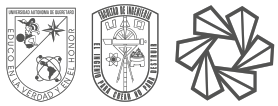

UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
FACULTAD DE INGENIERÍA
DIVISIÓN DE INVESTIGACIÓN Y POSGRADO

Algoritmos Metaheurísticos: Práctica 2. Algoritmo Genético: Problema del Vendedor Viajero.

Alumno:
Ing. Enrique Mena Camilo

Profesor:
Dr. Marco Antonio Aceves Fernández

Septiembre 2023

Índice

1. Objetivos

El objetivo de esta práctica consiste en encontrar soluciones a un problema de gran magnitud aplicado a la vida real, el problema del vendedor viajero. Esta solución deberá desarrollarse mediante el uso de algoritmos genéticos bajo las siguientes consideraciones:

- Deberá probar diferentes números de pobladores: 100, 250, 500, 1000.
- Deberá realizar la codificación mediante 18 genes de tipo numérico.
- Deberá diseñar una solución para medición de distancia (aptitud).
- Deberá implementar 2 métodos de selección.
- Deberá implementar 2 métodos de cruza: Partially Mapped Crossover y Cycle Crossover.
- Deberá implementar criterios de paro.
- Deberá implementar 2 métodos de mutación.
- Deberá implementar 1 criterio elitista.

2. Introducción

El Problema del Vendedor Viajero (TSP, por sus siglas en inglés: Traveling Salesman Problem) es uno de los problemas de optimización combinatoria más estudiados y conocidos en el ámbito de las ciencias de la computación y la investigación operativa. El desafío radica en encontrar el recorrido más corto que permita visitar una serie de ciudades y regresar al punto de origen, sin pasar más de una vez por cada ciudad. A pesar de su enunciado sencillo, el TSP es un problema NP-difícil, lo que significa que no se conoce una solución exacta que funcione en tiempo polinómico para instancias arbitrariamente grandes.

Dado que el número de posibles recorridos aumenta de manera factorial con el número de ciudades, resolver el TSP de forma exacta para una gran cantidad de ciudades se convierte en una tarea computacionalmente prohibitiva. Por lo tanto, la búsqueda de soluciones aproximadas y heurísticas se vuelve esencial.

En este contexto, los Algoritmos Genéticos (AG) han emergido como una herramienta poderosa y versátil para abordar el TSP. Inspirados en la teoría de la evolución natural, los AG simulan el proceso de selección natural donde los individuos más aptos son seleccionados para la reproducción, con el objetivo de producir descendencia de alta calidad para la próxima generación. En el caso del TSP, un individuo puede representar una posible solución o recorrido, y su aptitud puede estar relacionada con la distancia total o coste de ese recorrido.

La naturaleza adaptativa y probabilística de los AG los hace especialmente adecuados para explorar el amplio espacio de soluciones del TSP. Mediante la combinación de soluciones existentes (cruza) y la introducción de pequeñas perturbaciones aleatorias (mutaciones), los AG pueden explorar y explotar diferentes regiones del espacio de búsqueda, convergiendo a menudo a soluciones cercanas al óptimo global.

3. Marco teórico

3.1. Roulette Selection

El mecanismo de selección de ruleta, es una técnica popularmente usada en algoritmos genéticos para seleccionar posibles soluciones de acuerdo a su aptitud. La idea es dar a cada individuo una probabilidad de ser seleccionado que sea proporcional a su aptitud, de tal manera que los individuos con mayor aptitud tengan una mayor probabilidad de ser elegidos, pero aún permitiendo que los individuos con menor aptitud tengan alguna posibilidad.

En términos generales, este método consta de los siguientes pasos:

1. Cálculo de Aptitudes: En primer lugar, se calcula la aptitud de cada individuo en la población.
2. Suma de Aptitudes: Se suman todas las aptitudes para obtener una aptitud total de la población.
3. Cálculo de Probabilidades: Se determina la probabilidad de selección de cada individuo dividiendo su aptitud por la aptitud total de la población.
4. Selección: Se genera un número aleatorio entre 0 y 1. Luego, se selecciona el individuo cuya probabilidad acumulada incluye este número aleatorio. En otras palabras, imaginamos que giramos una ruleta donde cada segmento corresponde a un individuo, y el tamaño de cada segmento es proporcional a la aptitud del individuo.

Una representación gráfica de este mecanismo se puede observar en la Figura 1.

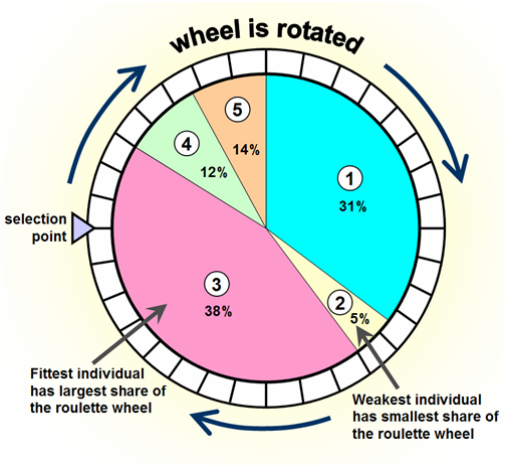

Figura 1: Diagrama de funcionamiento de selección por ruleta.

