Algorithmic problem solving in python pdf

Computer scientist, software engineer - Loadsmart, a machine learning enthusiastGenetic algorithms, also referred to simply as GA, are algorithms inspired by Charles Darwin's natural selection theory, which aims to find optimal solutions to problems we don't know much about. For example: How do I find this maximum or minimum function if you can't get it? It is based on three concepts: selection, reproduction and mutation. We generate a random set of people, choose the best, cross them and finally slightly muddy the result - over and over again until we find an acceptable solution. You can check out some comparisons on other search methods in Goldberg's book. Let's see how to write a simple implementation of a genetic algorithm with Python! The problem we'll try to solve here is to find a maximum 3D function similar to a hat. Defined as f(x, y) - sin (sqrt (x-2 and y2)). We will limit our problem to the boundaries of 4 ≥ x ≥ -4 and 4 ≥ at ≥ -4. (The plot of function between our defined boundaries, created with CalcPlot3D) The first step is to create our original population. The population or generation is our current set of possible solutions called individuals. We will iterate for generations, improving it until we find an acceptable solution. The first generation is randomly generated generate_population. x_boundaries, y_boundaries): lower_x_boundary, upper_x_boundary and x_boundaries lower_y_boundary, upper_y_boundary and y_boundaries population - for i in range (size): individual and x: random.uniform (lower_x_boundary, upper_x_boundary), y: random.uniform (lower_y_boundary, upper_y_boundary) - population.append (individual) population ReturnThis Genesis : The number of individuals that must have a population, dull, pointing boundaries on the x-axis and blunt, pointing boundaries on the axis, so our individuals randomly correspond to these boundaries. Moving on, let's define our fitness function. It will be our appraiser who will express how much better or worse a person is apart. Individuals with better fitness should be retained and reproduced while the worst of them should fall - just like in nature. In our case, as we want to find our feature to the maximum, we can just apply our objective function to the person and the greatest number will be the largest fitness as well. If we want to find a minimum, fitness can be expressed as a result of function once -1, so that smaller values become more fitness.import math def apply_function (individual): x individual y y individual return math.sin (math.sqrt (x x x 2 y 2)) Since we have a population generator and fitness evaluator, we can start reproducing our generation. We'll do it. Until we find it. Solution. There are several stop criteria, largely used by one n generation with outdated fitness, but we will use a simpler one that is just n generations - we will use 100. So far our entry function looks like this: Generations - 100 population - generate_population (size -10, x_boundaries (-4, 4), y_boundaries (-4, 4)) i - 1, while True: print (f GENERATION) for individuals in the population: print (individual) if I have generations: break i 1 - Make the next generation ... To select people to play we will use a widely adopted method called the roulette wheel, which consists of dividing the circle into pieces like a pie chart where each person has a piece proportional to his fitness and then spinning it. Thus, we ensure that the best people are more likely to be selected, while the worst still have a chance, although this is minor.def choice_by_roulette (sorted_population, fitness_sum lowest_fitness fitness_sum apply_function): the shift sorted_population lowest_fitness normalized_fitness_sum lowest_fitness of the zlt; No 0 normalized_fitness_sum sorted_population 1) accumulated 0 for a person in sorted_population: fitness - apply_function (individual) - probability compensation - fitness / normalized_fitness_sum accumulated probability, if the draw is zlt; accumulated: the return of the individual Illustreed our method, say, we have four people: A, B, C and D with fitness 0, 50, 200 and 250 respectively. The total fitness amount is 500, so each of them will have a fitness/total_fitness chance of being selected: 0%, 10%, 40%, 50%. We select a random number between 0 and 1, and then check which person is in the selected part: A (0, 0), B (0, 0.1), C (0.1, 0.5, D (0.5, 1). And then adding it to all of them (for example, if we have two people with fitness -10 and 5 respectively, we add 10 to become 0 and 15). and then crossing them. As a result, the person will get minor perturbations (mutations), so we do not stick to the comfort zone and look for even better solutions than what we