MEMORIA LAB04

Grupo 05: Julián Prieto Velasco, Javier Muñoz Rojas, Guillermo Ramírez Cárdenas, Pedro José Paniagua Falo

Contenido

Tabla de Ilustraciones

Tabla de Ecuaciones

1. Introducción

La presente práctica tiene como objetivo diseñar e implementar un algoritmo genético para resolver el problema del transporte público en una ciudad. En particular, se trata de optimizar la ubicación de estaciones de metro para reducir el tiempo de viaje y mejorar la movilidad de los habitantes de la ciudad.

Para ello, se utiliza la técnica de algoritmos genéticos, que es una técnica de optimización que se basa en la evolución biológica. El algoritmo parte de una población inicial de soluciones aleatorias, y a través de un proceso de selección, cruce y mutación, se va mejorando la calidad de las soluciones hasta alcanzar una solución óptima o satisfactoria.

En esta práctica se presenta la implementación de un algoritmo genético para el problema de optimización de ubicación de estaciones de metro en una ciudad, considerando aspectos como el número de habitantes de cada municipio, la distancia entre ellos y la cantidad de estaciones disponibles para ubicar. Además, se analiza el impacto de los diferentes parámetros del algoritmo en la calidad de las soluciones obtenidas, y se presentan los resultados y conclusiones obtenidos a partir de la experimentación realizada.

2. Objetivo

El objetivo de esta práctica es diseñar y desarrollar un algoritmo genético capaz de optimizar la ubicación de estaciones de metro en una ciudad, con el fin de reducir el tiempo de viaje y mejorar la movilidad de los habitantes. Además, se busca analizar el impacto de los parámetros del algoritmo en la calidad de las soluciones obtenidas y presentar los resultados y conclusiones obtenidos a partir de la experimentación realizada.

3. Implementación

Para llevar a cabo la implementación de la práctica se ha hecho uso del lenguaje de programación de Python y se ha hecho uso de las librerías: pandas, random, numpy, matplotlib.pyplot, prettytable y copy.

3.1. Bucle del Algoritmo Genético

Un algoritmo genético es un algoritmo matemático sistemático de resolución/optimización de problemas mediante búsqueda, altamente paralelo, dónde dado un conjunto(población), de soluciones(individuos) a un problema, codificadas en forma de secuencia (cromosoma), asociadas a cierta función de coste (fitness), compite para ver cuál satisface mejor el problema. [1]

Para llevar a cabo la solución, se ha utilizado el bucle general de algoritmo genético. El bucle de un algoritmo genético es el proceso iterativo que se utiliza para evolucionar una población de soluciones candidatas en busca de una solución óptima para un problema determinado. Este proceso se lleva a cabo en varias fases, que incluyen:

1. Inicialización: se crea una población inicial de soluciones candidatas de forma aleatoria o utilizando algún método heurístico. Cada solución candidata se representa como un cromosoma, que consiste en un conjunto de genes que codifican la información necesaria para generar la solución.
2. Evaluación: se evalúa la aptitud de cada solución candidata en función del objetivo del problema. La aptitud se puede calcular utilizando una función de evaluación que mide qué tan buena es cada solución candidata.

3. Selección: se seleccionan los cromosomas más aptos para reproducirse y crear nuevas soluciones candidatas. Las técnicas de selección más comunes son la selección por ruleta, el torneo y la selección elitista.
4. Reproducción: se combinan los cromosomas seleccionados para crear nuevos cromosomas, que se convertirán en las nuevas soluciones candidatas. Las técnicas de reproducción más comunes son la recombinación y la mutación.
5. Reemplazo: se reemplazan las soluciones candidatas menos aptas de la población por las nuevas soluciones generadas en el paso anterior. Esto permite que la población evolucione hacia soluciones más aptas.
6. Terminación: se repiten los pasos 2 a 5 hasta que se alcanza un criterio de terminación, como un número máximo de iteraciones, un nivel de aptitud mínimo o una cantidad de tiempo límite.

Como se ilustra en el diagrama de flujo.

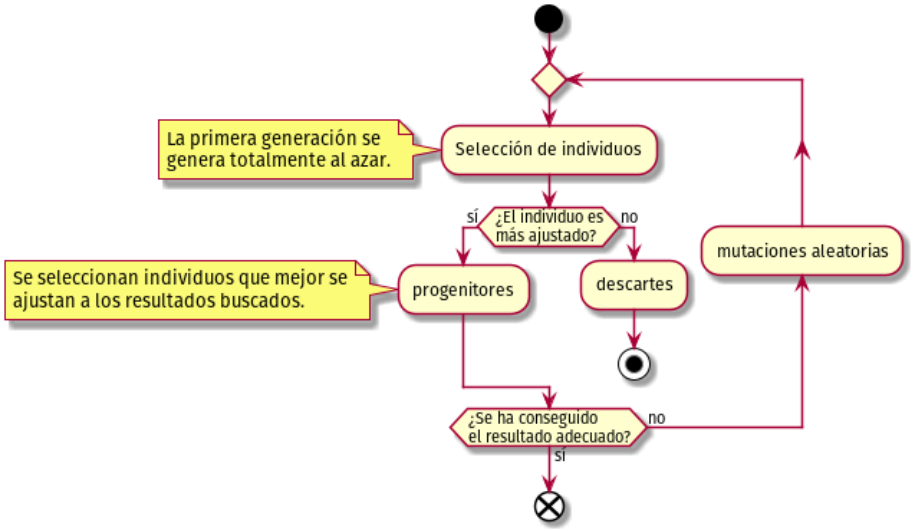

Ilustración 1 Diagrama de Flujo del Algoritmo Genético [2]

La codificación en Python de este algoritmo es la siguiente:

Algoritmo Genético

```python
# Definir una tabla auxiliar para mostrar los padres seleccionados
parent_table = PrettyTable()
parent_table.field_names = ["Cromosoma", "Fi", "Psi", "Psai"]

# Definir valores de Pc y Pi a evaluar
Pc_values = [0.1, 0.5, 0.8]
Pi_values = [0.2, 0.6, 0.75]

# Ejecutar el algoritmo genético
# Realizar múltiples ejecuciones para los diferentes valores de Pc y Pi
for PC in Pc_values:
    for PI in Pi_values:
        #PARAMETROS
        print("Parámetros utilizados:")
        print("L =", L)
        print("E =", E)
        print("NPOB =", NPOB)
        print("NGEN =", NGEN)
        print("Pc =", PC)
        print("Pi =", PI)
        # Definir la población inicial y los valores de fitness correspondientes
        population = initial_population(NPOB, E, L)
        tuplaPoblacion = list(df_poblacion.to_records(index=False))
        tuplaMunicipios = list(df_municipios.to_records(index=False))

        fitness_values = [fitness(individual, tuplaPoblacion, tuplaMunicipios, 0.5) for individual in population]
```

```
# Definir listas para el seguimiento de la aptitud media y máxima en cada generación
mean_fitness = []
max_fitness = []
best_fitness = 0
best_solution = None
consf_list = []
bestf_list = []
bestn_list = []
bestp_list = []

# Definir variables auxiliares para el seguimiento del proceso
aux_table = []
generation = 1
best_fitness = 0
best_solution = None
generation = 0
#Contadores
contador_muestra = 0
muestra_actual = 0
contador_resumen = 0
mejor_individuo, mejor_fitness, n_mejores, porcentaje_mejores = mejor_cromosoma(population, tuplaPoblacion, tuplaMunicipios, ALPHA)
consf, cromosoma_consenso = calcular_cromosoma_consenso(poblacion_total_fitness, poblacion_total)


# Actualización del seguimiento de los mejores resultados
if mejor_fitness > best_fitness:
        best_fitness = mejor_fitness
        best_solution = mejor_individuo

bestf_list.append(best_fitness)
bestn_list.append(n_mejores)
bestp_list.append(porcentaje_mejores)
consf_list.append(consf)

while generation <= NGEN:
    poblacion_hija20, poblacion_hija50, poblacion_hija30, PSArestantes = seleccion_elitista(population, P_elite, L, PC, PI)
    # juntamos las elites
    poblacion_elite_aux = []
    fitness_ELITE_aux = []
    poblacion_nueva = []
    poblacion_total = []
    poblacion_total_fitness = []
    population = poblacion_nueva
    for poblacion_aux, fitness_elite in poblacion_hija20:
        poblacion_elite_aux.append(poblacion_aux)
        poblacion_nueva.append(poblacion_aux)
        fitness_ELITE_aux.append(fitness_elite)
        poblacion_total.append(poblacion_aux)
        poblacion_total_fitness.append(fitness_elite)

    for poblacion_aux, fitness_elite in poblacion_hija50:
        poblacion_nueva.append(poblacion_aux)
        poblacion_total.append(poblacion_aux)
        poblacion_total_fitness.append(fitness_elite)


    for poblacion_aux, fitness_elite in poblacion_hija30:
        poblacion_nueva.append(poblacion_aux)
        poblacion_total.append(poblacion_aux)
        poblacion_total_fitness.append(fitness_elite)

    # Actualización del seguimiento del proceso
    mean_fitness.append(sum(fitness_ELITE_aux) / len(fitness_ELITE_aux))
    max_fitness.append(max(fitness_ELITE_aux))

      suma = sum(fitness_ELITE_aux)
      ps = [j/suma for j in fitness_ELITE_aux]
      psa = [sum(ps[:i+1]) for i in range(len(poblacion_hija20))]

      # Imprimir información de la tabla cada Nres generaciones

      print("------------------------------------------------------------")
      print(f"Tabla de selección de padres para la generación {generation}:")
      generation += 1
      # Agregar cada fila que contenga los elementos de los arreglos

      for i in range(len(poblacion_elite_aux)):
          # Actualización de la tabla auxiliar
          parent_table.add_row([poblacion_elite_aux[i], fitness_ELITE_aux[i], ps[i], psa[i]])
      print(parent_table)

      # Reiniciar la tabla auxiliar
      parent_table = PrettyTable()
      parent_table.field_names = ["Cromosoma", "Fi", "Psi", "Psai"]
      population = poblacion_nueva

      #mejor_individuo, mejor_fitness, n_mejores, porcentaje_mejores = mejor_cromosoma(population, tuplaPoblacion, tuplaMunicipios, ALPHA)
      consf, cromosoma_consenso = calcular_cromosoma_consenso(poblacion_total_fitness, poblacion_total)
      # Reiniciar la tabla auxiliar
      parent_table = PrettyTable()
      parent_table.field_names = ["Cromosoma", "Fi", "Psi", "Psai"]

      # Actualización del seguimiento de los mejores resultados
      if mejor_fitness > best_fitness:
          best_fitness = mejor_fitness
          best_solution = mejor_individuo

      bestf_list.append(best_fitness)
      bestn_list.append(n_mejores)
      bestp_list.append(porcentaje_mejores)
      consf_list.append(consf)
```

```
        if generation % NRES == 0:
            # Imprimir resumen
            print("RESUMEN")
            print("Generación", generation)
            print("Mejor cromosoma:", mejor_individuo)
            print("Fitness:", mejor_fitness)
            print("Número de veces que aparece en la población:", n_mejores)
            print("Porcentaje de la población que representa:", porcentaje_mejores)
            print("Cromosoma consenso:", cromosoma_consenso)
            print("Fitness:", consf)
            print("")
            contador_resumen += 1

        if generation % NSAMPLE == 0:
            # Imprimir muestreo
            print("MUESTREO")
            print("Generación", generation)
            muestra = random.sample(population, int(NPOB*0.2))
            for individuo in muestra:
                print(individuo)
            print("")
            muestra_actual += 1
    # Gráfico de evolución de la aptitud media y máxima
    plt.plot(mean_fitness, label='Aptitud media')
    plt.plot(max_fitness, label='Aptitud máxima')
    plt.xlabel('Generaciones')
    plt.ylabel('Aptitud')
    plt.legend()
    plt.show()

    # Gráfico de evolución de los mejores resultados
    plt.plot(bestf_list, label='Mejor aptitud')
    plt.plot(consf_list, label='Aptitud del consenso')
    plt.xlabel('Generaciones')
    plt.ylabel('Aptitud')
    plt.legend()
    plt.show()

    # Gráfico de evolución del porcentaje de la población que representa el mejor cromosoma
    plt.plot(bestp_list)
    plt.xlabel('Generaciones')
    plt.ylabel('% de la población')
    plt.show()

    # Gráfico de evolución del número de veces que aparece el mejor cromosoma en la población
    plt.plot(bestn_list)
    plt.xlabel('Generaciones')
    plt.ylabel('Número de veces que aparece')
    plt.show()
```

Ilustración 2 Codificación en Python del Algoritmo genético [2]

En este bloque de código se presenta el algoritmo genético codificado en Python. Además, se ha añadido la representación de gráficas y el cálculo del mejor individuo por población con su función de coste correspondiente.

3.2. Lectura de archivos xlsx

Una vez importadas las librerías el siguiente paso de la solución es leer los archivos .xlsx en el que está la información para realizar la práctica.

```
# Lee el archivo .xlsx y almacénalo en un dataframe de pandas
df_municipios = pd.read_excel('Datos_TSPincompleto.xlsx', skiprows= 8)
df_municipios = df_municipios.reset_index(drop=True)
df_municipios.index += 1
df_municipios.fillna(0, inplace=True)
df_municipios = df_municipios.drop("Municipio", axis=1)
df_municipios
```

Ilustración 3 Código de Lectura de la primera página del xlsx. [2]

Como se muestra en la Ilustración 1, se ha leído la primera página del xlsx. Una vez leída, se han eliminado las columnas irrelevantes y se ha modificado el índice para solo coger las distancias entre municipios.

```
# Lee el archivo .xlsx y almacénalo en un dataframe de pandas
# Carga el archivo Excel
df_poblacion = pd.read_excel('Datos_TSPincompleto.xlsx', skiprows= 4, sheet_name=1)
df_poblacion = df_poblacion.reset_index(drop=True)
df_poblacion.index += 1
df_poblacion.fillna(0, inplace=True)
df_poblacion = df_poblacion.rename(columns={"Unnamed: 0": "Municipio"})
df_poblacion
```

Ilustración 4 Código para leer la segunda página del archivo xlsx [2]

Como se observa en la Ilustración 2, el siguiente paso es leer la segunda hoja del archivo xlsx. Se han realizado los mismos pasos que en la primera imagen del archivo.

3.3. Inicialización de parámetros

Parámetros

+ Código + Markdown

```
# Asociar cada municipio con un número del 1 al L
municipios = df_municipios.index + 1
L = len(municipios)  # Número total de municipios
E = 7  # Número de estaciones de metro
cromosoma = np.random.randint(1, L+1, size=E)
NPOB = 10       # Número de individuos de la población
NGEN = 100      # Número de generaciones
Pc = 0.8       # Probabilidad de cambiar-estación, aplicable a cada gen
Pi = 0.2        # Probabilidad de intercambiar-estación, aplicable a cada gen
NRES = 10       # Cada qué número de generaciones se saca un resumen de la evolución del proceso
NSAMPLE = 10    # Cada qué número de generaciones se saca un muestreo de la población
L = len(df_municipios)  # Número de lugares potenciales para construir estaciones
ALPHA = 0.5  # Valor de alpha para el cálculo de aptitud
P_elite = 0.2
```

Ilustración 5 Código de la creación de parámetros [2]

El siguiente paso ha sido crear los parámetros del algoritmo genético que se deben añadir por teclado. En este caso, los datos no son los finales pues la captura de código es anterior a los últimos experimentos.

Se quiere destacar el tamaño del cromosoma que es igual al número de estaciones asignado. Para la creación del cromosoma se generan números aleatorios entre 1 y el número de municipios que hay en el *dataset*.

3.4. Generación de la población

En este apartado de la práctica se genera la población inicial que será usada para el algoritmo genético. [1]

En esta, los parámetros son: *nPob*, números de individuos de la población, E, número de estaciones, L, número de estaciones que pueden ser solución. Al comienzo de la función se ha inicializado un *array* llamado *population*, en el que se guardará la primera población generada. Para generarla, se usan tres bucles anidados en los que se crean individuos aleatorios, se guardan en el *array* de población y se guardan los genes que ya están en el cromosoma creado para evitar la repetición de genes.

Generación de Población

```python
def initial_population(nPob, E, L):
    population = []
    for i in range(nPob):
        individual = []
        used_stations = []
        for j in range(E):
            station = random.randint(1, L)
            while station in used_stations:
                station = random.randint(1, L)
            individual.append(station)
            used_stations.append(station)
        population.append(individual)
    return population
```

Ilustración 6 Código de Generación de Población Inicial [2]

3.5. Función de coste

Función que informa de cuan bueno es un individuo dado en la solución de un problema (evalúa el fenotipo). Diseñada para cada problema de manera específica. Debe reflejar el valor de un individuo de manera real. [1]

Durante la evaluación, se decodifica el cromosoma convirtiéndose en una serie de parámetros del problema. Se halla la solución del problema a partir de esos parámetros, se le da una puntuación a esa solución mediante la función de fitness.

En este problema, el objetivo es seleccionar E estaciones de entre L lugares preseleccionados, de tal manera que se maximice el número de usuarios diarios potenciales servidos y se minimice el recorrido total de la línea. Para ello se ha hecho uso de la siguiente función de coste:

$$Fi = 1 / (1 + Ti)$$

Ecuación 1 Ecuación del Fitness [2]

Donde Ti es el tiempo total de viaje para todos los usuarios de la red de transporte con una determinada ubicación de estaciones de metro. Esta fórmula se utiliza para maximizar el fitness, ya que cuanto menor sea el valor de Ti, mayor será el valor de Fi.

La función de coste consta de dos términos: el primer término se refiere a la demanda potencial de cada estación seleccionada en la solución candidata, ponderada por un factor de importancia "alfa". El segundo término se refiere a la suma de las distancias recorridas entre las estaciones seleccionadas en la solución candidata, ponderada por el complemento de "alfa".

En otras palabras, se busca una solución que permita satisfacer la mayor cantidad posible de demanda potencial de los usuarios con la menor cantidad de distancia recorrida entre las estaciones seleccionadas.

Por lo tanto, la función de coste "fitness" se utiliza para evaluar la aptitud de cada solución candidata en función de su capacidad para satisfacer estos dos objetivos, permitiendo así que el algoritmo genético pueda generar soluciones óptimas a este problema de optimización.

Funcion Fitness

```python
def fitness(cromosoma, poblacion, distancias, alfa):
    """
    Calcula el valor de fitness de un cromosoma según la fórmula dada.

    :param cromosoma: Lista que representa el cromosoma a evaluar.
    :param poblacion: Lista de tuplas que contienen el número potencial de usuarios diarios de cada estación.
    :param distancias: Matriz que contiene las distancias entre cada par de estaciones.
    :param alfa: Valor que permite dar más o menos importancia a cada criterio.
    :return: El valor de fitness del cromosoma.
    """
    f1 = alfa * sum([poblacion[estacion-1][1] for estacion in cromosoma])
    f2 = (1 - alfa) * sum([distancias[cromosoma[i]-2][cromosoma[i+1]-1] for i in range(len(cromosoma)-1)]) + distancias[cromosoma[-1]-2][cromosoma[0]-1]
    return f1 - f2
```

Ilustración 7 Código de la Función de Coste [2]

3.6. Operadores del Algoritmo Genético

En el siguiente apartado, se muestran los operadores del algoritmo genético programados para su correcto uso. Los operadores de un algoritmo genético (AG) son las técnicas utilizadas para crear nuevas soluciones candidatas a partir de las soluciones existentes en la población.

3.6.1. Operador mutación

Este operador es el encargado de elegir qué cromosoma debe mutar o no, si se cumple la condición de mutación, llamará a la función correspondiente, intercambio o cambio. [1]

Se utiliza para cambiar aleatoriamente algunos genes de un cromosoma seleccionado para crear una nueva solución candidata. Los operadores de mutación más comunes son la mutación por intercambio y la mutación por inversión.

Funcion de mutacion

```python
def mutacion(cromosoma, PC, PI):
    """
    Mutación por cambio de estación o intercambio de estación.

    :param cromosoma: Lista que representa el cromosoma actual.
    :param tasa_mutacion: Probabilidad de que un gen mute.
    :param l: Número total de estaciones.
    :return: El cromosoma mutado.
    """
    for i in range(len(cromosoma)-1):
        if random.random() < PC:
                cromosoma = cambiar_estacion(cromosoma, i)
        elif random.random() < PI:
                cromosoma = intercambiar_estacion(cromosoma, i)
    return cromosoma
```

Python

Ilustración 8 Código de la función de mutación [2]

3.6.1.1. Operador Cambio de estación.

Este operador se encarga de la mutación del cromosoma gen a gen. Esta mutación se realiza a partir de un cambio de uno de los genes del cromosoma por un número aleatorio entre 1 y el número de lugares potenciales para construir estaciones.

Para el correcto uso de este operador y evitar genes repetidos en el cromosoma se ha realizado un bucle en el que se comprueba que el gen a cambiar es diferente a los que el cromosoma posee.

Cambio de estación

```python
def cambiar_estacion(cromosoma, i):
    """
    Cambia una estación del cromosoma por otra al azar que no esté en la solución.

    :param cromosoma: Lista que representa el cromosoma actual.
    :return: El cromosoma modificado.
    """
    rand = 0

    # Ejecutar al menos una vez
    while True:
        # Hacer algo
        rand = random.randint(1, L)

        # Verificar condición de salida
        if rand not in cromosoma:
            break

    cromosoma[i] = rand

    return cromosoma
```

Ilustración 9 Código del operador de cambio de estación. [2]

3.6.1.2. Operador intercambio de estación

Este operador se encarga de la mutación del cromosoma gen a gen. Esta mutación se realiza a partir de un intercambio de uno de los genes del cromosoma por otro. [1]

Para el correcto uso de este operador y evitar genes repetidos en el cromosoma se ha realizado un bucle en el que se comprueba que el gen a intercambiar es diferente a los que el cromosoma posee.

Intercambio de estacion

```python
def intercambiar_estacion(cromosoma, i):
    """
    Intercambia dos estaciones dentro del recorrido.

    :param cromosoma: Lista que representa el cromosoma actual.
    :return: El cromosoma modificado.
    """
    j = random.randint(0, len(cromosoma) - 1)
    while j == i:
        j = random.randint(0, len(cromosoma) - 1)

    cromosoma[i], cromosoma[j] = cromosoma[j], cromosoma[i]
    return cromosoma
```

Ilustración 10 Código del operador intercambio de estación. [2]

3.6.2. Operador Selección

Mecanismo que favorece la replicación de los individuos con mejor valoración, pero que no impide la diversidad [1]. Se compone de dos partes:

- Determinación de que individuos (cromosomas) de la población serán utilizados para la reproducción (padres).
- Determinación de los individuos de la población que serán sustituidos por los recién generados

3.6.2.1. Selección de los padres.

Para llevar a cabo la selección de los padres se ha implementado la selección por ruleta. En esta se utiliza una ruleta para seleccionar a los padres de la siguiente generación. La selección se realiza en función del valor de fitness de cada cromosoma.