3.2. Tournament Selection

El mecanismo de selección por torneo es otro método popular utilizado en algoritmos genéticos para seleccionar individuos de una población basándose en su aptitud. Es un método relativamente simple y eficiente que puede adaptarse fácilmente para dar preferencia a individuos con mayor o menor aptitud.

La aproximación seleccionada para este trabajo consiste en realizar parejas de adentro hacia afuera dentro del conjunto de individuos. Esta aproximación presenta un problema en poblaciones grandes, ya que la diferencia de aptitud entre el individuo más alto y el menos apto es grande.

Este mecanismo se puede representar gráficamente mediante la Figura 2.

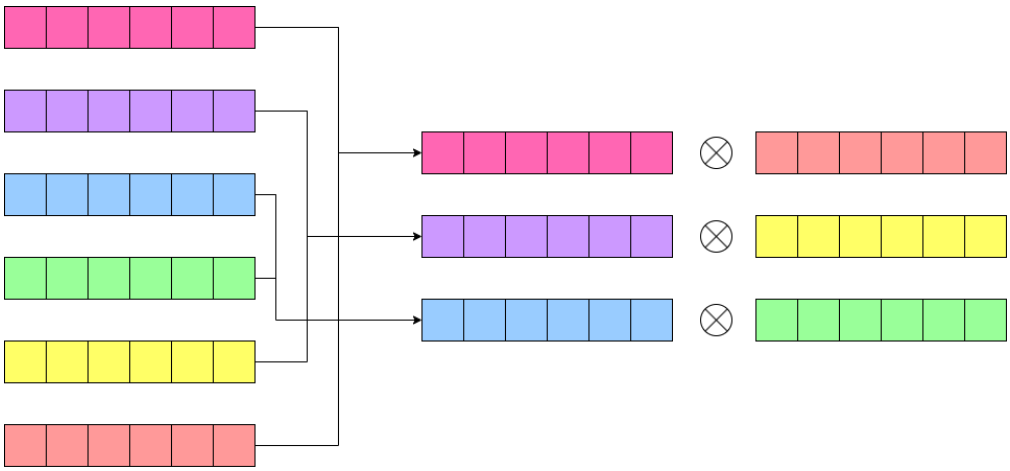

Figura 2: Diagrama de funcionamiento de selección por torneo.

3.3. Partially Mapped Crossover

El Partially Mapped Crossover (PMx) es una técnica de cruce especialmente diseñada para problemas de optimización combinatoria en los que las soluciones son representaciones de permutaciones de números. El objetivo del PMx es transferir sin ambigüedades segmentos de información entre dos padres, garantizando que los hijos resultantes sean permutaciones válidas.

El algoritmo consta de la siguiente serie de pasos:

1. Selección de un Segmento: Se seleccionan aleatoriamente dos puntos de corte en los padres, que definen un segmento.1
2. Intercambio de Segmentos: Se intercambian los segmentos entre los dos padres para formar la base de los dos hijos.

3. Reemplazo Fuera del Segmento: Para cada posición fuera del segmento intercambiado:
    a. Si el número en la posición correspondiente del otro padre ya está presente en el segmento intercambiado, se busca el número que ocupa esa posición en el segmento original y se sigue rastreando las posiciones hasta encontrar un número que no esté en el segmento intercambiado.
    b. Se reemplaza el número en la posición actual del hijo con el número encontrado en el paso anterior.
4. Finalización: Se repite el paso anterior hasta que todos los números fuera del segmento intercambiado en ambos hijos son permutaciones válidas de los números originales.

Este mecanismo se puede representar gráficamente como se muestra en la Figura 3.