have so far. There are several crossover methods for real numbers: for example, we could take x individual A and individual B, we could take the geometric mean of each or, most simply, take the arithmetic average of each. If we were dealing with binary data, a common method is to select part of the bit line A and part part of the строка Б. Для простоты причинам, давайте использовать арифметическое среднее. Для мутации есть много вариантов тоже - мы просто суммировать небольшое случайное число между фиксированным интервалом. Этот интервал является скорость мутации и может быть точно настроены соответствующим образом, давайте использовать 0,05, 0,05 . Для больших пробелов поиска можно выбрать большие интервалы и уменьшить их из поколения в поколение. When dealing with binary data you can simply flip randomly selected bits of the individual string.def sort_population_by_fitness(population): return sorted(population, key=apply_function) def crossover(individual_a, individual_b): xa = individual_a[x] ya = individual_a[y] xb = individual_b[x] yb = individual_b[y] return {x: (xa + xb) / 2, y: (ya + yb) / 2} def mutate(individual): next_x = individual[x] + random.uniform(-0.05, 0.05) next_y = individual[y] + random.uniform(-0.05, 0.05) lower_boundary, upper_boundary = (-4, 4) # Guarantee we keep inside boundaries next_x = min(max(next_x, lower_boundary), upper_boundary) next_y = min(max(next_y, lower_boundary), upper_boundary) return {x: next_x, y: next_y} def make_next_generation(previous_population): next_generation = [] sorted_by_fitness_population = sort_population_by_fitness(previous_population) population_size = len(previous_population) fitness_sum = sum(apply_function(individual) for individual in population) for i in range(population_size) : first_choice choice_by_roulette (sorted_by_fitness_population, fitness_sum) second_choice - choice_by_roulette (sorted_by_fitness_population, fitness_sum) индивидуальный кроссовер (first_choice, second_choice) индивидуальный и мутирующий (индивидуальный) next_generation.append (индивидуальный) возврат next_generationSo вот оно! Теперь у нас есть все три шага ГА: выбор, кроссовер и мутации. Наш основной метод, то просто так: поколения 100 населения - generate_population (размер 10, x_boundaries (-4, 4), y_boundaries (-4, 4)) i 1 в то время как True: печать (f GENERATION) для отдельных в популяции: печать (индивидуальный, индивидуальный, apply_function (индивидуальный)), если я поколений: перерыв я No 1 населения и make_next_generation (население) best_individual sort_population_by_fitness (население) -1 печати (↯ FINAL RESULT) печати (best_individual, apply_function (best_individual))) переменная best_individual будет держать нашего человека с самой высокой пригодности после этих 100 поколений. Это может быть точное оптимальное решение или нет, вам придется точно настроить параметры (скорость мутации, поколений и т.д.) и методы (выбор, кроссовер и мутации методы), пока вы не можете улучшить больше. Давайте посмотрим последние выходные строки для экспериментального запуска (обратите внимание, что из-за случайных параметров вы, скорее всего, получите различные, но аналогичные результаты): GENERATION 100 'x': -1.0665224807251312, 'y': 0.9745828000809058 'x': -1.0753606354537244, 'y': 'y': 0.976355423070003 {'x': -1.0580786664161246, 'y': -1.3693549033564183} 0.9872729309456848 {'x': -1.093601208942564, 'y': -1.3832920 0.9815156357267611 {'x': -1.0464963866796362, 'y': -1.3461172606906064} 0.9910018621648693 {'x': -0.987226479369966, 'y': -1.4569537217049857} 0.9821687265560713 {'x': -1.0501568673329658, 'y': -1.430577408679398} 0.9792937786319258 {'x': -1.0291192465186982, 'y': -1.4289167102720242} 0.9819781801342095 {'x': -1.098502968808768, 'y': -1.3738230550364259} 0.9823409690311633 {'x': -1.091317403073779, 'y': -1.4256574643591997} 0.9748817266026281 ↯ FINAL RESULT {'x': -1.0464963866796362, 'y': -1.3461172606906064} 0.9910018621648693Our final result was very close to one of the possible solutions (this function has multiple maximums inside our boundaries that is 1.0, as you can see on the plot at the beginning). Note that we've used less complex methods, so this result is somehow expected - it's a starting point for fine-tuning until we can find better solutions with fewer generations. It was a very introductory article about genetic algorithms using Python. If you liked it, you will definitely want to learn more about all the possible improvements you can make on it and the apps that you can use it. I highly recommend reading the book Genetic Algorithms in Search, Optimization and Machine Learning mentioned at the beginning. Sign up to get a daily preparation of top tech history! History! algorithmic problem solving in python ppt