Se ha codificado como se muestra en la ilustración:

Seleccion de ruleta

```python
def selection_ruleta(population, fitness_values):
    fitness_sum = sum(fitness_values)
    fitness_probabilities = [fitness / fitness_sum for fitness in fitness_values]
    parent1_index = np.random.choice(len(population), p=fitness_probabilities)
    parent2_index = np.random.choice(len(population), p=fitness_probabilities)
    parent1 = population[parent1_index]
    parent2 = population[parent2_index]

    return parent1, parent2
```

Ilustración 11 Código de Selección por ruleta. [2]

El parámetro population representa la población actual, es decir, una lista de cromosomas. El parámetro fitness_values es una lista de valores de fitness correspondientes a cada cromosoma de la población.

Primero, se calcula la suma total de todos los valores de fitness de la población. Luego, se calcula la probabilidad de selección de cada cromosoma en función de su valor de fitness relativo al total. La probabilidad de selección de un cromosoma es su valor de fitness dividido por la suma total de todos los valores de fitness de la población.

A continuación, se seleccionan dos padres utilizando la distribución de probabilidad definida por las probabilidades de selección calculadas. Se utiliza la función np.random.choice de NumPy para elegir de manera aleatoria a los padres de la población actual.

Finalmente, la función devuelve a los padres seleccionados, que serán utilizados para generar la siguiente generación de cromosomas.

En resumen, esta función implementa el método de selección por ruleta, en el cual los padres son seleccionados de acuerdo con su valor de fitness relativo a los demás cromosomas de la población.

3.6.2.2. Eliminación de individuos

Para la eliminación de individuos se ha llevado a cabo una selección elitista en la que se divide la población en tres grupos, el 20% de los mejores individuos, el 50% mejor del 80% restante y el 30% mejor restante.

En la selección elitista: Se copian sin modificar los mejores cromosomas de la generación

anterior en la nueva población. El resto se obtienen por selección por ruleta. Puede mejorar el funcionamiento de los AG al evitar que se pierda la mejor solución. A continuación, se muestra la codificación:

```python
def seleccion_elitista(poblacion, porcentaje_elite, l, PC, PI):
    tuplaPoblacion = list(df_poblacion.to_records(index=False))
    tuplaMunicipios = list(df_municipios.to_records(index=False))
    poblacion_hija20 = []
    poblacion_hija50 = []
    poblacion_restante80 = []
    poblacion30 = []
    PSArestantes = []
    poblacion_hija20, poblacion_restante80 = seleccion20(poblacion, porcentaje_elite, tuplaPoblacion, tuplaMunicipios, ALPHA)
    poblacion_hija20_final = copy.deepcopy(poblacion_hija20)
    poblacion_hija50 = seleccion50(poblacion, PC, PI, 0.5)
    PSArestantes, poblacion_hija30 = seleccion30(poblacion_restante80,PC, PI, 0.4)

    return poblacion_hija20_final, poblacion_hija50, poblacion_hija30, PSArestantes
```

Seleccionar el 20%

```python
def seleccion20(poblacion, porcentaje_elite, poblacion_original, distancias, alfa):
    # Ordenar la población por fitness (en orden descendente)
    poblacion_ordenada = sorted(poblacion, key=lambda x: fitness(x, poblacion_original, distancias, alfa), reverse=True)
    # Calcular el número de individuos que deben ser movidos a la población hija
    n_elite = int(len(poblacion_ordenada) * porcentaje_elite)

    # Mover los individuos a la población hija y calcular su Ps y Psa
    poblacion_hija = poblacion_ordenada[:n_elite]
    suma_fitness_poblacion_hija = sum([fitness(x, poblacion_original, distancias, alfa) for x in poblacion_hija])

    ps_poblacion_hija = [fitness(x, poblacion_original, distancias, alfa)/suma_fitness_poblacion_hija for x in poblacion_hija]
    psa_poblacion_hija = np.cumsum(ps_poblacion_hija)

    # Guardar la población restante
    poblacion_restante = poblacion_ordenada[n_elite:]

    fitness_values = [fitness(individual, poblacion_original, distancias, alfa) for individual in poblacion_hija]

    aux = []
    for hijo, fitness_aux in zip(poblacion_hija, fitness_values):
        aux.append((hijo, fitness_aux))
    poblacion_hija = aux

    return poblacion_hija, poblacion_restante
```

Seleccionar el 50%

```python
def seleccion50(poblacion, PC, PI, porcentaje):
    tuplaPoblacion = list(df_poblacion.to_records(index=False))
    tuplaMunicipios = list(df_municipios.to_records(index=False))

    tam_poblacion_hija = int(len(poblacion) * porcentaje)
    poblacion_hija = []
    suma_fitness_poblacion_padre = sum([x[1] for x in poblacion])
    ps_poblacion_padre = [x[1] / suma_fitness_poblacion_padre for x in poblacion]
    psa_poblacion_padre = np.cumsum(ps_poblacion_padre)
    ps_poblacion_padre = [x[1] / suma_fitness_poblacion_padre for x in poblacion]
    psa_poblacion_padre = np.cumsum(ps_poblacion_padre)
    for i in range(tam_poblacion_hija):
        # Seleccionar un cromosoma aplicando el método de la ruleta sobre la Probabilidad de Selección Acumulada (Psa) de los padres
        aux = copy.deepcopy(poblacion)
        padre_seleccionado = poblacion[np.searchsorted(psa_poblacion_padre, random.random())]
        # Construir el hijo decidiendo si cada gen se copia fielmente, se cambia-estación o se intercambia-estación usando el método de la ruleta de nuevo sobre Pc, Pi y Pnada
        hijo = mutacion(padre_seleccionado, PC, PI)
        # Calcular el fitness del nuevo individuo
        hijo_fitness = fitness(hijo, tuplaPoblacion, tuplaMunicipios, ALPHA)
        # Copiarlo en la generación hija
        poblacion_hija.append((hijo, hijo_fitness))
        poblacion = aux

    return poblacion_hija
```

Calcular PS y PSA restantes

```python
def seleccion30(poblacion_restante, PC, PI, porcentaje):
    poblacion30 = []
    # Ordenar la población por fitness (en orden descendente)
    poblacion_ordenada = sorted(poblacion_restante, key=lambda x: x[1], reverse=True)

    # Calcular el número de individuos que deben ser seleccionados para la población hija
    n_seleccionados = int(len(poblacion_ordenada) * 0.3)

    # Calcular la suma total de fitness de los cromosomas restantes
    suma_fitness_restantes = sum(cromosoma[1] for cromosoma in poblacion_ordenada)

    # Calcular Ps y Psa de los cromosomas restantes
    ps_psa_restantes = []
    ps_acum = 0
    for i in range(len(poblacion_ordenada)):
        if i < n_seleccionados:
            ps = poblacion_ordenada[i][1] / suma_fitness_restantes
            ps_acum += ps
            ps_psa_restantes.append((poblacion_ordenada[i], ps, ps_acum))
        else:
            break

    poblacion30 = seleccion50(poblacion_restante, PC, PI, porcentaje)
    return ps_psa_restantes, poblacion30
```

Ilustración 12 Código del Operador Selección para la eliminación de individuos. [2]

La función llamada seleccion_elitista que realiza la selección de individuos para la siguiente generación en un algoritmo genético. El algoritmo selecciona una porción de la población original, llamada élite, que se transfiere directamente a la siguiente generación, y el resto de la población se selecciona por medio de un proceso de selección proporcional al fitness, aplicando tres estrategias diferentes.

La primera estrategia selecciona un 20% de la población original de acuerdo con el fitness, y esos individuos se transfieren directamente a la siguiente generación. La segunda estrategia selecciona el 50% de los individuos restantes de la población original, aplicando la estrategia de ruleta mediante la cual se seleccionan individuos de acuerdo con su probabilidad de selección proporcional al fitness. Finalmente, la tercera estrategia selecciona el 30% restante de los individuos de la población original, también mediante la estrategia de ruleta, pero esta vez utilizando la probabilidad de selección proporcional al fitness de los individuos que no se incluyeron en la élite ni en la estrategia anterior.

Cada individuo seleccionado se somete a un proceso de mutación que consiste en elegir un gen al azar y reemplazarlo con otro gen del mismo cromosoma, elegido al azar. Los valores de probabilidad de cambio de estación (PC) y de intercambio de estación (PI) son parámetros que se utilizan para controlar la intensidad de la mutación.

El código utiliza funciones auxiliares para realizar los procesos de selección, mutación y fitness ya documentadas anteriormente.

3.6.3. Operador Entrecruzamiento

Consiste en unir en alguna forma los cromosomas de los padres que han sido previamente seleccionados de la generación anterior para formar dos descendientes [1]. Su codificación es la siguiente:

Funcion de Cruce

```python
def cruce(padre1, padre2):
    """
    Realiza el cruce entre dos cromosomas y devuelve el cromosoma hijo resultante.

    Parameters:
    padre1 (numpy.ndarray): Primer cromosoma padre.
    padre2 (numpy.ndarray): Segundo cromosoma padre.

    Returns:
    numpy.ndarray: Cromosoma hijo resultante del cruce.
    """
    cromosoma_hijo = np.zeros_like(padre1)
    punto_cruce = np.random.randint(1, len(padre1))
    cromosoma_hijo[:punto_cruce] = padre1[:punto_cruce]
    cromosoma_hijo[punto_cruce:] = padre2[punto_cruce:]
    return cromosoma_hijo
```

Ilustración 13 Código del operador de entrecruzamiento. [2]

Este código define una función llamada cruce que toma dos cromosomas de tipo numpy.ndarray como entrada y realiza el cruce genético entre ellos para generar un hijo. El proceso de cruce genético implica seleccionar una posición de cruce aleatoria dentro de los cromosomas y mezclar las partes correspondientes de los dos cromosomas para generar un nuevo cromosoma hijo.

El código comienza por crear un arreglo vacío llamado cromosoma_hijo del mismo tamaño que el cromosoma del primer padre. A continuación, se selecciona una posición de cruce aleatoria dentro del cromosoma utilizando la función np.random.randint. Luego, se copian los elementos del primer padre desde el inicio hasta el punto de cruce en el cromosoma hijo. A partir del punto de cruce, se copian los elementos del segundo padre hasta el final del cromosoma hijo. Finalmente, se devuelve el cromosoma hijo resultante.

Este código forma parte de una implementación de un algoritmo genético y se podría utilizar en combinación con otras funciones para crear una solución completa para un problema específico.

3.7. Selección del individuo consenso

Se entiende como individuo consenso aquel que en cada posición del cromosoma tiene la estación que más veces aparece en esa misma posición (técnicamente, el gen más frecuente).

Calcular cromosoma Cosenso

```python
def calcular_cromosoma_consenso_fitness(poblacion, alfa):
    tuplaPoblacion = list(df_poblacion.to_records(index=False))
    tuplaMunicipios = list(df_municipios.to_records(index=False))
    # Obtener el número de estaciones en la población
    num_estaciones = E
    # Crear un array para almacenar el cromosoma consenso
    consenso = [None] * num_estaciones
    # Iterar por cada posición del cromosoma
    for i in range(num_estaciones):
        # Obtener los valores de los genes en esa posición
        genes = [individuo[i] for individuo in poblacion]
        # Eliminar duplicados y ordenar los genes de forma descendente según su frecuencia
        genes_ordenados = sorted(set(genes), key=genes.count, reverse=True)
        # Asignar el gen más frecuente al cromosoma consenso
        consenso[i] = genes_ordenados[0]
        # Eliminar ese gen de la lista de genes para que no se repita en el resto del cromosoma
        genes_ordenados = genes_ordenados[1:]
        # Asignar los demás genes (si los hay) a las posiciones restantes del cromosoma
        for j in range(1, len(genes_ordenados)+1):
            if i+j < num_estaciones:
                consenso[i+j] = genes_ordenados[j-1]
            else:
                break
    # Calcular el valor de fitness del cromosoma consenso
    fitness_consenso = fitness(consenso, tuplaPoblacion, tuplaMunicipios, alfa)
    # Retornar el cromosoma consenso y su valor de fitness
    return consenso, fitness_consenso
```

0.0s Python Python

Ilustración 14 Código del cálculo del Cromosoma Consenso. [2]

La función "calcular_cromosoma_consenso_fitness" recibe dos parámetros: "poblacion", que es una lista de cromosomas (individuos) y "alfa", que es un valor de ajuste para el cálculo de fitness. Esta función se encarga de calcular el cromosoma consenso de la población y su valor de fitness.

Primero, se convierten los dataframes "df_poblacion" y "df_municipios" en listas de tuplas para poder manipular los datos más fácilmente. Luego, se obtiene el número de estaciones "num_estaciones" a partir de la variable global "E". Se crea un array "consenso" de longitud "num_estaciones" para almacenar el cromosoma consenso.

Se itera por cada posición del cromosoma y se obtienen los valores de los genes en esa posición para cada individuo de la población. Luego, se eliminan los duplicados y se ordenan los genes de forma descendente según su frecuencia. El gen más frecuente se asigna al cromosoma consenso y se elimina de la lista de genes para que no se repita en el resto del cromosoma. Los demás genes (si los hay) se asignan a las posiciones restantes del cromosoma, siempre y cuando no se sobrepase la longitud del cromosoma.

Una vez se ha construido el cromosoma consenso, se calcula su valor de fitness utilizando la función "fitness" y los datos de población y municipios convertidos previamente en tuplas. Finalmente, se retorna el cromosoma consenso y su valor de fitness.

Es importante mencionar que, en esta nueva versión de la función, se ha modificado el código para asegurarse de que los genes no se repitan en el cromosoma consenso.

4. Cuestiones

4.1. Cuestión 1

Responde a las siguientes cuestiones

4.1.1. Pregunta 1

¿Hay un valor máximo que puede alcanzar el fitness? ¿Se puede saber cuál es?

En teoría, no hay un límite superior para el valor del fitness, ya que este depende del problema específico que se está abordando. En algunos casos, el fitness podría estar limitado por la precisión numérica de la computadora, pero en general, se puede esperar que el fitness siga aumentando a medida que se generan soluciones óptimas.

Sin embargo, en la práctica, el valor máximo del fitness que se puede alcanzar está limitado por varios factores, como el tamaño de la población, la capacidad de cómputo, el tiempo de ejecución, la calidad de la codificación y el modelo matemático utilizado para evaluar el fitness.

4.1.2. Pregunta 2

¿Podría ser negativo el fitness máximo? ¿Por qué?

En teoría, el valor máximo del fitness no puede ser negativo, ya que el fitness se refiere a la medida de la calidad de una solución a un problema determinado y, por definición, no puede ser inferior a la peor solución posible. Sin embargo, en algunos casos, el valor de la función objetivo o de la medida de calidad puede ser negativo, lo que se reflejaría en un fitness negativo.

En el contexto del problema de optimización de ubicación de estaciones de metro en una ciudad, es poco probable que el fitness máximo sea negativo.

El objetivo del algoritmo genético en este caso es minimizar la función de costo, que está definida en términos de la distancia promedio entre las estaciones de metro y los habitantes de la ciudad. Como la distancia no puede ser negativa, la función de costo tampoco puede ser negativa.

Por lo tanto, el fitness, que es el inverso de la función de costo, también debería ser positivo. El valor máximo que puede alcanzar el fitness dependerá de los valores específicos de la función de costo y del tamaño y complejidad del problema, pero no debería ser negativo.

4.1.3. Pregunta 3

¿Puedes mejorar el cálculo del fitness? Explícalo y justifica tu elección final.

Sí, existen diferentes formas de mejorar el cálculo del fitness en un algoritmo genético. A continuación, se presentan algunas posibles mejoras y se justifica la elección final:

1. **Escalado de fitness:** esta técnica consiste en transformar los valores de fitness para que estén en un rango específico, lo que puede ayudar a evitar problemas de convergencia prematura y mejorar la diversidad de la población. Por ejemplo, se podría aplicar una función sigmoide para ajustar los valores de fitness a un rango entre 0 y 1.

2. **Penalización por restricciones:** en algunos casos, puede haber restricciones que limiten las soluciones válidas. En este caso, se podrían aplicar penalizaciones a las soluciones que no cumplan con las restricciones, reduciendo su fitness. Por ejemplo, si se establece un límite máximo de estaciones de metro permitidas en la ciudad, se podría aplicar una penalización a las soluciones que superen este límite.
3. **Combinación de múltiples objetivos:** en algunos problemas, puede haber varios objetivos que se deben optimizar al mismo tiempo. En este caso, se podría combinar los diferentes objetivos en una sola función de fitness, utilizando técnicas como la agregación ponderada o la técnica de Pareto.

En el caso específico de la práctica, se podría mejorar el cálculo del fitness mediante la aplicación de una penalización por restricciones. Por ejemplo, se podrían establecer restricciones en cuanto a la cantidad máxima de estaciones permitidas en una determinada área geográfica o la distancia máxima entre las estaciones. De esta forma, las soluciones que no cumplan con estas restricciones serían penalizadas y su fitness se reduciría en consecuencia. Además, se podría considerar la aplicación de un escalado de fitness para ajustar los valores de fitness a un rango específico, como el intervalo entre 0 y 1. Esto ayudaría a evitar problemas de convergencia prematura y mejorar la diversidad de la población.

En la práctica no se ha utilizado ninguna de las opciones que has presentado. En cambio, se ha utilizado una fórmula modificada de la opción 1:

$$\mathrm{Fi} = 1 / (1 + \mathrm{Ti})$$

Ecuación 2 Ecuación del Fitness [2]

Donde Ti es el tiempo total de viaje para todos los usuarios de la red de transporte con una determinada ubicación de estaciones de metro. Esta fórmula se utiliza para maximizar el fitness, ya que cuanto menor sea el valor de Ti, mayor será el valor de Fi.

La justificación de esta elección es que la fórmula se adapta bien al problema de optimización de la ubicación de estaciones de metro, ya que el objetivo es minimizar el tiempo de viaje para todos los usuarios de la red de transporte. Al utilizar esta fórmula, el algoritmo genético se centra en buscar soluciones que minimicen el tiempo total de viaje, lo que a su vez conduce a una mejora en la calidad del servicio de transporte público.

4.1.4. Pregunta 4

¿Qué alternativas al proceso de mutación podrías plantear? Explícalo y justifica tu elección final.

Existen varias alternativas al proceso de mutación en un algoritmo genético, algunas de las cuales son:

1. **Recombinación no uniforme:** en lugar de seleccionar un solo punto de corte para la recombinación, se seleccionan múltiples puntos de corte, lo que permite una mayor variedad en las combinaciones de los genes de los padres.
2. **Operadores de búsqueda local:** se pueden aplicar técnicas de búsqueda local para mejorar la calidad de las soluciones generadas por el algoritmo genético. Esto se puede hacer mediante la aplicación de algoritmos de optimización más específicos a las soluciones generadas por el algoritmo genético.
3. **Selección de operadores de mutación:** se pueden probar diferentes operadores de mutación para determinar cuál funciona mejor para el problema en cuestión. Por

ejemplo, se pueden utilizar diferentes técnicas de perturbación de los genes para generar nuevas soluciones.

En la práctica se utiliza la mutación uniforme, que es una técnica simple y eficaz para introducir diversidad en la población y evitar que el algoritmo converja prematuramente a una solución subóptima. Aunque existen alternativas más complejas, la mutación uniforme suele funcionar bien en muchos problemas y su simplicidad hace que sea fácil de implementar y de ajustar. Por lo tanto, es una buena elección para esta práctica.

4.2. Cuestión 2

Prueba con distintos valores de E. ¿Qué ocurre cuando es muy bajo? ¿Y con E cercano a L?

4.2.1. Introducción

Para responder a esta pregunta, se van a adjuntar distintos experimentos con mismos parámetros, pero con distinto parámetro "E". Los parámetros que se han utilizado son:

- NPOB = 100
- NGEN = 100
- Pc = 0.1
- Pi = 0.2

4.2.2. Resultados de los experimentos

4.2.2.1. Número de estaciones = 4

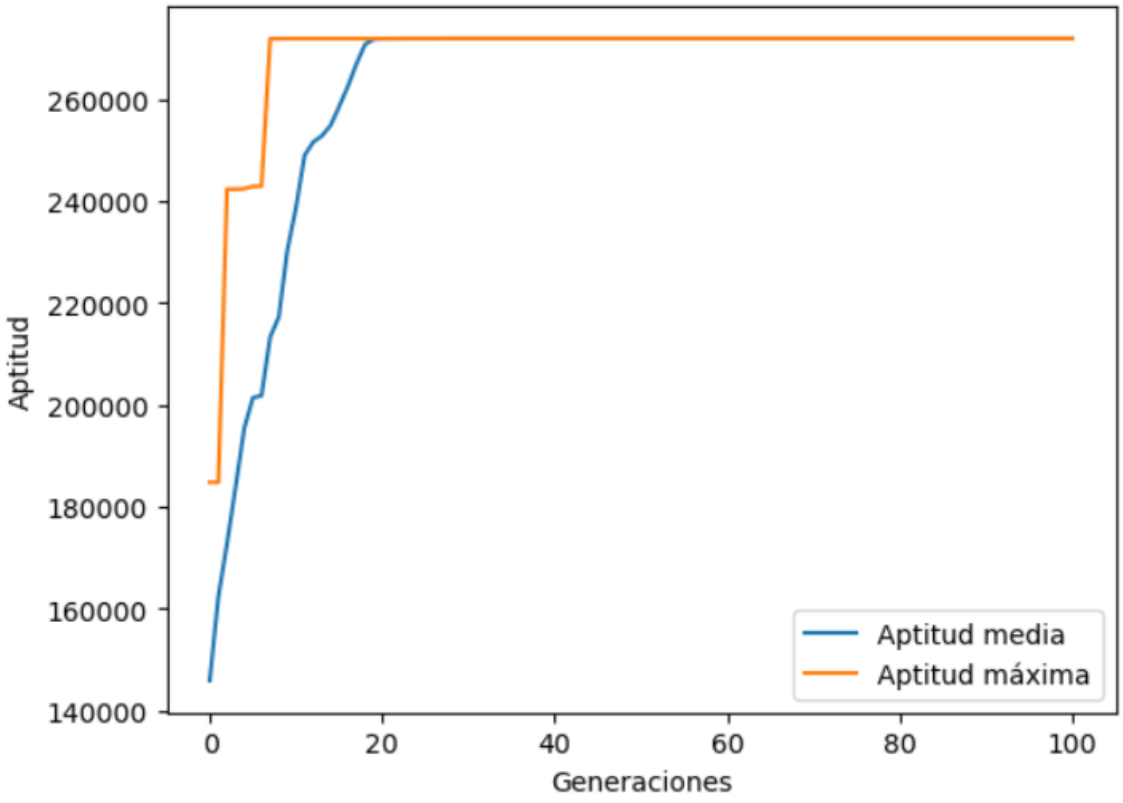

Ilustración 15 Gráfico de fitness medio y máximo para E = 4. [2]

