**Report on Genetic Algorithm Adaptation for Mountain Climbing Task**

This code is purely written by me

import pybullet as p

import pybullet\_data

import time

import numpy as np

import random

import creature

import math

# Connect to PyBullet physics server in GUI mode

client = p.connect(p.GUI)

if client < 0:

    raise Exception("Failed to connect to the PyBullet physics server.")

p.setAdditionalSearchPath(pybullet\_data.getDataPath())

def make\_mountain(num\_rocks=100, max\_size=0.25, arena\_size=10, mountain\_height=5):

    def gaussian(x, y, sigma=arena\_size/4):

        """Return the height of the mountain at position (x, y) using a Gaussian function."""

        return mountain\_height \* math.exp(-((x\*\*2 + y\*\*2) / (2 \* sigma\*\*2)))

    for \_ in range(num\_rocks):

        x = random.uniform(-1 \* arena\_size/2, arena\_size/2)

        y = random.uniform(-1 \* arena\_size/2, arena\_size/2)

        z = gaussian(x, y)  # Height determined by the Gaussian function

        # Adjust the size of the rocks based on height. Higher rocks (closer to the peak) will be smaller.

        size\_factor = 1 - (z / mountain\_height)

        size = random.uniform(0.1, max\_size) \* size\_factor

        orientation = p.getQuaternionFromEuler([random.uniform(0, 3.14), random.uniform(0, 3.14), random.uniform(0, 3.14)])

        rock\_shape = p.createCollisionShape(p.GEOM\_BOX, halfExtents=[size, size, size])

        rock\_visual = p.createVisualShape(p.GEOM\_BOX, halfExtents=[size, size, size], rgbaColor=[0.5, 0.5, 0.5, 1])

        rock\_body = p.createMultiBody(baseMass=0, baseCollisionShapeIndex=rock\_shape, baseVisualShapeIndex=rock\_visual, basePosition=[x, y, z], baseOrientation=orientation)

def make\_arena(arena\_size=10, wall\_height=1):

    wall\_thickness = 0.5

    floor\_collision\_shape = p.createCollisionShape(shapeType=p.GEOM\_BOX, halfExtents=[arena\_size/2, arena\_size/2, wall\_thickness])

    floor\_visual\_shape = p.createVisualShape(shapeType=p.GEOM\_BOX, halfExtents=[arena\_size/2, arena\_size/2, wall\_thickness], rgbaColor=[1, 1, 0, 1])

    floor\_body = p.createMultiBody(baseMass=0, baseCollisionShapeIndex=floor\_collision\_shape, baseVisualShapeIndex=floor\_visual\_shape, basePosition=[0, 0, -wall\_thickness])

    wall\_collision\_shape = p.createCollisionShape(shapeType=p.GEOM\_BOX, halfExtents=[arena\_size/2, wall\_thickness/2, wall\_height/2])

    wall\_visual\_shape = p.createVisualShape(shapeType=p.GEOM\_BOX, halfExtents=[arena\_size/2, wall\_thickness/2, wall\_height/2], rgbaColor=[0.7, 0.7, 0.7, 1])  # Gray walls

    # Create four walls

    p.createMultiBody(baseMass=0, baseCollisionShapeIndex=wall\_collision\_shape, baseVisualShapeIndex=wall\_visual\_shape, basePosition=[0, arena\_size/2, wall\_height/2])

    p.createMultiBody(baseMass=0, baseCollisionShapeIndex=wall\_collision\_shape, baseVisualShapeIndex=wall\_visual\_shape, basePosition=[0, -arena\_size/2, wall\_height/2])

    wall\_collision\_shape = p.createCollisionShape(shapeType=p.GEOM\_BOX, halfExtents=[wall\_thickness/2, arena\_size/2, wall\_height/2])

    wall\_visual\_shape = p.createVisualShape(shapeType=p.GEOM\_BOX, halfExtents=[wall\_thickness/2, arena\_size/2, wall\_height/2], rgbaColor=[0.7, 0.7, 0.7, 1])  # Gray walls

    p.createMultiBody(baseMass=0, baseCollisionShapeIndex=wall\_collision\_shape, baseVisualShapeIndex=wall\_visual\_shape, basePosition=[arena\_size/2, 0, wall\_height/2])

    p.createMultiBody(baseMass=0, baseCollisionShapeIndex=wall\_collision\_shape, baseVisualShapeIndex=wall\_visual\_shape, basePosition=[-arena\_size/2, 0, wall\_height/2])

def evaluate\_creature(creature\_id):

    """Evaluate a creature and return its fitness score based on height achieved."""

    max\_height = 0

    start\_time = time.time()

    while time.time() - start\_time < 10:  # Simulate for 10 seconds

        p.stepSimulation()

        pos, \_ = p.getBasePositionAndOrientation(creature\_id)

        max\_height = max(max\_height, pos[2])

        time.sleep(1./240.)  # Adjust to your simulation speed

    return max\_height

def run\_ga(gene\_count=3, population\_size=10, generations=10):

    """Run the genetic algorithm."""

    population = [creature.Creature(gene\_count) for \_ in range(population\_size)]

    best\_creature = None

    best\_fitness = 0

    for generation in range(generations):

        fitness\_scores = []

        for cr in population:

            # Save the creature to XML and load it into the simulation

            with open('test.urdf', 'w') as f:

                f.write(cr.to\_xml())

            creature\_id = p.loadURDF('test.urdf', (0, 0, 10))

            # Evaluate the creature

            fitness = evaluate\_creature(creature\_id)

            fitness\_scores.append((cr, fitness))

            # Remove the creature from the simulation

            p.removeBody(creature\_id)

        # Select the best creature

        fitness\_scores.sort(key=lambda x: x[1], reverse=True)

        best\_creature = fitness\_scores[0][0]

        best\_fitness = fitness\_scores[0][1]

        # Print the best fitness of the generation

        print(f"Generation {generation}, Best Fitness: {best\_fitness}")

        # Apply genetic algorithm operators (selection, crossover, mutation) to create the next generation

        next\_population = []

        for \_ in range(population\_size):

            parent1, parent2 = random.choices(fitness\_scores, k=2)

            child = parent1[0].crossover(parent2[0])

            child.mutate()

            next\_population.append(child)

        population = next\_population

    return best\_creature, best\_fitness

p.setGravity(0, 0, -10)

arena\_size = 20

make\_arena(arena\_size=arena\_size)

mountain\_position = (0, 0, -1)  # Adjust as needed

mountain\_orientation = p.getQuaternionFromEuler((0, 0, 0))

p.setAdditionalSearchPath('shapes/')

mountain = p.loadURDF("gaussian\_pyramid.urdf", mountain\_position, mountain\_orientation, useFixedBase=1)

# Set the camera to a suitable position and orientation

p.resetDebugVisualizerCamera(cameraDistance=10, cameraYaw=50, cameraPitch=-35, cameraTargetPosition=[0, 0, 0])

# Run the genetic algorithm

best\_creature, best\_fitness = run\_ga()

# Print the best creature's fitness

print(f"Best Creature Fitness: {best\_fitness}"

# Visualize the best creature

with open('best\_creature.urdf', 'w') as f:

    f.write(best\_creature.to\_xml())

best\_creature\_id = p.loadURDF('best\_creature.urdf', (0, 0, 10))

p.setRealTimeSimulation(1)

while True:

    p.stepSimulation()

    time.sleep(1./240.)