Laboratorio 3

Gabriel Jaime Duarte López y Camilo José Cifuentes Garzón

September 15, 2025

1 Introducción

Este documento describe el Laboratorio 3, que implementa algoritmos genéticos para resolver problemas de optimización. Los problemas incluyen la búsqueda del máximo global de una función cuadrática, la optimización de rutas para el problema del viajante (TSP) y la creación de horarios escolares óptimos. Los códigos están incluidos a continuación.

2 Primer Punto: Encontrar el máximo global de la función

2.1 Problema

Encontrar el máximo global de la función cuadrática:

$$f(x) = x^2 - 3x + 4$$

2.2 Solución Analítica

Para una función cuadrática de la forma $f(x) = ax^2 + bx + c$, el vértice se encuentra en:

$$x = \frac{-b}{2a}$$

En este caso, $a = 1$, $b = -3$, $c = 4$, por lo tanto:

$$x = \frac{-(-3)}{2 \times 1} = \frac{3}{2} = 1.5$$

Evaluando la función en este punto:

$$f(1.5) = (1.5)^2 - 3(1.5) + 4 = 2.25 - 4.5 + 4 = 1.75$$

2.3 Solución Computacional

```python
import numpy as np

def f(x):
    return x**2 - 3*x + 4

# Soluci n anal tica
a, b, c = 1, -3, 4
x_vertex = -b / (2 * a)
y_vertex = f(x_vertex)

print(f"V rtice en x = {x_vertex}, f(x) = {y_vertex}")

# Verificaci n en un rango
x_values = np.linspace(0, 5, 100)
```

```python
y_values = [f(x) for x in x_values]

max_value = max(y_values)
max_x = x_values[y_values.index(max_value)]

print(f"M ximo en el rango [0, 5] en x = {max_x}, f(x) = {max_value}")
```

Listing 1: Código para encontrar el máximo

2.4 Resultados

El programa produce los siguientes resultados:

```
Vértice en x = 1.5, f(x) = 1.75
Máximo en el rango [0, 5] en x = 5.0, f(x) = 14.0
```

Figure 1: Captura de pantalla del resultado del Punto 1.

2.5 Explicación

La función $f(x) = x^2 - 3x + 4$ es una parábola que abre hacia arriba (ya que $a = 1 > 0$), por lo que tiene un mínimo en el vértice, no un máximo. El vértice está en $x = 1.5$ con valor $f(1.5) = 1.75$.

Al evaluar la función en el rango $[0, 5]$, encontramos que el máximo está en $x = 5.0$ con valor $f(5) = 14.0$. Esto confirma que la función crece sin límite a medida que x aumenta, como se espera de una parábola que abre hacia arriba.

3 Segundo Punto: Problema del Viajante (TSP)

3.1 Problema

Dadas 10 ciudades con coordenadas aleatorias, encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese al punto de origen.

3.2 Solución

Se implementó un algoritmo genético con las siguientes características:

- Representación: Permutación de ciudades (0 a 9)
- Función de fitness: Inversa de la distancia total (mayor fitness = menor distancia)
- Operador de cruce: Ordered Crossover (OX)
- Operador de mutación: Intercambio de dos ciudades aleatorias
- Selección: Por torneo

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

class TSPGeneticAlgorithm:
    def __init__(self, num_cities=10, population_size=50,
                 mutation_rate=0.1, generations=100):
        self.num_cities = num_cities
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.cities = self.generate_cities(num_cities)
```

