Examen: Algoritmos Genéticos: Problema TPL

Realizar un algoritmo genético que permita calcular la ruta más rápida o cercana a la más rápida para recorrer ciertos puntos en el mapa.

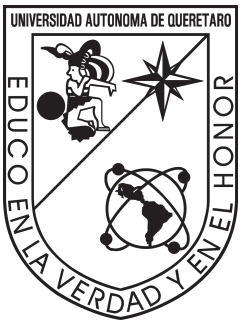
LICENCIATURA
EN
INGENIERÍA EN SOFTWARE

Documento realizado por:

VANEGAS ARCEGA ALFREDO (EXP. 297121)

Evaluado por el docente:

MALAGÓN GARCÍA JESÚS SALVADOR



FACULTAD DE INFORMÁTICA
UNIVERSIDAD AUTONÓMA DE QUERÉTARO

INTELIGENCIA ARTIFICIAL

MAYO, 2023

Índice general

Capítulo 1

Introducción

El problema del viajante (Travelling Salesman problem en inglés) trata de un viajante que debe cruzar todas las ciudades de la forma más rápida posible. Este es un clásico problema NP de optimización.

Su dificultad es el factorial crecimiento de combinaciones posibles para los viajes, para ello, se utiliza los algoritmos genéticos para facilitar una solución utilizando combinatoria.

Los algoritmos genéticos se pueden utilizar para una variedad grande de problemas que requieran de optimización. Los métodos de mutación y cruzamiento se pueden utilizar para cualquier tipo de problema que sea abstraído a una colección de elementos únicos.

El caso a resolver es de cruzar unos puntos arbitrarios en la ciudad de Querétaro.

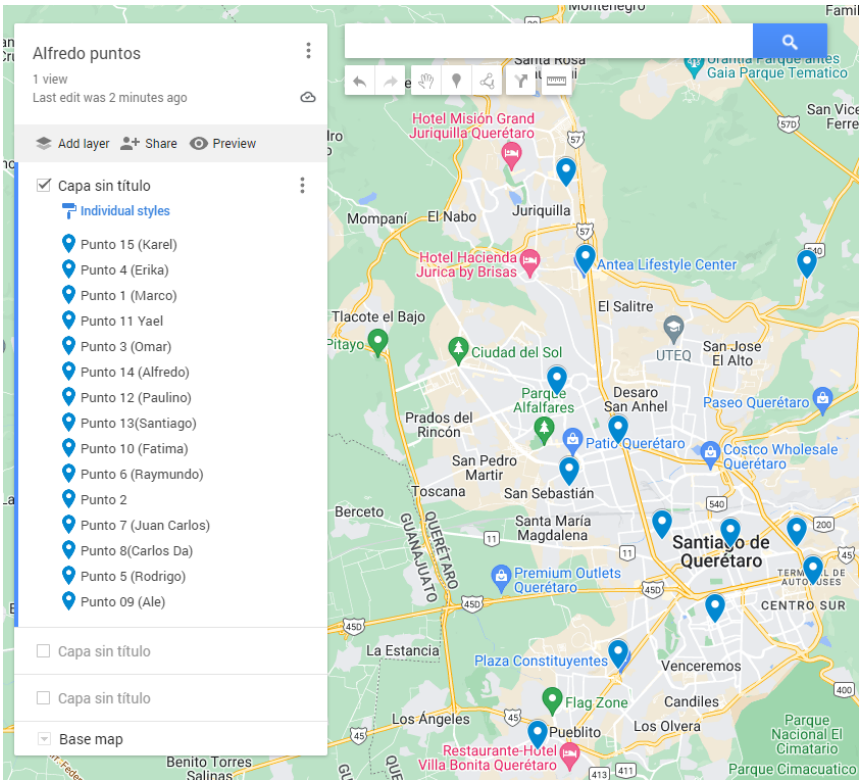


Figura 1.1: Puntos arbitrarios escogidos para el problema.

