+34 691 225 633
Rev. Febrero 2021

Autor: Marcelo Guato

Lógica difusa y algoritmos de adaptación social

Objetivo y Alcance del Trabajo

El objetivo de este trabajo es aplicar los conocimientos teóricos de Lógica Difusa y algoritmos de adaptación social en implementaciones informáticas mediante el lenguaje Python. Se presentarán dos enunciados, en primer lugar un enunciado que debe ser trasladado a una implementación de un concepto descrito a una solución en términos de lógica difusa.

Un segundo enunciado describirá un problema y parte de la implementación en código del algoritmo de Optimización por Enjambre de Partículas (PSO) y se deberá completar el desafío presentado.

Estructura

El proyecto se divide en las siguientes partes:

PARTE 1, experimentación con Lógica Difusa (50%):

1. Considere las siguientes situaciones:
    - ***"Conducir en un día lluvioso puede ser peligroso, en caso de derrapar se debe mantener la calma y evitar movimientos bruscos. Si derrapa, puede ayudar soltar el acelerador, pisar suavemente el freno si es necesario, y girar el volante en la dirección opuesta a la del derrape. Si el derrape es leve, puede corregirlo con movimientos suaves del volante."***
    - ***"En base a una reflexión de su entorno de trabajo o una área donde usted es experto/a plante una situación donde pueda aplicar la lógica difusa y proponga una solución"***

2. Proponer una solución escrita en Python con un programa que determine mediante lógica difusa si un vehículo esta frente a una situación de derrape y aplique la corrección de dirección y frenado.

   - Definir universos de discurso.
   - Crear variables lingüísticas (antecedentes y consecuentes).
   - Definir funciones de pertenencia.
   - Establecer reglas difusas.
   - Construir y simular sistemas de control difuso.
   - Realizar la defuzzificación.

PARTE 2, experimentación con algoritmo PSO (50%).

Conocido el pseudocódigo de PSO:

```
Pseudo código algoritmo PSO:
Asignar posiciones y velocidades aleatorias iniciales a las partículas
Repetir
   Cada partícula:
      Actualizar su velocidad considerando:
         Inercia de la partícula (la hace seguir con la misma velocidad)
         Atracción al mejor personal
         Atracción al mejor global
      Actualizar la posición de la partícula
      Calcular el valor de fitness en la nueva posición
      Actualizar su mejor personal
   Actualizar el mejor global del sistema
Devolver el mejor global
```

1. El problema es la implementación de algoritmo PSO (Optimización por Enjambre de Partículas) , en el cual se dé solución al problema del vuelo de una flota de drones manteniendo una formación dada.

2. Estudiar, corregir, documentar mediante comentarios que describan lo que hacen las diferentes secciones de código e implementar el código mostrado en el punto 3 considerando lo siguiente:

- Función de Fitness:
  - ¿Calcula correctamente el error de formación para las geometrías solicitadas?
  - ¿Implementa adecuadamente la penalización por colisión?
  - ¿Considera los límites del área de vuelo?
- Resultados y Convergencia:
  - ¿El algoritmo converge a una solución de bajo fitness?
  - ¿La formación final visualmente se asemeja a la formación objetivo?
  - ¿Se evitan colisiones en la solución final?.
- Experimentar y analizar:
  - Probar con diferentes parámetros de PSO y comentar su impacto.
  - Probar con las diferentes formaciones objetivo.

- El código implementa la formación triangular, implementar adicionalmente la formación en H.

