interactivos. Desarrollada inicialmente por el neurobiólogo John Hunter en 2002 para visualizar señales eléctricas del cerebro en personas epilépticas, para conseguir ello debía replicar las funcionalidades de MATLAB en Python.
- **NetworX:** Es una biblioteca de Python para la creación, manipula-

ción y estudio de la estructura, dinámica y funciones de redes complejas. Incluye algoritmos para la generación de grafos aleatorios, la búsqueda de caminos más cortos, la detección de comunidades, el cálculo de centralidades, la identificación de cliques, entre otros.

3.4.2. Herramientas vinculadas a la evaluación del algoritmo

- **Timeit:** Este módulo ofrece una manera fácil de medir el tiempo de ejecución de pequeños fragmentos de código en Python. Cuenta con una interfaz de línea de comandos y una invocable. Permite evitar diversos errores comunes al medir los tiempos de ejecución.
- **cProfile**: Es un conjunto de estadísticas que describe con qué frecuencia y durante cuánto tiempo se ejecutaron varias partes del programa. Es una extensión C con una sobrecarga razonable que la hace adecuada para perfilar programas de larga duración. Basado en lsprof, aportado por Brett Rosen y Ted Czotter.
- **Snakeviz**: Proporciona una interfaz web que muestra gráficamente el tiempo de ejecución de las funciones de tu programa, permitiéndote identificar fácilmente las áreas de tu código que consumen más tiempo de CPU.

4. Desarrollo

4.1. Algoritmo genético 1

4.1.1. Introducción

Este primer algoritmo propuesto, el cual está elaborado en lenguaje python, está destinado a optimizar rutas para que un agente pueda visitar un conjunto de ciudades reduciendo la distancia total recorrida. Este algoritmo pertenece a la tradición de los algoritmos genéticos, que son conocidos por su capacidad para abordar problemas de optimización combinatoria complejos utilizando técnicas basadas en la evolución biológica.

El algoritmo sugerido utiliza una población inicial de rutas generadas aleatoriamente, cada una de las cuales representa una solución potencial al TSP. El algoritmo determina las rutas más prometedoras basándose en su aptitud (distancia recorrida), las combina con operadores de cruce para generar nuevas soluciones y las somete a mutaciones aleatorias para fomentar la exploración del espacio de soluciones a través de iteraciones sucesivas. Este

proceso evolutivo se repite varias generaciones antes de llegar a una ruta óptima o subóptima que representa la solución cercana al TSP.

4.1.2. Descripción el algoritmo

La primera etapa del algoritmo genético para resolver el problema de TSP es representar las ciudades como objetos de ciudad con coordenadas (x, y). Luego, la función initialPopulation se usa para inicializar una población inicial de rutas aleatorias. El algoritmo ha pasado por varias generaciones y cada una incluye:

1. Usando selection, elija las mejores rutas de la población actual según su aptitud (distancia más corta).
2. Combinando las rutas seleccionadas a través de operaciones de cruce, se crea una nueva población donde se crean hijos a partir de padres seleccionados.
3. MutePopulation se utiliza para introducir cambios aleatorios en la población para mantener la diversidad.
4. Evaluación: utiliza rankRoutes para calcular la distancia de cada ruta en la población para evaluar su aptitud.

Estos pasos se repiten a lo largo de varias generaciones (generations) hasta que se alcance un criterio de parada, que puede ser un número máximo de generaciones o una mejora mínima en la distancia recorrida. Finalmente, como solución al problema del TSP, se devuelve la mejor ruta encontrada, que se representa gráficamente con las ciudades y la ruta óptima encontrada utilizando matplotlib.

4.1.3. Análisis de complejidad

1. Crear clase "Ciudades"

```
class City:
    def __init__(self,name, x, y):
        self.name = name
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.name)+ ")"

    devolver true
```

- Inicialización (`__init__`): $O(1)$ ya que se asignan valores.
- Distancia (`distance`): $O(1)$ ya que se calculan operaciones matemáticas.
- Representación (`__repr__`): $O(1)$ puesto que se convierte cadena y concatena.

2. Crear funcion fitness

```
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
```

```
17                  pathDistance += fromCity.distance(
      toCity)
18              self.distance = pathDistance
19          return self.distance
20
21      def routeFitness(self):
22          if self.fitness == 0:
23              self.fitness = 1 / float(self.routeDistance
      ())
24          return self.fitness
```

- Inicialización (`__init__`): Se asignan valores $O(1)$.
- Distancia de la ruta (`routeDistance`): El bucle `for` recorre todas las ciudades en la ruta, dentro del bucle son operaciones constantes $O(n)$.
- Aptitud de la ruta (`routeFitness`): $O(n)$ en el peor caso (cuando `self.fitness` es 0), de lo contrario $O(1)$.

3. Crear población inicial

```
1  def createRoute(cityList):
2      route = random.sample(cityList, len(cityList))
3      return route
4
5  def initialPopulation(popSize, cityList):
6      population = []
7
8      for i in range(0, popSize):
9          population.append(createRoute(cityList))
10     return population
```

- createRoute: Selecciona una muestra aleatoria de tamaño igual al tamaño de `cityList`.
- La complejidad de `random.sample` para una lista de tamaño n es $O(n)$.
- initialPopulation: $O(\texttt{popSize} \times n)$.

4. Ranquear rutas

```
1  def rankRoutes(population):
2      fitnessResults = {}
3      for i in range(0,len(population)):
4          fitnessResults[i] = Fitness(population[i]).
       routeFitness()
5      sorted_results=sorted(fitnessResults.items(), key =
        operator.itemgetter(1), reverse = True)
6      return sorted_results
```

- Creación del diccionario: $O(1)$
- Llenar el diccionario con `routeFitness`: El bucle `for` itera sobre cada individuo en `population`, dentro del bucle la llamada a `routeFitness()` incluye la llamada a `routeDistance()`. Por tanto, $O(\texttt{population} \times n)$
- Ordenar los resultados: $O(m \log m)$
- Retornar los resultados: $O(1)$

5. Selección padres

```
1  def selection(popRanked, eliteSize):
2      selectionResults = []
3      df = pd.DataFrame(np.array(popRanked), columns=["
       Index","Fitness"])
4      df['cum_sum'] = df.Fitness.cumsum()
5      df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()
6
7      for i in range(0, eliteSize):
8          selectionResults.append(popRanked[i][0])
9      for i in range(0, len(popRanked) - eliteSize):
10         pick = 100*random.random()
11         for i in range(0, len(popRanked)):
12             if pick <= df.iat[i,3]:
13                 selectionResults.append(popRanked[i
       ][0])
14                 break
15     return selectionResults
```

- Inicialización y creación del DataFrame: Crear lista $O(\texttt{popRanked})$
- Selección de la élite: El bucle `for` itera `eliteSize` veces, $O(\texttt{eliteSize})$
- Selección del resto de la población: $O(\texttt{popRanked}^2)$

- Retorno de los resultados: $O(1)$
- Complejidad de la función `selection`: $O(\texttt{popRanked}^2)$

6. Crear grupo de aparamiento

```
1  def matingPool(population, selectionResults):
2      matingpool = []
3      for i in range(0, len(selectionResults)):
4          index = selectionResults[i]
5          matingpool.append(population[index])
6      return matingpool
```

- Inicialización de `matingpool`: $O(1)$
- Llenado de `matingpool`: El bucle `for` itera sobre cada elemento en `selectionResults`. Si el tamaño de `selectionResults` es s, el bucle iterará s veces. $O(s)$
- Retorno de `matingpool`: Retorna lista $O(1)$