De los siguientes puntos, se saco la distancia entre cada punto en kilométros por medio del mapa de Google Maps, asegurando que sea el mismo recorrido de ida y regreso. Los resultados fueron los siguientes:

Cuadro 1.1: Tabla de distancias entre cada ciudad en kilométros

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	15	8	23	5	9	14	23	3	18	7	8	11	20	8
2	15	0	17	28	24	28	16	23	15	12	17	20	33	21	17
3	8	17	0	27	9	21	13	21	11	9	3	11	18	14	4
4	23	28	27	0	20	25	34	51	19	30	24	17	27	18	28
5	5	24	9	20	0	7	23	39	4	19	13	13	10	5	17
6	9	28	21	25	7	0	26	38	8	22	17	12	5	9	21
7	14	16	13	34	23	26	0	13	18	6	11	12	35	22	11
8	23	23	21	51	39	38	13	0	26	21	20	27	36	31	19
9	3	15	11	19	4	8	18	26	0	14	8	4	11	4	12
10	18	12	9	30	19	22	6	21	14	0	8	15	29	19	6
11	7	17	3	24	13	17	11	20	8	8	0	9	23	13	5
12	8	20	11	17	9	12	12	27	4	15	9	0	17	5	15
13	11	33	18	27	10	5	35	36	11	29	23	17	0	17	20
14	20	21	14	18	5	9	22	31	4	19	13	5	17	0	16
15	8	17	4	28	17	21	11	19	12	6	5	15	20	16	0

Esos resultados de distancias serán tratadas como un clásico problema de Dijkstra, ya que la distancia entre ambas ciudades es la misma y tenemos una diagonal representando que la distancia entre un punto sobre si mismo es de 0.

Capítulo 2

Desarrollo

2.1. Codificación de dominio

El dominio del problema consiste en un diccionario nativo de Python para almacenar los genotipos:

```
def get_initial_genotipes(total_genotipes):
    generated_genotipes = []
    for _ in range(0, total_genotipes):
        genotipe = []
        # La lista inicial consiste de todas las ciudades disponibles
        available_choices = [x for x in range(len(PATHS))]
        for _ in range(0, len(PATHS)):
            letter_index = np.random.randint(0, len(available_choices))
            letter = available_choices[letter_index]
            # Se quita la ciudad para que no se repita de nuevo
            available_choices.remove(letter)
            genotipe.append(letter)
        generated_genotipes.append({'alleles': genotipe, 'fitness': 0})
    generated_genotipes = evaluate_genotipes(generated_genotipes)
    return generated_genotipes
```

Los genotipos tienen dos llaves:

- Alleles: Son los aleles del genotipo que serían las ciudades.
- Fitness: La distancia total del recorrido de todas las ciudades en los aleles.

El dominio también en consiste en los siguientes paramétros que son es la figura 1.1 en forma de un array multidimensional llamado PATHS y los parámetros para la renovación general que serán explicados más adelante.

```
PATHS = [
    [0,15,8,23,5,9,14,23,3,18,7,8,11,20,8],
    [15,0,17,28,24,28,16,23,15,12,17,20,33,21,17],
    [8,17,0,27,9,21,13,21,11,9,3,11,18,14,4],
    [23,28,27,0,20,25,34,51,19,30,24,17,27,18,28],
    [5,24,9,20,0,7,23,39,4,19,13,13,10,5,17],
    [9,28,21,25,7,0,26,38,8,22,17,12,5,9,21],
    [14,16,13,34,23,26,0,13,18,6,11,12,35,22,11],
    [23,23,21,51,39,38,13,0,26,21,20,27,36,31,19],
    [3,15,11,19,4,8,18,26,0,14,8,4,11,4,12],
    [18,12,9,30,19,22,6,21,14,0,8,15,29,19,6],
    [7,17,3,24,13,17,11,20,8,8,0,9,23,13,5],
    [8,20,11,17,9,12,12,27,4,15,9,0,17,5,15],
    [11,33,18,27,10,5,35,36,11,29,23,17,0,17,20],
    [20,21,14,18,5,9,22,31,4,19,13,5,17,0,16],
    [8,17,4,28,17,21,11,19,12,6,5,15,20,16,0],
]

evolution_loop(
    total_genotipes=800,
```

```
    best_genotipes_kept=50,
    total_generations=100,
    operator="inversion"
)
```