3. **Código parta trabajar**:

```
NUM_DRONES =
DIMENSIONS =
AREA_WIDTH =
AREA_HEIGHT =
MIN_SAFE_DISTANCE =
FORMATION_DISTANCE_L =

NUM_PARTICLES =
MAX_ITERATIONS =

W = 0.5
C1 = 1.5
C2 = 1.5
V_MAX = 10.0

def get_target_formation_relative_positions(formation_type="line"):
    targets = np.zeros((NUM_DRONES, DIMENSIONS))
    if formation_type == "line":

        start_x = - (NUM_DRONES - 1) * FORMATION_DISTANCE_L / 2.0
        for i in range(NUM_DRONES):
```

```
            targets[i, 0] = start_x + i * FORMATION_DISTANCE_L
            targets[i, 1] = 0 # Todos en la misma línea y
    elif formation_type == "triangle" and NUM_DRONES == :
        h = (np.sqrt(3)/2) * FORMATION_DISTANCE_L # Altura
        targets[0, :] = [0, h * 2/3]
        targets[1, :] = [-FORMATION_DISTANCE_L / 2, -h / 3]
        targets[2, :] = [FORMATION_DISTANCE_L / 2, -h / 3]
    # ... otras formaciones ...
    else:
        raise ValueError("Tipo de formación no soportado o número de drones incorrecto.")
    return targets

# --- Función de Fitness ---
def fitness_function(particle_position, target_relative_pos):

    drone_positions = particle_position.reshape((NUM_DRONES, DIMENSIONS))
    current_centroid = np.mean(drone_positions, axis=0)

    e_form = 0
    for i in range(NUM_DRONES):
        # Posición ideal del dron i = centroide_actual + pos_relativa_ideal_i
        ideal_pos_drone_i = current_centroid + target_relative_pos[i, :]
        e_form += np.sum((drone_positions[i, :] - ideal_pos_drone_i)**2)

    # 2. Penalización por Colisión (P_coll)
    p_coll = 0
    collision_penalty_value = 10000 # Valor alto por cada colisión
    for i in range(NUM_DRONES):
        for j in range(i + 1, NUM_DRONES):
            dist_sq = np.sum((drone_positions[i, :] - drone_positions[j, :])**2)
            if np.sqrt(dist_sq) < MIN_SAFE_DISTANCE:
                p_coll += collision_penalty_value # Penalización fija
                # O una penalización proporcional a la invasión:
                # p_coll += (MIN_SAFE_DISTANCE - np.sqrt(dist_sq)) * factor_grande

    # 3. Penalización por Límites (Opcional pero recomendable)
    p_boundary = 0
    boundary_penalty_value = 1000
    for i in range(NUM_DRONES):
        if not (0 <= drone_positions[i, 0] <= AREA_WIDTH and \
                0 <= drone_positions[i, 1] <= AREA_HEIGHT):
            p_boundary += boundary_penalty_value
            # También se puede hacer un clipping dentro del bucle de PSO

    # Pesos (ajustar según sea necesario)
    w1 =    # Error de formación
    w2 =   # Penalización por colisión
    w3 =    # Penalización por límites

    total_fitness = w1 * e_form + w2 * p_coll + w3 * p_boundary
    return total_fitness
```

+34 691 225 633
Rev. Febrero 2021

```
# --- Implementación de PSO ---
# Posiciones de las partículas (swarm)
# Cada fila es una partícula, cada partícula es un vector de posiciones de drones
particles_pos = np.random.rand(NUM_PARTICLES, NUM_DRONES * DIMENSIONS)
# Escalar a las dimensiones del área
particles_pos[:, ::2] *= AREA_WIDTH  # Coordenadas X
particles_pos[:, 1::2] *= AREA_HEIGHT # Coordenadas Y

particles_vel = np.random.rand(NUM_PARTICLES, NUM_DRONES * DIMENSIONS) * V_MAX * 0.1 #
Velocidades iniciales pequeñas

pbest_pos = np.copy(particles_pos)
pbest_fitness = np.array([float('inf')] * NUM_PARTICLES)

gbest_pos = np.zeros(NUM_DRONES * DIMENSIONS)
gbest_fitness = float('inf')

# Selección de la formación objetivo
TARGET_FORMATION_TYPE = "triangle" # o "line"
target_rel_pos = get_target_formation_relative_positions(TARGET_FORMATION_TYPE)

fitness_history = [] # Para graficar convergencia

# Bucle principal de PSO
for iter_num in range(MAX_ITERATIONS):
    for i in range(NUM_PARTICLES):
        current_fitness = fitness_function(particles_pos[i], target_rel_pos)

        # Actualizar pBest
        if current_fitness < pbest_fitness[i]:
            pbest_fitness[i] = current_fitness
            pbest_pos[i] = np.copy(particles_pos[i])

        # Actualizar gBest
        if current_fitness < gbest_fitness:
            gbest_fitness = current_fitness
            gbest_pos = np.copy(particles_pos[i])

    fitness_history.append(gbest_fitness)

    # Actualizar velocidades y posiciones
    for i in range(NUM_PARTICLES):
        r1 = np.random.rand(NUM_DRONES * DIMENSIONS)
        r2 = np.random.rand(NUM_DRONES * DIMENSIONS)

        cognitive_vel = C1 * r1 * (pbest_pos[i] - particles_pos[i])
        social_vel = C2 * r2 * (gbest_pos - particles_pos[i])

        particles_vel[i] = W * particles_vel[i] + cognitive_vel + social_vel
```