7. Cruce

```
1  def breed(parent1, parent2):
2      child = []
3      childP1 = []
4      childP2 = []
5
6      geneA = int(random.random() * len(parent1))
7      geneB = int(random.random() * len(parent1))
8
9      startGene = min(geneA, geneB)
10     endGene = max(geneA, geneB)
11
12     for i in range(startGene, endGene):
13         childP1.append(parent1[i])
14
15
16     childP2 = [item for item in parent2 if item not in
       childP1]
17     print(startGene, endGene)
18
19     print(parent1)
20     print(parent2)
21
22     print(childP1)
23     print(childP2)
```

```
24      child = childP1 + childP2
25
26      print(child)
27      return child
```

- Llenar `childP1` con genes de `parent1`: $O(n)$
- Llenar `childP2` con genes de `parent2`: $O(n^2)$
- Combinar `childP1` y `childP2`: Concatenar 2 listas $O(n)$
- Complejidad de la función: $O(n^2)$

7. Crear población cruzada

```
1  def breedPopulation(matingpool, eliteSize):
2      children = []
3      length = len(matingpool) - eliteSize
4      pool = random.sample(matingpool, len(matingpool))
5
6      for i in range(0,eliteSize):
7          children.append(matingpool[i])
8
9      for i in range(0, length):
10         child = breed(pool[i], pool[len(matingpool)-i
       -1])
11         children.append(child)
12     return children
```

- Crear una muestra aleatoria de `matingpool`: $O(m)$
- Añadir los individuos de la élite a `children`: $O($`eliteSize`$)$
- Generar el resto de los niños a través de `breed`: $O(mn^2)$, donde m es el tamaño de `matingpool` y n es el tamaño de los padres (`parent1` y `parent2`).
- Complejidad de la función: $O(mn^2)$

8. Mutacion

```
1  def mutate(individual, mutationRate):
2      for swapped in range(len(individual)):
3          if(random.random() < mutationRate):
4              swapWith = int(random.random() * len(
       individual))
5
6              city1 = individual[swapped]
7              city2 = individual[swapWith]
8
9              individual[swapped] = city2
10             individual[swapWith] = city1
11     return individual
12
13     def mutatePopulation(population, mutationRate):
14     mutatedPop = []
15
16     for ind in range(0, len(population)):
17         mutatedInd = mutate(population[ind],
       mutationRate)
18         mutatedPop.append(mutatedInd)
19     return mutatedPop
```

- Función `Mutate`: $O(n)$ donde n es la longitud del individuo.
- Función `mutatePopulation`: $O(mn)$ donde m es el tamaño de `population` y n es la longitud de cada individuo en `population`.

9. Creación de la siguiente generación

```
1  def nextGeneration(currentGen, eliteSize, mutationRate)
       :
2      popRanked = rankRoutes(currentGen)
3      selectionResults = selection(popRanked, eliteSize)
4      matingpool = matingPool(currentGen,
       selectionResults)
5      children = breedPopulation(matingpool, eliteSize)
6      nextGeneration = mutatePopulation(children,
       mutationRate)
7      return nextGeneration
```

- Llamar a `rankRoutes`: $O(mn \log n)$
- Llamar a `selection`: $O(m \log m)$

- Llamar a `matingPool`: $O(m)$

- Llamar a `breedPopulation`: $O(mn^2)$

- Llamar a `mutatePopulation`: $O(mn)$

10. Creación del algoritmo genético

```
1  def geneticAlgorithm(population, popSize, eliteSize,
       mutationRate, generations):
2      pop = initialPopulation(popSize, population)
3      progress = [1 / rankRoutes(pop)[0][1]]
4      print("Initial distance: " + str(progress[0]))
5
6      for i in range(1, generations+1):
7
8          pop = nextGeneration(pop, eliteSize,
       mutationRate)
9          progress.append(1 / rankRoutes(pop)[0][1])
10         if i%50==0:
11           print('Generation '+str(i),"Distance: ",
       progress[i])
12
13
14     bestRouteIndex = rankRoutes(pop)[0][0]
15     bestRoute = pop[bestRouteIndex]
16
17     plt.plot(progress)
18     plt.ylabel('Distance')
19     plt.xlabel('Generation')
20     plt.title('Best Fitness vs Generation')
21     plt.tight_layout()
22     plt.show()
23
24
25
26     return bestRoute
```

- Inicialización y creación de la población inicial: $O(\text{popSize} \times n)$

- Inicialización de `progress`: $O(mn \log n)$

- Bucle principal (`for loop`): $O(\text{generations} \times (mn^2 + mn \log n))$

- Determinación de la mejor ruta: $O(mn \log n)$

- Visualización con Matplotlib: $O(\text{generations})$

```
1  cityList = []
2
3  for i in range(0,5):
4      cityList.append(City(name = i, x=int(random.random
       () * 200), y=int(random.random() * 200)))
5
6
7  best_route=geneticAlgorithm(population=cityList,
       popSize=30, eliteSize=20, mutationRate=0.01,
       generations=1)
8  x=[]
9  y=[]
10 for i in best_route:
11   x.append(i.x)
12   y.append(i.y)
13 x.append(best_route[0].x)
14 y.append(best_route[0].y)
15 plt.plot(x, y, '--o')
16 plt.xlabel('X')
17 plt.ylabel('Y')
18 ax=plt.gca()
19 plt.title('Final Route Layout')
20 bbox_props = dict(boxstyle="circle,pad=0.3", fc='C0',
       ec="black", lw=0.5)
21 for i in range(1,len(cityList)+1):
22   ax.text(cityList[i-1].x, cityList[i-1].y, str(i), ha
       ="center", va="center",
23             size=8,
24             bbox=bbox_props)
25 plt.tight_layout()
26 plt.show()
```

Dado que la mayoría de las operaciones son $O(1)$ o $O(n)$ donde n es pequeño y fijo (5 ciudades en `cityList`), la complejidad total de todo el código proporcionado es $O(n)$, donde n es el número de ciudades en `cityList`. En este caso particular, como `cityList` tiene 5 ciudades, la complejidad es $O(5) = O(1)$.

4.1.4. Complejidad total del programa

Para resumir, la complejidad total del algoritmo genético está dominada por las operaciones dentro de nextGeneration, que a su vez dependen de rankRoutes. Por lo tanto, la complejidad total del algoritmo genético es:

$$O(\text{generations} \times (\text{popSize} \times n^2 + \text{popSize} \times n \log n))$$

Donde:

- **generations**: número de generaciones.
- **popSize**: tamaño de la población.
- **n**: tamaño de los individuos en la población, es decir, el número de ciudades en el problema del TSP.

4.2. Algoritmo genético 2

4.2.1. Introducción

La elección de Java como lenguaje de implementación se debe a sus características robustas, su orientación a objetos y su extensa biblioteca estándar, que facilitan la manipulación de estructuras de datos complejas y la realización de operaciones matemáticas.

La implementación desarrollada en Java aprovecha la modularidad y la capacidad de manejo de colecciones del lenguaje para representar y manipular las rutas entre ciudades. Cada ciudad se modela como un objeto con atributos que incluyen su nombre y coordenadas. Las rutas (o caminos) se representan como listas de ciudades, y la población de rutas se gestiona mediante listas de objetos de tipo `Path`.

4.2.2. Descripción el algoritmo

El algoritmo genético desarrollado para resolver el TSP en este informe consta de los siguientes pasos principales:

1. **Inicialización**: Se crea una población inicial de caminos (soluciones) de tamaño P. Cada camino se genera de manera aleatoria.
2. **Evaluación**: Se calcula la distancia euclidiana total de cada camino en la población, que actúa como la función de aptitud (fitness) a minimizar.
3. **Selección**: Se seleccionan los caminos más aptos para la reproducción. En este caso, se utiliza una selección determinística basada en ordenar los caminos según su aptitud.
4. **Crossover**: Se cruzan pares de caminos seleccionados para producir nuevos caminos (descendientes). Se utilizan puntos de corte aleatorios

para combinar segmentos de los padres.