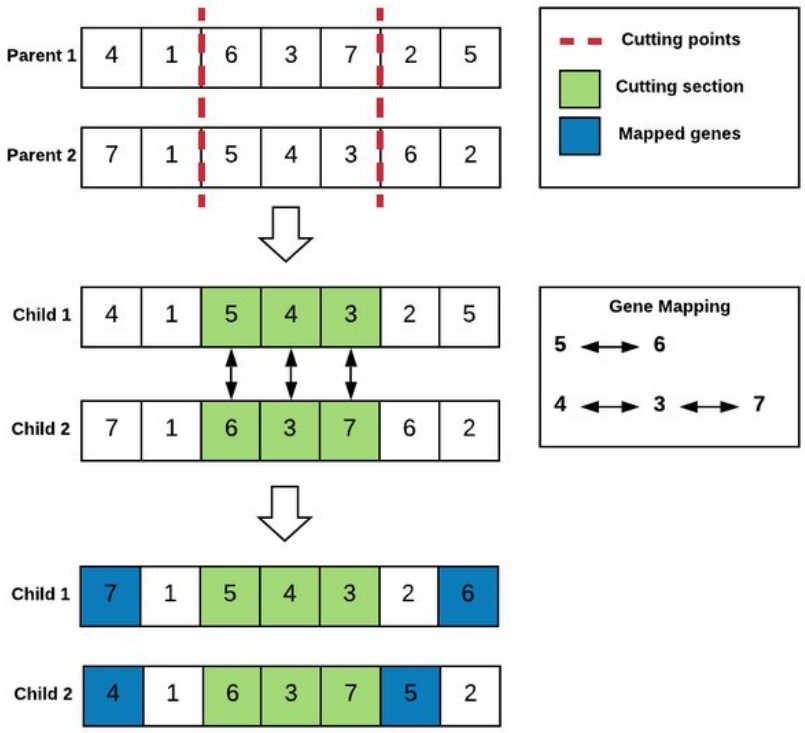

Figura 3: Diagrama de funcionamiento de partially mapped crossover.

3.4. Cycle Crossover

El Cycle Crossover (Cx) es otro operador de cruce específico para representaciones de permutación en algoritmos genéticos. A diferencia del PMx, el Cx se centra en la preservación de la posición absoluta de cada alelo, lo que puede conducir a una transmisión más consistente de ciertos rasgos de los padres a los hijos.

Los pasos generales de dicho mecanismo son:

1. Inicio del Primer Ciclo: Comienza con el primer elemento del primer padre.
2. Formación de Ciclos:
    a. Mira la posición de ese elemento en el segundo padre.
    b. Ve al elemento del primer padre en esa misma posición.
    c. Repite este proceso hasta que regrese al elemento inicial del primer padre. Ahora, tienes un ciclo.
3. Transferencia de Ciclos a los Hijos:
    a. Transfiere todos los elementos del primer ciclo del primer padre al primer hijo en las mismas posiciones. Haz lo mismo para el segundo hijo pero usando los alelos del segundo padre.
4. Formación de Ciclos Adicionales:
    a. Si aún quedan elementos sin asignar en los hijos, repite el proceso de formación de ciclos comenzando desde el primer elemento no asignado en el primer padre y crea otro ciclo.
    b. Alterna entre los padres para determinar qué alelos se transfieren a los hijos para cada ciclo adicional.
5. Finalización: Continúa hasta que todos los elementos están asignados en ambos hijos.

Este mecanismo se puede representar graficamente como se muestra en la Figura 4.

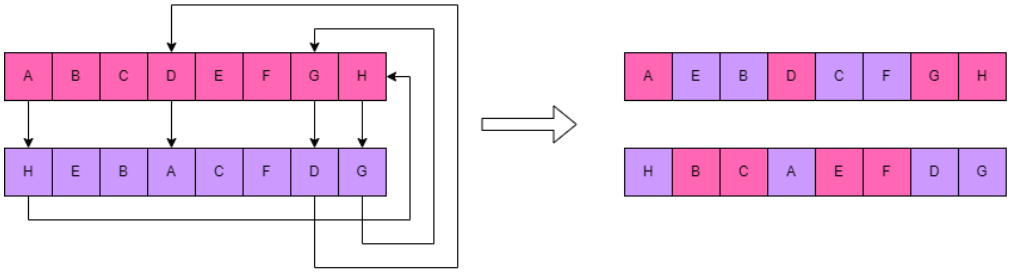

Figura 4: Diagrama de funcionamiento de cycle crossover.

3.5. Insert Mutation

El proceso Insert Mutation, es una técnica de mutación utilizada en algoritmos genéticos para modificar soluciones que se representan como permutaciones. Es especialmente útil en problemas donde las soluciones son representaciones de rutas o secuencias.

Los pasos de este proceso son:

1. Selección de segmentos: Se seleccionan aleatoriamente dos segmentos en la permutación, asegurándose de que no sean el misma segmento.

2. Inserción: Se intercambian los segmentos entre ellos.

Este mecanismo se puede observar en la Figura 5

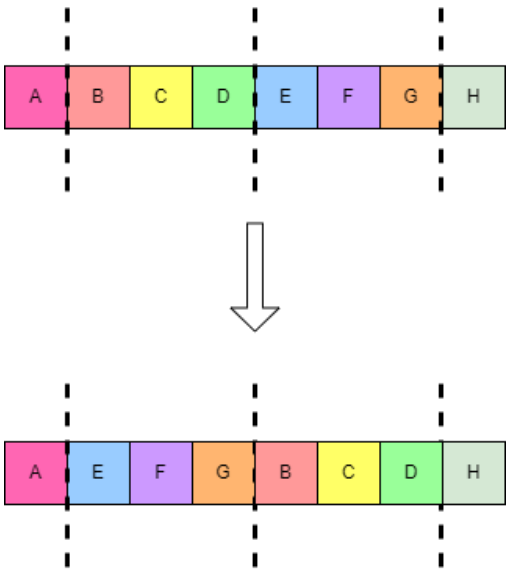

Figura 5: Diagrama de funcionamiento de insert mutation.