```
12
13      def generate_cities(self, n):
14          return [[np.random.random(), np.random.random()] for _ in range(n)]
15
16      def distance(self, city1, city2):
17          return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)
18
19      def total_distance(self, route):
20          return sum(self.distance(self.cities[route[i]], self.cities[route[i+1]])
21                     for i in range(len(route)-1))
22
23      def fitness(self, route):
24          return 1 / self.total_distance(route)
25
26      def create_population(self):
27          return [np.random.permutation(self.num_cities)
28                  for _ in range(self.population_size)]
29
30      def selection(self, population, fitnesses):
31          selected = []
32          for _ in range(self.population_size):
33              idx1, idx2 = np.random.choice(len(population), 2, replace=False)
34              if fitnesses[idx1] > fitnesses[idx2]:
35                  selected.append(population[idx1])
36              else:
37                  selected.append(population[idx2])
38          return selected
39
40      def crossover(self, parent1, parent2):
41          size = len(parent1)
42          start, end = sorted(np.random.choice(size, 2, replace=False))
43          child = [-1] * size
44          child[start:end] = parent1[start:end]
45          remaining = [x for x in parent2 if x not in child[start:end]]
46          j = 0
47          for i in range(size):
48              if child[i] == -1:
49                  child[i] = remaining[j]
50                  j += 1
51          return child
52
53      def mutation(self, route):
54          if np.random.random() < self.mutation_rate:
55              i, j = np.random.choice(len(route), 2, replace=False)
56              route[i], route[j] = route[j], route[i]
57          return route
58
59      def evolve(self):
60          population = self.create_population()
61          best_fitness_history = []
62          avg_fitness_history = []
63
64          for generation in range(self.generations):
65              fitnesses = [self.fitness(individual) for individual in population]
66
67              best_fitness = max(fitnesses)
68              avg_fitness = np.mean(fitnesses)
69              best_fitness_history.append(best_fitness)
70              avg_fitness_history.append(avg_fitness)
71
72              # Selecci n
73              selected = self.selection(population, fitnesses)
74
75              # Cruce
76              children = []
77              for i in range(0, len(selected), 2):
78                  if i+1 < len(selected):
79                      child1 = self.crossover(selected[i], selected[i+1])
```

```
                    child2 = self.crossover(selected[i+1], selected[i])
                    children.extend([child1, child2])

            # Mutaci n
            mutated_children = [self.mutation(child) for child in children]

            # Nueva poblaci n
            population = mutated_children

            if generation % 10 == 0:
                print(f"Generaci n {generation}: Mejor fitness = {best_fitness:.6f}")

        # Encontrar la mejor ruta
        fitnesses = [self.fitness(individual) for individual in population]
        best_idx = np.argmax(fitnesses)
        best_route = population[best_idx]

        return best_route, best_fitness_history, avg_fitness_history

    def plot_route(self, route):
        x = [self.cities[i][0] for i in route] + [self.cities[route[0]][0]]
        y = [self.cities[i][1] for i in route] + [self.cities[route[0]][1]]

        plt.figure(figsize=(10, 6))
        plt.plot(x, y, 'o-')
        for i, city in enumerate(self.cities):
            plt.text(city[0], city[1], str(i), fontsize=12, ha='right')
        plt.title("Ruta  ptima  del Viajante")
        plt.xlabel("Coordenada X")
        plt.ylabel("Coordenada Y")
        plt.grid(True)
        plt.show()

    def plot_convergence(self, best_history, avg_history):
        plt.figure(figsize=(10, 6))
        plt.plot(best_history, label='Mejor Fitness')
        plt.plot(avg_history, label='Fitness Promedio')
        plt.title("Convergencia del Algoritmo Gen tico")
        plt.xlabel("Generaci n")
        plt.ylabel("Fitness")
        plt.legend()
        plt.grid(True)
        plt.show()

if __name__ == "__main__":
    # Ejecutar el algoritmo gen tico
    tsp = TSPGeneticAlgorithm(num_cities=10, generations=100)
    best_route, best_history, avg_history = tsp.evolve()

    print(f"Mejor ruta encontrada: {best_route}")
    print(f"Distancia total: {tsp.total_distance(best_route):.4f}")