2.2. Evaluación de la población

La evaluación se realizó agarrando el índice de la ciudad, y eso agarra sus respectivas distancias con las otras ciudades, y ya solo se agarra la distancia de la siguiente ciudad (que sería i+1).

```
def evaluate_genotipes(genotipes):
    for genotipe in genotipes:
        total_score = 0
        alleles = genotipe['alleles']
        for i in range(len(alleles)-1):
            cost = PATHS[alleles[i]][alleles[i+1]]
            total_score += cost
        # se considera anadir la puntuacion de la ultima ciudad a la primera
        total_score += PATHS[alleles[len(alleles)-1]][alleles[0]]
        genotipe['fitness'] = total_score
    return genotipes
```

2.3. Selección

El argumento **best_genotipes_kept** es la cantidad de genotipos con calificación de *fitness* más baja. Ya que en este problema se trata de encontrar la distancia más corta. Se pasan todas las puntuaciones, se sortean de forma ascendiente, y solo se mantienen las que especifica **best_genotipes_kept**. Se itera sobre todos los genotipos hasta que se tengan los mejores.

```
def get_best_genotipes(genotipes, best_genotipes_kept):
    best_genotipes = []
    scores = [genotipe['fitness'] for genotipe in genotipes]
    scores.sort(reverse=False)
    best_scores = list(set(scores[:best_genotipes_kept]))
    for i in range(0, len(genotipes)):
        if len(best_genotipes) == best_genotipes_kept:
            break
        if genotipes[i]['fitness'] in best_scores:
            best_genotipes.append(genotipes[i])
    return best_genotipes
```

2.4. Cruzamiento

El cruzamiento utilizado fue *Partially mapped crossover (PMX)* y en el código se observa que se escogen dos padres de forma aleatoria que generan dos hijos en la función de *pmx_crossing*.

```python
def pmx_offspring(parent1, parent2):
    cut1, cut2 = np.sort(rng.choice(np.arange(len(parent1)+1), size=2, replace=False))
    offspring = ['' for x in range(0, len(parent1))]

    offspring[cut1:cut2] = parent1[cut1:cut2]
    rest_index = np.concatenate([np.arange(0, cut1), np.arange(cut2, len(parent1))])
    for i in rest_index:
        candidate = parent2[i]
        while candidate in parent1[cut1:cut2]:
            candidate = parent2[np.where(parent1 == candidate)[0][0]]

        offspring[i] = candidate
    return offspring


def pmx_crossing(genotipes, total_offspring):
    childs = []
    for _ in range(0, int(np.floor(total_offspring / 2))):
        p1, p2 = np.sort(rng.choice(np.arange(len(genotipes)), size=2, replace=False))
        parent1, parent2 = np.array(genotipes[p1]['alleles']), np.array(genotipes[p2]['alleles'])
        offspring0 = pmx_offspring(parent1, parent2)
        offspring1 = pmx_offspring(parent2, parent1)
        childs.append({'alleles': offspring0, 'fitness': 0})
        childs.append({'alleles': offspring1, 'fitness': 0})
    return childs
```

La función de PMX funciona de la siguiente forma:

- La sección cortada del primer padre se copia directamente en las posiciones correspondientes del arreglo de descendencia.
- Las posiciones restantes fuera de la sección mapeada se llenan a partir del segundo padre, pero teniendo en cuenta evitar valores duplicados. Para cada posición i, se verifica si el valor correspondiente en padre 2 ya está presente en la sección mapeada de padre 1. En tal caso, se busca el mapeo de ese valor en padre 1 y se continúa este proceso hasta encontrar un valor de padre 2 que no esté presente en la sección mapeada. Esto evita crear valores duplicados en la descendencia.
- Se devuelve el arreglo resultante de descendencia.