3.6. Inverse Mutation

La mutación por inversión, o Ïnverse Mutation", es otra técnica de mutación usada en algoritmos genéticos que trabajan con soluciones representadas como permutaciones. El procedimiento de esta técnica consta de los siguientes pasos:

1. Selección de segmento: Se seleccionan aleatoriamente dos puntos en la permutación. Estos puntos determinarán el inicio y el fin de un segmento.
2. Inversión: El segmento delimitado por estos dos puntos se invierte, es decir, el orden de los alelos dentro de este segmento se revierte.

Este mecanismo se puede observar en la Figura 6

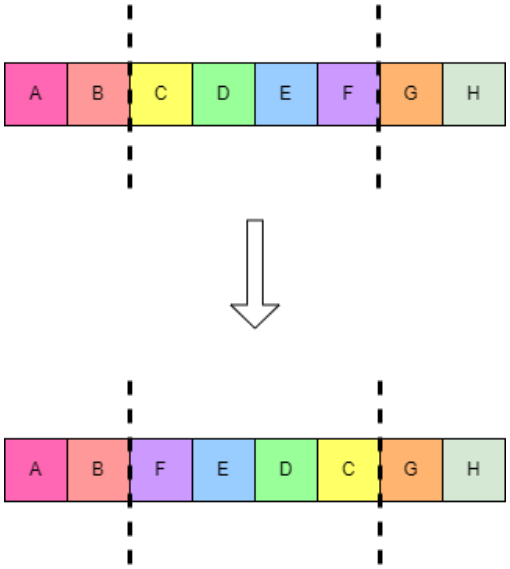

Figura 6: Diagrama de funcionamiento de inverse mutation.

4. Materiales y métodos

4.1. Algoritmo genético

Para esta práctica se desarrolló un algoritmo genético que se apega al proceso mostrado en la Figura 7.

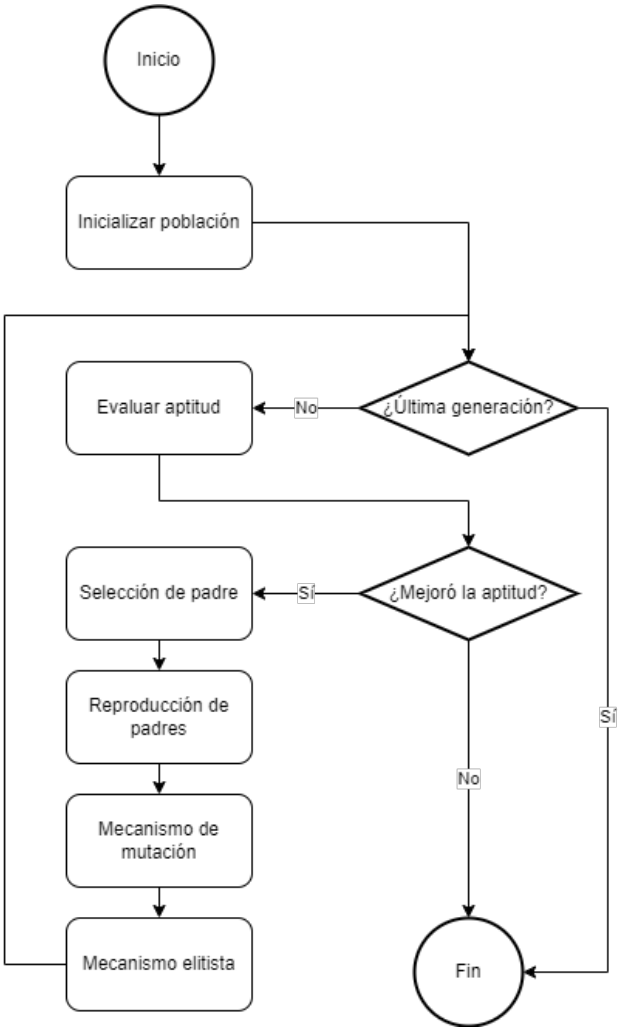

Figura 7: Diagrama de flujo del algoritmo genético implementado.

Para llevar a cabo dicho proceso, se optó por utilizar el lenguaje de programación Python, donde se diseñaron métodos genéricos para realizar los procesos de generación de población, evaluación de aptitud, selección de individuos, reproducción de individuos, mutación y mecanismos elitistas.