5. **Mutación**: Se aplican mutaciones aleatorias a algunos de los caminos nuevos para mantener la diversidad genética. Esto se hace intercambiando ciudades dentro de un camino.

6. **Reemplazo**: Se forma una nueva población con los caminos seleccionados y los nuevos caminos generados, manteniendo el tamaño de la población constante.

7. **Iteración**: Se repiten los pasos de evaluación, selección, crossover y mutación hasta alcanzar un número máximo de generaciones G o hasta que no se observe una mejora significativa en las soluciones.

4.2.3. Análisis del algoritmo

1. Inicialización de la población de caminos

```
1  funcion createPaths():
2      si (3 * n) es par:
3          populationSize = 3 * n
4      sino:
5          populationSize = 3 * n + 1
6      para i de 0 a populationSize - 1:
7          crear un nuevo Path: paths[i]
8          para j de 0 a n - 1:
9              city = GenerateNewCityBanning()
10             paths[i].addCity(Map[city])
11         paths[i].addCity(paths[i].getCityFromPath(0))
12         paths[i].CalculateEuclideanDistance()
13         limpiar bannedCities
```

- Bucle externo: O(populationSize) = O(n), ya que populationSize es proporcional a n.
- Bucle interno: O(n), donde n es el número de ciudades.
- GenerateNewCityBanning(): En el peor caso, O(n) si casi todas las ciudades están prohibidas.
- CalculateEuclideanDistance(): O(n), como veremos más adelante.

2. Función principal del algoritmo genético

```
1  funcion LaunchSimulation():
2      para i de 0 a MaxGenerations - 1:
```

```
3           ordenar paths por distancia (QuickSort)
4           crear NewPaths
5           seleccionar los mejores caminos en NewPaths
6           para j de 0 a (fatherSize - 2) paso 2:
7               si NewPaths[j+1] existe:
8                   GenerateNewValidPath(NewPaths[j],
        NewPaths[j+1], s1, s2, cutPoint1, cutPoint2)
9                   agregar el mejor de s1 y s2 a NewPaths
10          para a de 0 a NewPaths.size() - 1:
11              mutar NewPaths[a] si mejora
12          paths = NewPaths
13          imprimir mejor camino
```

- Bucle externo: O(MaxGenerations).
- Ordenar (Collections.sort()): O(n * log(n)) en promedio (QuickSort).
- Selección: O(n), donde n es proporcional a populationSize.
- Cruce (GenerateNewValidPath()): O(n^2) en el peor caso, como veremos.
- Mutación: O(n) por cada camino, así que O(n^2) en total.

3. Realiza el cruce entre dos caminos

```
1   funcion GenerateNewValidPath(f1, f2, s1, s2,
        minCutPoint, maxCutPoint):
2       si no hay conflicto en puntos de corte:
3           copiar f1 a s1 y f2 a s2
4           intercambiar segmentos entre minCutPoint y
        maxCutPoint
5           para i de 0 a (maxCutPoint - minCutPoint - 1):
6               para x de 0 a (n - 1):
7                   si hay ciudad duplicada en s1 o s2:
8                       reemplazar con ciudad faltante
9           calcular distancias de s1 y s2
10          devolver true
11      sino:
12          devolver false
```

- Copiar e intercambiar: O(n)
- Bucles anidados para corregir duplicados: O(n * n) = O(n^2)
- CalculateEuclideanDistance(): O(n)

4. Calcula la distancia de un camino

```
1  funcion CalculateEuclideanDistance():
2      EuclideanDistance = 0
3      para i de 0 a path.size() - 1:
4          si i < path.size() - 1:
5              EuclideanDistance += distancia(path[i],
       path[i+1])
6          sino:
7              EuclideanDistance += distancia(path[i],
       path[0])
8      totalDistance = EuclideanDistance
```

- Bucle: O(n), donde n es el tamaño del camino (número de ciudades).
- Operaciones dentro del bucle: O(1), ya que calcular la distancia entre dos puntos es constante.
- CalculateEuclideanDistance(): O(n)

5.Verificar la validez de un camino

```
1  funcion isAValidPath(check):
2      para i de 0 a check.path.size() - 2:
3          para j de 0 a check.path.size() - 2:
4              si check.path[i] == check.path[j] y i != j:
5                  devolver false
6      devolver true
```

- Bucles anidados: O(n * n) = O(n^2), donde n es el tamaño del camino.

El análisis de complejidad del algoritmo genético para el TSP considera varios factores clave:

- **Inicialización**: La generación de la población inicial tiene una complejidad de $O(P \cdot n)$, donde P es el tamaño de la población y n es el número de ciudades.
- **Evaluación**: Calcular la distancia total para cada camino tiene una complejidad de $O(n)$. Evaluar toda la población en cada generación tiene una complejidad de $O(P \cdot n)$.
- **Selección**: Ordenar la población basada en la aptitud tiene una complejidad de $O(P \log P)$.

- **Crossover**: Generar nuevos caminos mediante crossover depende de la implementación específica, pero en el peor de los casos puede ser $O(P \cdot n)$ si se realiza un crossover completo para toda la población.
- **Mutación**: La mutación de caminos tiene una complejidad de $O(P \cdot n)$, asumiendo que cada camino puede ser mutado una vez por generación.
- **Reemplazo**: El reemplazo de la población tiene una complejidad de $O(P)$.

Por lo tanto, la complejidad total por generación del algoritmo es $O(P \cdot n + P \log P)$. Dado que el algoritmo se ejecuta durante G generaciones, la complejidad total es $O(G \cdot (P \cdot n + P \log P))$.

En la práctica, P y G son parámetros que se eligen para equilibrar la precisión y el tiempo de ejecución del algoritmo. A pesar de su complejidad, los algoritmos genéticos son efectivos para encontrar soluciones aproximadas a problemas NP-completos como el TSP en un tiempo razonable.

4.3. Interfaz gráfica

Para facilitar la interacción y la visualización del algoritmo genético que hemos implementado, se ha desarrollado una interfaz gráfica utilizando React. Esta interfaz permite a los usuarios visualizar en tiempo real la evolución de las soluciones al problema del agente viajero (TSP), así como interactuar con los parámetros del algoritmo. La interfaz gráfica incluye las siguientes funcionalidades:

- Visualización de la ruta final: Muestra la ruta final generada por el algoritmo en un plano cartesiano, permitiendo observar la solución al problema del TSP.
- Ingreso de puntos (ciudades): Los usuarios pueden colocar puntos en el plano que representan las ciudades. Estas ciudades son enumeradas en el orden de colocación como City 1, City 2, etc.
- Ejecución del algoritmo: Un botón Run Algorithm"que, al ser presionado, ejecuta el algoritmo y muestra la ruta final.
- Reinicio de la interfaz: Un botón Çlear Canvas"que permite resetear el plano para empezar de nuevo.
- Información de la solución: Texto que muestra el número de ciudades y la distancia total de la ruta generada.

```
import { useState, useEffect, useRef } from 'react';
import { GeneticsAlgorithm } from '../logic/
    GeneticsAlgorithm';

const TSPVisualizer = () => {
    const [cities, setCities] = useState([]);
    const [bestPath, setBestPath] = useState([]);
    const [totalDistance, setTotalDistance] = useState
    (0);
    const canvasRef = useRef(null);

    const canvasWidth = 800;
    const canvasHeight = 600;

    useEffect(() => {
        drawCanvas();
    }, [cities, bestPath]);

    const drawCanvas = () => {
        const canvas = canvasRef.current;
        const ctx = canvas.getContext('2d');
        ctx.clearRect(0, 0, canvasWidth, canvasHeight);

        // Dibujar cuadr cula
        ctx.strokeStyle = '#d4d4d8';
        for (let i = 0; i <= canvasWidth; i += 50) {
            ctx.beginPath();
            ctx.moveTo(i, 0);
            ctx.lineTo(i, canvasHeight);
            ctx.stroke();
        }
        for (let i = 0; i <= canvasHeight; i += 50) {
            ctx.beginPath();
            ctx.moveTo(0, i);
            ctx.lineTo(canvasWidth, i);
            ctx.stroke();
        }

        // Dibujar ciudades
        cities.forEach((city, index) => {
            ctx.fillStyle = '#3b82f6';
            ctx.beginPath();
            ctx.arc(city.x, city.y, 5, 0, 2 * Math.PI);
            ctx.fill();
            ctx.fillStyle = '#115e59';
            ctx.fillText(`City ${index + 1}`, city.x +
    10, city.y - 10);
        });
```

```
46
47          // Dibujar mejor ruta
48          if (bestPath.length > 0) {
49              const originalLineWidth = ctx.lineWidth; //
        Paso 1: Guardar el grosor de l nea actual
50              ctx.strokeStyle = '#fb923c';
51              ctx.lineWidth = 3; // Paso 2: Establecer el
        nuevo grosor de l nea
52              // Paso 3: Dibujar la mejor ruta
53              ctx.beginPath();
54              ctx.moveTo(bestPath[0].getCityPosition().x,
        bestPath[0].getCityPosition().y);
55              bestPath.forEach((city) => {
56                  ctx.lineTo(city.getCityPosition().x,
       city.getCityPosition().y);
57              });
58              ctx.closePath();
59              ctx.stroke();
60              ctx.lineWidth = originalLineWidth; // Paso
       4: Restablecer el grosor de l nea
61          }
62      };
63
64      const handleCanvasClick = (event) => {
65          const rect = canvasRef.current.
       getBoundingClientRect();
66          const x = event.clientX - rect.left;
67          const y = event.clientY - rect.top;
68          setCities([...cities, { x, y }]);
69      };
70
71      const runAlgorithm = () => {
72          const tsp = new GeneticsAlgorithm(20);
73          cities.forEach((city) => tsp.setNew({ x: city.x
       , y: city.y }));
74          tsp.createPaths();
75          tsp.launchSimulation();
76
77          const bestSolution = tsp.paths[0];
78          setBestPath(bestSolution.path);
79          setTotalDistance(bestSolution.
       getEuclideanDistance());
80      };
81
82      const clearCanvas = () => {
83          setCities([]);
84          setBestPath([]);
85          setTotalDistance(0);
86      };
```

```
87
88      return (
89          <div className='min-h-screen flex flex-col
        justify-center items-center p-5 gap-5'>
90              <h1 className='text-5xl font-bold mb-6 text
        -orange-400'>TSP - GeneticsAlgorithm</h1>
91              <div
92                  className='flex items-center justify-
        center gap-16'
93              >
94                  <canvas
95                      className='cursor-pointer shadow-xl
        '
96                      ref={canvasRef}
97                      width={canvasWidth}
98                      height={canvasHeight}
99                      onClick={handleCanvasClick}
100                 />
101                 <div
102                     className='flex flex-col gap-5
        items-center shadow-xl p-5'
103                 >
104                     <p
105                         className='text-lg font-medium'
106                     >
107                         Number of cities: {cities.
        length}
108                     </p>
109                     <p
110                         className='text-lg font-medium'
111                     >
112                         Total Distance: {totalDistance.
        toFixed(2)}
113                     </p>
114                     <div className='flex gap-2'>
115                         <button
116                             onClick={runAlgorithm}
117                             disabled={cities.length <
        6}
118                             className={`bg-blue-500
        hover:bg-blue-700 text-white font-bold py-2 px-4
        rounded cursor-pointer ${cities.length < 6 ? 'bg-
        gray-500 cursor-not-allowed hover:bg-gray-500' : '
        bg-blue-500'}`}
119                         >
120                             Run Algorithm
121                         </button>
122                         <button
123                             onClick={clearCanvas}
```

```
124                                 className='bg-red-500 hover
       :bg-red-700 text-white font-bold py-2 px-4 rounded
       cursor-pointer'
125                             >
126                                 Clear Canvas
127                             </button>
128                         </div>
129
130                     </div>
131                 </div>
132             </div>
133         );
134 };
135
136 export default TSPVisualizer;
```

4.3.1. Captura de pantalla de la intefaz

A continuación, se muestra una captura de pantalla de la interfaz gráfica desarrollada, donde se pueden observar las diferentes funcionalidades mencionadas.

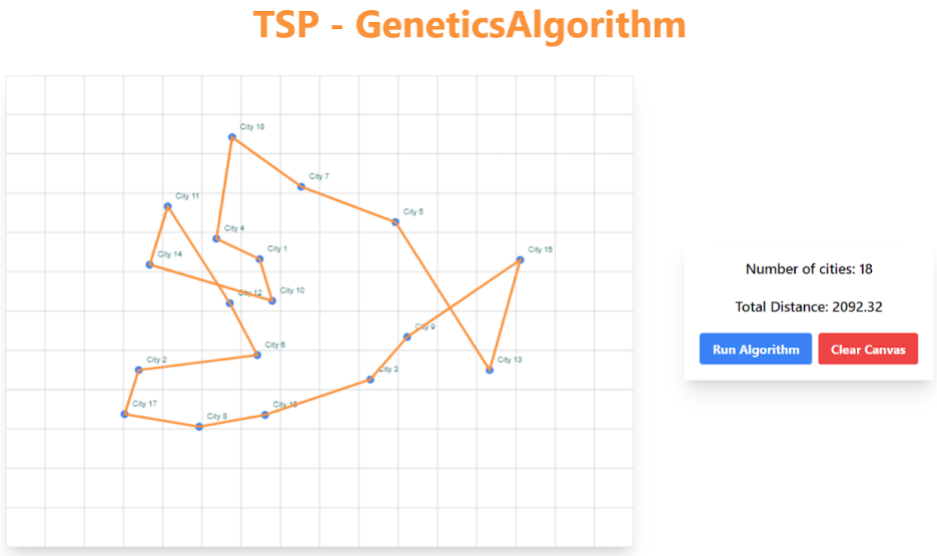

Figura 1: Algoritmo implementando `https://tsp-reactjs.netlify.app/`

En esta imagen, se pueden ver los puntos que representan las ciudades en el plano y el camino final generado por el algoritmo al presionar el botón Run Algorithm".

5. Pruebas y resultados

Para validar el desempeño y la efectividad del algoritmo genético desarrollado para el problema del viajante (TSP), se realizaron diversas pruebas. Estas pruebas se enfocaron en dos aspectos críticos del algoritmo: el operador de selección y el operador de mutación. La finalidad de estas pruebas fue evaluar cómo diferentes configuraciones de parámetros afectan la capacidad del algoritmo para encontrar rutas óptimas o cercanas a la óptima en términos de distancia recorrida. Se compararon diferentes configuraciones de tamaño de población, tamaño de la élite, tasa de mutación y número de generaciones. Además, se visualizaron los resultados mediante gráficos que muestran la distancia mínima por generación y la disposición final de la mejor ruta encontrada. Precisamente se configuraron los valores en la siguiente instrucción del código que presentaban valores por default:

```
1  best_route=geneticAlgorithm(population=cityList,
       popSize=30, eliteSize=20, mutationRate=0.01,
       generations=10)
2       devolver true
```

5.1. Pruebas sobre el operador de selección

Para evaluar la efectividad del operador de selección, se realizaron varias pruebas variando el tamaño de la élite (eliteSize). El tamaño de la élite determina cuántos de los mejores individuos se seleccionan directamente para la siguiente generación sin someterse a cruce o mutación. Tomando la configuración:

- Tamaño de población (popSize): 30
- Tasa de mutación (mutationRate): 0.01
- Número de generaciones (generations): 5

Se probaron tres tamaños de élite: 5, 10 y 20. A continuación, se presentan los resultados de estas pruebas:

1. Prueba con eliteSize = 5: Esto significa que solo los mejores 5 individuos de cada generación se seleccionarán directamente para la siguiente generación sin cambios. Esto puede llevar a una convergencia más rápida hacia soluciones de alta calidad, ya que se conservan las mejores soluciones en cada iteración.