Lo más importante a considerar es que no se debe añadir la primera ciudad otra vez al último índice, ya que el algoritmo tendría que ser reescrito, aparte que en el cruzamiento se puede seleccionar el ultimo índice y cruzarse, pero sería inútil ya que el último índice debe ser la primera ciudad. Así que en la función de evaluación se considera ese recorrido sin agregarlo explícitamente a los aleles del genotipo.

2.5. Mutación

2.5.1. Inversion

La mutación de inversión selecciona dos posiciones dentro de un cromosoma/recorrido al azar y luego invierte las ciudades en la subcadena entre estas dos posiciones.

```
def inversion_mutation(genotipes):
    for genotipe in genotipes:
        ran_start = np.random.randint(0, len(genotipe['alleles']) - 1)
        ran_end = np.random.randint(ran_start + 2, len(genotipe['alleles']) + 1)
        mutation = genotipe['alleles'][ran_start:ran_end]
        mutation.reverse()
        genotipe['alleles'][ran_start:ran_end] = mutation
    return genotipes
```

2.5.2. Displacement

La mutación de desplazamiento selecciona un recorrido secundario al azar y lo inserta en una posición aleatoria fuera del recorrido secundario. La inserción puede ser visto como un caso especial de desplazamiento en el que la subcadena contiene solo una ciudad.

```
def displacement_mutation(genotipes):
    for genotipe in genotipes:
        ran_start = np.random.randint(0, len(genotipe['alleles']) - 1)
        ran_end = np.random.randint(ran_start + 1, len(genotipe['alleles']))
        unavailable_choices = [x for x in range(ran_start, ran_end+1)]
        choices = [x for x in range(0, len(genotipe['alleles'])) if not x in
    unavailable_choices]
        if not len(choices) == 0:
            ran_pos = np.random.choice(choices)
            choices.remove(ran_pos)
        else:
            continue
        #selection = genotipe['alleles'][ran_start:ran_end+1]
        #displacement = genotipe['alleles'][ran_pos:ran_start]
        genotipe['alleles'][ran_start:ran_end+1], genotipe['alleles'][ran_pos:
    ran_start] = genotipe['alleles'][ran_pos:ran_start], genotipe['alleles'][ran_start
    :ran_end+1]
    return genotipes
```

2.5.3. Exchange

La mutación de intercambio selecciona dos posiciones al azar e intercambia las ciudades en estas posiciones.

```
def exchange_mutation(genotipes):
    mutations = []
    for genotipe in genotipes:
        ran_pos1, ran_pos2 = np.sort(rng.choice(np.arange(len(genotipe['alleles'])),
    size=2, replace=False))
        aux = genotipe['alleles'][ran_pos1]
```

```
        genotipe['alleles'][ran_pos1] = genotipe['alleles'][ran_pos2]
        genotipe['alleles'][ran_pos2] = aux
        mutations.append(genotipe)
    return mutations
```

2.5.4. Insertion

La mutación de inserción selecciona una ciudad al azar y la inserta en una posición aleatoria.

```
def insertion_mutation(genotipes):
    for genotipe in genotipes:
        ran_index = np.random.randint(0, len(genotipe['alleles']))
        ran_pos = np.random.randint(0, len(genotipe['alleles']))
        letter = genotipe['alleles'][ran_index]
        aux = genotipe['alleles'][ran_pos]
        genotipe['alleles'][ran_pos] = letter
        genotipe['alleles'][ran_index] = aux
    return genotipes
```