4.2. Implementación en Python

4.2.1. Inicializador de población

```python
#! /usr/bin/env python3
"""Population utilities."""

import numpy as np


def init_nobinary_population(n: int, genes: int = 10):
    available_genes = np.arange(1, genes+1)
    population = [np.random.choice(available_genes, (1, genes), replace=False)[0]
                  for _ in range(n)]

    return np.vstack(population)
```

4.2.2. Evaluación de individuos

```python
#! /usr/bin/env python3
"""cities."""

cities_list = [
    "Mexico City",
    "Quito",
    "Miami",
    "San Salvador",
    "Mendoza",
    "Guadalajara",
    "Merida",
    "Washington",
    "Monterrey",
    "Managua",
    "Caracas",
    "Boston",
    "Buenos Aires",
    "New York",
    "Panama City",
    "Brasilia",
    "Montevideo",
    "Bogotá",
]

cities_mapper = {idx+1: city for idx, city in enumerate(cities_list)}
```

```python
#! /usr/bin/env python3
"""Distance helper."""

import requests
import pandas as pd
from geopy.distance import geodesic

import cities


class DistanceHelper:
    def __init__(self, offline_mode: bool = False, on_demad_measure: bool = True):
        if offline_mode:
            self.distances = pd.read_csv("./data/predefined_distances.csv")
```

```
            self.distances = self.distances.set_index("X")
            self.cities = self.distances.columns.to_list()

        else:
            self.cities = cities.cities_list
            self.distances = pd.DataFrame(index=self.cities, columns=self.cities)

            if on_demad_measure is False:
                for city1 in self.cities:
                    for city2 in self.cities:
                        self.distances.loc[city1][city2] = self.measure_distance(city1, city2)

    def get_coordinates(self, city: str):
        base_url = 'https://nominatim.openstreetmap.org/search'
        params = {'q': city, 'format': 'json'}
        response = requests.get(base_url, params=params).json()
        location = response[0]

        return float(location['lat']), float(location['lon'])

    def measure_distance(self, city1: str, city2: str):
        if city1 == city2:
            return 0

        coordinates1 = self.get_coordinates(city1),
        coordinates2 = self.get_coordinates(city2)

        return geodesic(coordinates1, coordinates2).kilometers

    def get_distance(self, city1: str, city2: str):
        distance = self.distances.loc[city1][city2]

        if pd.isnull(distance):
            distance = self.measure_distance(city1, city2)
            self.distances.loc[city1][city2] = distance

        return distance
```

```
#! /usr/bin/env python3
"""Aptitude utilities."""

import numpy as np


def evaluate_tsp_individue(individue: np.ndarray, distances, cities_mapper: dict):
    aptitude = 0
    genes_ = len(individue)
    for idx in range(genes_):
        city_start = individue[idx]
        city_end = individue[idx+1] if idx < genes_-1 else individue[0]
        aptitude += distances.get_distance(cities_mapper[city_start], cities_mapper[city_end])

    return aptitude


def evaluate_tsp_population(population:np.ndarray, distances, cities_mapper: dict, desc: bool = False):
    aptitudes = np.vstack([evaluate_tsp_individue(individue, distances,cities_mapper) for individue in population])

    direction = -1 if desc else 1
    sorted_indexes = aptitudes[:, -1].argsort()[::direction]
```

```
23      sorted_population = population[sorted_indexes]
24      sorted_aptitudes = aptitudes[sorted_indexes]
25      avg_aptitude = np.mean(sorted_aptitudes)
26      min_aptitude = np.min(sorted_aptitudes)
27
28      return sorted_population, sorted_aptitudes, avg_aptitude, min_aptitude
```

4.2.3. Selección de parejas

```
1   #! /usr/bin/env python3
2   """Selection utilities."""
3
4   import numpy as np
5
6
7   def roulette_selection(aptitudes: np.ndarray):
8       total_aptitude = aptitudes.sum()
9
10      couples = []
11      couple = []
12
13      for _ in range(aptitudes.shape[0]):
14          accumulated = 0
15          selection = np.random.uniform(0, total_aptitude)
16
17          for idx, aptitude  in enumerate(aptitudes):
18              accumulated += aptitude
19              if accumulated >= selection:
20                  couple.append(idx)
21                  break
22
23          if len(couple) == 2:
24              couples.append(couple)
25              couple = []
26
27      return np.vstack(couples)
28
29
30  def tournament_selection(population: np.ndarray):
31      population_len = population.shape[0]
32
33      couples = []
34      for idx in range(population_len//2):
35          couples.append([idx, population_len-idx-1])
36
37      return np.vstack(couples)
```