+34 691 225 633


```
        # Limitar velocidad
        particles_vel[i] = np.clip(particles_vel[i], -V_MAX, V_MAX)

        particles_pos[i] += particles_vel[i]

        # Aplicar límites del área de vuelo (clipping)
        # Asegurar que las coordenadas X estén entre 0 y AREA_WIDTH
        particles_pos[i, ::2] = np.clip(particles_pos[i, ::2], 0, AREA_WIDTH)
        # Asegurar que las coordenadas Y estén entre 0 y AREA_HEIGHT
        particles_pos[i, 1::2] = np.clip(particles_pos[i, 1::2], 0, AREA_HEIGHT)

    if (iter_num + 1) % 10 == 0:
        print(f"Iteración {iter_num + 1}/{MAX_ITERATIONS}, Mejor Fitness: {gbest_fitness:.2f}")

print("\n--- Mejor Solución Encontrada ---")
print(f"Fitness: {gbest_fitness:.4f}")
final_drone_positions = gbest_pos.reshape((NUM_DRONES, DIMENSIONS))
print("Posiciones de los drones:")
for i in range(NUM_DRONES):
    print(f"  Dron {i}: ({final_drone_positions[i,0]:.2f}, {final_drone_positions[i,1]:.2f})")

# --- Visualización ---
# 1. Convergencia del Fitness
plt.figure(figsize=(10, 5))
plt.plot(fitness_history)
plt.title(f'Convergencia de PSO (Formación: {TARGET_FORMATION_TYPE})')
plt.xlabel('Iteración')
plt.ylabel('Mejor Fitness (gBest)')
plt.grid(True)
plt.show()

# 2. Formación Final
plt.figure(figsize=(8, 8))
plt.scatter(final_drone_positions[:, 0], final_drone_positions[:, 1], s=100, label='Drones')
for i in range(NUM_DRONES):
    plt.text(final_drone_positions[i,0]+1, final_drone_positions[i,1]+1, f'D{i}')
    # Dibujar círculo de seguridad
    circle = plt.Circle((final_drone_positions[i,0], final_drone_positions[i,1]), MIN_SAFE_DISTANCE,
color='r', fill=False, linestyle='--')
    plt.gca().add_artist(circle)

# Visualizar el centroide y la forma objetivo ideal (opcional)
final_centroid = np.mean(final_drone_positions, axis=0)
plt.scatter(final_centroid[0], final_centroid[1], color='green', marker='x', s=100, label='Centroide Actual')

ideal_positions_at_final_centroid = final_centroid + target_rel_pos
plt.scatter(ideal_positions_at_final_centroid[:,0], ideal_positions_at_final_centroid[:,1],
            color='gray', marker='o', s=150, facecolors='none', label='Formación Objetivo Ideal')

plt.title(f'Formación Final de Drones ({TARGET_FORMATION_TYPE})')
plt.xlabel('Coordenada X')
```

```
plt.ylabel('Coordenada Y')
plt.xlim(0, AREA_WIDTH)
plt.ylim(0, AREA_HEIGHT)
plt.legend()
plt.grid(True)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

Conclusiones

- El presente trabajo se configura como una herramienta pedagógica que permite llevar a la práctica la teoría de la Lógica Difusa y los algoritmos de adaptación social, como PSO. A través de la implementación práctica en Python, se trata de fortalecer los conceptos y el desarrollar habilidades de diseño, codificación y depuración de soluciones para problemas complejos, ejemplificados en el control de vehículos ante derrapes y la optimización de formaciones de drones.

- Se debe ir más allá de la simple transcripción de pseudocódigo, y avanzar a la comprensión de los fundamentos de la Lógica Difusa para modelar la incertidumbre, y también se intenta estudiar la sensibilidad paramétrica de algoritmos de optimización como PSO. La necesidad de analizar, corregir, extender y experimentar con el código para el problema de los drones tiene el propósito de estimular el trabajo en la validación y el ajuste fino en la aplicación de técnicas bio inspiradas de inteligencia artificial.