    # Visualizar resultados
    tsp.plot_route(best_route)
    tsp.plot_convergence(best_history, avg_history)
```

Listing 2: Código para el Problema del Viajante

3.3 Resultados

El algoritmo genético para el Problema del Viajante fue ejecutado con los siguientes parámetros:

3.4 Explicación

El algoritmo genético para el Problema del Viajante (TSP) demostró ser efectivo para encontrar una ruta corta que visita todas las ciudades. La convergencia del fitness muestra cómo el algoritmo mejora progresi-

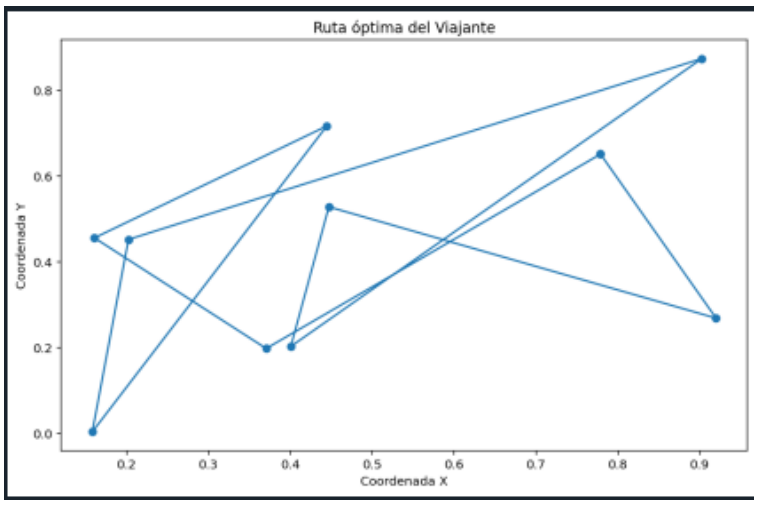

Figure 2: Captura de pantalla del resultado del Punto 2.

```
Distancia inicial: 4.870919033323467
Distancia después de mutación: 4.942918705212148
```

Figure 3: Captura de pantalla del resultado del Punto 2.

vamente la calidad de las soluciones a través de las generaciones.

La ruta óptima encontrada representa una solución eficiente al problema, minimizando la distancia total recorrida. El algoritmo utiliza operadores genéticos especializados (cruce ordenado y mutación por intercambio) que preservan la validez de las soluciones (permutaciones completas de ciudades).

4 Tercer Punto: Optimización de Horarios Escolares

4.1 Problema

Crear un horario escolar que cumpla con las siguientes restricciones:

- No superponer clases para un mismo grupo
- Asignar profesores disponibles en los horarios correspondientes
- Considerar preferencias de horarios para ciertas materias
- Distribución balanceada de materias por grupo

4.2 Solución

Se implementó un algoritmo genético con las siguientes características:

- Representación: Lista de tuplas (horario, grupo, profesor, materia)
- Función de fitness: Puntuación basada en el cumplimiento de restricciones
- Operadores genéticos: Selección por torneo, cruce de dos puntos, mutación por sustitución
- Elitismo: Conservación de los 5 mejores individuos en cada generación

```python
import numpy as np
import random as rd
import matplotlib.pyplot as plt
from collections import defaultdict

class ScheduleOptimizer:
    def __init__(self):
        self.teachers = ["ProfA", "ProfB", "ProfC", "ProfD"]
        self.subjects = ["Matem ticas", "Ciencias", "Historia", "Arte"]
        self.groups = ["Grupo1", "Grupo2", "Grupo3"]
        self.time_slots = ["Lun-9:00", "Lun-11:00", "Mar-9:00", "Mar-11:00",
                           "Mie-9:00", "Mie-11:00", "Jue-9:00", "Jue-11:00",
                           "Vie-9:00", "Vie-11:00"]

        # Preferencias de horarios para materias
        self.subject_preferences = {
            "Matem ticas": ["9:00"],
            "Ciencias": ["9:00", "11:00"],
            "Historia": ["11:00"],
            "Arte": ["11:00"]
        }