2.5.5. Inverted exchange

- Se selecciona dos posiciones dentro de un cromosoma/recorrido al azar y luego se invierte las ciudades en la subcadena entre estas
- En segundo lugar, después de seleccionar aleatoriamente una ciudad/nodo de las ciudades en la subcadena invertida, luego se intercambia esta ciudad con una ciudad elegida al azar fuera de la subcadena invertida.

```
def inverted_exchange_mutation(genotipes):
    for genotipe in genotipes:
        ran_start, ran_end = np.sort(rng.choice(np.arange(len(genotipe['alleles'])),
    size=2, replace=False))
        selection = genotipe['alleles'][ran_start:ran_end]
        selection.reverse()
        genotipe['alleles'][ran_start:ran_end] = selection
        ran_ex1 = np.random.randint(0, len(selection))
        ran_ex2 = np.random.randint(0, len(genotipe['alleles']))
        aux = genotipe['alleles'][ran_ex1]
        genotipe['alleles'][ran_ex1] = genotipe['alleles'][ran_ex2]
        genotipe['alleles'][ran_ex2] = aux
    return genotipes
```

2.5.6. Inverted displacement

- El primer paso es seleccionar dos posiciones dentro de un cromosoma/recorrido al azar y luego invertir las ciudades en la subcadena entre estas dos posiciones.
- En el segundo paso, la subcadena obtenida en el primer paso se inserta en una posición aleatoria fuera de la subcadena.

```python
def inverted_displacement_mutation(genotipes):
    for genotipe in genotipes:
        ran_start, ran_end = np.sort(rng.choice(np.arange(len(genotipe['alleles'])),
    size=2, replace=False))
        ran_pos = np.random.randint(0, ran_end - ran_start)
        genotipe['alleles'][ran_start:ran_end].reverse() #selection
        #displacement = genotipe['alleles'][ran_pos:ran_start]
        genotipe['alleles'][ran_start:ran_end], genotipe['alleles'][ran_pos:ran_start]
     =  genotipe['alleles'][ran_pos:ran_start], genotipe['alleles'][ran_start:ran_end]
    return genotipes
```

2.6. Renovación generacional

En la siguiente función se selecciona el operador de mutación y se realiza todo el proceso anterior. Esto se repite por una cantidad de generaciones especificada por el usuario.

```python
def evolution_loop(
    total_genotipes: int,
    best_genotipes_kept: int,
    total_generations: int,
    operator: Literal['inversion',
                      'displacement',
                      'exchange',
                      'insertion',
                      'inverted_exchange',
                      'inverted_displacement']
):
    mutation_operator = inversion_mutation
    if operator == "inversion":
        mutation_operator = inversion_mutation
    elif operator == "displacement":
        mutation_operator = displacement_mutation
    elif operator == "exchange":
        mutation_operator = exchange_mutation
    elif operator == "insertion":
        mutation_operator = insertion_mutation
    elif operator == "inverted_exchange":
        mutation_operator = inverted_exchange_mutation
    elif operator == "inverted_displacement":
        mutation_operator = inverted_displacement_mutation

    # Initial variables
    generations_count = 0
    genotipes = get_initial_genotipes(total_genotipes)
    for _ in range(total_generations):
        best_genotipes = get_best_genotipes(genotipes, best_genotipes_kept)
        genotipes = pmx_crossing(best_genotipes, total_genotipes - len(best_genotipes)
    )
        genotipes.extend(best_genotipes)
        genotipes = mutation_operator(genotipes)
        genotipes = evaluate_genotipes(genotipes)
        best_genotipe = get_best_genotipe(genotipes)
        generations_count += 1
        print("Generation: ", generations_count)
        print("Best:", best_genotipe)
```