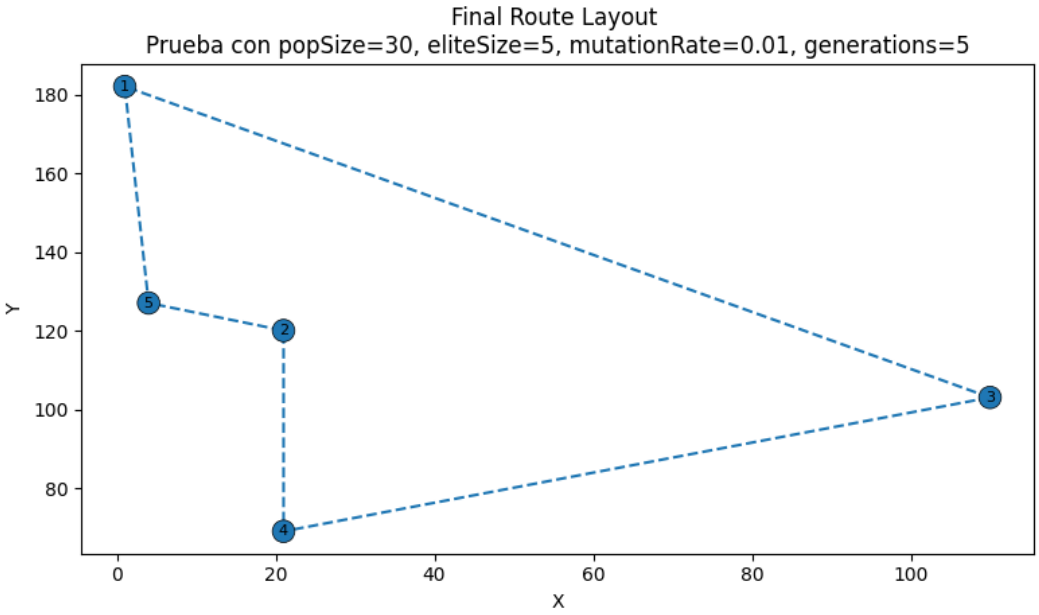

Figura 2: Initial distance: 354.35780283914016

2. Prueba con eliteSize = 10: Al aumentar eliteSize a 10, se conservan más soluciones de élite entre generaciones. Esto puede aumentar la intensificación, es decir, la búsqueda en áreas de alta calidad del espacio de soluciones, pero también puede reducir la diversidad genética si no hay suficiente exploración de otras soluciones.

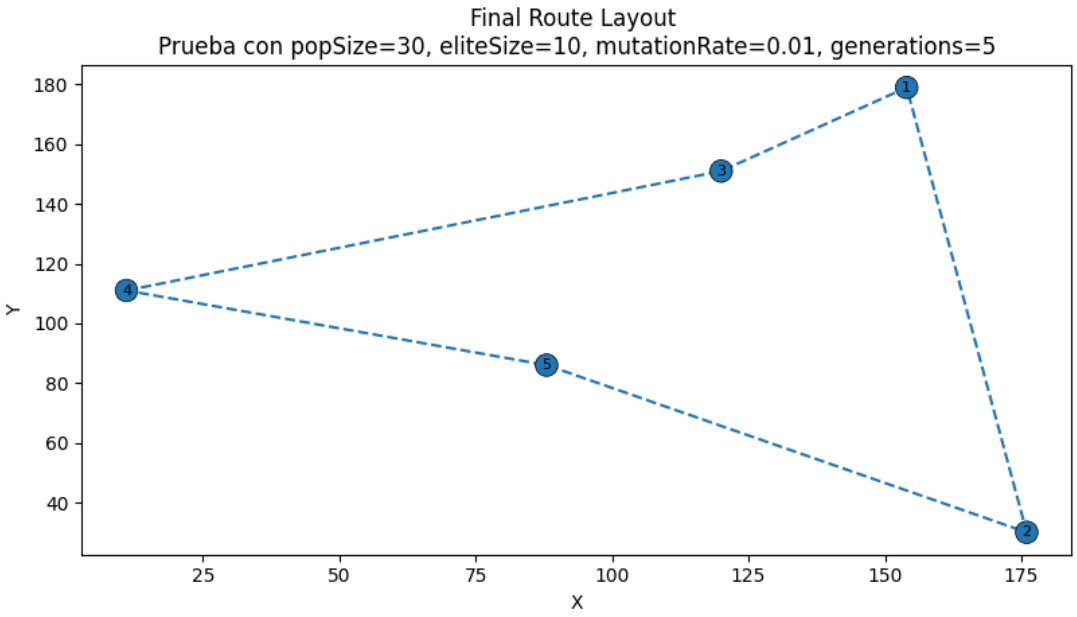

Figura 3: Initial distance: 496.03256104078105

3. Prueba con eliteSize = 20: Un eliteSize de 20 conserva aún más soluciones de élite. Esto puede llevar a una mayor intensificación y a convergir más rápidamente hacia soluciones de alta calidad, pero a expensas de la diversidad genética y la capacidad de explorar nuevas áreas del espacio de soluciones.

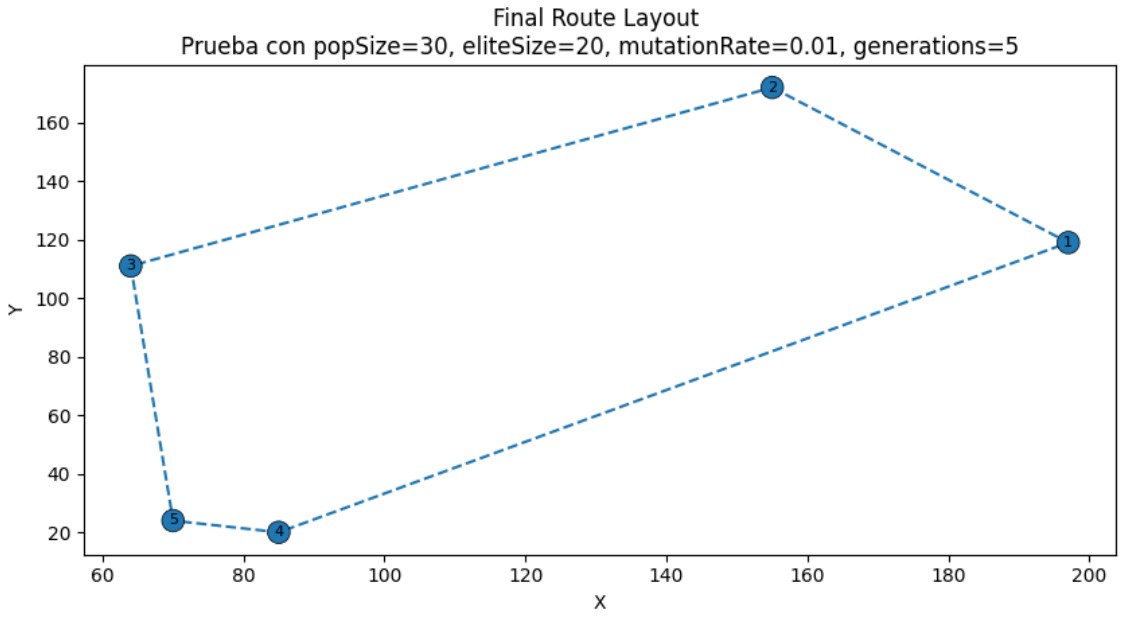

Figura 4: Initial distance: 429.3908663398732

5.2. Pruebas sobre el operador de mutación

Para evaluar la efectividad del operador de mutación, se realizaron varias pruebas variando la tasa de mutación (mutationRate). La tasa de mutación determina la probabilidad de que un individuo sea mutado en cada generación. Se probaron tres tasas de mutación: 0.01, 0.05 y 0.10. A continuación, se presentan los resultados de estas pruebas. Tomando la configuración:

- Tamaño de población (popSize): 30
- Tamaño de la élite (eliteSize): 10
- Número de generaciones (generations): 5

1. Prueba con mutationRate= 0.02: Una tasa de mutación baja significa que las probabilidades de que se produzcan cambios aleatorios en los individuos son bajas. Esto puede conducir a una convergencia rápida hacia óptimos locales y puede no ser suficiente para explorar adecuadamente el espacio de soluciones.

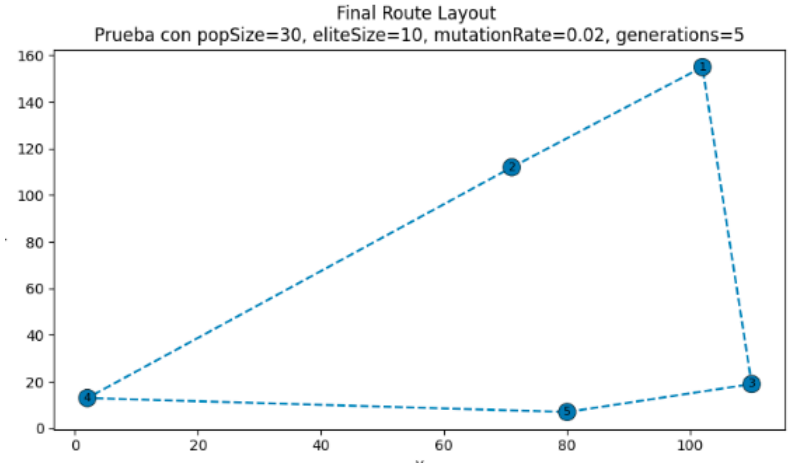

Figura 5: Initial distance: 354.35780283914016

2. Prueba con mutationRate= 0.05: Una tasa de mutación moderada puede permitir un equilibrio entre la explotación (explotar soluciones actuales conocidas) y la exploración (explorar nuevas soluciones). Puede ayudar a evitar la convergencia prematura hacia óptimos locales y mantener cierta diversidad genética.

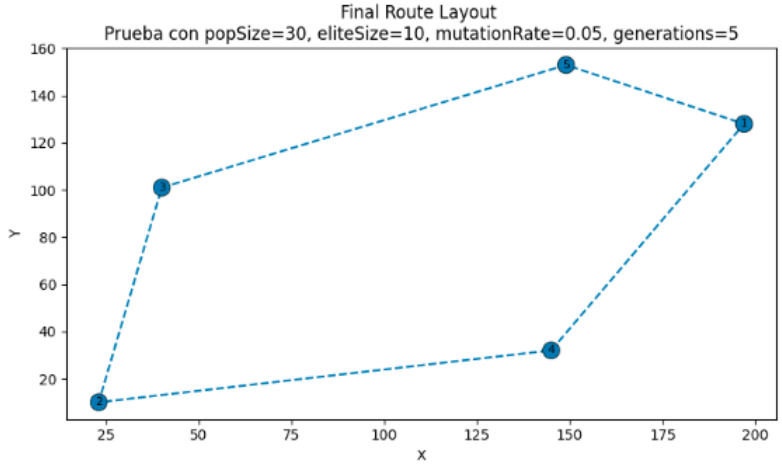

Figura 6: Initial distance: 420.4590539221569

3. Prueba con mutationRate= 0.10: Con una tasa de mutación alta, hay más probabilidades de que se introduzcan cambios aleatorios en los individuos en cada generación. Esto fomenta una mayor exploración del espacio de soluciones, lo que puede ayudar a escapar de óptimos locales y encontrar soluciones más diversas y potencialmente mejores.

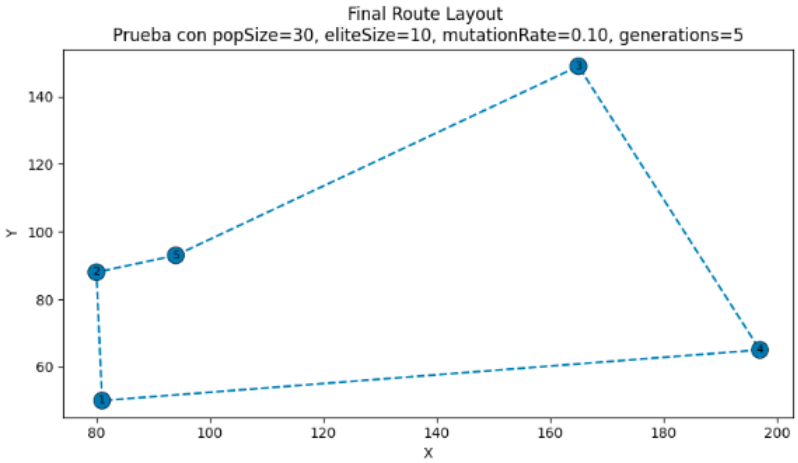

Figura 7: Initial distance: 350.16061750020566

5.3. Efectos Generales

* Elitismo (eliteSize): Un aumento en eliteSize generalmente conduce a una convergencia más rápida hacia soluciones de alta calidad, pero puede limitar la exploración del espacio de soluciones y la diversidad genética.

* Mutación (mutationRate): Una tasa de mutación más alta promueve la exploración del espacio de soluciones, lo que puede ayudar a encontrar soluciones mejores, pero también aumenta el riesgo de perder soluciones de alta calidad por cambios aleatorios.

5.4. Comparación de los algoritmos

5.4.1. Costos de algoritmos

1. Representación de la solución

- Algoritmo 1: Utiliza una representación basada en permutaciones de

ciudades, donde cada individuo en la población representa un orden aleatorio de las ciudades. Estas permutaciones forman circuitos cerrados que representan rutas completas del TSP. Esta estructura facilita la aplicación de operadores genéticos estándar como selección, cruce y mutación

- Algoritmo 2: Emplea una representación basada en caminos ordenados de ciudades, donde cada individuo es una secuencia ordenada de ciudades que debe visitar cada ciudad exactamente una vez. Aunque inicialmente no forman un circuito cerrado, la condición de visita única asegura que todos los individuos sean soluciones válidas del TSP. Esta representación permite aplicar estrategias específicas para manejar caminos ordenados y operadores adaptados a esta estructura

2. Estrategia de evolución

- Algoritmo 1: La selección de élites y la tasa de mutación afectan directamente la diversidad genética y la convergencia del algoritmo hacia soluciones óptimas.

- Algoritmo 2: Implementa estrategias más específicas para manipular caminos ordenados de ciudades, como la selección de padres basada en la distancia recorrida y la generación de nuevos caminos mediante técnicas de recombinación y mutación adaptadas a esta representación. Esto permite un control más detallado sobre cómo se exploran y explotan las soluciones en el espacio de búsqueda.