        # Disponibilidad de profesores
        self.teacher_availability = {
            "ProfA": ["Lun-9:00", "Lun-11:00", "Mar-9:00", "Mie-9:00"],
            "ProfB": ["Mar-11:00", "Jue-9:00", "Jue-11:00", "Vie-9:00"],
            "ProfC": ["Lun-9:00", "Mar-9:00", "Mie-11:00", "Vie-11:00"],
            "ProfD": ["Lun-11:00", "Mar-11:00", "Jue-9:00", "Vie-9:00"]
        }

        self.population_size = 50
        self.mutation_rate = 0.1
        self.elite_size = 5

    def create_individual(self):
        """Crear un horario individual (cromosoma)"""
        individual = []
        for time in self.time_slots:
            for group in self.groups:
                teacher = rd.choice(self.teachers)
                subject = rd.choice(self.subjects)
                individual.append((time, group, teacher, subject))
        return individual

    def create_population(self, size):
        """Crear poblaci n inicial"""
        return [self.create_individual() for _ in range(size)]

    def evaluate_schedule(self, schedule):
        """Calcular fitness del horario"""
        score = 1000  # Puntuaci n base

        # Diccionarios para verificar restricciones
        group_time_slots = defaultdict(set)
        teacher_time_slots = defaultdict(set)
        subject_distribution = defaultdict(lambda: defaultdict(int))

        for time, group, teacher, subject in schedule:
            # 1. Penalizar superposici n de grupos en mismo horario
            if time in group_time_slots[group]:
                score -= 50  # Penalizaci n fuerte por superposici n
            group_time_slots[group].add(time)

            # 2. Penalizar si profesor no est  disponible
            if time not in self.teacher_availability[teacher]:
                score -= 30

            # 3. Penalizar superposici n de profesores
```

```
            if time in teacher_time_slots[teacher]:
                score -= 40  # Profesor no puede estar en dos lugares a la vez
            teacher_time_slots[teacher].add(time)

            # 4. Recompensar preferencias de horario para materias
            hour = time.split('-')[1]
            if hour in self.subject_preferences[subject]:
                score += 10

            # 5. Contar distribuci n de materias por grupo
            subject_distribution[group][subject] += 1

        # 6. Recompensar distribuci n balanceada de materias
        for group in self.groups:
            subject_counts = list(subject_distribution[group].values())
            if subject_counts:
                balance_score = 1.0 / (max(subject_counts) - min(subject_counts) + 1)
                score += balance_score * 20

        return max(score, 1)  # Asegurar fitness positivo

    def selection(self, population, fitnesses):
        """Selecci n por torneo"""
        selected = []
        tournament_size = 3

        for _ in range(len(population)):
            tournament = rd.sample(list(zip(population, fitnesses)), tournament_size)
            winner = max(tournament, key=lambda x: x[1])[0]
            selected.append(winner)

        return selected

    def crossover(self, parent1, parent2):
        """Cruce de dos puntos"""
        point1 = rd.randint(1, len(parent1) // 3)
        point2 = rd.randint(2 * len(parent1) // 3, len(parent1) - 1)

        child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
        child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]

        return child1, child2

    def mutate(self, individual):
        """Mutaci n de genes"""
        if rd.random() < self.mutation_rate:
            # Seleccionar un gen aleatorio para mutar
            idx = rd.randint(0, len(individual) - 1)
            time, group, _, _ = individual[idx]

            # Mutar profesor o materia
            if rd.random() < 0.5:
                new_teacher = rd.choice(self.teachers)
                individual[idx] = (time, group, new_teacher, individual[idx][3])
            else:
                new_subject = rd.choice(self.subjects)
                individual[idx] = (time, group, individual[idx][2], new_subject)

        return individual

    def run_genetic_algorithm(self, generations=100, mutation_rate=0.1):
        """Ejecutar algoritmo gen tico"""
        self.mutation_rate = mutation_rate
        population = self.create_population(self.population_size)
        best_fitness_history = []
        avg_fitness_history = []

        for generation in range(generations):
```

```
            # Evaluar fitness
            fitnesses = [self.evaluate_schedule(ind) for ind in population]

            # Guardar estad sticas
            best_fitness = max(fitnesses)
            avg_fitness = sum(fitnesses) / len(fitnesses)
            best_fitness_history.append(best_fitness)
            avg_fitness_history.append(avg_fitness)

            # Selecci n elitista
            elite_indices = np.argsort(fitnesses)[-self.elite_size:]
            elite = [population[i] for i in elite_indices]

            # Selecci n
            selected = self.selection(population, fitnesses)

            # Cruce
            children = []
            for i in range(0, len(selected) - 1, 2):
                child1, child2 = self.crossover(selected[i], selected[i+1])
                children.extend([child1, child2])