Capítulo 3

Resultados

3.1. Resultados sin regresar a la primera ciudad

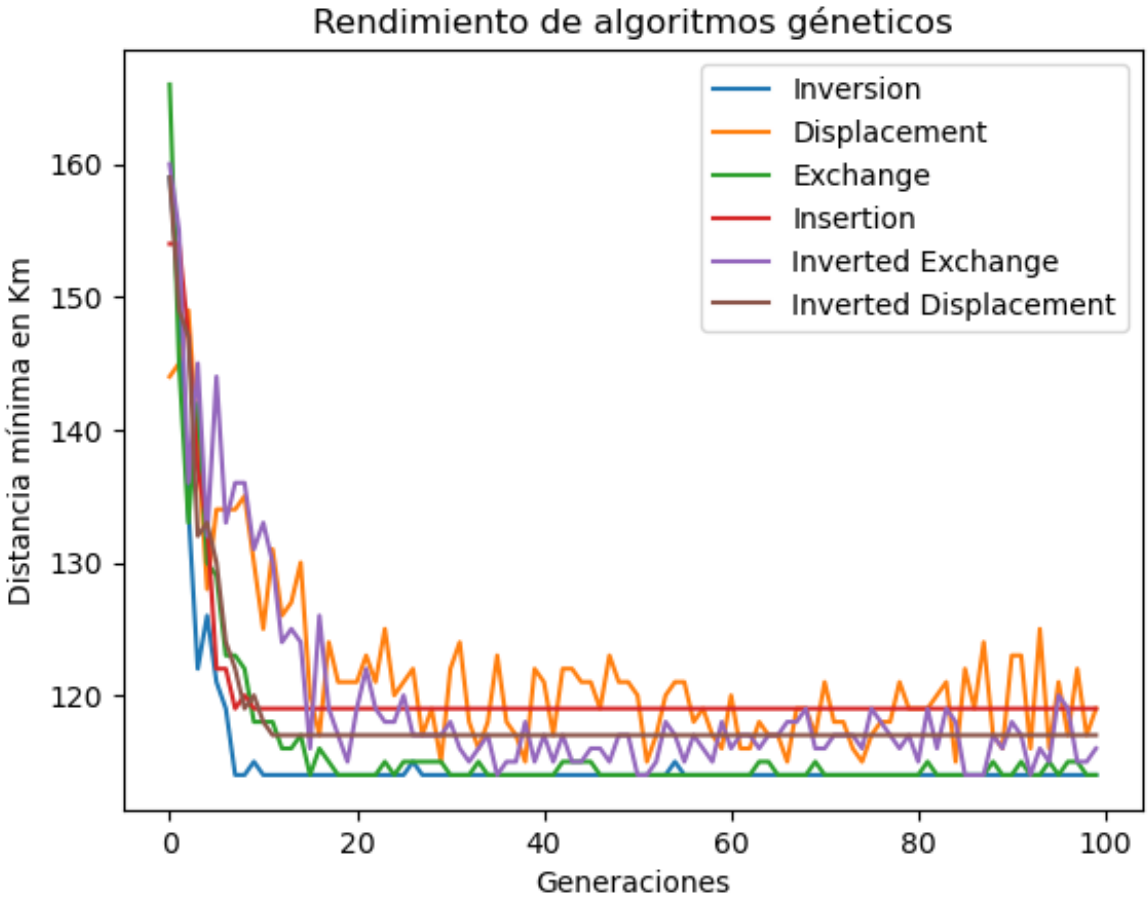


Figura 3.1: Resultados de los diferentes operadores de mutuación

Se puede observar que algunos algoritmos son inestables en los resultados. Esto se debe a que los algoritmos más inestables cambian más aleles por mutación, y puede que perderse progreso. Se puede ver que el de inversión es el que más rápido alcanza la solución y no hay tanta inestabilidad. Ya que solo se modifica dos aleles por mutación. La solución más corta que se encontró fue: **114km**.