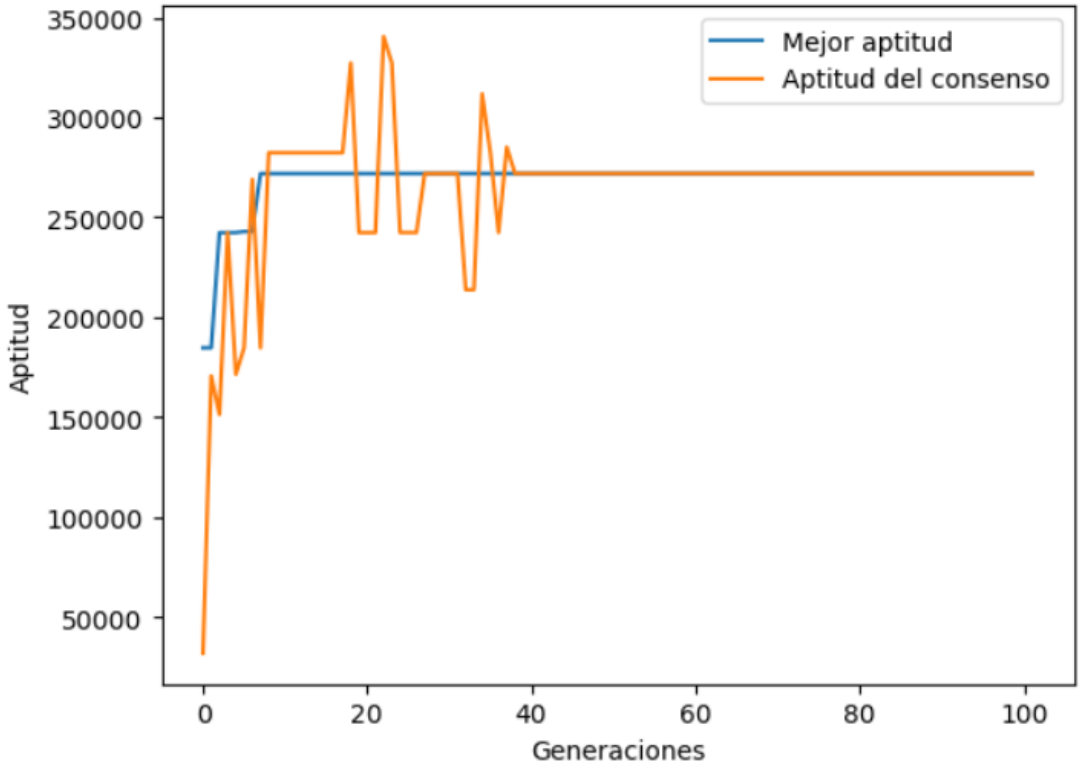

Ilustración 16 Gráfico de mejor fitness de la población VS Mejor fitness del individuo consenso para E = 4. [2]

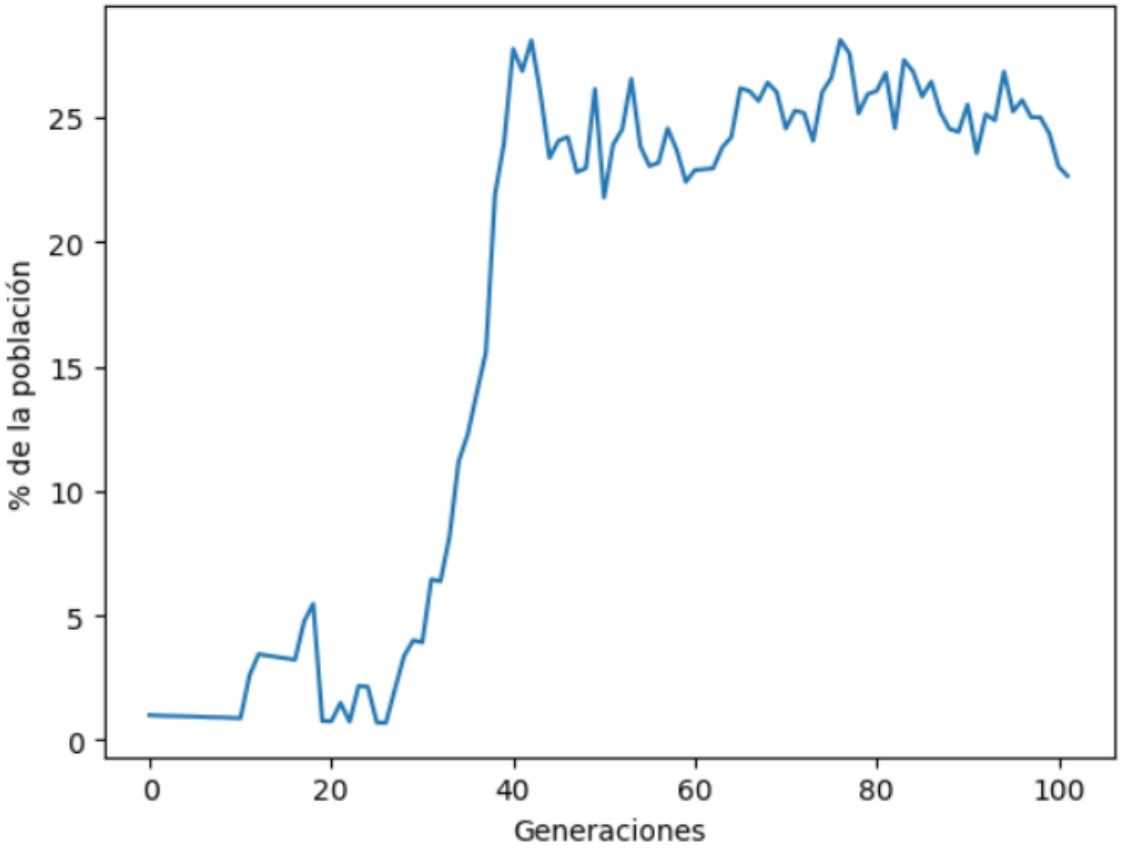

Ilustración 17 Gráfico de porcentaje de la población que representa el mejor cromosoma para E = 4. [2]

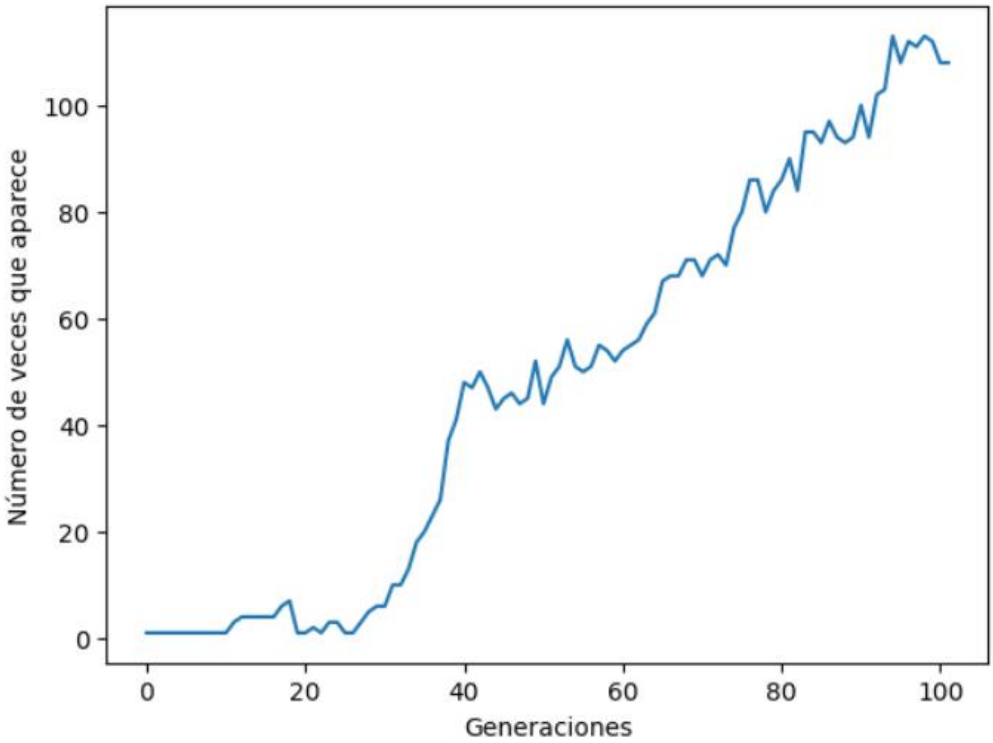

Ilustración 18 Gráfico que representa el número de veces que aparece el mejor indivduo de la población para E = 4. [2]

4.2.2.2. Número de estaciones = 8

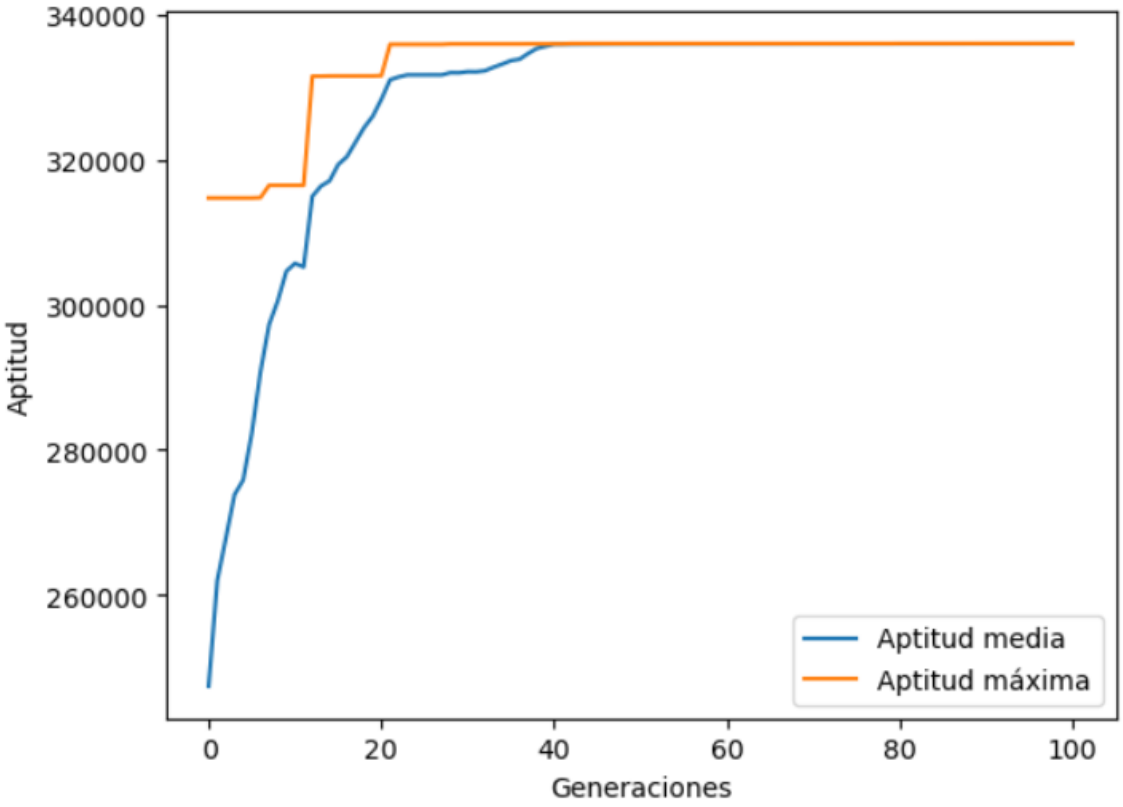

Ilustración 19 Gráfico de fitness medio y máximo para E = 8. [2]

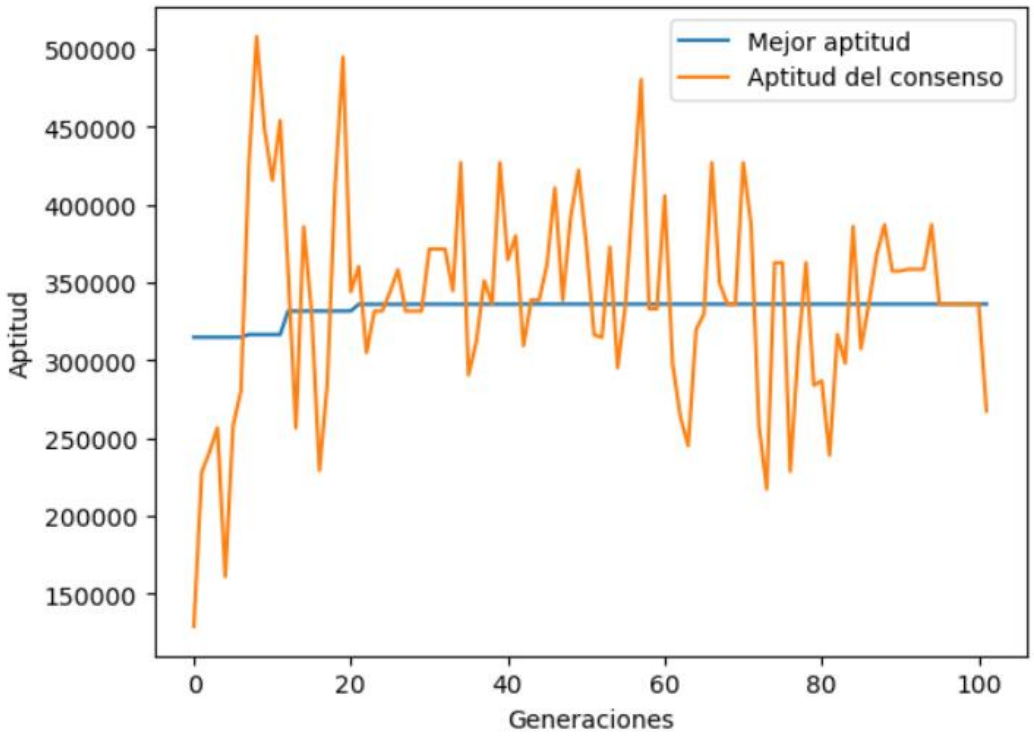

Ilustración 20 Gráfico de mejor fitness de la población VS Mejor fitness del individuo consenso para E = 8. [2]

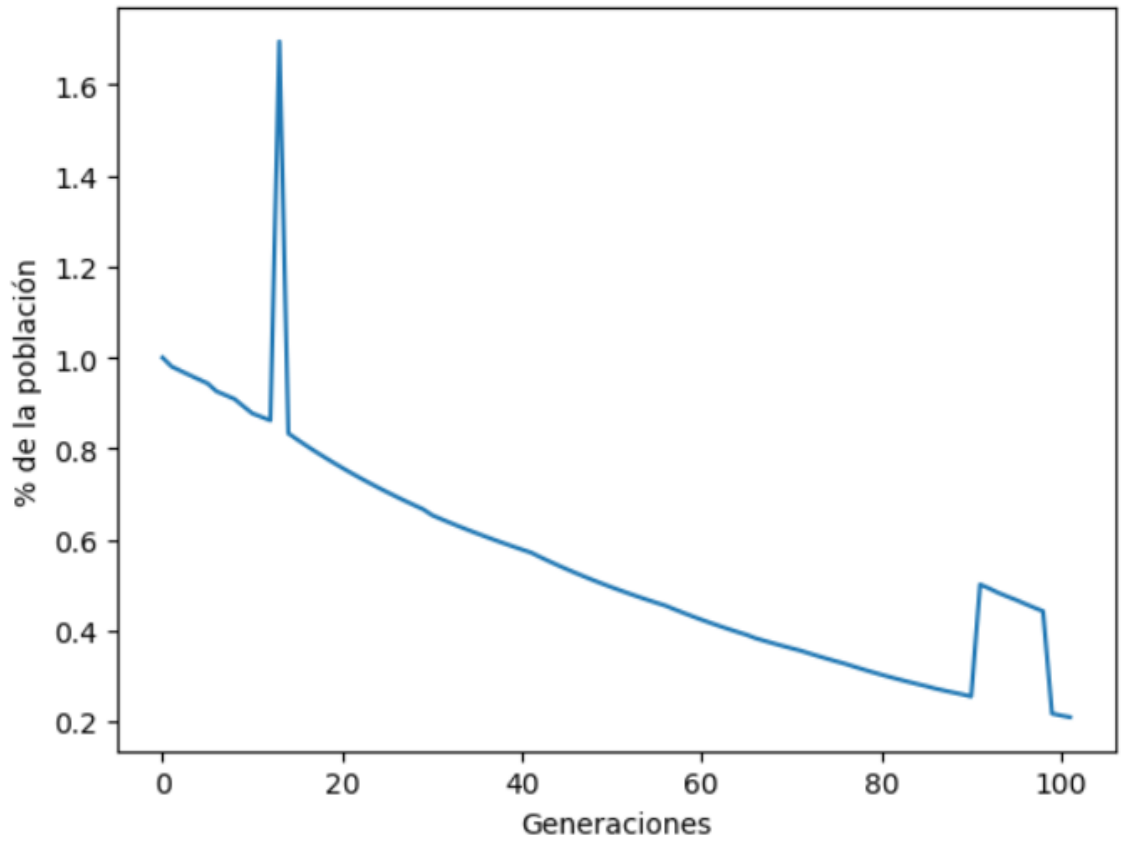

Ilustración 21 Gráfico de porcentaje de la población que representa el mejor cromosoma para E = 8. [2]

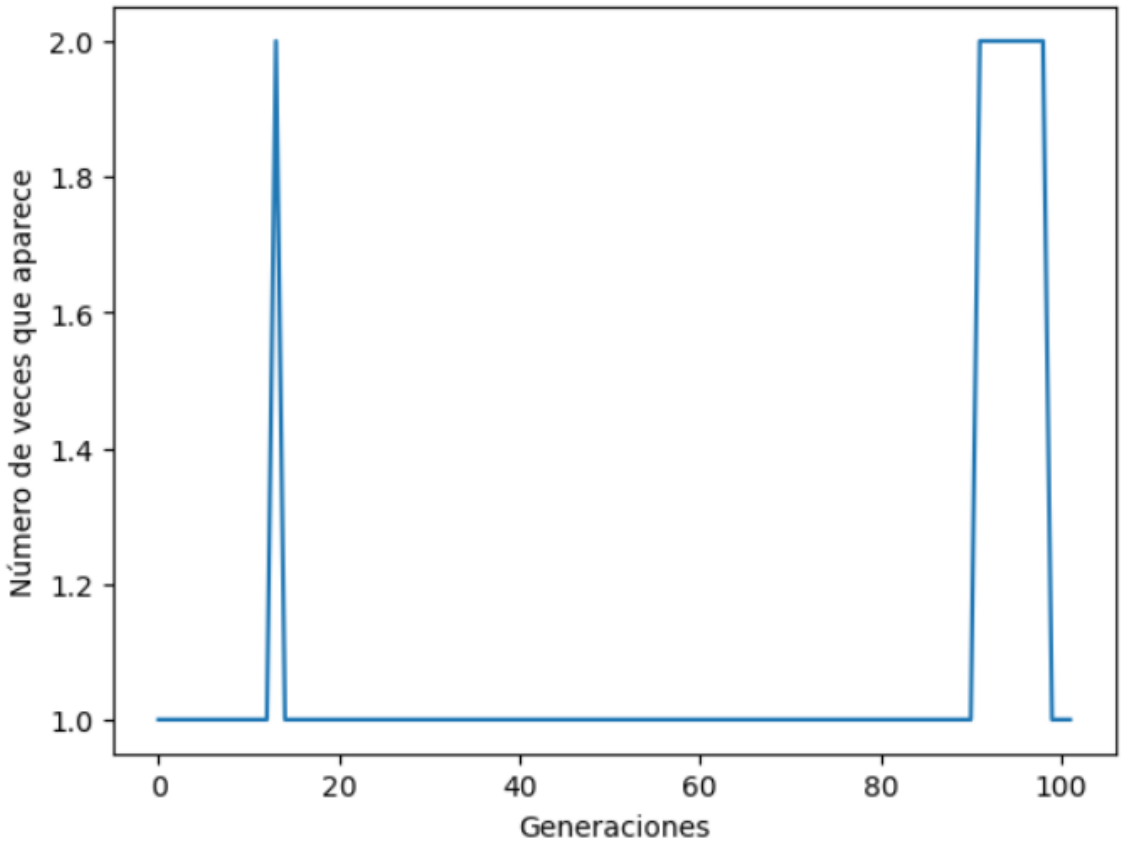

Ilustración 22 Gráfico que representa el número de veces que aparece el mejor individuo de la población para E = 8. [2]

4.2.2.3. Número de estaciones = 12

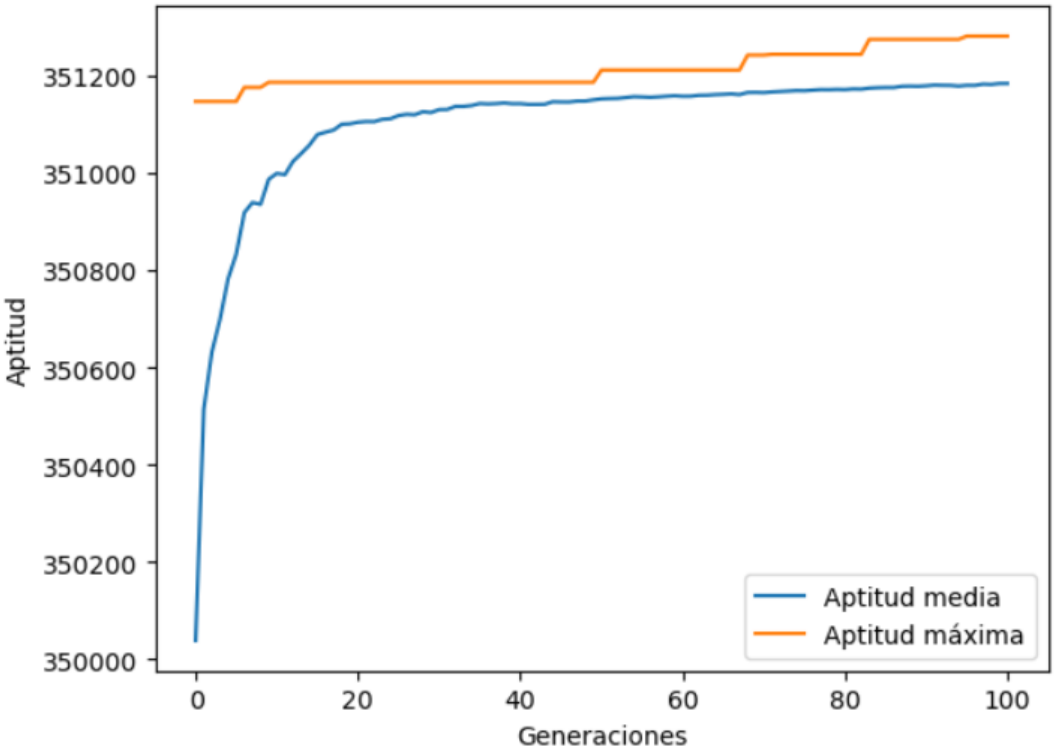

Ilustración 23 Gráfico de fitness medio y máximo para E = 12. [2]