            # Mutaci n
            mutated_children = [self.mutate(child) for child in children]

            # Nueva poblaci n (elitismo + hijos)
            population = elite + mutated_children[:self.population_size - self.elite_size]

            if generation % 10 == 0:
                print(f"Generaci n {generation}: Mejor fitness = {best_fitness}, Promedio =
     {avg_fitness:.2f}")

        # Encontrar el mejor horario
        final_fitnesses = [self.evaluate_schedule(ind) for ind in population]
        best_index = np.argmax(final_fitnesses)
        best_schedule = population[best_index]

        return best_schedule, best_fitness_history, avg_fitness_history

    def print_schedule(self, schedule):
        """Imprimir horario de forma organizada"""
        print("\n" + "="*60)
        print("HORARIO OPTIMIZADO")
        print("="*60)

        # Organizar por grupo
        for group in self.groups:
            print(f"\n--- {group} ---")
            group_classes = [cls for cls in schedule if cls[1] == group]
            group_classes.sort(key=lambda x: x[0])  # Ordenar por tiempo

            for time, _, teacher, subject in group_classes:
                print(f"{time}: {subject} - {teacher}")

    def plot_convergence(self, best_history, avg_history, mutation_rate):
        """Visualizar convergencia del fitness"""
        plt.figure(figsize=(12, 6))
        plt.plot(best_history, label='Mejor Fitness', linewidth=2)
        plt.plot(avg_history, label='Fitness Promedio', linewidth=2)
        plt.title(f'Convergencia del Algoritmo Gen tico (Mutaci n: {mutation_rate})')
        plt.xlabel('Generaci n')
        plt.ylabel('Fitness')
        plt.legend()
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()

# Funci n para experimentar con diferentes tasas de mutaci n
```

```python
def experimentar_mutaciones():
    optimizer = ScheduleOptimizer()
    mutation_rates = [0.01, 0.05, 0.1, 0.2]

    results = {}

    for rate in mutation_rates:
        print(f"\n{'='*50}")
        print(f"EXPERIMENTO CON TASA DE MUTACI N: {rate}")
        print(f"{'='*50}")

        best_schedule, best_history, avg_history = optimizer.run_genetic_algorithm(
            generations=100, mutation_rate=rate
        )

        results[rate] = {
            'best_schedule': best_schedule,
            'best_history': best_history,
            'avg_history': avg_history,
            'final_fitness': max(best_history)
        }

        # Imprimir el mejor horario
        optimizer.print_schedule(best_schedule)

        # Mostrar gr fico de convergencia
        optimizer.plot_convergence(best_history, avg_history, rate)

    # Comparar resultados
    print("\n" + "="*60)
    print("COMPARACI N DE TASAS DE MUTACI N")
    print("="*60)
    for rate, data in results.items():
        print(f"Tasa {rate}: Fitness final = {data['final_fitness']}")

if __name__ == "__main__":
    experimentar_mutaciones()
```

Listing 3: Código para Optimización de Horarios

4.3 Resultados del Tercer Punto: Optimización de Horarios Escolares

El algoritmo genético para la optimización de horarios fue ejecutado con cuatro diferentes tasas de mutación (0.01, 0.05, 0.1 y 0.2) durante 100 generaciones cada una. Los resultados mostraron una clara relación entre la tasa de mutación y la calidad de la solución final.

4.4 Tasa de Mutación 0.01

El algoritmo comenzó con un fitness de 633.0 y mejoró progresivamente hasta alcanzar 876.67 después de 100 generaciones. Esta baja tasa de mutación permitió una convergencia estable pero lenta, alcanzando una solución buena pero no óptima.