5.4.2. Tiempo de ejecución

En este análisis comparativo, se evalúa el rendimiento de dos algoritmos genéticos aplicados al problema del viajante de comercio (TSP, por sus siglas en inglés) utilizando una configuración común de 50 ciudades y un máximo de 10 generaciones. Los algoritmos, denominados Algoritmo 1 y Algoritmo 2, difieren en su implementación y enfoque para resolver el TSP. El objetivo es determinar cuál de los dos algoritmos exhibe un mejor rendimiento en términos de tiempo de ejecución para esta configuración específica.

- **Algoritmo 1:** Utiliza una población inicial de 50 individuos (rutas de ciudades) y una tasa de mutación del 0,01, y se ejecuta durante 10 generaciones.

- **Algoritmo 2**: Inicia con un número de generaciones máximo de 10. Cada generación implica la evaluación y posible evolución de rutas

basadas en el cálculo de distancias euclidianas entre ciudades.

Para realizar una evaluación exhaustiva del tiempo de ejecución de dos algoritmos genéticos aplicados al problema del viajante de comercio (TSP), se llevaron a cabo 10 iteraciones variando el número de ciudades entre 10, 20, 30, 40 y 50. Los resultados de estas pruebas se detallan a continuación:

Iteración	Algoritmo 1	Algoritmo 2
1	0,06	0,06
2	0,06	0,02
3	0,06	0,02
4	0,07	0,02
5	0,06	0,02
6	0,07	0,02
7	0,06	0,02
8	0,07	0,02
9	0,08	0,03
10	0,06	0,03

Cuadro 1: Tiempos de ejecución - 10 ciudades

El promedio de tiempo de ejecución para el Algoritmo 1 es 0.0647 segundos, mientras que para el Algoritmo 2 es 0.0249 segundos.

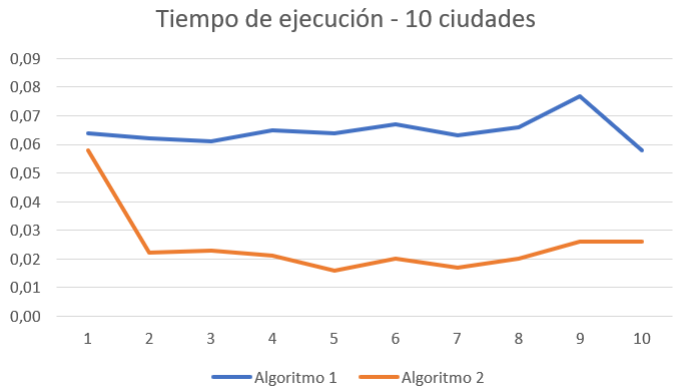

Figura 8: Tiempo de ejecución para 10 ciudades

Iteración	Algoritmo 1	Algoritmo 2
1	0,25	0,10
2	0,25	0,11
3	0,23	0,09
4	0,23	0,18
5	0,24	0,10
6	0,25	0,11
7	0,24	0,11
8	0,23	0,16
9	0,23	0,12
10	0,23	0,10

Cuadro 2: Tiempo de ejecución - 20 ciudades

El promedio de tiempo de ejecución para el Algoritmo 1 es 0.2395 segundos, mientras que para el Algoritmo 2 es 0.118 segundos.

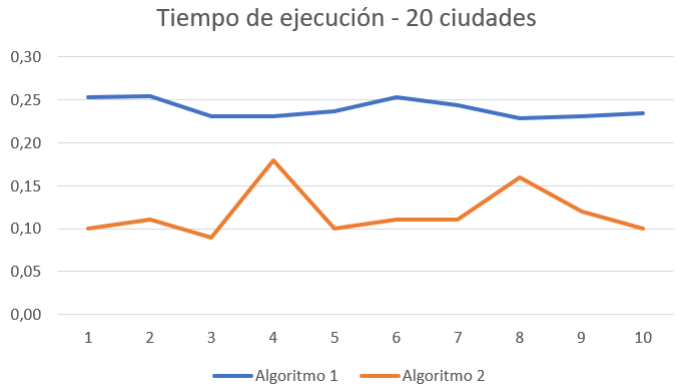

Figura 9: Tiempo de ejecución para 20 ciudades

Iteración	Algoritmo 1	Algoritmo 2
1	0,27	0,26
2	0,29	0,28
3	0,26	0,52
4	0,28	0,26
5	0,27	0,30
6	0,31	0,26
7	0,27	0,27
8	0,28	0,29
9	0,28	0,24
10	0,29	0,25

Cuadro 3: Tiempo de ejecución - 30 ciudades

El promedio de tiempo de ejecución para el Algoritmo 1 es 0.28 segundos, mientras que para el Algoritmo 2 es 0.293 segundos.

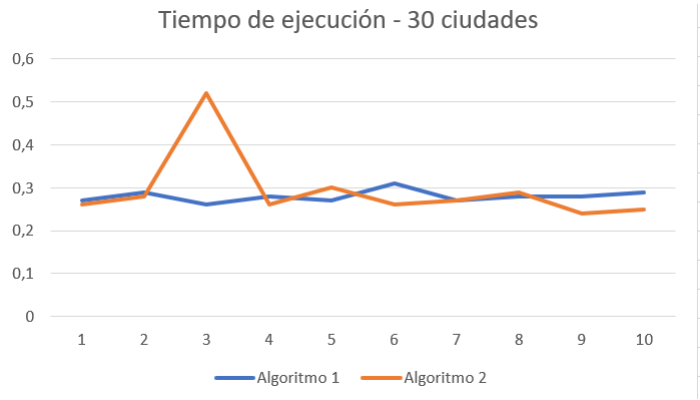

Figura 10: Tiempo de ejecución para 30 ciudades

Iteración	Algoritmo 1	Algoritmo 2
1	0,30	0,63
2	0,31	0,52
3	0,30	0,55
4	0,31	0,51
5	0,31	0,45
6	0,54	0,51
7	0,29	0,52
8	0,29	0,55
9	0,30	0,59
10	0,29	0,61

Cuadro 4: Tabla con los nuevos datos proporcionados

El promedio de tiempo de ejecución para el Algoritmo 1 es 0.325 segundos, mientras que para el Algoritmo 2 es 0.544 segundos.

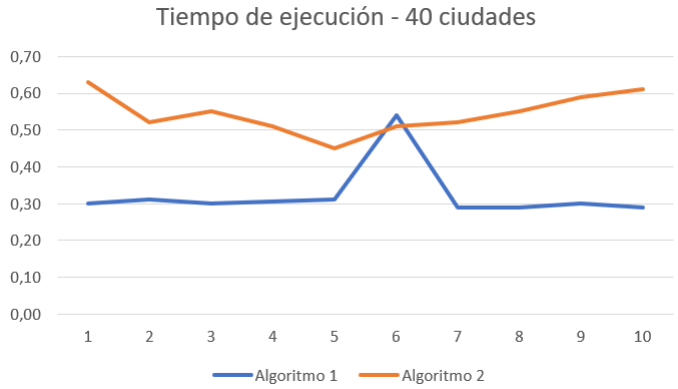

Figura 11: Tiempo de ejecución para 40 ciudades

Iteración	Algoritmo 1	Algoritmo 2
1	0,39	0,72
2	0,36	0,65
3	0,35	0,64
4	0,35	0,76
5	0,36	0,60
6	0,38	0,70
7	0,37	0,67
8	0,38	0,65
9	0,38	0,78
10	0,37	0,58

Cuadro 5: Tabla con los últimos datos proporcionados

El promedio de tiempo de ejecución para el Algoritmo 1 es 0.369 segundos, mientras que para el Algoritmo 2 es 0.675 segundos.