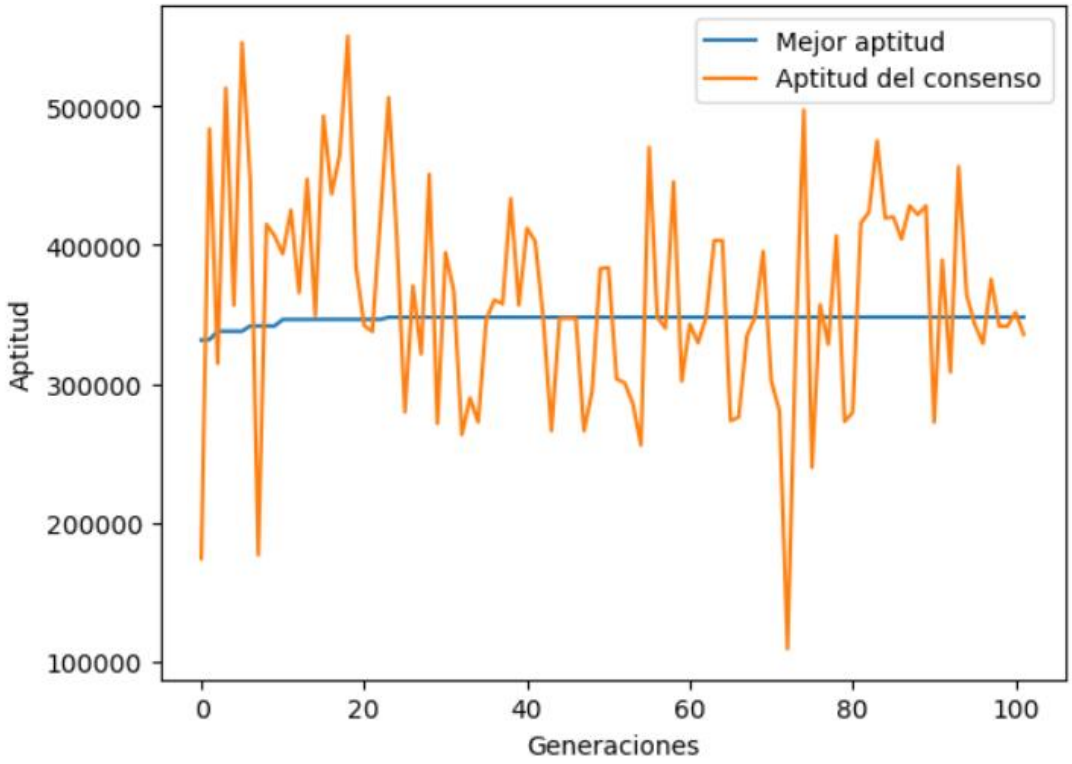

Ilustración 24 Gráfico de mejor fitness de la población VS Mejor fitness del individuo consenso para E = 12. [2]

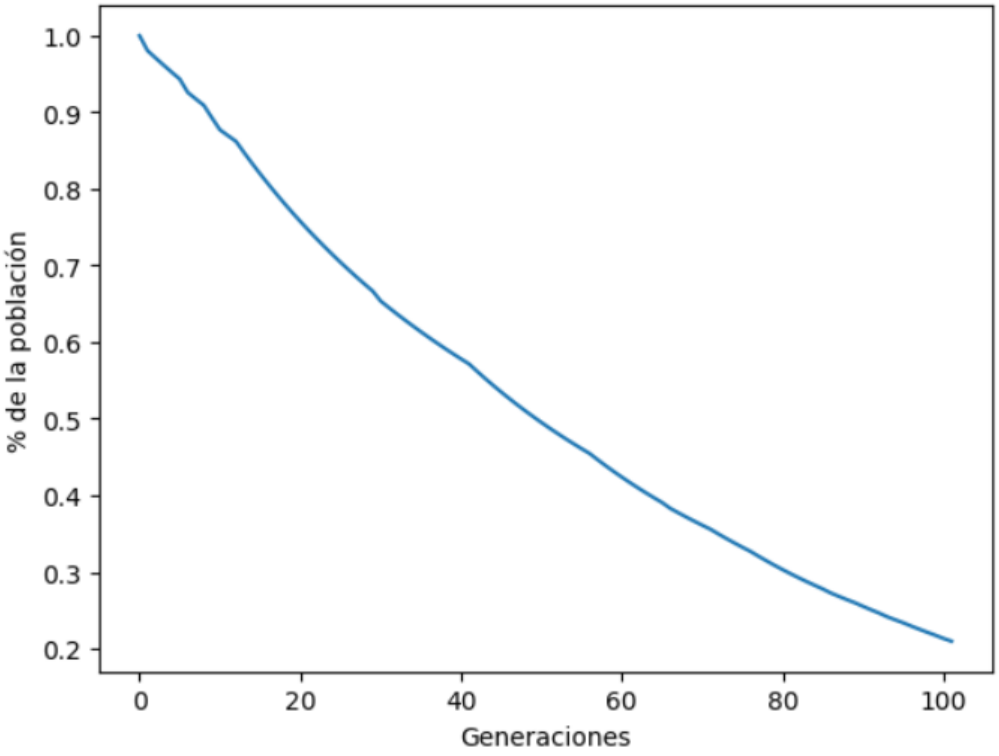

Ilustración 25 Gráfico de porcentaje de la población que representa el mejor cromosoma para E = 12. [2]

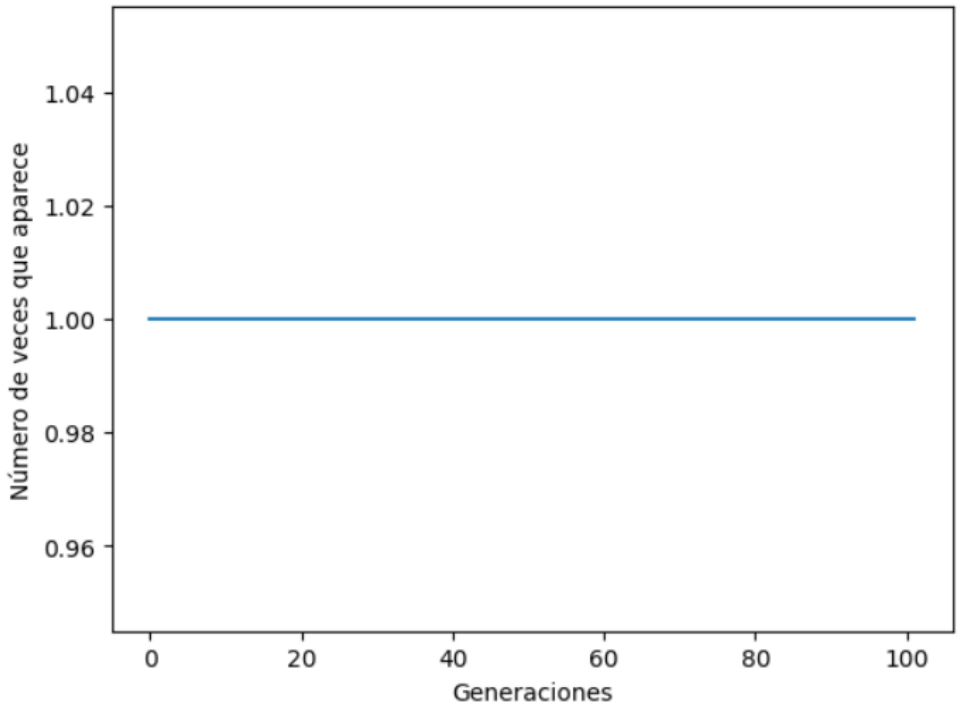

Ilustración 26 Gráfico que representa el número de veces que aparece el mejor individuo de la población para E = 12. [2]

4.2.2.4. Número de estaciones = 16

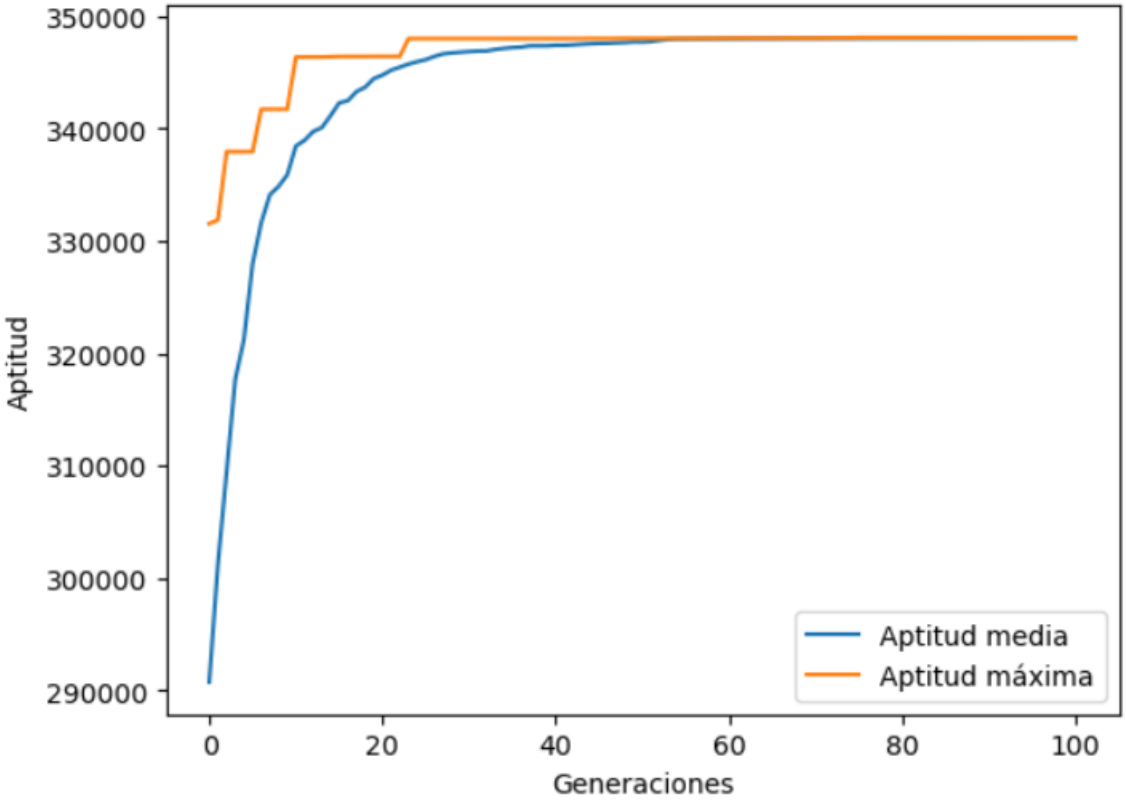

Ilustración 27 Gráfico de fitness medio y máximo para E = 18. [2]

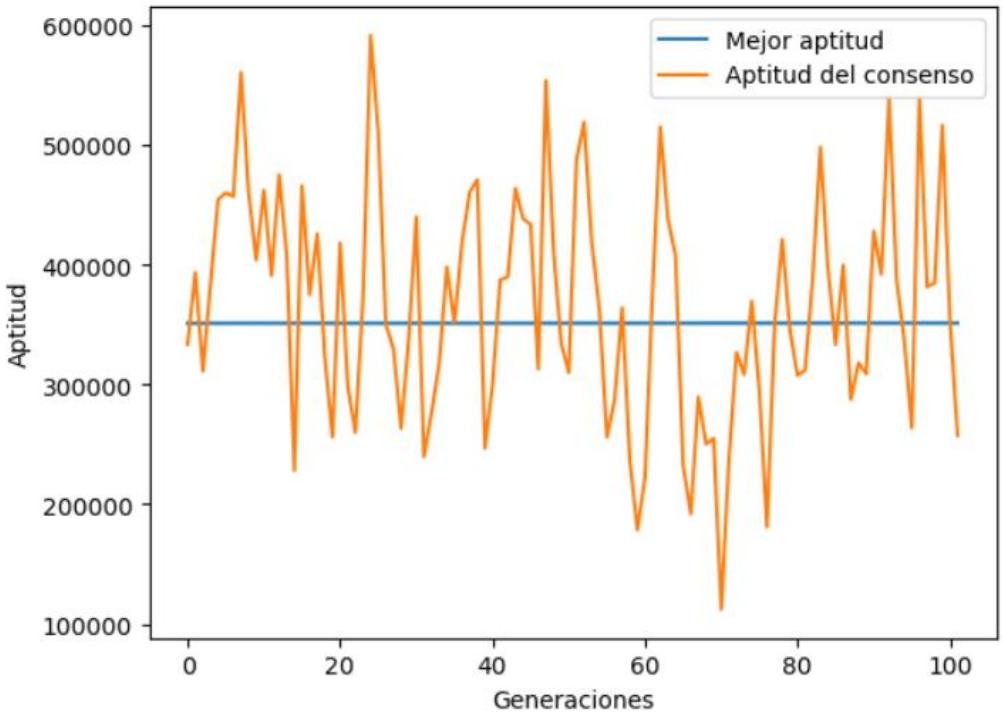

Ilustración 28 Gráfico de mejor fitness de la población VS Mejor fitness del individuo consenso para E = 18. [2]

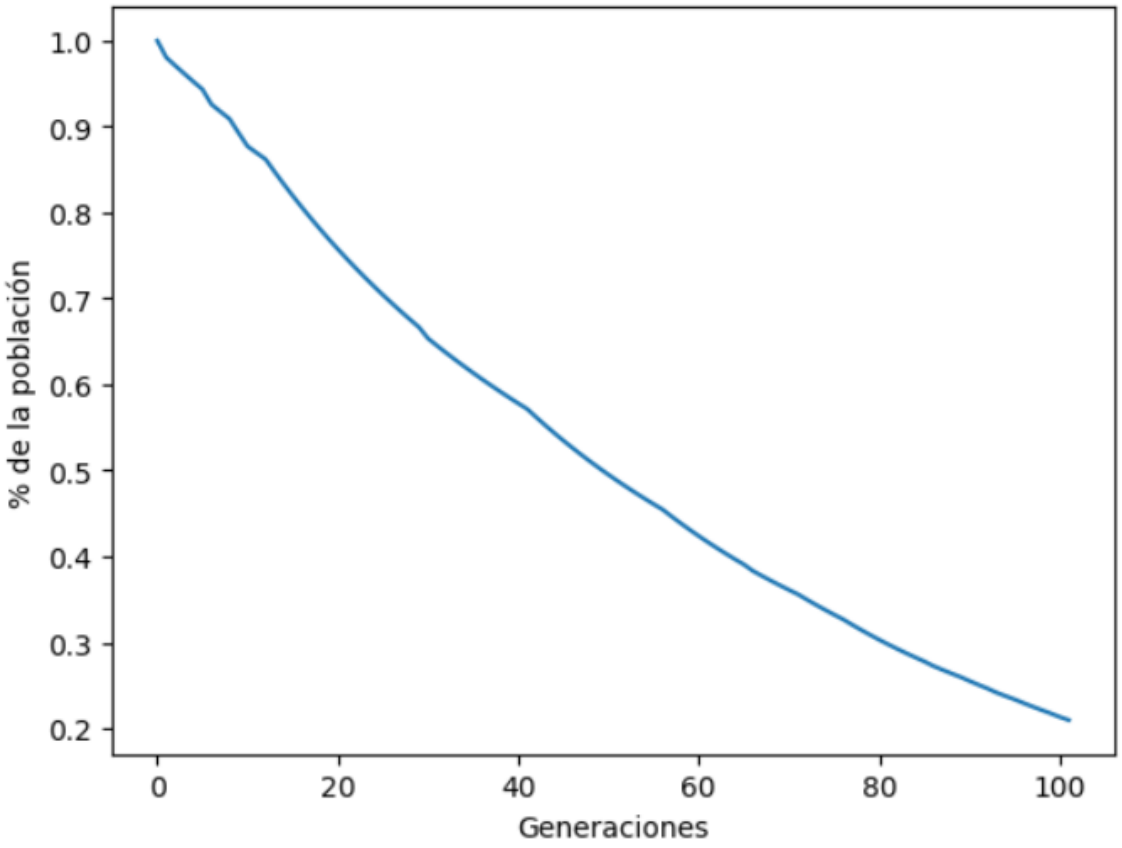

Ilustración 29 Gráfico de porcentaje de la población que representa el mejor cromosoma para E = 18. [2]

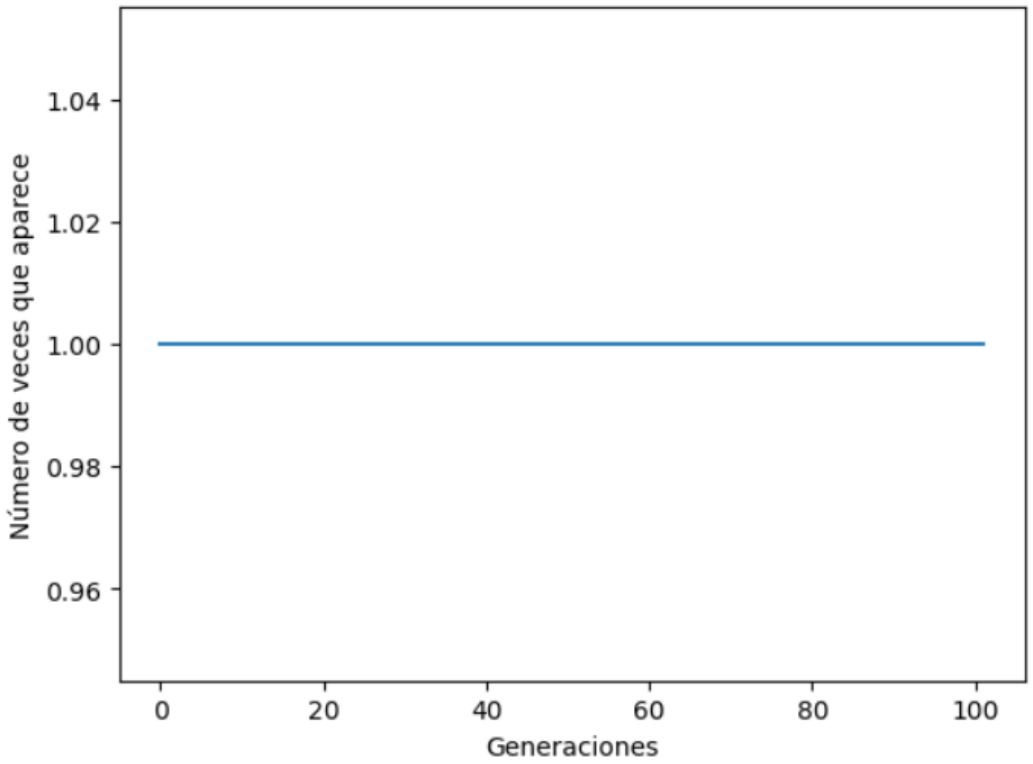

Ilustración 30 Gráfico que representa el número de veces que aparece el mejor individuo de la población para E = 18. [2]

4.2.3. Conclusiones

Cuando el número de estaciones es bajo, como en el caso de nuestro ejemplo, se presenta un fenómeno conocido como deriva genética. Esto ocurre cuando la población es pequeña y por probabilidad los alelos de los cromosomas van cambiando de generación en generación. En otras palabras, como hay pocos individuos en la población, es más probable que se generen cromosomas aleatorios que se alejen del mejor cromosoma encontrado hasta el momento, lo que causa una alteración en la frecuencia del mejor cromosoma en la población.

Además, con un número de estaciones bajo, la cantidad de genes en el cromosoma es muy pequeña en comparación con la población existente, lo que hace que la variabilidad genética sea menor y que haya menos combinaciones posibles de cromosomas. Por lo tanto, los porcentajes de cambio e intercambio de genes son bajos y esto afecta la variabilidad genética y la capacidad del algoritmo para encontrar soluciones óptimas.

Cuando el número de estaciones es cercano al número de municipios, se espera una distribución más uniforme del número de veces que se repite el mejor cromosoma en la generación, así como del porcentaje de la población que ocupa el mejor cromosoma. Esto se debe a que hay más genes en el cromosoma y, por lo tanto, una mayor diversidad genética, lo que resulta en una mayor probabilidad de que la población encuentre soluciones diferentes y óptimas.

Al querer minimizar el recorrido total de la línea de metro, se hace más difícil encontrar una solución óptima cuando el número de estaciones es cercano al número de municipios. Esto se debe a que, al haber más estaciones, aumenta el espacio de búsqueda y las posibles combinaciones de rutas, lo que hace que el algoritmo tarde más en converger hacia una solución óptima.

Además, cuando el número de estaciones es cercano al número de municipios, es posible que algunas estaciones queden demasiado lejos o demasiado cerca de otras, lo que dificulta la creación de rutas óptimas. En estos casos, es importante tener en cuenta la topografía y la distribución de los municipios para encontrar una solución que sea eficiente y práctica.

4.3. Cuestión 3

Estudia cómo evolucionan BESTf, BESTn, BEST% y CONSf con especto a alfa, NPOB y NGEN. Utilizar gráficos donde se recoja la relación entre estas variables, elige el menor conjunto e intentar encontrar alguna relación que garantice el mejor resultado. Explica lo que ocurre.

Se han realizado varios experimentos para distintos valores de NPOB y NGEN, los experimentos más destacados son los que tienen valores de 100 y 200 tanto de NPOB y NGEN.