3.2. Resultados con regresar a la primera ciudad

Se puede observar más inestabilidad con algunos algoritmos y el resultado más bajo fue: **143km**

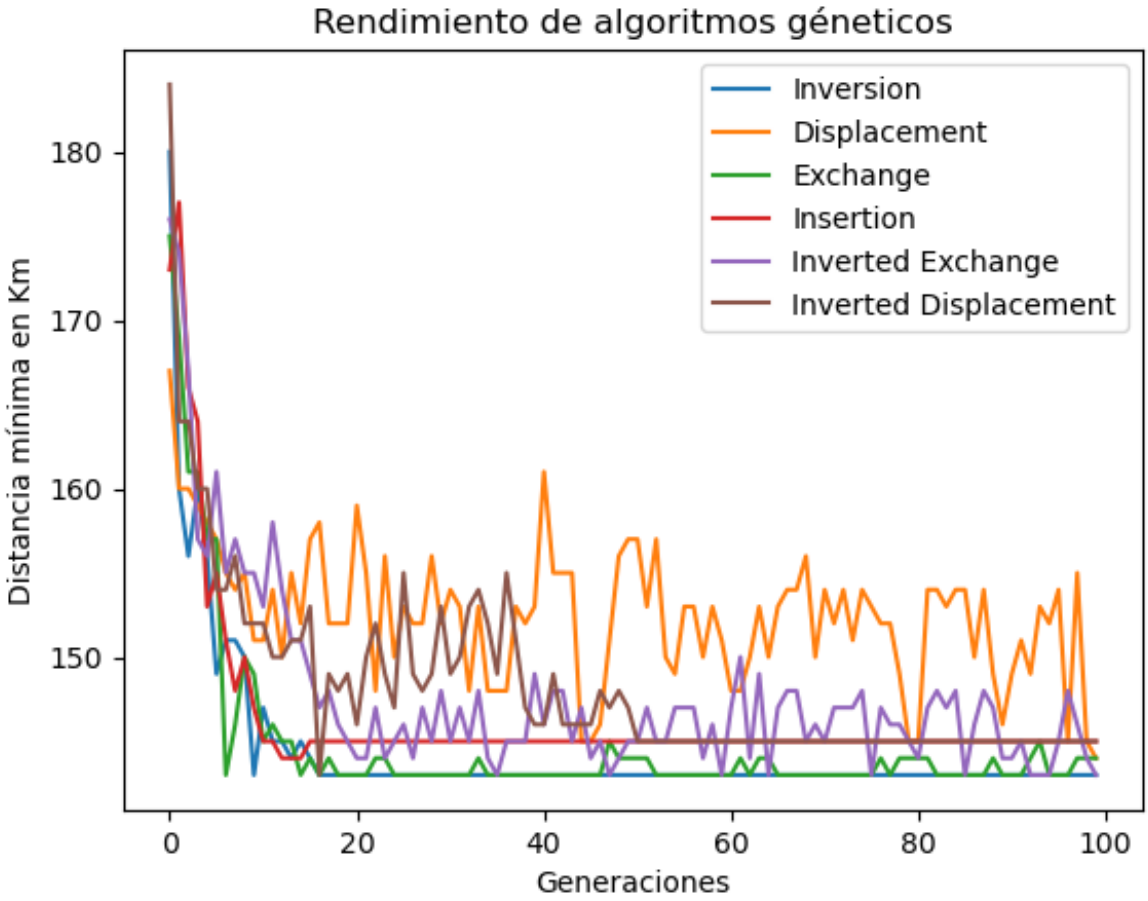


Figura 3.2: Resultados de los diferentes operadores de mutuación

Capítulo 4

Conclusión

Los algoritmos genéticos son una solución que puede ser eficiente para cualquier tipo de problema NP de combinatoria, ya que solo es cuestión de tener una lista de elementos no repetidos y adaptar la función de evaluación para el problema especifico. Ya dependerá la cantidad de generaciones ingresadas para determinar si la solución puede ser la más eficiente, lo cuál es una característica de este problema NP, ya que es posible que exista una solución más óptima, pero por complejidad algoritmíca, es posible que no se conozca hasta un número grande de generaciones.