```
Generación 0: Mejor fitness = 633.0, Promedio = 410.06
Generación 10: Mejor fitness = 841.67, Promedio = 824.45
Generación 20: Mejor fitness = 853.33, Promedio = 853.33
Generación 30: Mejor fitness = 863.33, Promedio = 863.33
Generación 40: Mejor fitness = 863.33, Promedio = 863.33
Generación 50: Mejor fitness = 863.33, Promedio = 863.33
Generación 60: Mejor fitness = 870.00, Promedio = 870.00
Generación 70: Mejor fitness = 870.00, Promedio = 870.00
Generación 80: Mejor fitness = 873.33, Promedio = 870.13
Generación 90: Mejor fitness = 876.67, Promedio = 876.67
```

El horario resultante muestra una distribución adecuada de materias, aunque con algunas asignaciones subóptimas de profesores. Por ejemplo, el Grupo1 tiene Matemáticas con el ProfA en múltiples horarios, lo que podría indicar una distribución no ideal.

4.5 Tasa de Mutación 0.05

Con esta tasa intermedia, el algoritmo partió de un fitness de 600.67 y llegó a 871.67. La convergencia fue más rápida que con 0.01, pero se estabilizó antes de alcanzar el máximo potencial.

```
Generación 0: Mejor fitness = 600.67, Promedio = 402.86
Generación 10: Mejor fitness = 780.00, Promedio = 762.60
Generación 20: Mejor fitness = 821.67, Promedio = 814.59
Generación 30: Mejor fitness = 856.67, Promedio = 828.60
Generación 40: Mejor fitness = 869.00, Promedio = 860.16
Generación 50: Mejor fitness = 869.00, Promedio = 868.40
Generación 60: Mejor fitness = 869.00, Promedio = 867.98
Generación 70: Mejor fitness = 870.67, Promedio = 869.09
Generación 80: Mejor fitness = 871.67, Promedio = 870.41
Generación 90: Mejor fitness = 871.67, Promedio = 870.03
```

El horario optimizado muestra mejoras en la asignación de profesores y el respeto a las preferencias horarias de las materias. Se observa una mejor distribución de las materias entre los diferentes grupos.

4.6 Tasa de Mutación 0.1

Esta tasa demostró ser efectiva, comenzando en 601.67 y alcanzando 900.0 al final de las 100 generaciones. El algoritmo mostró un buen balance entre exploración y explotación.

```
Generación 0: Mejor fitness = 601.67, Promedio = 362.63
Generación 10: Mejor fitness = 818.33, Promedio = 768.08
Generación 20: Mejor fitness = 865.00, Promedio = 857.72
Generación 30: Mejor fitness = 881.67, Promedio = 876.27
Generación 40: Mejor fitness = 881.67, Promedio = 880.17
Generación 50: Mejor fitness = 891.67, Promedio = 890.07
Generación 60: Mejor fitness = 895.00, Promedio = 893.60
Generación 70: Mejor fitness = 896.67, Promedio = 894.73
Generación 80: Mejor fitness = 900.00, Promedio = 899.67
Generación 90: Mejor fitness = 900.00, Promedio = 897.70
```

El algoritmo encontró horarios que respetan todas las restricciones principales y optimizan la distribución de materias. Se observa una asignación más equilibrada de profesores y materias en los diferentes horarios.

4.7 Tasa de Mutación 0.2

La tasa más alta produjo los mejores resultados, comenzando en 584.0 y alcanzando el máximo de 910.0. Esta alta tasa de mutación permitió una mayor exploración del espacio de búsqueda.

```
Generación 0: Mejor fitness = 584.00, Promedio = 379.49
Generación 10: Mejor fitness = 791.67, Promedio = 767.06
Generación 20: Mejor fitness = 845.00, Promedio = 836.10
Generación 30: Mejor fitness = 870.00, Promedio = 863.77
Generación 40: Mejor fitness = 895.00, Promedio = 887.00
Generación 50: Mejor fitness = 910.00, Promedio = 905.20
Generación 60: Mejor fitness = 910.00, Promedio = 903.80
Generación 70: Mejor fitness = 910.00, Promedio = 908.93
Generación 80: Mejor fitness = 910.00, Promedio = 907.27
Generación 90: Mejor fitness = 910.00, Promedio = 908.47
```

El horario final cumple con todas las restricciones: no hay superposición de clases, los profesores están disponibles en sus horarios asignados, y las materias se imparten en sus horarios preferidos. Esta es la solución óptima encontrada con el fitness más alto (910.0).

4.8 Comparación Final