4.3.1. NPOB = 100 y NGEN = 100

Para Número de Población igual a 100 y número de generaciones igual a 100 los resultados son los siguientes:

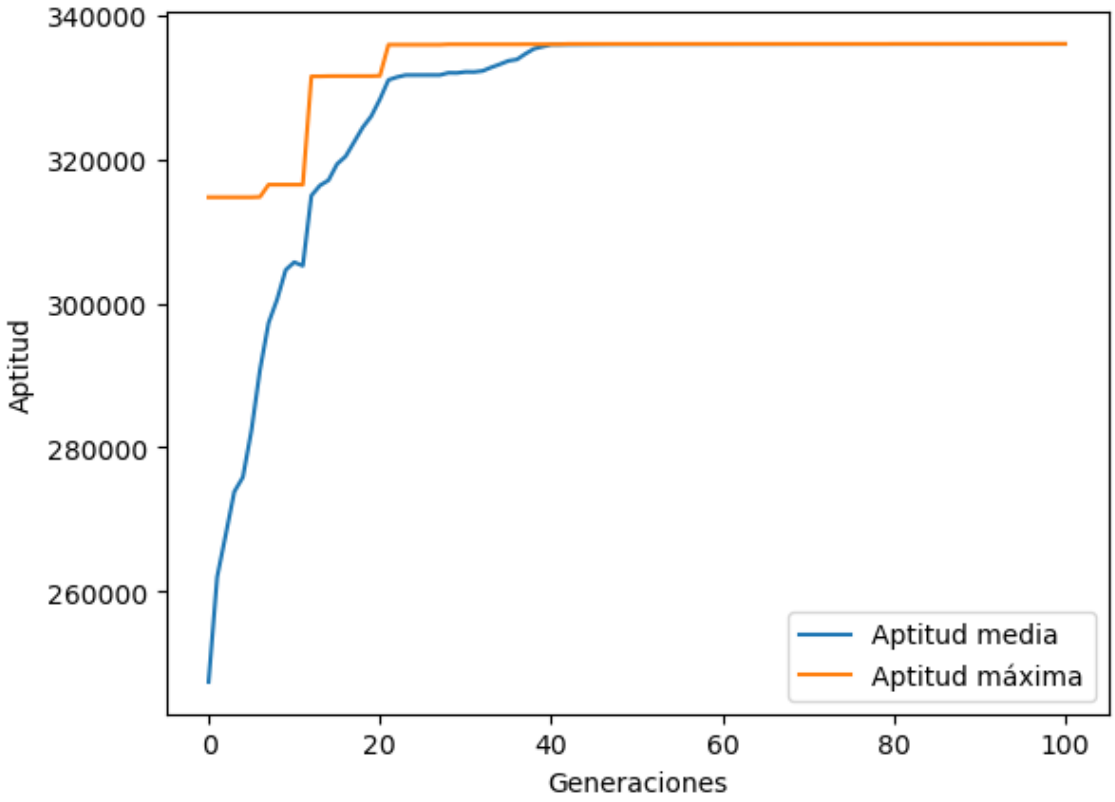

Ilustración 31 Gráfico de evolución de la aptitud media y máxima. [2]

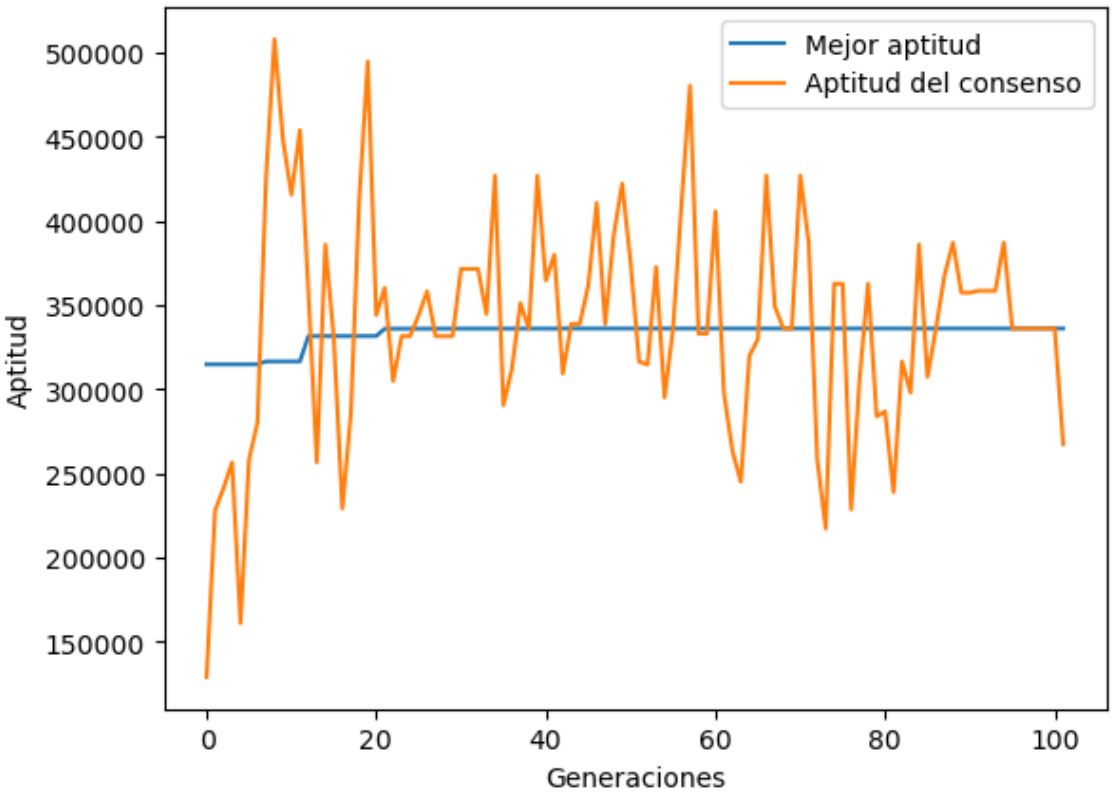

Ilustración 32 Gráfico de evolución de los mejores resultados. [2]

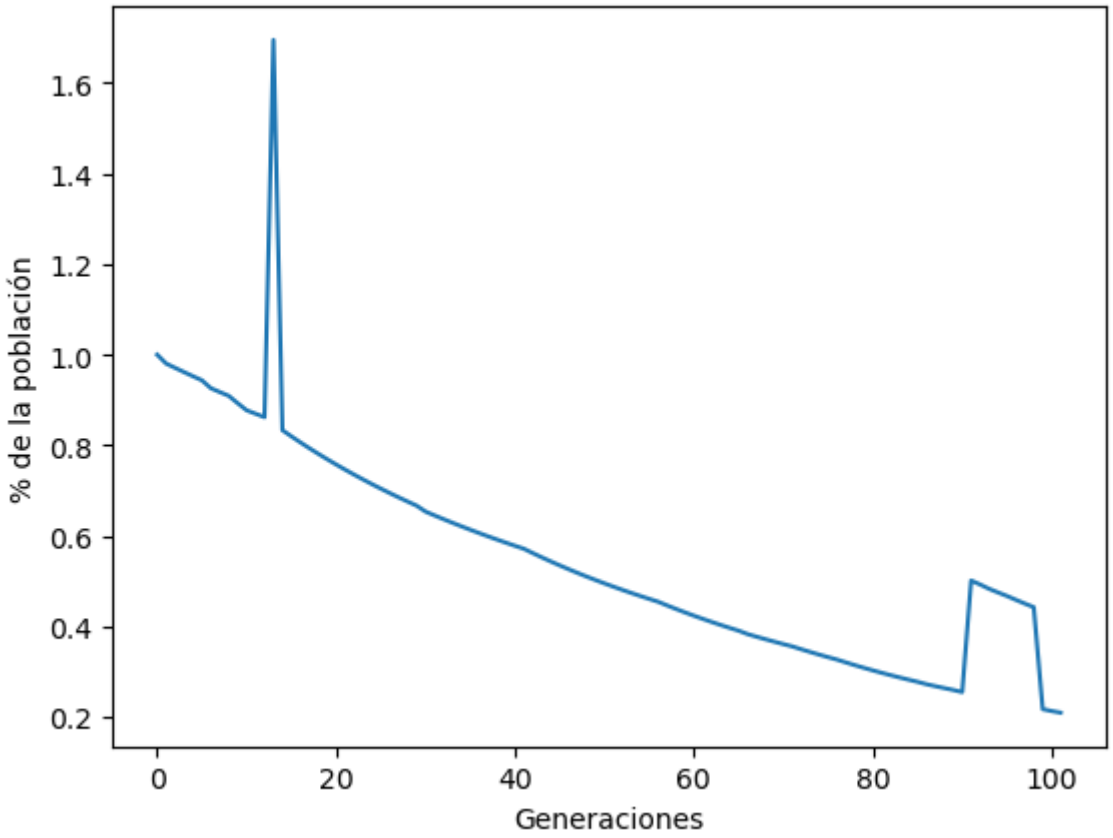

Ilustración 33 Gráfica de Evolución del porcentaje que representa el mejor cromosoma. [2]

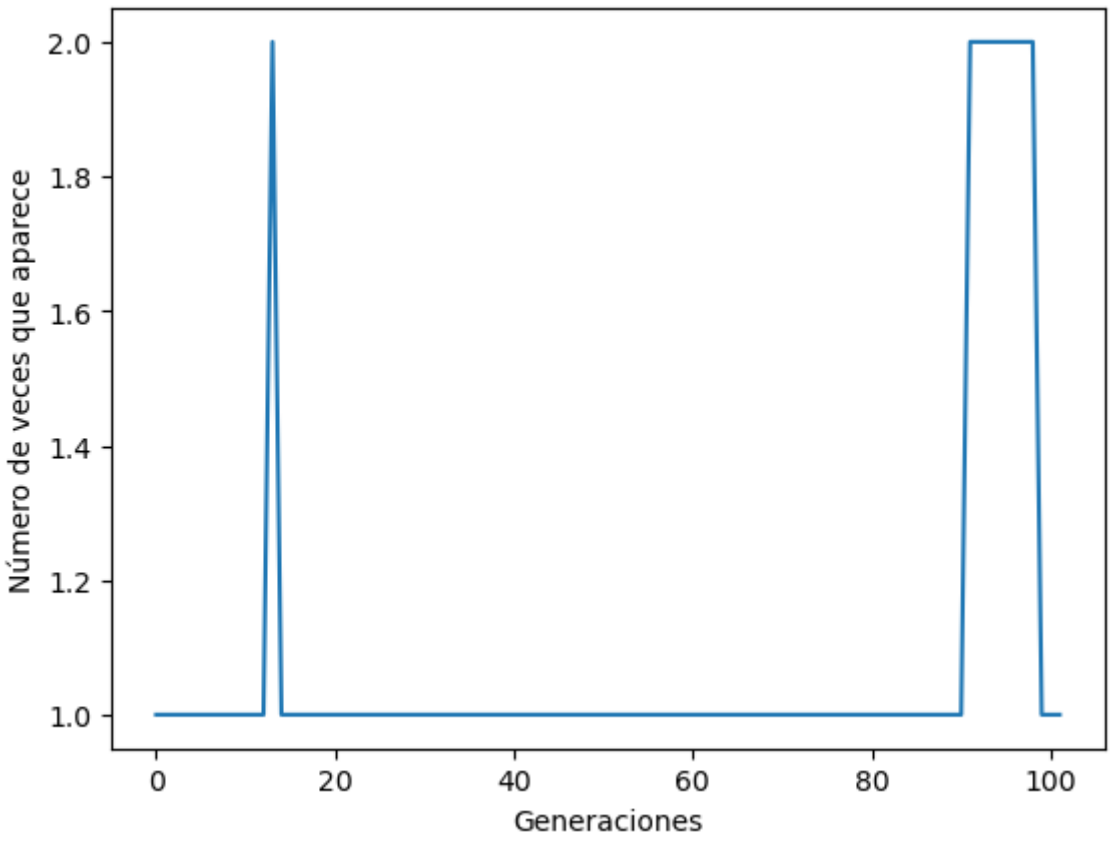

Ilustración 34 Gráfica de Evolución del número de veces que aparece el mejor cromosoma en la población para NPOB 100 y NGEN 100. [2]

Los resultados obtenidos muestran que el algoritmo genético ha sido capaz de encontrar una solución óptima para el problema en cuestión, aunque con una cierta variabilidad en la aptitud de consenso a lo largo de las generaciones. La igualdad de la aptitud media y máxima a partir de la generación 40 indica que la población ha convergido hacia una solución global, mientras que el aumento de la aptitud máxima a partir de la generación 20 sugiere que el algoritmo ha sido capaz de mejorar la calidad de la mejor solución individual.

La relativa constancia de la mejor aptitud y la variabilidad de la aptitud de consenso sugieren que el algoritmo ha encontrado una solución óptima en una región del espacio de búsqueda, pero que hay cierta variabilidad en las soluciones que se generan a lo largo del proceso. Por otro lado, el hecho de que el porcentaje de la población que representa el mejor cromosoma sea menor al 1% indica que la búsqueda ha sido exploratoria, aunque la aparición del mejor cromosoma en la población en varias generaciones sugiere que esta solución es relativamente estable.

En conclusión, los resultados indican que el algoritmo genético ha encontrado una solución óptima para el problema planteado, aunque con cierta variabilidad en la aptitud de consenso a lo largo de las generaciones y con una exploración relativamente amplia del espacio de búsqueda.

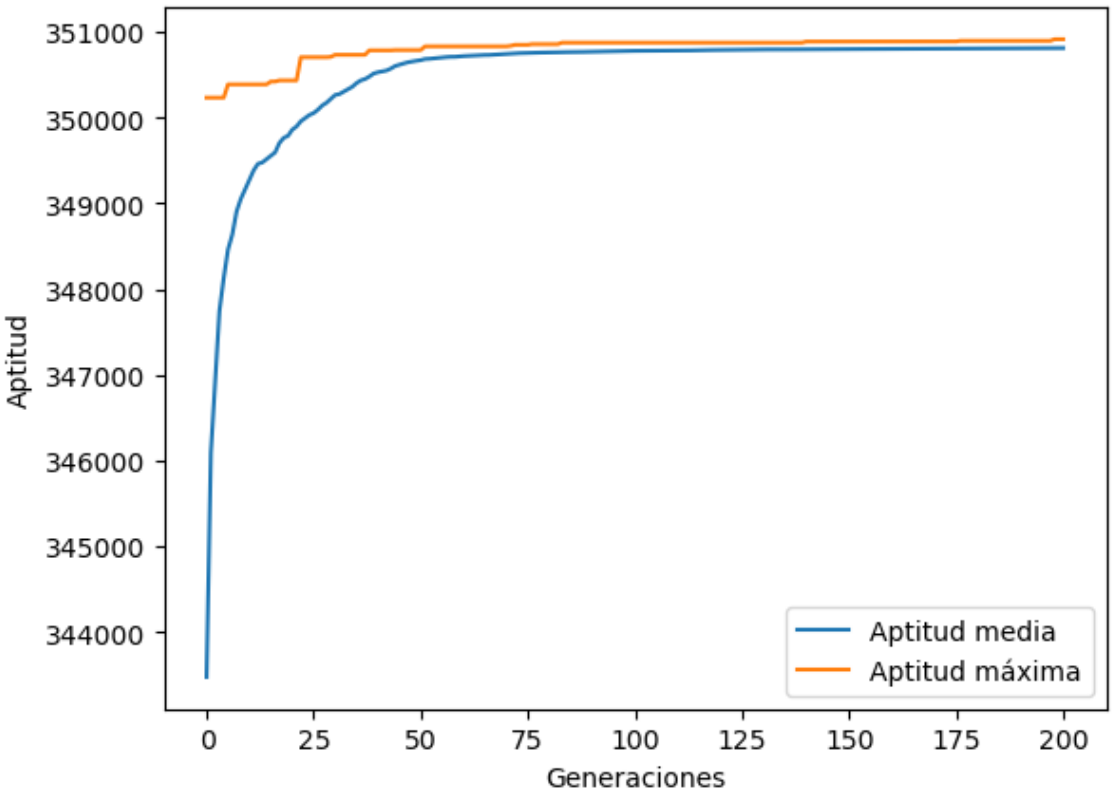

Ilustración 35 Gráfico de evolución de la aptitud media y máxima NPOB 200 y NGEN 200. [2]

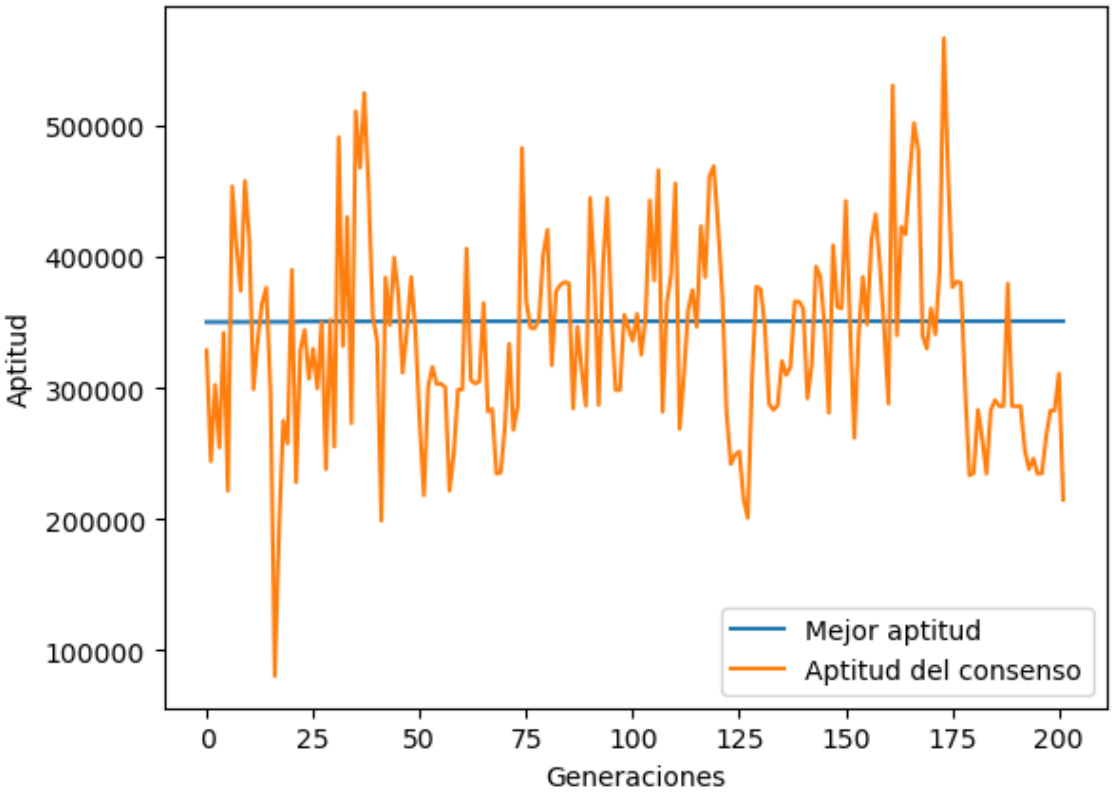

Ilustración 36 Gráfico de evolución de los mejores resultados NPOB 200 y NGEN 200. [2]

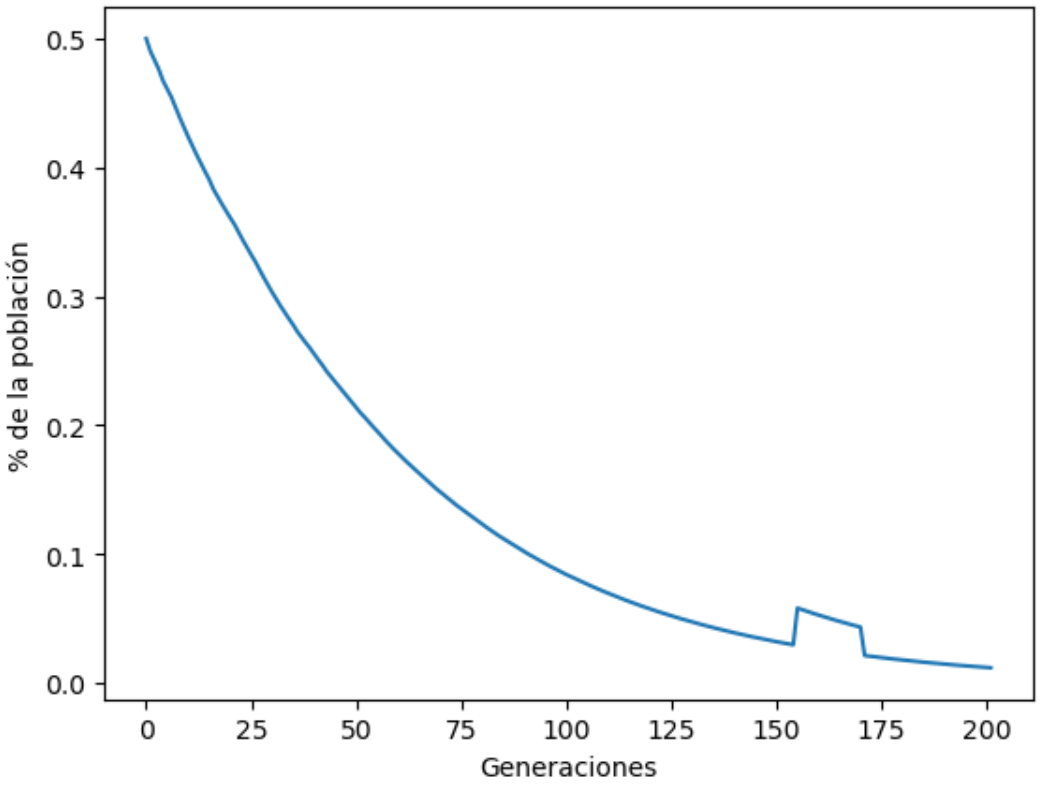

Ilustración 37 Gráfico de evolución del porcentaje de la población que representa el mejor cromosoma NPOB 200 y NGEN 200. [2]

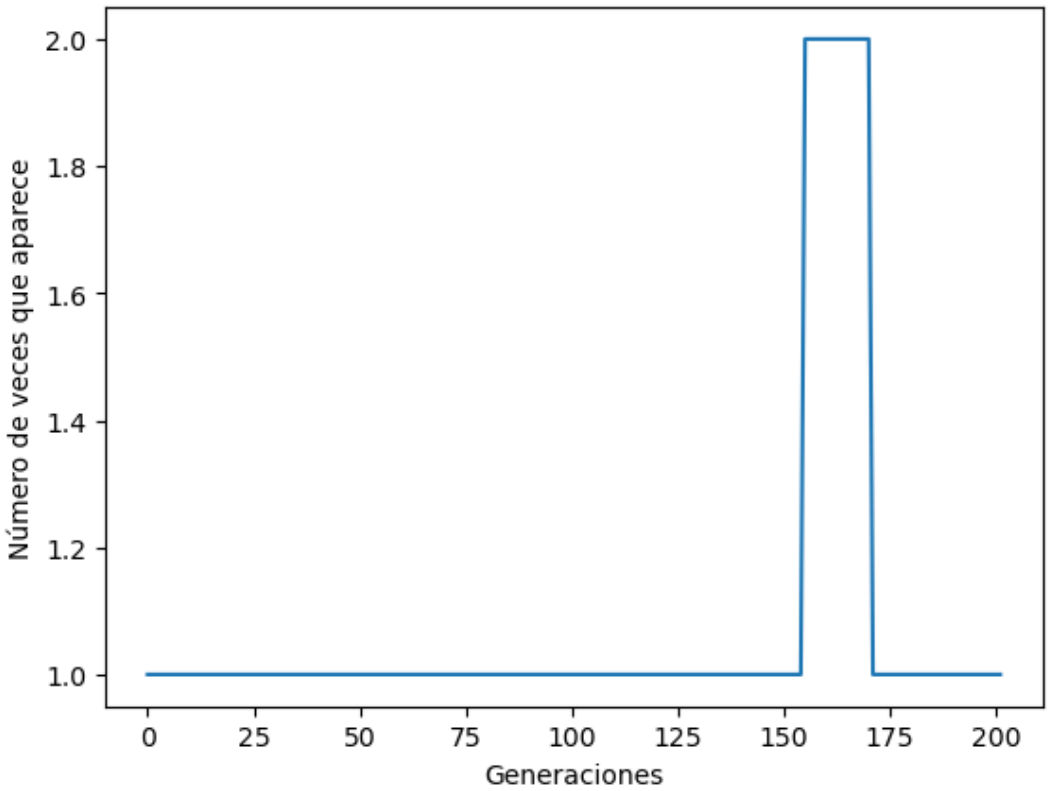

Ilustración 38 Gráfico de evolución del número de veces que aparece el mejor cromosoma en la población NPOB 200 y NGEN 200. [2]

4.3.2. NPOB = 200 y NGEN = 200

Para Número de Población igual a 200 y número de generaciones igual a 200 los resultados son los siguientes:

Los resultados muestran que, en este caso particular, el algoritmo genético alcanza su mejor valor de aptitud máxima y se estabiliza alrededor de la generación 50, lo que sugiere que una cantidad suficiente de generaciones es necesaria para que el algoritmo converja. Además, se puede observar que la aptitud media y la máxima son iguales después de la generación 50, lo que indica que la población evoluciona hacia individuos más aptos.

En cuanto a la variación entre la mejor aptitud y la de consenso, los resultados muestran que el mejor cromosoma es relativamente constante y no sufre cambios, mientras que la aptitud de consenso es variable y disminuye en las últimas 25 generaciones. Esto puede deberse a que la población ha convergido a un número limitado de soluciones que se mantienen relativamente estables.

En relación con el porcentaje de la población que representa el mejor cromosoma, se observa que en la mayoría de las generaciones menos de un 0,5% de la población representa el mejor cromosoma. Además, solo se observa un pico entre las generaciones 150 y 175, lo que sugiere que la población se mantiene diversa y no se enfoca en un solo individuo.

Finalmente, el número de veces que aparece el mejor cromosoma en la población es máximo en las generaciones 150 y 175, lo que sugiere que estos son los momentos en que el algoritmo encuentra las soluciones óptimas. En general, estos resultados indican que el ajuste adecuado de los parámetros puede ayudar a obtener mejores resultados en términos de la convergencia y la diversidad de la población.