4.2.4. Reproducción de individuos

```
1   #! /usr/bin/env python3
2   """Crossover utilities."""
3
4   import numpy as np
5
6
7   def partially_mapped_crossover(popoulation: np.ndarray, parents: np.ndarray):
8       def map_value(value, mapping):
9           while value in mapping:
```

```
            value = mapping[value]
        return value

    childrens = np.empty_like(popoulation)
    size = len(popoulation[0])
    middle_ = size//2
    crop_len = size//6
    point1, point2 = middle_-crop_len, middle_+crop_len

    for idx, couple in enumerate(parents):
        childrens[idx*2,   point1:point2+1] = popoulation[couple[0], point1:point2+1]
        childrens[idx*2+1, point1:point2+1] = popoulation[couple[1], point1:point2+1]

        parent1_mapping = dict(zip(popoulation[couple[0], point1:point2+1], popoulation[couple[1], point1:point2+1]))
        parent2_mapping = dict(zip(popoulation[couple[1], point1:point2+1], popoulation[couple[0], point1:point2+1]))

        for i in range(size):
            if i < point1 or i > point2:
                childrens[idx*2  ][i] = map_value(popoulation[couple[1]][i], parent1_mapping)
                childrens[idx*2+1][i] = map_value(popoulation[couple[0]][i], parent2_mapping)

    return childrens


def cycle_crossover(population: np.ndarray, parents: np.ndarray):
    childrens = np.empty_like(population)
    for idx, couple in enumerate(parents):
        parent1, parent2 = population[couple[0]], population[couple[1]]
        child1, child2 = np.empty_like(parent1), np.empty_like(parent2)
        visited = np.zeros_like(parent1, dtype=bool)
        cycle_start = 0
        while not visited[cycle_start]:
            visited[cycle_start] = True
            child1[cycle_start] = parent1[cycle_start]
            child2[cycle_start] = parent2[cycle_start]
            cycle_start = np.where(parent1 == parent2[cycle_start])[0][0]
        for i in range(len(parent1)):
            if not visited[i]:
                child1[i] = parent2[i]
                child2[i] = parent1[i]
        childrens[idx*2] = child1
        childrens[idx*2+1] = child2
    return childrens

```

4.2.5. Mutación

```
#! /usr/bin/env python3
"""Mutation utilities."""

import numpy as np


def insert_mutation(individues: np.ndarray, mr: float = 0.02):
    total_individues = individues.shape[0]
    to_mutate = np.random.choice(total_individues, int(total_individues*mr), replace=False)

    individue_len = individues.shape[1]
    segment_len = individue_len//5
    segments = [[idx, idx+segment_len] for idx in range(0, individue_len, segment_len)]
```

```
14
15      for idx in to_mutate:
16          segment1, segment2 = np.random.choice(len(segments), 2, replace=False)
17          segment1, segment2 = segments[segment1], segments[segment2]
18          org_individue = individues[idx].copy()
19          individues[idx][segment1[0]:segment1[1]] = org_individue[segment2[0]:segment2[1]]
20          individues[idx][segment2[0]:segment2[1]] = org_individue[segment1[0]:segment1[1]]
21
22      return individues
23
24
25  def inverse_mutation(individues: np.ndarray, mr: float = 0.02):
26      total_individues = individues.shape[0]
27      individue_len = individues.shape[1]
28      to_mutate = np.random.choice(total_individues, int(total_individues*mr))
29
30      for idx in to_mutate:
31          idx_l, idx_r = sorted(np.random.choice(individue_len, 2))
32          idx_r = idx_r if idx_r == 17 else idx_r + 1
33          individues[idx][idx_l:idx_r] = individues[idx][idx_l:idx_r][::-1]
34
35      return individues
```

4.2.6. Algoritmo completo

```
1   import sys
2
3   import numpy as np
4
5   from tqdm import tqdm
6
7   sys.path.append("..")
8
9   from cities import cities_mapper
10  from distance_helper import DistanceHelper
11  from utils.population import init_nobinary_population
12  from utils.aptitude import evaluate_tsp_population
13  from utils.selection import roulette_selection, tournament_selection
14  from utils.crossover import partially_mapped_crossover, cycle_crossover
15  from utils.mutation import insert_mutation, inverse_mutation
16  from utils.visualization import plot_aptitude
17
18
19  def genetic_competence(population: np.ndarray, childrens: np.ndarray):
20      all_population = np.vstack([population, childrens])
21      sorted_population, _, _, _ = evaluate_tsp_population(all_population, distances, cities_mapper)
22      return sorted_population[:population.shape[0]]
23
24
25  distances = DistanceHelper(offline_mode=True)
26
27  genes = 18
28  generations = 50
29  max_stagnant_generations = 10
30  delta = 100
31
32  # Roulette + PMx
33  population = init_nobinary_population(n=100, genes=18)
34  avg_aptitudes_ = []
35  min_aptitudes_ = []
```

```
evolution_ = []
last_best_aptitude = float("inf")
stagnant_generations = 0

for _ in tqdm(range(generations), desc="Generation"):
    population, aptitudes, avg_aptitude, min_aptitude = evaluate_tsp_population(population, distances, cities_mapper)
    avg_aptitudes_.append(avg_aptitude)
    min_aptitudes_.append(min_aptitude)

    if abs(min_aptitude - last_best_aptitude) < delta:
        stagnant_generations += 1
        if stagnant_generations > max_stagnant_generations:
            break
    else:
        stagnant_generations = 0

    last_best_aptitude = min_aptitude
    parents = roulette_selection(aptitudes)
    childrens = partially_mapped_crossover(population, parents)
    childrens = insert_mutation(childrens)
    population = genetic_competence(population, childrens)

print(f"Best aptitude: {min_aptitudes_[-1]}")
print(f"Avg aptitude: {avg_aptitudes_[-1]}")

title = "Roulette selection + PMx"
plot_aptitude(avg_aptitudes_, min_aptitudes_, generations, title)
```