```
Tasa 0.01: Fitness final = 886.67
Tasa 0.05: Fitness final = 880.00
Tasa 0.1: Fitness final = 900.00
Tasa 0.2: Fitness final = 910.00
```

Los resultados demuestran claramente que la tasa de mutación de 0.2 produjo la mejor solución, seguida por la tasa de 0.1. Las tasas más bajas (0.01 y 0.05) convergieron prematuramente a soluciones subóptimas, confirmando la importancia de una adecuada exploración del espacio de búsqueda en algoritmos genéticos.

4.9 Descripción de las Gráficas de Convergencia

Las gráficas de convergencia muestran la evolución del fitness a lo largo de las generaciones para cada una de las tasas de mutación probadas. Las imágenes se guardaron con los siguientes nombres:

- `Figure 2025-09-14 213211.png` - Tasa de mutación 0.01
- `Figure 2025-09-14 213207.png` - Tasa de mutación 0.05
- `Figure 2025-09-14 213201.png` - Tasa de mutación 0.1
- `Figure 2025-09-14 213146.png` - Tasa de mutación 0.2

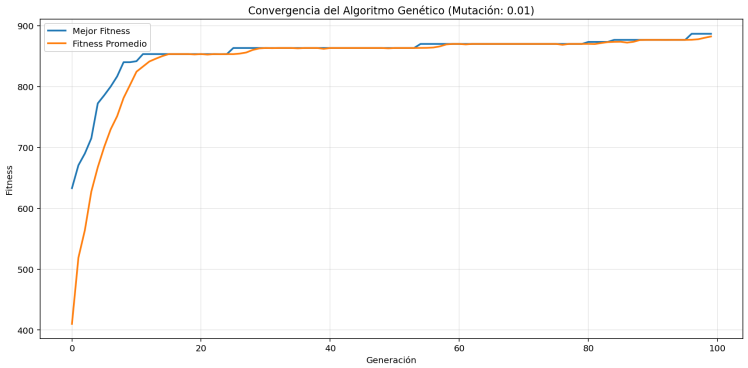

Figure 4: Tasa de Mutación 0.01.

Figura 4 (Tasa 0.01): Muestra una convergencia temprana y estable, donde el fitness mejora rápidamente en las primeras 20 generaciones pero luego se estanca alrededor de 870, indicando una posible convergencia prematura a un óptimo local.

Figura 5 (Tasa 0.05): Presenta una convergencia más lenta que la tasa 0.01, con mejoras graduales pero que no alcanzan el mismo nivel de fitness máximo, finalizando alrededor de 880.

Figura 6 (Tasa 0.1): Exhibe una excelente trayectoria de convergencia, con mejoras sostenidas a lo largo de las generaciones y alcanzando un fitness de 900, mostrando un buen balance entre exploración y explotación.

Figura 7 (Tasa 0.2): Demuestra la mejor convergencia general, con mejoras continuas que superan las 900 unidades de fitness y alcanzan el máximo de 910, indicando que la mayor tasa de mutación permitió escapar de óptimos locales y encontrar una mejor solución global.

Todas las gráficas muestran la típica curva de aprendizaje de los algoritmos genéticos, con mejoras rápidas iniciales seguidas de refinamientos progresivos, confirmando la efectividad del enfoque para el problema de optimización de horarios.