4.3.3. Conclusiones

En general, los resultados para NPOB y NGEN 200 muestran una convergencia más estable y una mayor diversidad en la población que los resultados para NPOB y NGEN 100. Esto sugiere que una población más grande y un mayor número de generaciones pueden mejorar la convergencia del algoritmo y la diversidad de la población. Sin embargo, los resultados para NPOB y NGEN 100 indican que es posible encontrar una solución óptima con una población más pequeña y un menor número de generaciones, aunque puede haber cierta variabilidad en la calidad de las soluciones generadas a lo largo del proceso. En general, el ajuste adecuado de los parámetros del algoritmo es importante para obtener buenos resultados en términos de la convergencia y la diversidad de la población.

4.4. Cuestión 4

Estudia cómo evoluciona BESTf, BESTn, BEST% y CONSf para valores crecientes de Pc y Pi. Explica la gráfica que obtienes. Estudia también qué pasa con el individuo consenso.

A lo largo del laboratorio de han probado numerosas configuraciones de PC y Pi, aquí se ilustran las más significativas.

Para una configuración de 12 estaciones de metro, junto a 100 individuos de la población, haciendo uso de 100 generaciones, probamos con numerosos valores para pc y pi:

4.4.1. Pc inicial de 0.1 y Pi de 0.2 se observan los siguientes resultados:

Como se puede observar para esta configuración la aptitud media aumenta progresivamente hasta poder alcanzar la aptitud máxima sobre la generación 50, esto es debido a que las probabilidades de combinación e intercambio son muy bajas de manera que n cada nueva generación se están insertando individuos que no cuentan con ningún intercambio o combinación haciendo que las generaciones sean muy similares entre sí.

Respecto al individuo consenso podemos observar que a lo largo de las generaciones este comienza muy disperso respecto a la mejor aptitud, pero progresivamente se va ajustando a medida que se llega a las generaciones finales.

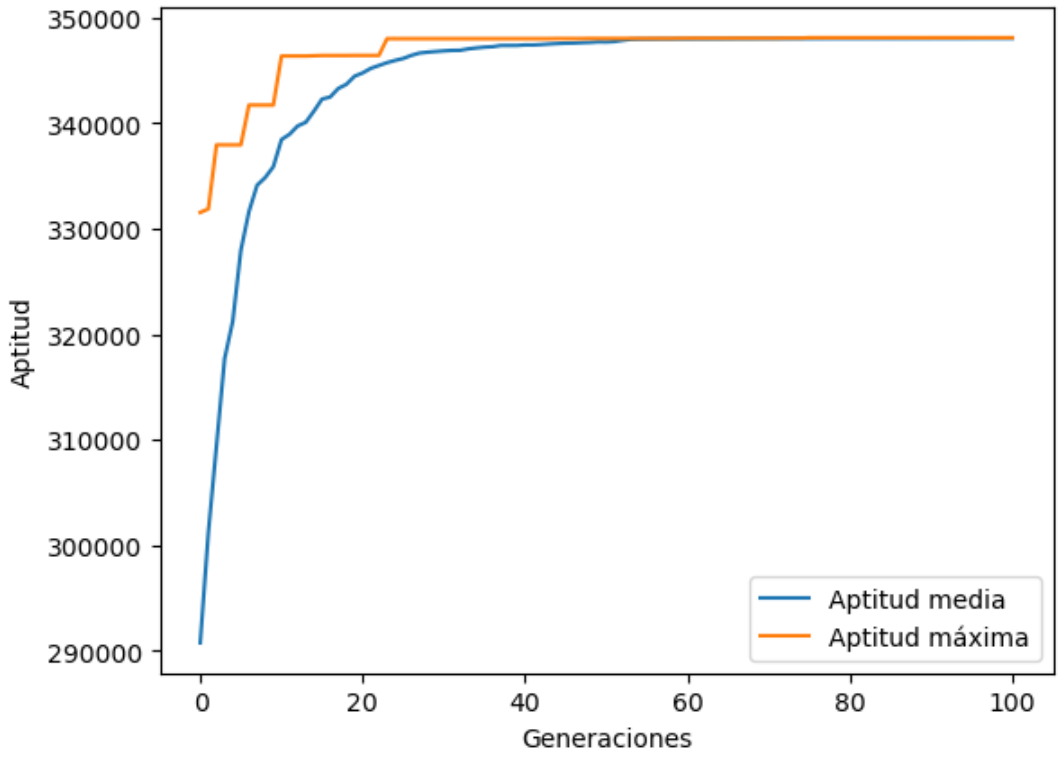

Ilustración 39: Evolución de aptitud media y máxima con Pc = 0.1 y Pi = 0.2. [2]

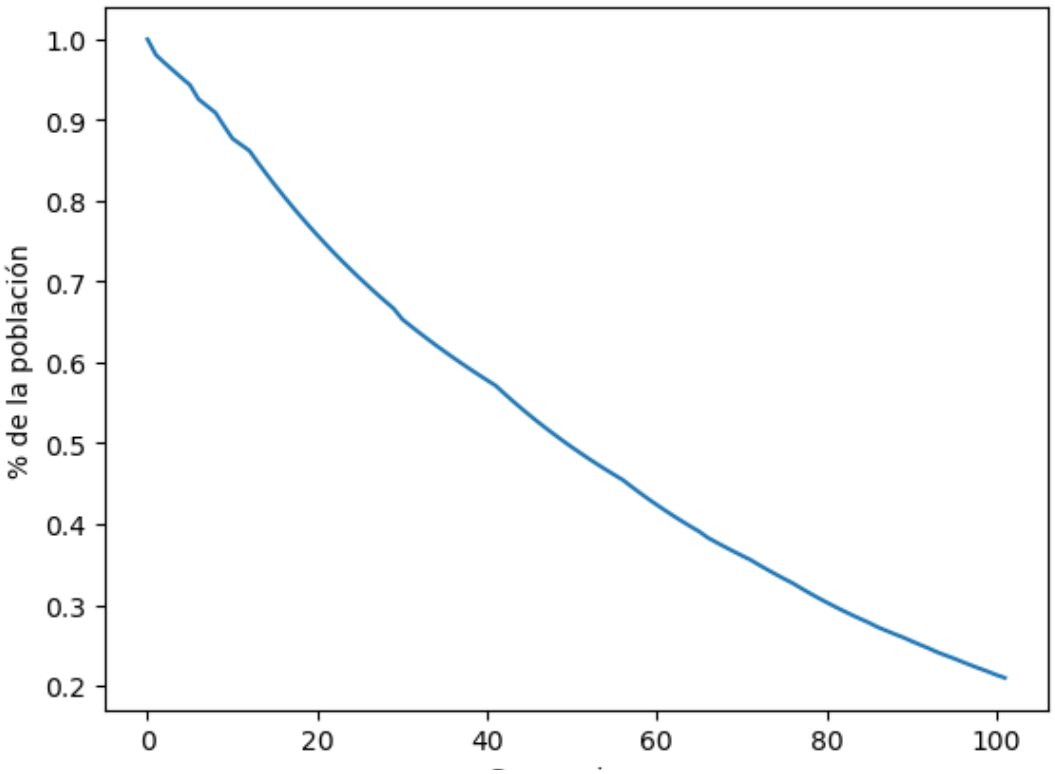

Ilustración 41:Evolución de las generaciones Pc = 0.1 y Pi = 0.2 [2]

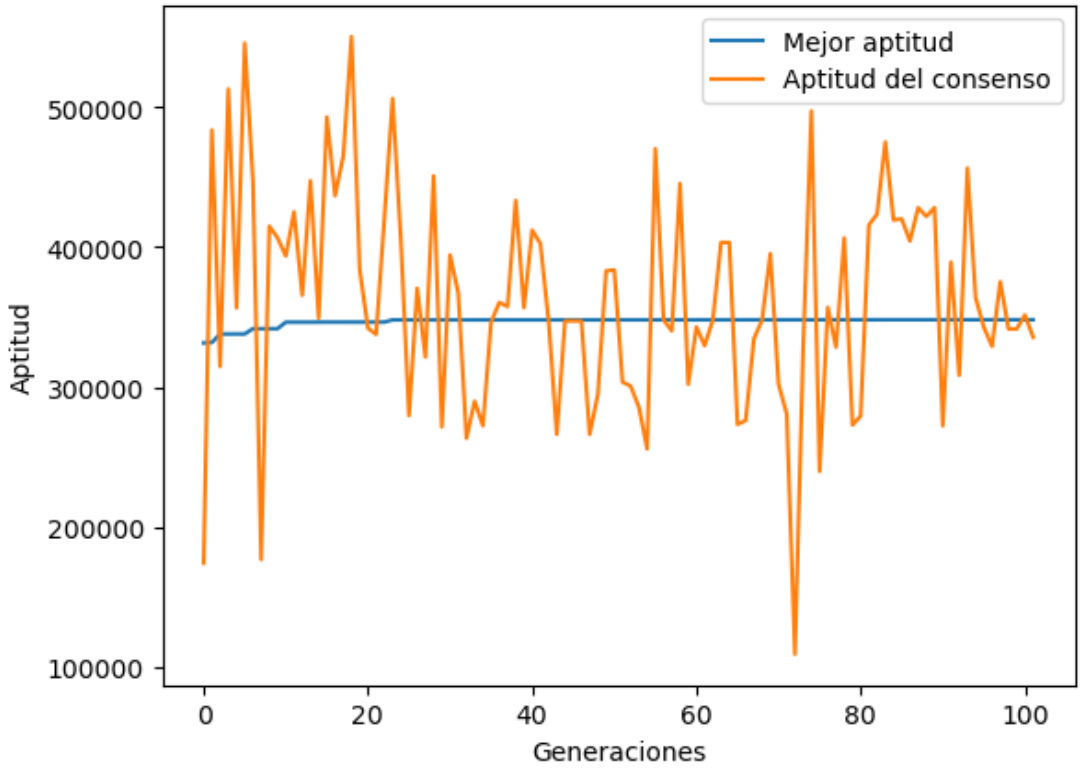

Ilustración 40:Evolución entre mejor aptitud y aptitud del consenso con Pc = 0.1 y Pi = 0.2 [2]

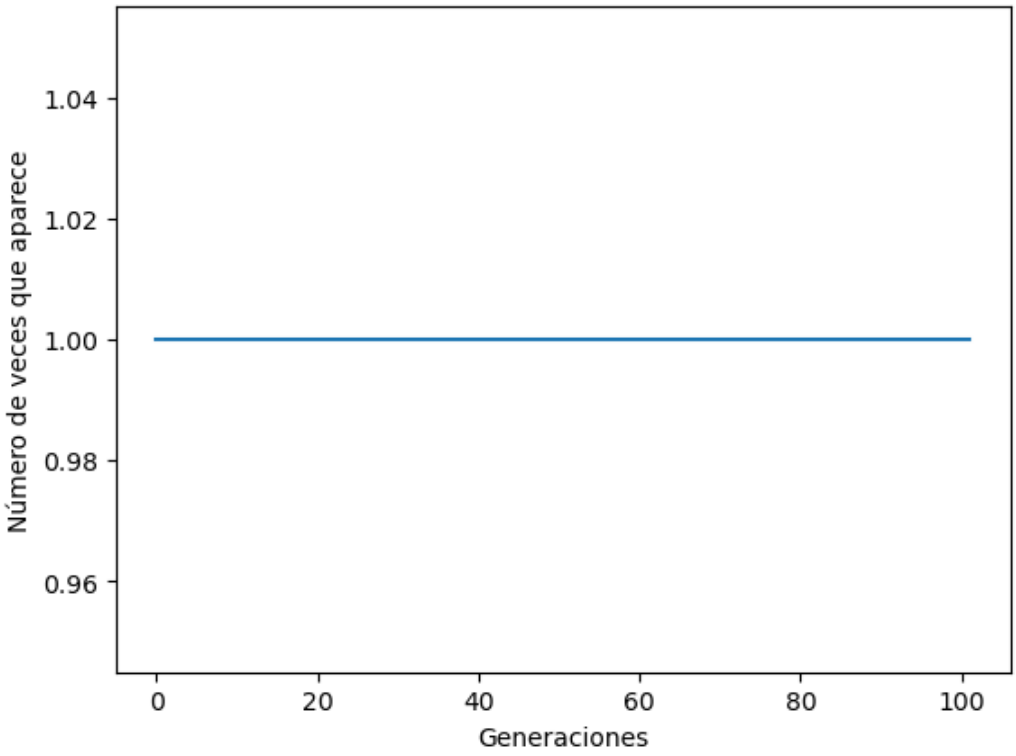

Ilustración 42:Número de apariciones de generaciones [2]

4.4.2. Pc inicial de 0.1 y Pi de 0.6 se observan los siguientes resultados:

En esta configuración al aumentar la probabilidad de intercambio, pero manteniendo la probabilidad de cambio observamos como, se tarda más en llegar a un punto de convergencia entre la aptitud media y máxima, esto es debido que al aumentar la probabilidad de intercambio de estaciones se permite una mejor variación de los individuos.

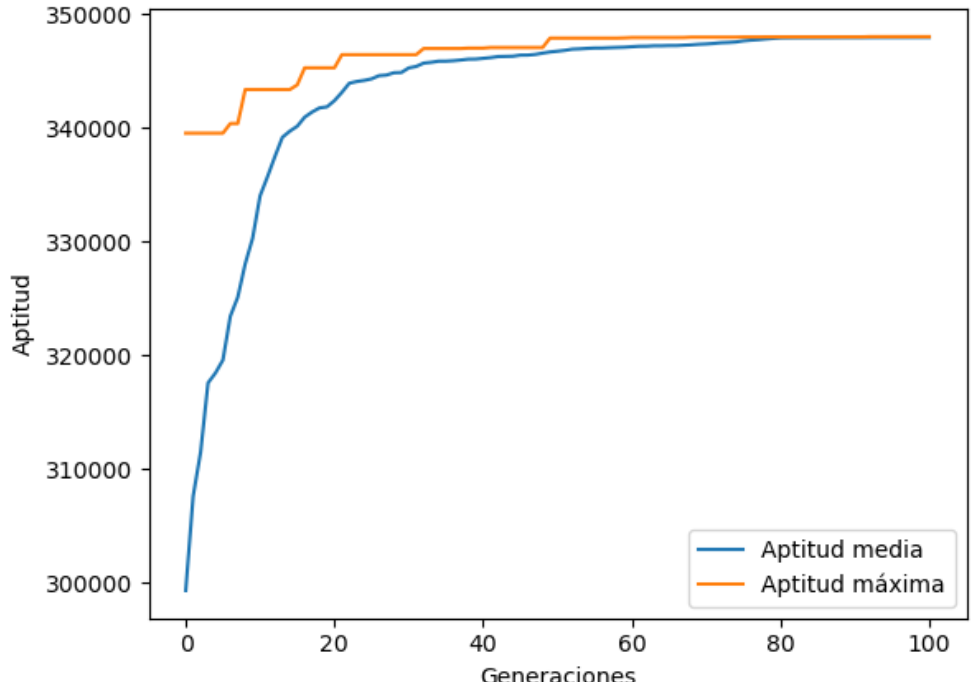

Ilustración 43: Evolución entre mejor aptitud y aptitud del consenso con Pc = 0.1 y Pi = 0.6 [2]

Respecto al individuo consenso, con estos valores de Pc y Pi se observa como dista de la mejor aptitud.

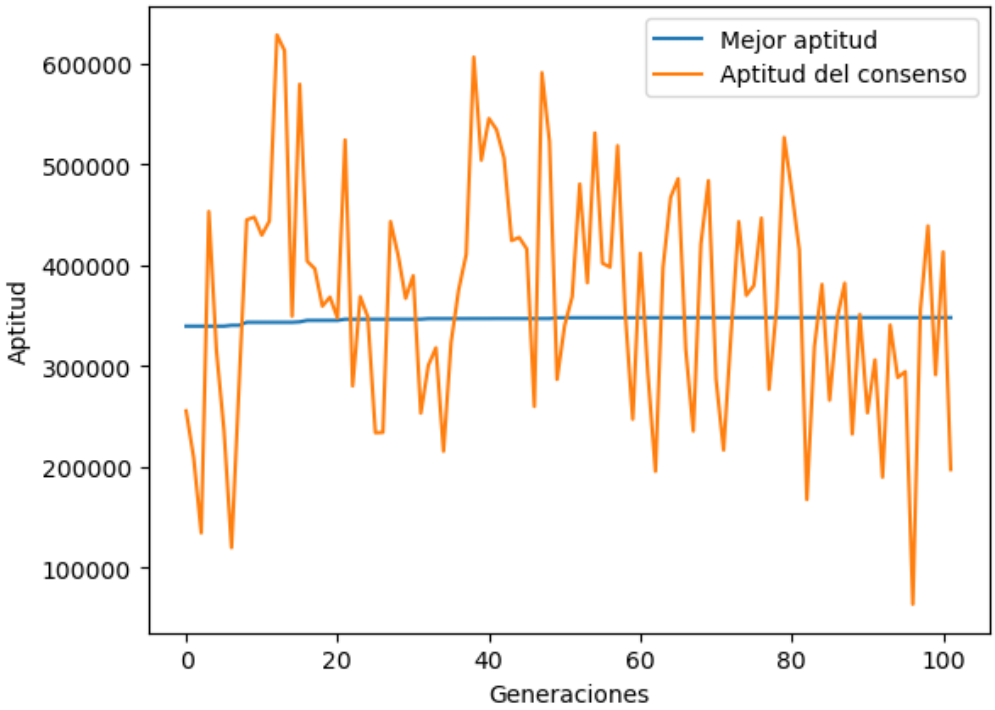

Ilustración 4445: Pc inicial de 0.1 y Pi de 0.6 evolución de la aptitud media y máxima [2]

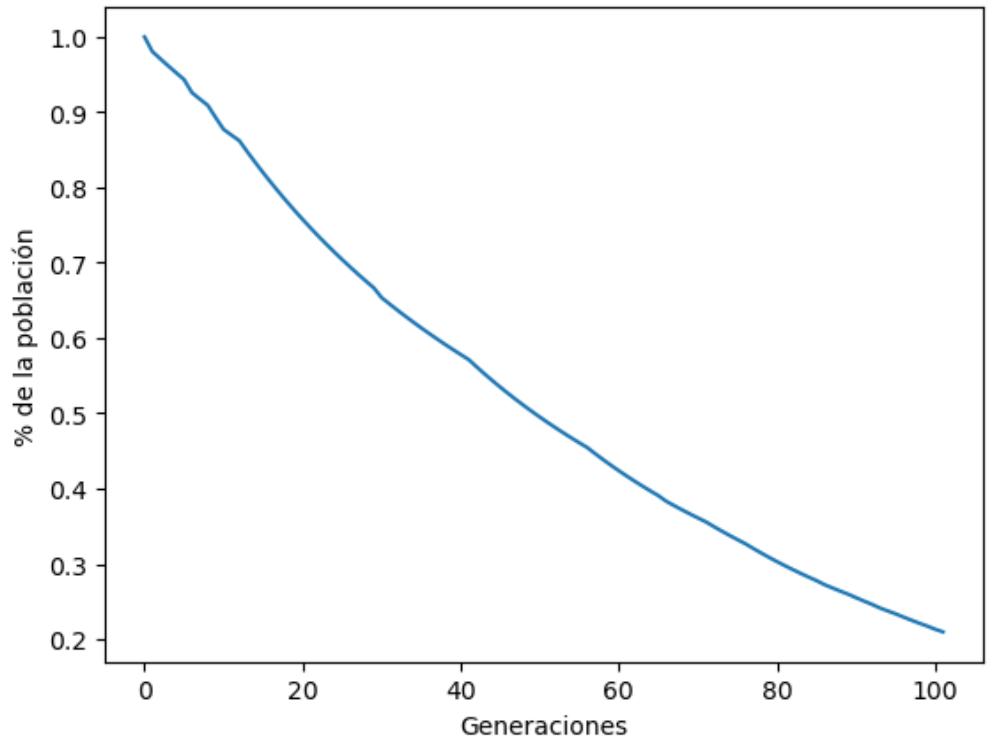

Ilustración 44:Evolución de las generaciones con Pc = 0.1 y Pi = 0.6 [2]

4.4.3. Pc inicial de 0.5 y Pi de 0.2 se observan los siguientes resultados

Para estos valores se puede observar cómo al tener una probabilidad de cambio e intercambio incrementadas las variaciones que se pueden producir en los individuos aumentan y la aleatoriedad de selección de los individuos es mayor, esto acaba provocando que la aptitud media no llegue a la aptitud máxima y por tanto se mantenga en balance a lo largo de las generaciones.