4.3. Pruebas realizadas

Se implementaron un total de 2 mecanismos de selección, 2 mecanismos de cruza, 2 mecanismos de mutación y 1 mecanismo elitista. Se optó por realizar una combinación de todos los mecanismos de selección con todos los mecanismos de cruza, intercalando los mecanismos de mutación. Las combinaciones resultantes fueron:

- Algoritmo 1: Selección Ruleta + PMx + Insert Mutation + 100 individuos.
- Algoritmo 2: Selección Ruleta + Cx + Inverse Mutation + 250 individuos.
- Algoritmo 3: Selección Torneo + PMx + Insert Mutation + 500 individuos.
- Algoritmo 4: Selección Torneo + Cx + Inverse Mutation + 1000 individuos.

Cabe destacar que, dado que solamente se implementó 1 mecanismo elitista, este fue aplicado a todos los algoritmos por igual. También es importante mencionar que se implementó 1 criterio de paro híbrido:

- Número de generaciones. Se estableció un límite de 10 generaciones para todos los algoritmos.
- Término por δ. Se estableció un valor $\delta = 1000$ con un máximo de 50 generaciones.

5. Resultados

Relacionado a los resultados del algoritmo 1. Este algoritmo tomó un total de 17 generaciones. Se obtuvo un aptitud promedio final de 36,557.61 km y su mejor individuo consiguió una aptitud de 36,557.61 km. El proceso de convergencia se puede observar en la Figura 8.

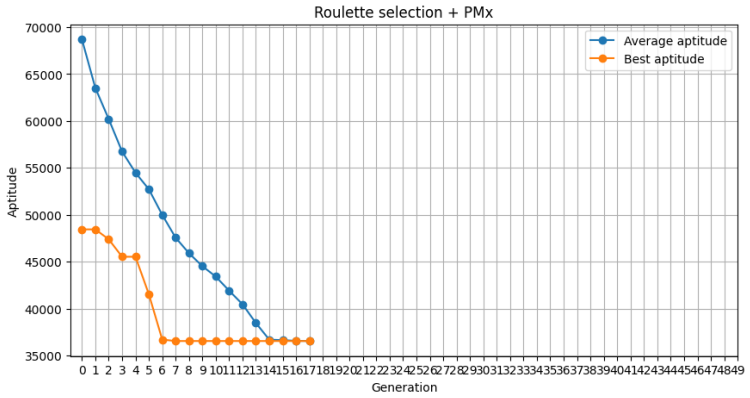

Figura 8: Evolución de la aptitud de los individuos del algoritmo 1.

Relacionado a los resultados del algoritmo 2. Este algoritmo tomó un total de 50 generaciones. Se obtuvo un aptitud promedio final de 31,214.21 km y su mejor individuo consiguió una aptitud de 29,746.86 km. El proceso de convergencia se puede observar en la Figura 9.

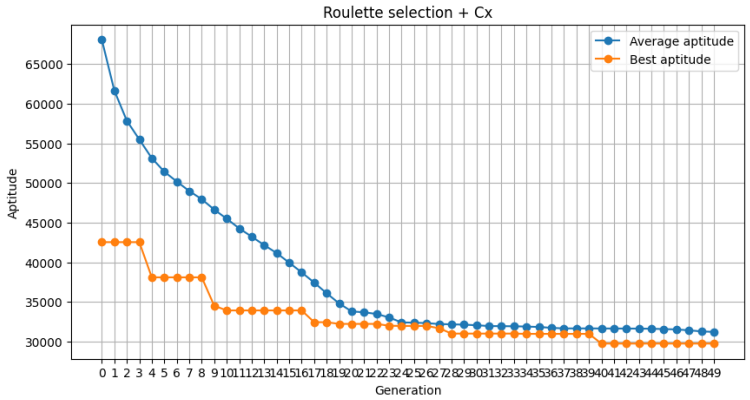

Figura 9: Evolución de la aptitud de los individuos del algoritmo 2.