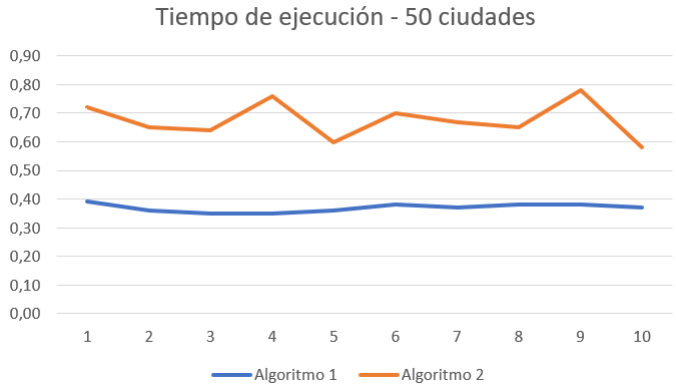

Figura 12: Tiempo de ejecución para 50 ciudades

Al observar el tiempo de ejecución de ambos algoritmos con un número de ciudades entre 10 y 20, se nota claramente que el Algoritmo 2 tiene una ventaja significativa. Sin embargo, esta ventaja se reduce gradualmente a medida que el número de ciudades aumenta. Esta diferencia puede explicarse por los parámetros establecidos en el Algoritmo 1, donde variaciones sutiles pueden afectar su desempeño conforme crece el tamaño del problema. Modificaciones en estos parámetros pueden influir en el tiempo de ejecución, ya que ajustes ineficientes pueden llevar a un aumento en el tiempo necesario

para encontrar soluciones óptimas.

5.4.3. Conclusiones del análisis

1. **Simplicidad vs. Escalabilidad:** Aunque el Algoritmo 2 es más simple conceptualmente, el Algoritmo 1 generalmente ofrece mayor escalabilidad y eficiencia computacional para problemas de tamaño considerable. Esto se debe a su enfoque más robusto y estructurado en la aplicación de operadores genéticos, lo cual optimiza la búsqueda de soluciones óptimas en entornos complejos.

2. **Importancia de los Parámetros:** La configuración de parámetros en el Algoritmo 1 juega un papel crucial. Pequeños ajustes en la selección, cruce y mutación pueden tener un impacto significativo en la eficiencia del algoritmo a medida que aumenta el número de ciudades. Una configuración óptima permite al Algoritmo 1 adaptarse mejor a problemas más grandes sin un aumento desproporcionado en el tiempo de ejecución.

Conclusiones

En este trabajo de investigación se realiza un análisis detallado y la implementación del Algoritmo Genético (GA) para resolver el Problema de las Agencias de Viajes (TSP). El estudio se realizó hasta el análisis de complejidad, donde se muestra que los resultados muestran que los algoritmos genéticos son herramientas potentes y flexibles capaces de resolver problemas complejos de optimización como el TSP. Su capacidad para explorar grandes espacios de soluciones y adaptarse a diferentes configuraciones proporciona ventajas significativas sobre otros enfoques tradicionales.

Durante el desarrollo del proyecto se identificaron las ventajas y limitaciones del GA. Entre sus múltiples ventajas destaca su adaptabilidad y capacidad para encontrar una solución casi óptima en un plazo de tiempo razonable. Sin embargo, el rendimiento de los algoritmos genéticos puede verse afectado por el establecimiento de parámetros clave como el tamaño de la población, las tasas de mutación y cruce y las estrategias de selección. Por tanto, la correcta configuración de estos parámetros es fundamental para obtener buenos resultados, y los experimentos realizados confirmaron que una mala configuración puede conducir a soluciones subóptimas o a un aumento significativo del tiempo de cálculo. Las pruebas y ajustes son esenciales para adaptar el algoritmo genético a las características específicas del problema.

Además, los resultados obtenidos por GA se comparan con otros métodos de análisis de TSP, como algoritmos exactos y algoritmos heurísticos. Aunque los algoritmos genéticos no pueden garantizar una optimización absoluta como los algoritmos exactos, proporcionan una buena relación entre la calidad de la solución y el tiempo de ejecución y, en algunos casos, superan a otros métodos heurísticos en términos de velocidad y precisión.

La implementación de algoritmos genéticos utilizando Python y herramientas de concatenación demostró ser eficiente y práctica. Python tiene un extenso ecosistema de bibliotecas que facilita el desarrollo y la experimentación, permitiendo un análisis detallado de la complejidad y el rendimiento del algoritmo. Las investigaciones confirman que los algoritmos genéticos son aplicables no sólo a TSP sino también a otros problemas de optimización en campos que van desde la logística hasta la bioinformática. La flexibilidad del modelo permite adaptarlo y ampliarlo para resolver problemas específicos con diferentes restricciones y objetivos. Se han identificado varias áreas para investigaciones futuras, incluida la combinación de GA con otros métodos híbridos, la exploración de nuevas estrategias de selección y mutación y la

aplicación de GA a problemas de TSP más complejos y grandes.

En resumen, los algoritmos genéticos son una solución factible y eficaz al problema de las agencias de viajes que proporciona un equilibrio adecuado entre complejidad y precisión. Su implementación en Python proporciona una plataforma sólida para futuras investigaciones y aplicaciones prácticas, lo que demuestra su potencial para resolver problemas complejos de optimización.

Referencias

1. Agapitos, A., O'Neill, M., & Brabazon, A. (2010). Evolutionary learning of technical trading rules without data-mining bias. En *Parallel Problem Solving from Nature, PPSN XI* (pp. 294–303). Springer Berlin Heidelberg.

2. Martínez, E., Rivera, M., Marcial, L., & Sandoval, L. (2016). Implementación paralela de un algoritmo genético para el problema del agente viajero usando OpenMP. *Research in Computing Science, 128*, 9-19. Recuperado de `https://rcs.cic.ipn.mx/2016_128/Implementacion%20paralela%20de%20un%20algoritmo%20genetico%20para%20el%20problema%20del%20agente%20viajero.pdf`

3. Morales, J. y Castro, M. (2016). Un algoritmo memético paralelo para TSP. *Investigación y Desarrollo en Robótica y Computación*. México. Recuperado de `https://posgrado.lapaz.tecnm.mx/uploads/archivos/2016-1.pdf`

4. Martínez, E. (2017). Implementación paralela de un algoritmo genético para resolver el problema del agente viajero. Tesis de maestría, Benemérita Universidad Autónoma de Puebla. Recuperado de `https://repositorioinstitucional.buap.mx/server/api/core/bitstreams/287d64ed-69fd-4272-b0d3-dcc313cb914a/content`

5. López, E., Salas, Ó., & Murillo, Á. (2014). El problema del agente viajero: un algoritmo determinístico usando búsqueda tabú. *Revista de Matemática: teoría y aplicaciones, 21*(1), 127-144.

6. Del Cartero Viajante, T. S. P. (2004). Técnicas heurísticas aplicadas al problema. *Scientia et Technica, 10*(24).

7. Flores, J. L. M., & Robles, F. S. (2017). Modelos de Optimización para la Logística en una Cadena de Suministro Agroalimentaria. *Revista de la Ingeniería Industrial, 11*(1), 1-7.

8. Challenger-Pérez, I., Díaz-Ricardo, Y., & Becerra-García, R. A. (2014). El lenguaje de programación Python. *Ciencias Holguín, XX*(2), 1-13.

9. Soto Gómez, E. (2021). Python en Ingeniería en Ciencias Informáticas. *Serie Científica De La Universidad De Las Ciencias Informáticas, 14*(12), 1-15. Recuperado de `https://publicaciones.uci.cu/index.php/serie/article/view/879`