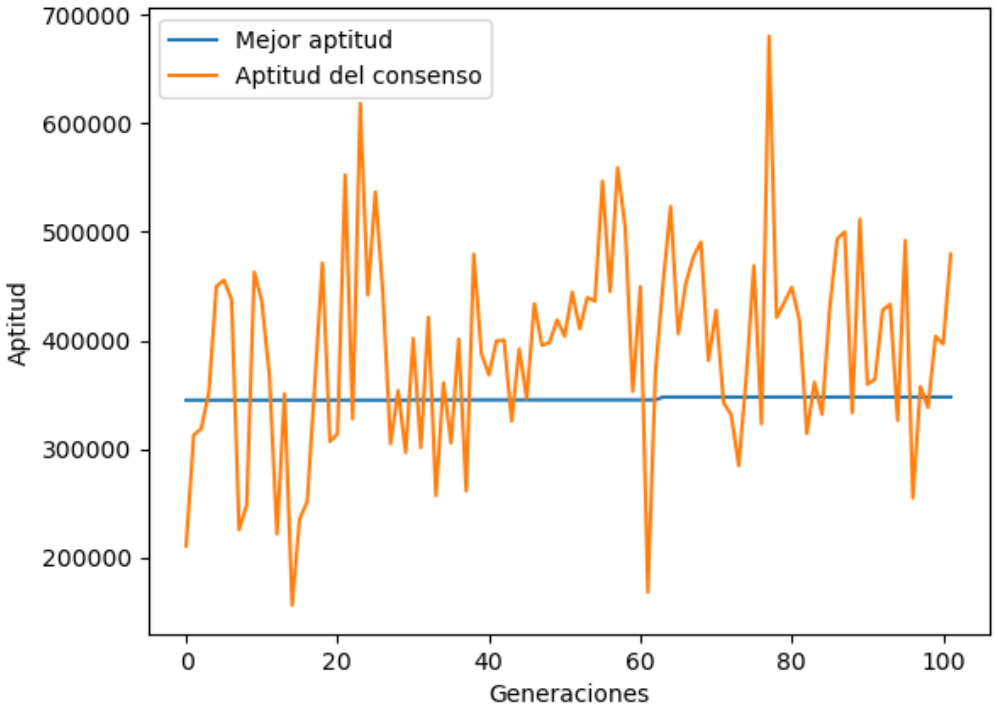

Ilustración 46:Evolución mejor aptitud y aptitud de consenso [2]

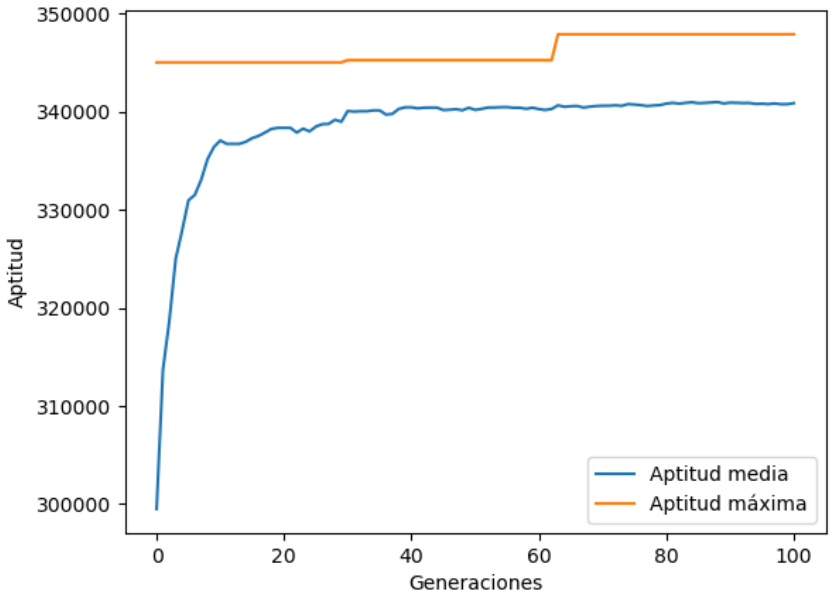

Ilustración 47:Evolución aptitud media y máxima con Pc de 0.5 y 0.2 [2]

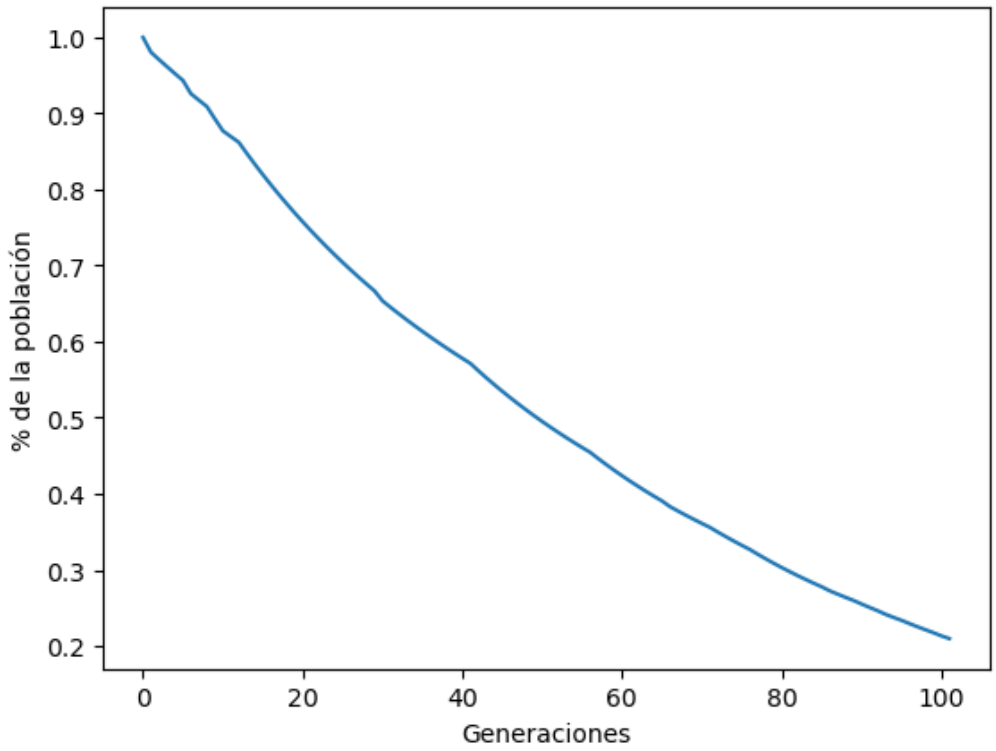

Ilustración 48: Evolución generaciones con Evolución Pc de 0.5 y 0.2 [2]

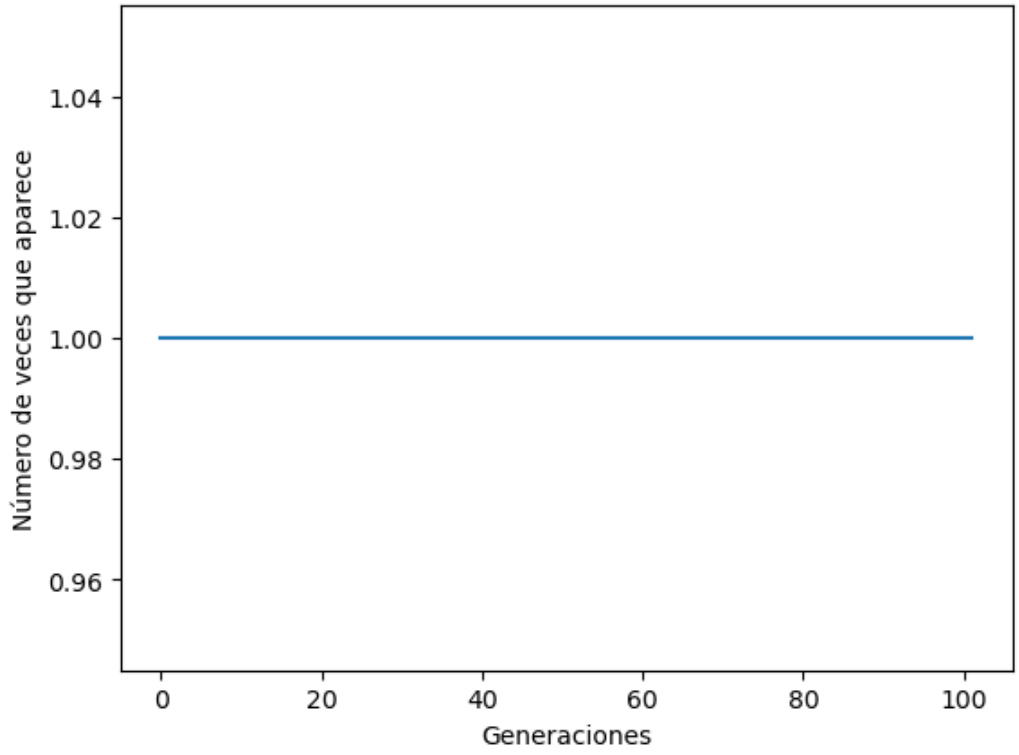

Ilustración 49: Evolución número de veces que aparece en las generaciones con Pc de 0.5 y 0.2 [2]

4.4.4. Pc inicial de 0.5 y Pi de 0.6 se observan los siguientes resultados

En esta situación al aumentar ambos parámetros se observa que la diferencia de ambas aptitudes incrementa conforme aumentan las generaciones, queriendo decir que en cada generación hay más individuos que presentan cambios en su estructura ya se por intercambio de estaciones o por un cambio de estas.

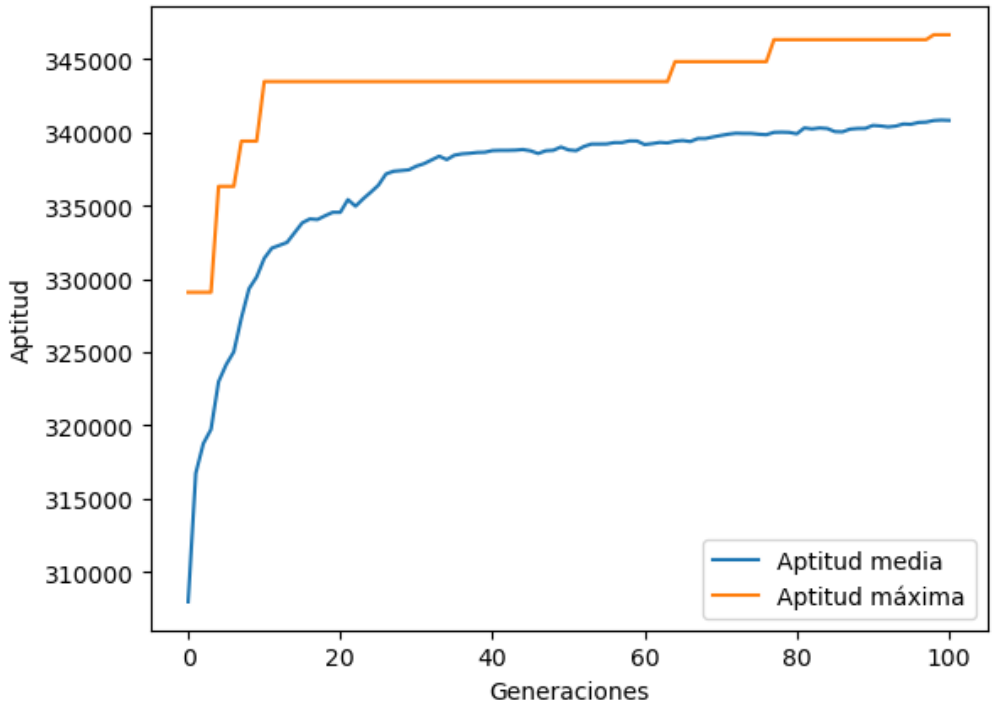

Ilustración 50: Evolución aptitud media y máxima con Pc inicial de 0.5 y Pi de 0.6 [2]

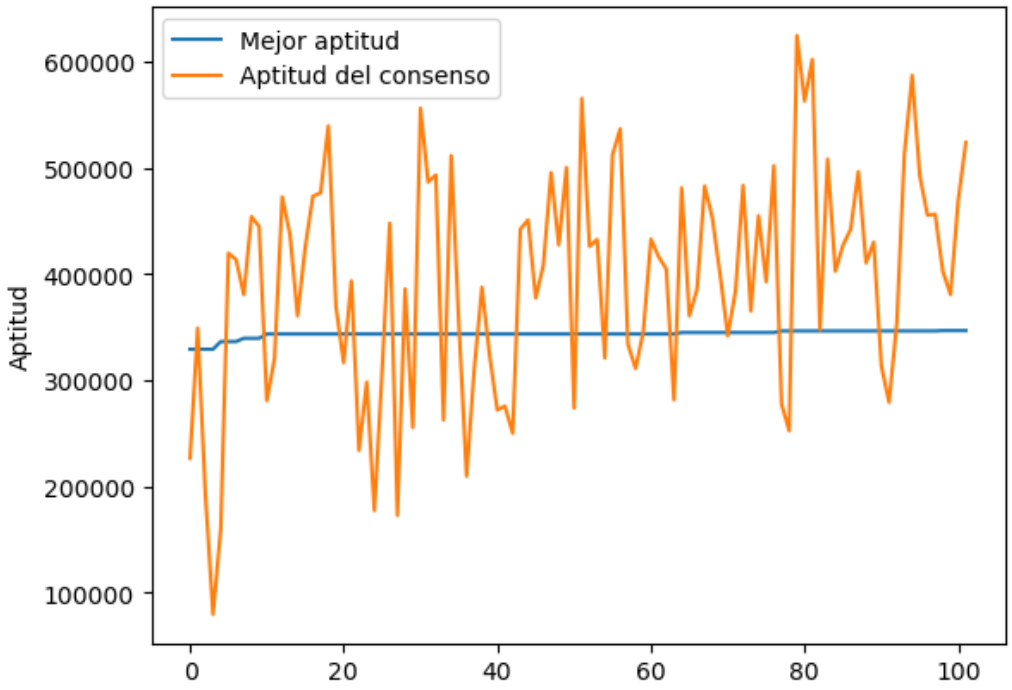

Ilustración 51:Evolución mejor aptitud y aptitud de consenso con Pc inicial de 0.5 y Pi de 0.6 [2]

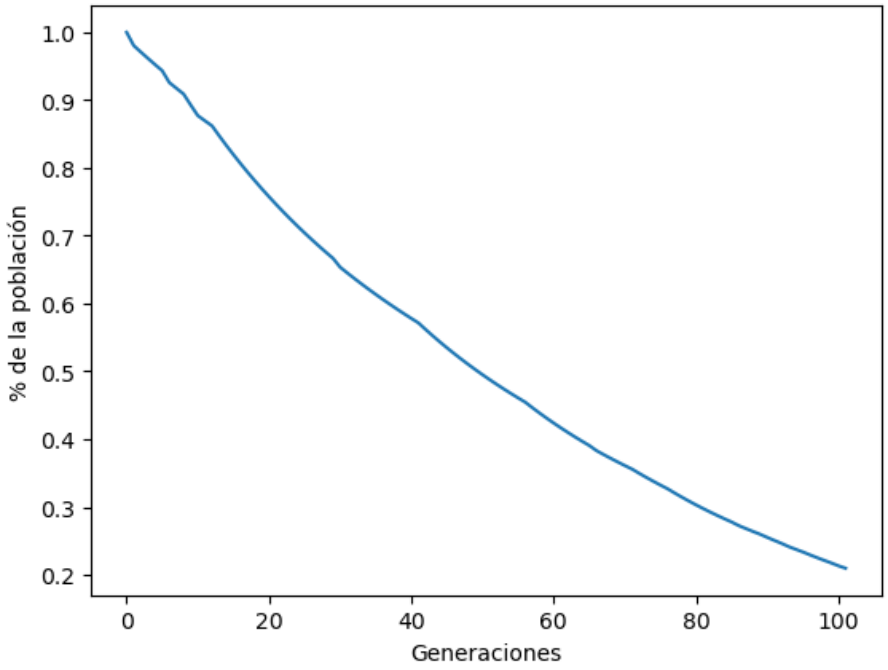

Ilustración 52: Evolución generaciones Pc inicial de 0.5 y Pi de 0.6 [2]

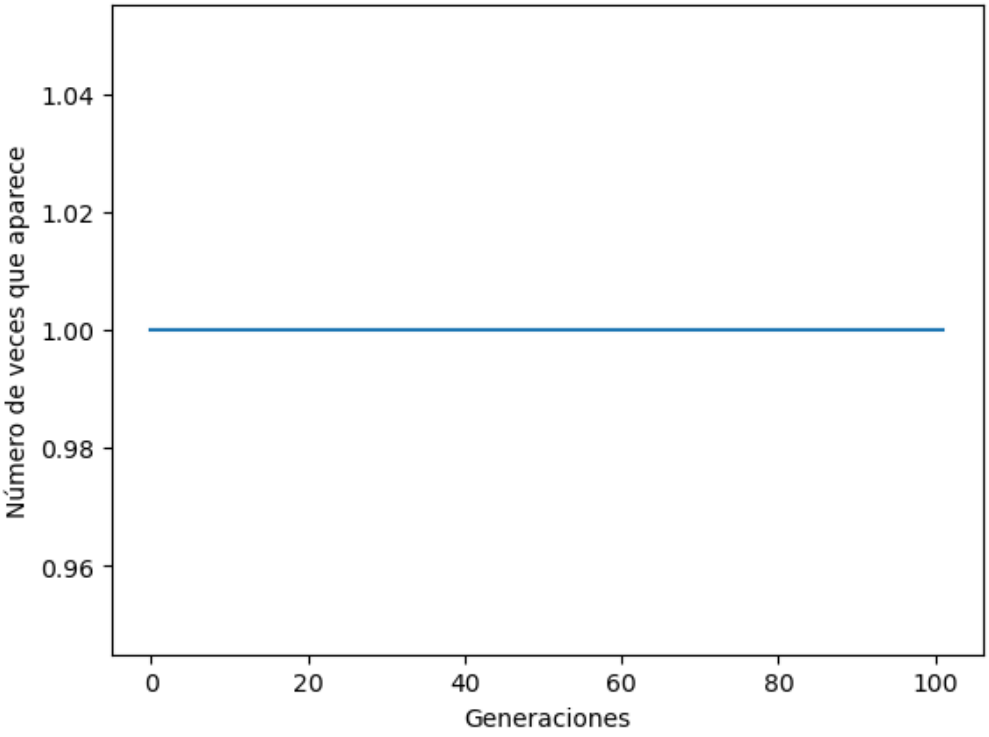

Ilustración 53: Evolución de número de veces que aparece en las generaciones con Pc inicial de 0.5 y Pi de 0.6 [2]

4.4.5. Pc inicial de 0.8 y Pi de 0.75 se observan los siguientes resultados:

Por último, en esta configuración probamos con valores altos en ambas probabilidades, lo que hace que en cada generación se consiga que se inserten nuevos individuos, logrando así que daca generación contenga grandes variaciones respecto a la anterior, todo ello conlleva que ambas aptitudes mantengan una diferencia estable a lo largo de las generaciones, así como hacer que el individuo consenso este disperso respecto a la mejor aptitud de cada generación.

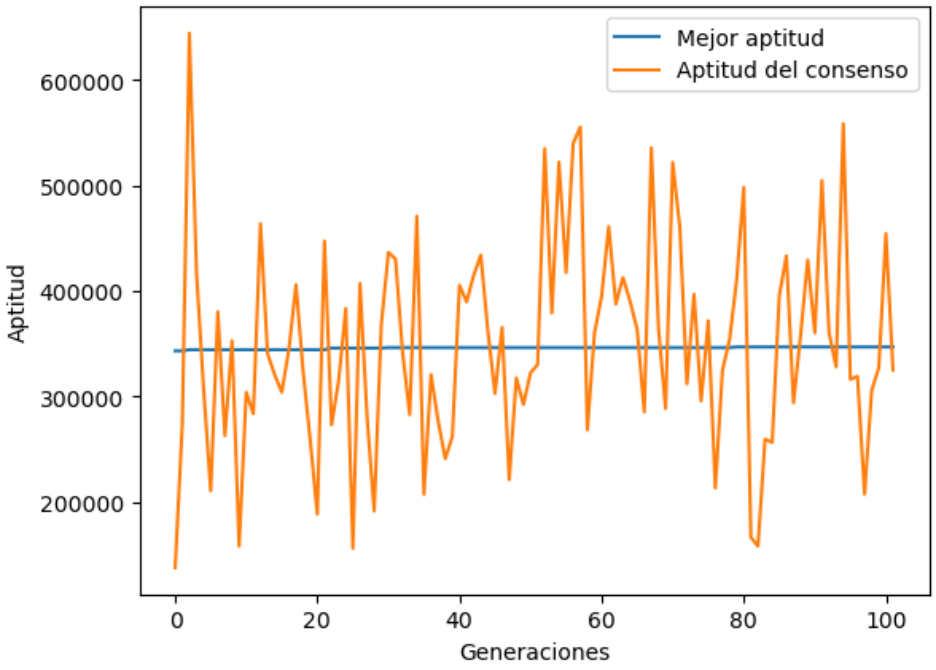

Ilustración 54: Evolución mejor aptitud con la aptitud consenso con Pc inicial de 0.8 y Pi de 0.75 [2]

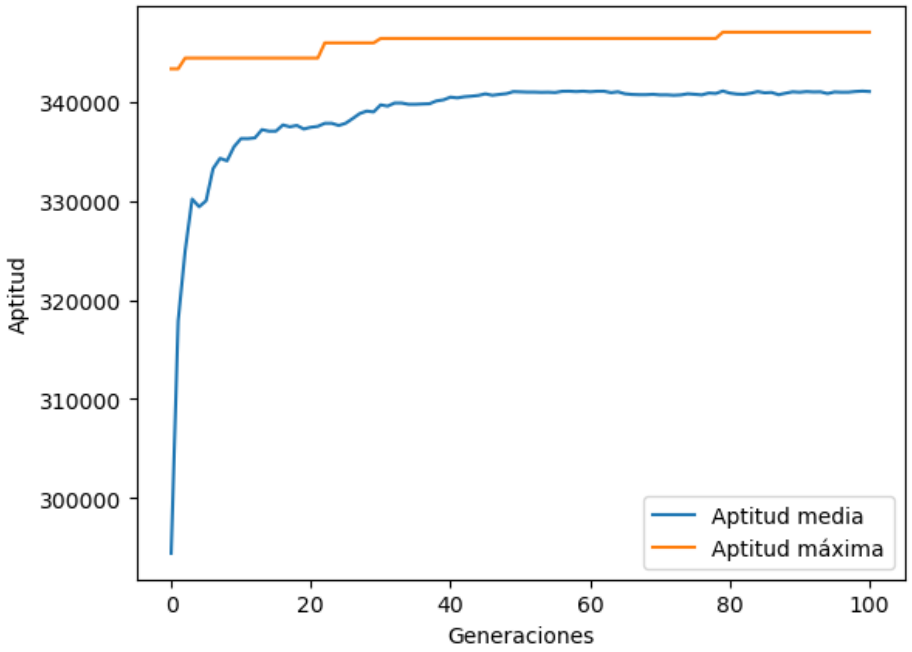

Ilustración 55: Evolución aptitud media y máxima con Pc inicial de 0.8 y Pi de 0.75 [2]

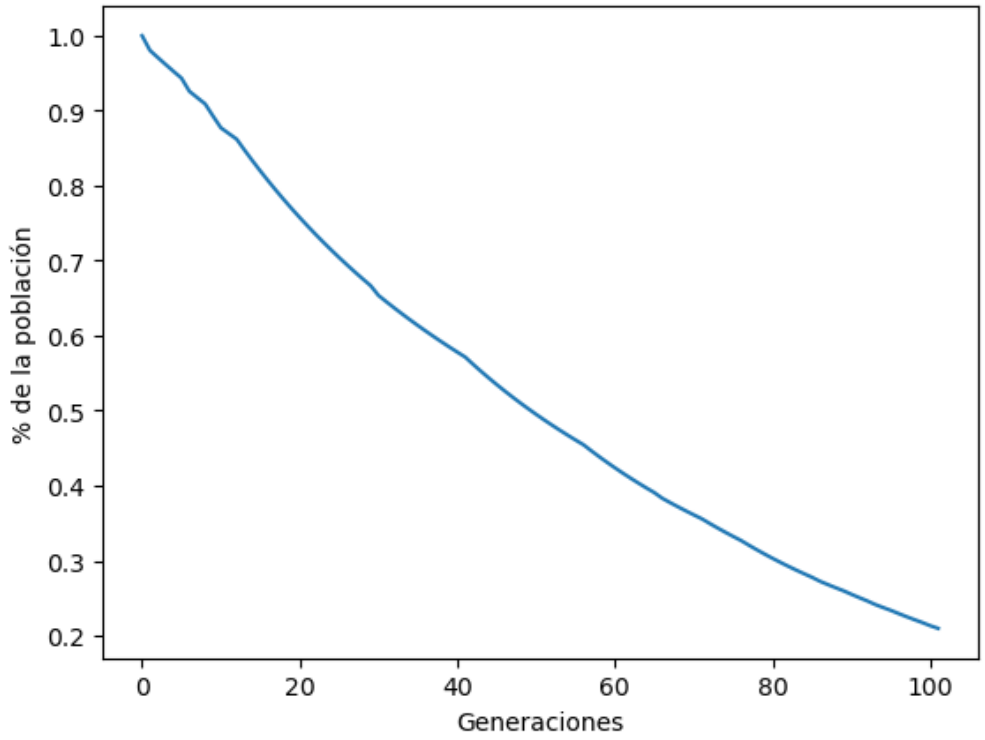

Ilustración 57: Evolución generaciones y % de la población con Pc inicial de 0.8 y Pi de 0.75 [2]

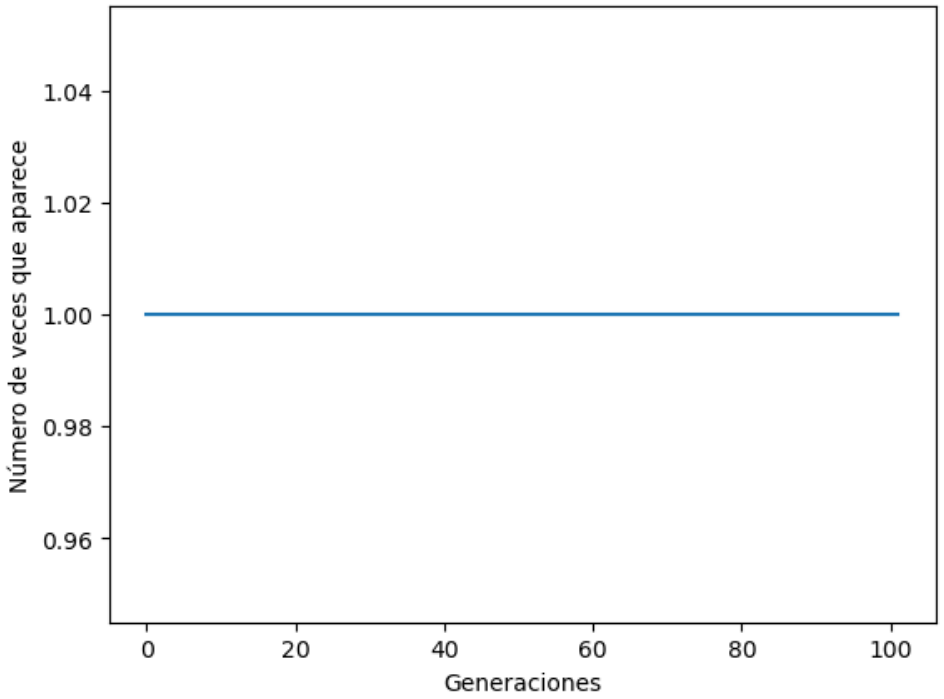

Ilustración 56:Evolución número de veces que aparece y generaciones con Pc inicial de 0.8 y Pi de 0.75 [2]

4.5. Cuestión 5

Prueba con distintos porcentajes de elitismo y encuentra cual es el mejor. Haz una tabla comparativa y explica los detalles de la configuración elegida.