Relacionado a los resultados del algoritmo 3. Este algoritmo tomó un total de 17 generaciones. Se obtuvo un aptitud promedio final de 32,423.60 km y su mejor individuo consiguió una aptitud de 30,304.17 km. El proceso de convergencia se puede observar en la Figura 10.

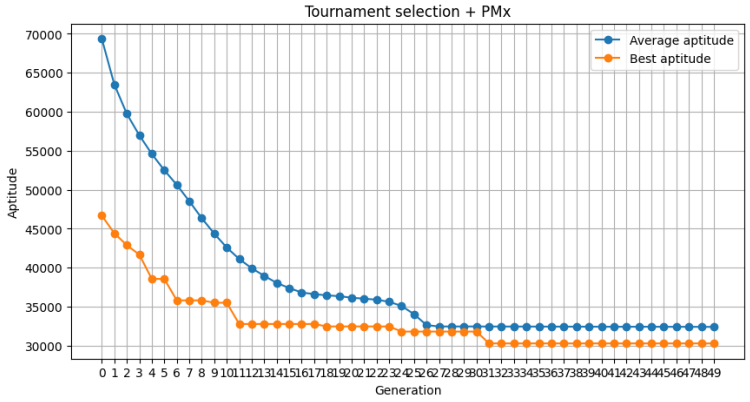

Figura 10: Evolución de la aptitud de los individuos del algoritmo 3.

Relacionado a los resultados del algoritmo 4. Este algoritmo tomó un total de 17 generaciones. Se obtuvo un aptitud promedio final de 25,060.90 km y su mejor individuo consiguió una aptitud de 24,571.73 km. El proceso de convergencia se puede observar en la Figura 11.

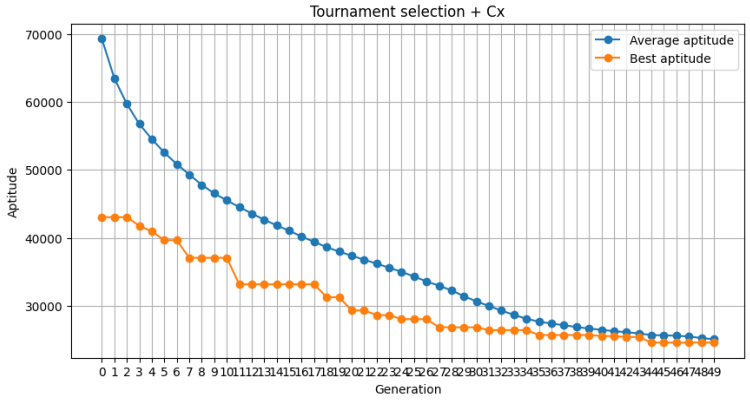

Figura 11: Evolución de la aptitud de los individuos del algoritmo 4.

Todos estos resultados se puden resumir en la Tabla 1, en la cual se puede observar que los mejores resultados fueron los obtenidos por el algoritmo 4.

Tabla 1: Resumen de resultados de todos los algoritmos.

Algoritmo	Mejor Aptitud [km]	Aptitud Promedio [km]	Total de individuos
Algoritmo 1	36,557.61	36,557.61	100
Algoritmo 2	29,746.86	31,214.21	250
Algoritmo 3	30,304.17	32,423.60	500
Algoritmo 4	24,571.73	25,060.90	1,000

6. Conclusiones

Al analizar los resultados de todos los algoritmos, se destaca el rendimiento superior del algoritmo número 4. A primera vista, uno podría atribuir este éxito a la combinación efectiva de los mecanismos de selección por torneo, Cx y la mutación inversa. Sin embargo, es plausible que la verdadera razón detrás de estos resultados sea la cantidad de individuos utilizados en este algoritmo: 1,000 individuos. Esta amplia muestra permitió al algoritmo explorar un espacio de búsqueda más extenso en comparación con los demás, lo que si bien implicó un mayor tiempo de ejecución, también se tradujo en resultados superiores.

Aunque los demás algoritmos no tuvieron un rendimiento deficiente, y partiendo de la premisa de que este problema no tiene una única solución sino un conjunto de soluciones óptimas, se podría inferir que cualquier algoritmo genético es adecuado para resolver el TSP. No obstante, es importante considerar que una población más numerosa tiende a garantizar resultados más óptimos.

Referencias bibliográficas

[1] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer, 2015, ISBN: 9783662448731.

[2] K. Dahal, K. C. Tan, and P. I. Cowling, *Evolutionary Scheduling*, 1st ed. Springer, 2007, ISBN: 9783540485827.

[3] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer, 2008, ISBN: 9783540731894.