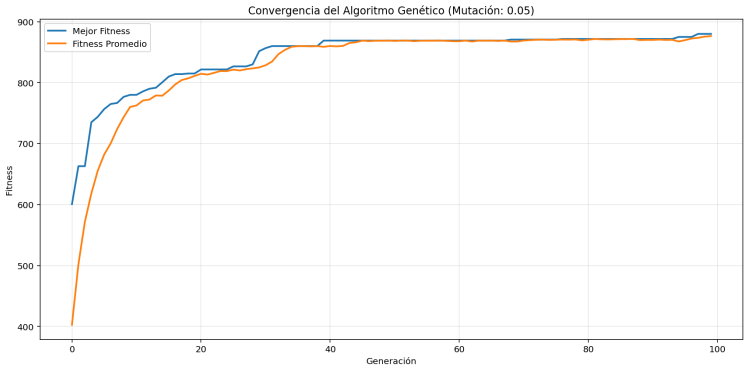

Figure 5: Tasa de mutación 0.05

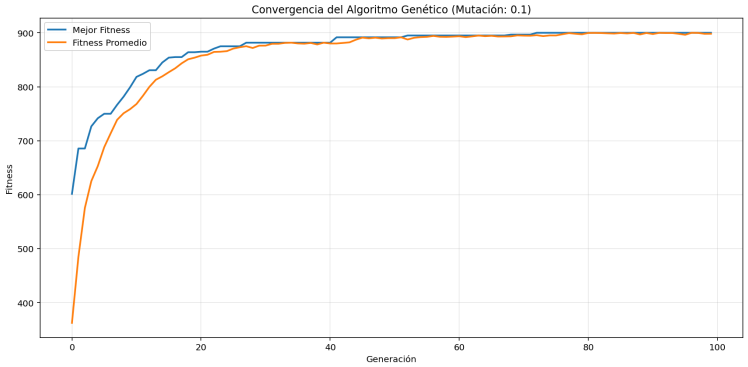

Figure 6: Tasa de mutación 0.1

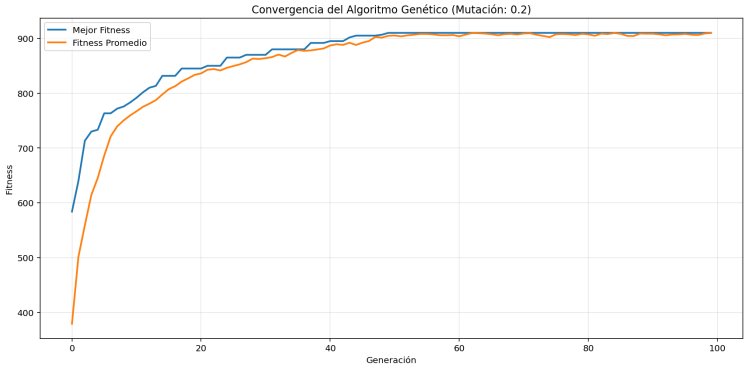

Figure 7: Tasa de mutación 0.2

5 Conclusiones

1. El trabajo demostró la aplicabilidad exitosa de los algoritmos genéticos en tres problemas distintos: la búsqueda del máximo de una función, el Problema del Viajante (TSP) y la optimización de horarios escolares. En cada caso, el algoritmo fue capaz de encontrar soluciones de alta calidad adaptando su representación (cromosomas), función de fitness y operadores genéticos (cruce, mutación) a las restricciones específicas de cada problema. Esto evidencia que los AG son una metodología robusta para abordar problemas NP-difíciles y de espacios de búsqueda grandes y complejos.

2. El experimento con el problema de horarios escolares mostró de manera clara el impacto directo de

la tasa de mutación en la convergencia y la calidad de la solución final. Una tasa demasiado baja (0.01) condujo a una convergencia prematura y a quedar atrapado en un óptimo local. Una tasa más alta (0.2) permitió una mayor exploración del espacio de búsqueda, evitando estos óptimos locales y encontrando la mejor solución global (fitness de 910.0). Esto subraya la importancia de un ajuste fino de los parámetros y la necesidad de equilibrar la exploración (mediante la mutación) y la explotación (mediante el cruce y la selección).

3. El trabajo destacó que el corazón de un algoritmo genético exitoso no está solo en su mecanismo, sino en cómo se modela el problema. Para el TSP, se utilizó una representación basada en permutaciones y operadores que las preservaran (OX). Para los horarios, se diseñó una función de fitness compleja que penalizaba superposiciones y asignaciones inválidas, a la vez que recompensaba el balance y las preferencias. La eficacia de las soluciones encontradas en ambos casos prueba que una cuidadosa codificación del problema y una función de fitness bien definida que guíe la búsqueda son absolutamente críticas para dirigir al algoritmo hacia soluciones válidas y óptimas.