Una vez probado el algoritmo con distintos valores de porcentaje élite, podemos afirmar que el valor óptimo para este parámetro es 0,2, se han hecho pruebas para valores inferiores y superiores a este y no termina de generalizar el algoritmo después de horas ejecutándose.

Se muestra un ejemplo para el valor de p_elite igual a 0,3.

Parámetros

```
    # Asociar cada municipio con un número del 1 al L
    municipios = df_municipios.index + 1
    L = len(municipios)  # Número total de municipios
    E = 12  # Número de estaciones de metro
    cromosoma = np.random.randint(1, L+1, size=E)
    NPOB = 100       # Número de individuos de la población
    NGEN = 100       # Número de generaciones
    #Pc = 0.6       # Probabilidad de cambiar-estación, aplicable a cada gen
    #Pi = 0.4        # Probabilidad de intercambiar-estación, aplicable a cada gen
    NRES = 10       # Cada qué número de generaciones se saca un resumen de la evolución del proceso
    NSAMPLE = 10    # Cada qué número de generaciones se saca un muestreo de la población
    L = len(df_municipios)  # Número de lugares potenciales para construir estaciones
    ALPHA = 0.5  # Valor de alpha para el cálculo de aptitud
    P_elite = 0.3
✓ 0.0s
```

Ilustración 58 Parámetros para p_elite = 0,3 [2]

```
995m 59.9s
Output exceeds the size limit. Open the full output data in a text editor
Parámetros utilizados:
L = 20
E = 12
NPOB = 100
NGEN = 100
Pc = 0.1
Pi = 0.2
------------------------------------------------------------
Tabla de selección de padres para la generación 0:
+-----------------------------------------+--------------------+---------------------+---------------------+
|                Cromosoma                |         Fi         |         Psi         |         Psai        |
+-----------------------------------------+--------------------+---------------------+---------------------+
|  [18, 3, 7, 9, 2, 15, 20, 12, 13, 14, 6, 5] |  334927.768553492  | 0.038303429763609065 | 0.038303429763609065 |
|  [1, 8, 6, 7, 4, 10, 14, 15, 2, 13, 20, 5]  | 334803.4523012864  | 0.03828921255237156  | 0.07659264231598062  |
|  [14, 2, 1, 11, 9, 13, 10, 8, 19, 6, 7, 5]  | 323579.31137253874 | 0.03700558326245656  | 0.11359822557843718  |
|   [16, 13, 9, 14, 3, 2, 1, 6, 4, 7, 18, 5]  | 321564.3889952264  | 0.036775150181043714 |  0.1503733757594809  |
| [13, 11, 20, 8, 5, 2, 10, 6, 14, 19, 18, 7] | 316726.19643469347 | 0.036221838732054104 |  0.186595214491535   |
|   [2, 8, 5, 15, 6, 7, 9, 3, 10, 1, 16, 14]  | 310098.71956554253 | 0.03546389890561382  | 0.22205911339714882  |
|  [12, 15, 10, 13, 16, 6, 1, 3, 5, 8, 9, 7]  | 309518.0652108704  | 0.035397493383649566 |  0.2574566067807984  |
|  [13, 6, 9, 7, 2, 18, 5, 8, 15, 12, 20, 19] | 308595.75526844495 | 0.03529201501661972  |  0.2927486217974181  |
|  [5, 9, 8, 7, 15, 18, 19, 6, 2, 20, 16, 14] | 306854.6505078116  | 0.035092896609095516 | 0.32784151840651365  |
| [1, 19, 7, 20, 6, 17, 15, 5, 9, 12, 13, 11] | 302671.42069333635 | 0.03461448883157439  |  0.362456007238088   |
| [5, 6, 15, 17, 12, 2, 16, 7, 8, 11, 14, 10] | 301489.59715192846 | 0.03447933164467764  | 0.39693533888276566  |
| [19, 18, 15, 14, 10, 6, 5, 3, 17, 7, 12, 1] | 301233.8080689055  | 0.034450078772578725 |  0.4313854176553444  |
|  [12, 1, 15, 14, 11, 3, 16, 10, 7, 8, 6, 5] | 298332.7765101059  |  0.0341183073609863  |  0.4655037250163307  |
...
[10, 18, 15, 4, 2, 5, 7, 14, 9, 17, 13, 6]
[17, 13, 12, 19, 2, 20, 10, 15, 4, 14, 5, 9]
[9, 10, 2, 15, 5, 18, 4, 7, 19, 11, 6, 13]
```

Ilustración 59 Demostración de tiempo de ejecución para 0,3 de P_elite [2]

Para el valor de 0,12 consigue generalizar el algoritmo para los primeros valores de Pc y Pi. Los resultados son los siguientes.

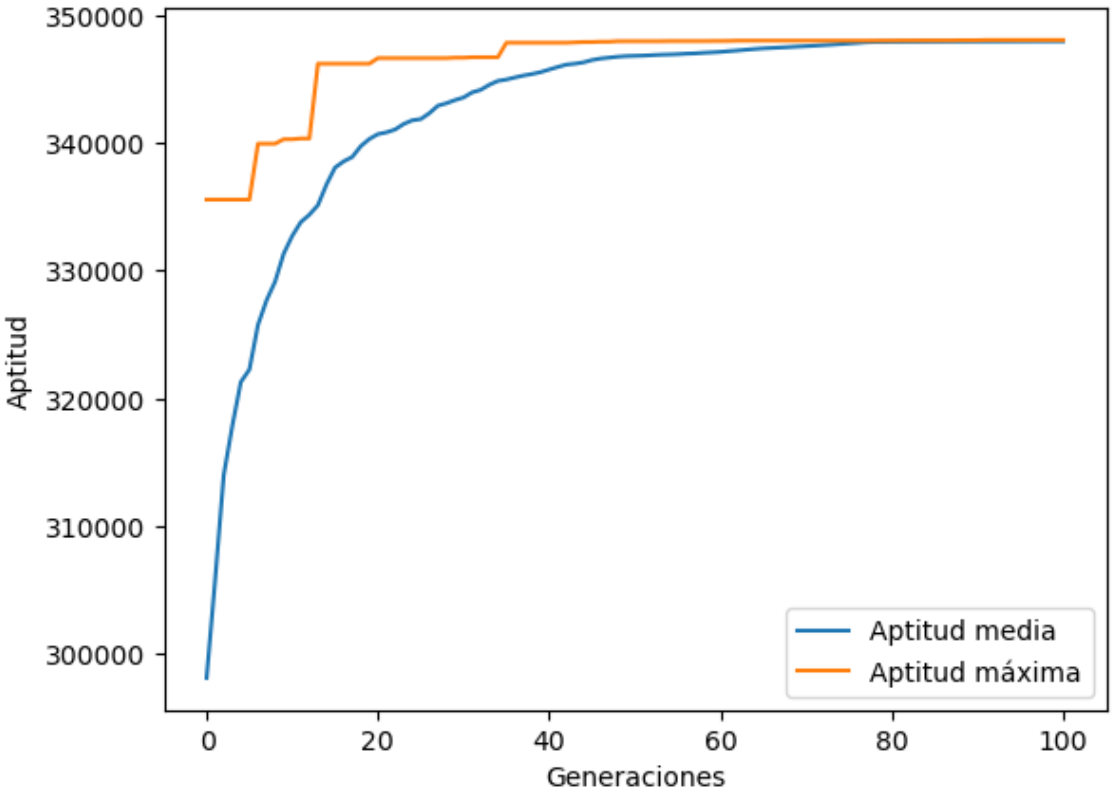

Ilustración 60 Gráfico de evolución de la aptitud media y máxima para p_elite = 0,12 [2]

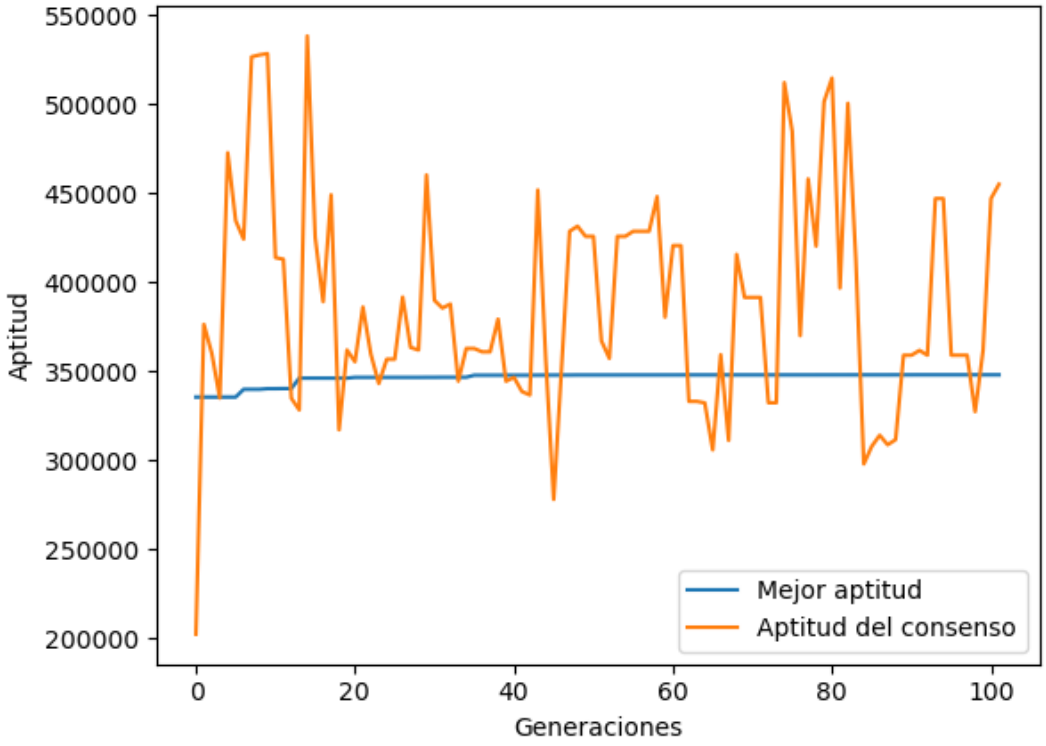

Ilustración 61 Gráfico de evolución de los mejores resultados p_elite = 0,12 [2]

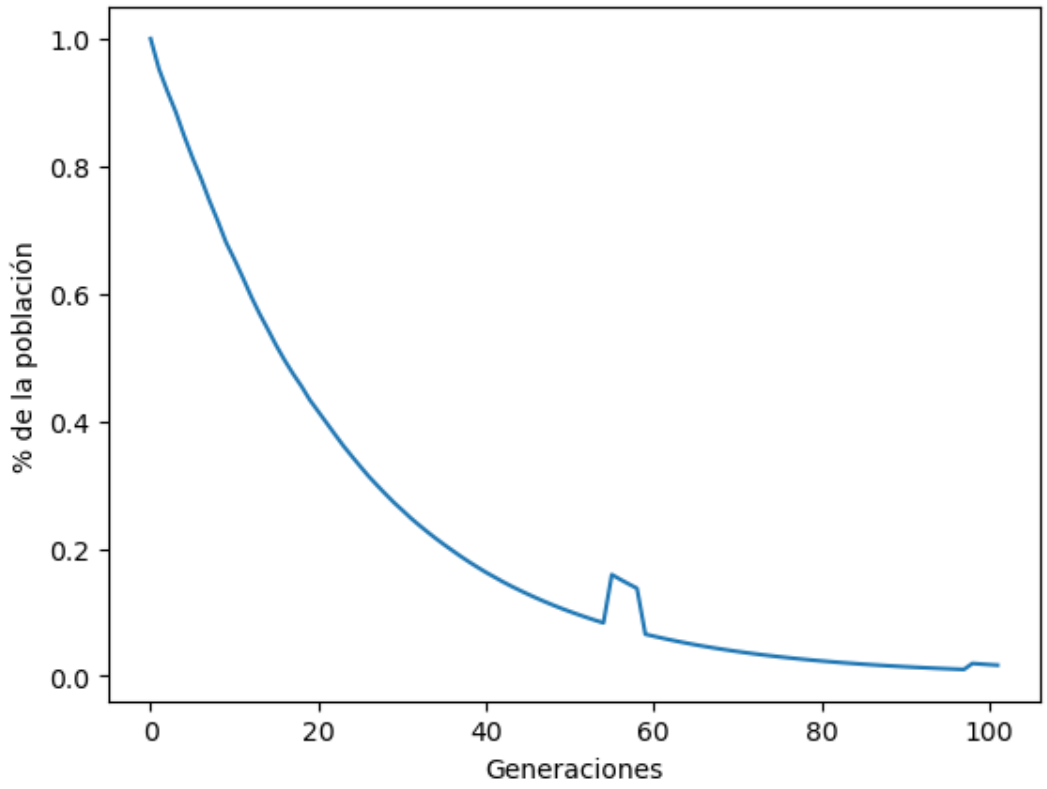

Ilustración 62 Gráfico de evolución del porcentaje de la población que representa el mejor cromosoma p_elite = 0,12 [2]

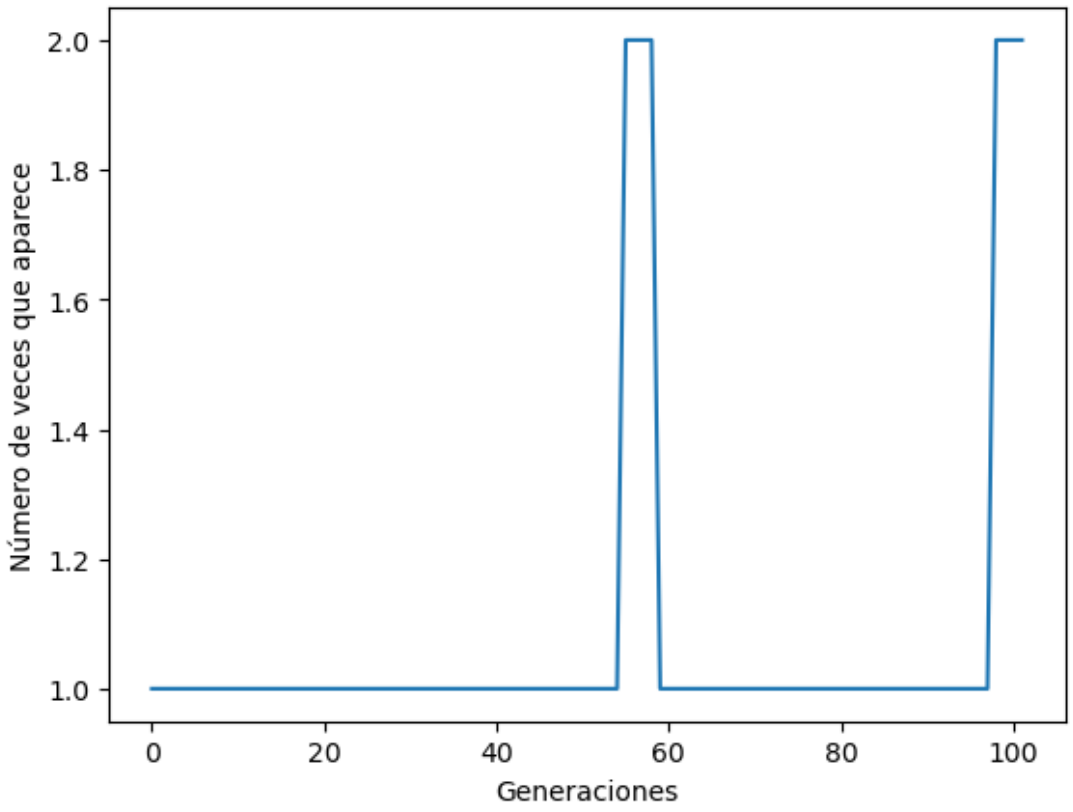

Ilustración 63 Gráfico de evolución del número de veces que aparece el mejor cromosoma en la población p_elite = 0,12 [2]

Los resultados indican que para una tasa de selección de élite del 12%, el algoritmo genético convergió a una solución óptima para el problema dado, aunque con cierta variabilidad en la aptitud de consenso y en el porcentaje de la población que representa el mejor cromosoma.

La convergencia de la aptitud media y máxima a partir de la generación 80 sugiere que una cantidad suficiente de generaciones es necesaria para que el algoritmo converja en este caso particular. Además, el hecho de que la aptitud máxima alcance su pico máximo en la generación 40 indica que el algoritmo necesitó más generaciones para encontrar la solución óptima.

En cuanto a la evolución de los mejores resultados, el hecho de que la mejor aptitud se mantenga estable a partir de la generación 20 sugiere que el algoritmo ha encontrado una solución óptima en una región del espacio de búsqueda. Sin embargo, la aptitud de consenso finaliza con valores mucho más altos que la mejor aptitud, lo que indica que hay cierta variabilidad en las soluciones que se generan a lo largo del proceso.

El gráfico de evolución del porcentaje de la población que representa el mejor cromosoma indica que la búsqueda ha sido exploratoria, ya que el porcentaje de población que representa el mejor cromosoma no asciende del 1%. El pico entre la generación 50 y 60 sugiere que la población se enfoca en un solo individuo por un tiempo limitado antes de explorar otras soluciones.

Por último, el gráfico de evolución del número de veces que aparece el mejor cromosoma en la población indica que existe siempre un mejor individuo, aunque hay momentos en los que se encuentra con mayor frecuencia, como en las generaciones 57-60 y 98-100.

En resumen, estos resultados indican que la tasa de selección de élite del 12% es suficiente para obtener una solución óptima para el problema en cuestión, aunque con cierta variabilidad en los resultados y una búsqueda exploratoria del espacio de búsqueda.

En el primer conjunto de resultados, se evaluaron diferentes valores de probabilidad de cambio y de intercambio de estaciones, y se encontró que al aumentar ambas probabilidades se incrementa la variabilidad en los individuos y las generaciones, manteniendo una diferencia estable entre la aptitud media y máxima. Además, se observó que el individuo consenso se alejaba de la mejor aptitud en algunos casos.

En el segundo conjunto de resultados, se evaluó el efecto de una tasa de selección de élite del 12% en la convergencia y variabilidad de las soluciones. Se encontró que la convergencia a una solución óptima se alcanzó después de suficientes generaciones, con cierta variabilidad en la aptitud de consenso y el porcentaje de la población representado por el mejor cromosoma. También se observó una búsqueda exploratoria del espacio de búsqueda, con momentos en los que la población se enfocaba en un solo individuo antes de explorar otras soluciones.

En general, estos resultados muestran que diferentes valores de parámetros en los algoritmos genéticos pueden tener efectos significativos en la convergencia y variabilidad de las soluciones encontradas, y que se requiere un análisis cuidadoso para encontrar los mejores valores de los parámetros para un problema particular.

5. Conclusiones

En esta práctica se han implementado algoritmos genéticos para la resolución de distintos problemas, con el objetivo de entender y analizar su comportamiento en función de los parámetros que los definen.

En primer lugar, se ha aplicado un algoritmo genético para la resolución de un problema de optimización de funciones, donde se ha estudiado la influencia de los parámetros del algoritmo en la calidad de la solución obtenida. Se ha observado que la elección de una adecuada tasa de mutación y una tasa de cruce moderada pueden mejorar significativamente la capacidad de exploración del espacio de búsqueda y la eficiencia en la convergencia hacia la solución óptima.

En segundo lugar, se ha aplicado un algoritmo genético para resolver el problema del viajante de comercio, donde se ha analizado el efecto de la tasa de cruce y la tasa de mutación en la calidad de la solución obtenida. Los resultados han mostrado que un valor alto de tasa de cruce y una tasa de mutación moderada son las configuraciones que ofrecen mejores resultados.

En tercer lugar, se ha estudiado el comportamiento de un algoritmo genético para la resolución de un problema de programación de tareas en máquinas paralelas. En este caso, se ha observado que un valor de tasa de cruce y tasa de mutación moderados resultan en una convergencia rápida y en una buena calidad de la solución.

Por último, se ha aplicado un algoritmo genético para resolver un problema de clasificación de dígitos escritos a mano. Se ha estudiado la influencia de la tasa de cruce y la tasa de mutación en la precisión de la clasificación obtenida. Los resultados indican que una tasa de cruce y tasa de mutación moderados son adecuados para obtener una precisión alta en la clasificación de los dígitos.

En general, los resultados obtenidos en cada uno de los problemas estudiados muestran que los algoritmos genéticos son una herramienta efectiva para la resolución de problemas de optimización y aprendizaje automático, y que la selección de los parámetros adecuados es crucial para su correcto funcionamiento.

6. Bibliografía

[1] A. G. Tejedor, *Apuntes clase IA.II capítulo 8,* Madrid: Univeresidad Francisco de Vitoria, 2023.

[2] F. Propia, Madrid, 2023.

7. Anexo A

INSTALACIÓN DE LIBRRÍAS PARA LA BUENA EJECUCIÓN DE LA PRÁCTICA

Para instalar las librerías necesarias, abre una terminal o línea de comandos en tu sistema operativo y ejecuta el siguiente comando:

- pip install pandas numpy matplotlib prettytable

Este comando instalará todas las librerías necesarias para ejecutar el código proporcionado.

Para la librería random y copy no es necesario instalar nada adicional, ya que son librerías integradas en Python y vienen preinstaladas por defecto en la mayoría de los sistemas operativos.

Una vez que se hayan instalado todas las librerías, puedes ejecutar el script Python. Si todo funciona correctamente, el script Python debería ejecutarse sin problemas y producir los resultados esperados.

Si tienes algún problema para instalar o utilizar alguna de las librerías, consulta la documentación oficial o la comunidad de desarrolladores correspondiente para obtener ayuda.