Continue

[illegible] Cookies Policy, Privacy Policy [illegible] Terms & Conditions [illegible] Agree [illegible] Agree In this article, we'll discuss "genetic algorithms", an adaptation of evolution that allows data scientists to "evolve" solutions to problems.To build a thorough understanding of genetic algorithms, we'll first review how evolution works from a high level, then we'll then explore how common problems can be re-thought as generations of solutions undergoing evolution. By the end of this article, we'll use the principles of evolution to solve scheduling conflicts and choose the best route in complicated traffic patterns.This is the concept that got me into data science, so I'm super excited! Let's get into it!Doing some back of the napkin math, that's 120^{440} possible schedules. There's roughly 10^{82} atoms in the observable universe, meaning brute forcing this problem would be much harder than counting every atom in every known galaxy one at a time. Let's solve this problem in a few minutes by making schedules have sex.— From later in the articleShow off your knowledge to friends, colleagues, and recruiters.ShareWho is this useful for? Anyone who wants to form a complete understanding of AI.How advanced is this post? This post is beginner accessible, but is also a fascinating topic for experienced data scientists who might not have had the privledge of working with genetic algorithms before.Pre-requisites: NonesourceEvolution is the process of lifeforms changing over time as a result of "natural selection" influencing which characteristics are most common over successive generations.Natural selection is the idea that only certain members of a population will live to the point of sexual maturity. If a member of a population has some disadvantageous trait they will be less likely to mate, and if a member of a population has some advantageous trait they will be more likely to mate.A conceptual diagram of how natural selection influences a population. Here, a dot being a cool color is "disadvantageous".In the short term, this can allow a species to adapt to shifting environments.The peppered moth is a famous example of rapid evolution as a result of environmental changes. SourceAnd in the long term, it can lead to the rise of new, highly optimized species.Every lifeform you see today is the result of slow adaptation over billions of years. Source.Evolution relies on Deoxyribonucleic acid (DNA), a long and complex molecule that defines the genetic identity of an organism.DNA, a complex polymer, encodes the fundamental information of life. With the exception of some viruses, all living things contain DNA. Source.This structure can be simplified as a long series of genetic information. This genetic information influences things like eye color, height, proclivity to get certain diseases, etc.DNA, in it's simplest conceptualization, is a list of instructions. SourceDNA is long and fragile, so for practical purposes it's wrapped up into a structure called a chromosome.the chromosome, an X shaped construction, is how DNA is commonly stored within lifeforms. Source.During sexual reproduction, an offspring is created by dividing the chromosome of each parent at a random point, and swapping their genetic information. Thus, the offspring is a combination of both parent's genetic information. This process is called "crossover".During mating, chromosomes from each parent interact with one another by swapping components in random locations. This results in an offspring which is the combination of both parents. Source."Mutation" is another important concept in evolution, which involves the random modification of genetic information in a child offspring. Mutations can take a variety of forms: certain genetic information can be duplicated, removed, flipped around, etc.Chromosomes can undergo mutation throughout an organisms life. Source.Mutations can have no effect on an organism, can have a massive negative impact, or can occasionally introduce new traits in a species which are tremendously advantageous. The ability to see color, for instance, was the result of a single example of genetic material being produced four times.the human eye uses four genes to make structures that sense light: three for cone cell or colour vision and one for rod cell or night vision; all four arose from a single ancestral gene. — SourceMost mutations cause no impact, or a negative impact. However, variation is vitally important for the species to maintain genetic diversity and evolve, so some mutation is critical for evolution to function.That's how evolution works in biology in a nutshell. Let's explore how genetic algorithms manipulate these ideas to solve complex problems."Genetic Algorithms" (GAs) leverage the core principles of evolution to solve problems. The essential idea is to turn a problem into something resembling a chromosome, perform survival of the fittest, create breeding pairs and perform crossover, and then perform some random mutation to create a new population.Let's explore this through a practical example.Imagine you were in charge of creating a schedule in a university. There are a bunch of rooms, classes, and teachers to teach them. Certain teachers may be capable of teaching only a handful of classes, certain classes need to be held in certain classrooms, etc. We can use a GA to help us solve this problem.First, we can imagine the problem like DNA, where there's a big list of locations and certain information that can exist in each of those locations. We can imagine each time slot in each classroom as a location, and the class and teacher as the information.We can start off with a few randomly defined schedules, creating our initial "population"We cab create a "population" of schedules by creating many different examples randomly.Then, we can define something called a "fitness function". We can look through our schedule and make some basic rules:If a professor is teaching two classes at the same time, -20 fitness pointsIf a class is being held in a room that's too small, -10 fitness pointsetc.We can apply this fitness function to our population, and preserve only the members of the population who are most fit.We can create a score for each member of our population based on the fitness function. If we had 100 schedules in our population, we could perform selection by preserving the schedules in the top 50% of scores.Once we have a relatively fit population, we can randomly create breeding pairs and perform crossover on their information.In our GA, we can perform crossover by randomly swapping over information within our list of schedules.Then we can perform random mutation to inject some genetic diversity into the population.To perform mutation, we can randomly change information in some of the schedules in our population.If we do this, many many times, we'll create successive generations which will slowly evolve to better fit our fitness function, thus giving us a good schedule!Let's code up this scheduling example and see how it works practically.Have any questions about this article? Join the IAEE Discord.Join The DiscordFirst, let's define the problem we're dealing with. Full code can be found here.Imagine we have some collection of teachers, each of which is qualified to teach a certain set of subjects.# teachers, and the subjects they can teach teachers = { 'Morgan': ['math', 'physics', 'statistics', 'computer science'], 'Jordan': ['english', 'creative writing', 'drama', 'debate'], 'Casey': ['biology', 'chemistry', 'psychology', 'sociology'], 'Riley': ['history', 'geography', 'economics', 'business'], 'Quinn': ['art', 'music', 'theater', 'french', 'german', 'spanish'], 'Taylor': ['philosophy', 'ethics', 'english', 'health', 'physical education'], 'Alex': ['robotics', 'engineering', 'environmental science'] }Those teachers have to teach some number of classes, where each class has to be taught some number of times a week, may or may not need a computer lab, has a certain class size, etc.classes = { 'physics1': {'type': 'physics', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 20}, 'modern sculpture': {'type': 'art', 'weekly_instances': 2, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 15}, 'jazz': {'type': 'music', 'weekly_instances': 2, 'requires_media': True, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 16}, 'calculus AB': {'type': 'math', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 18}, 'british literature': {'type': 'english', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 22}, 'french 1': {'type': 'french', 'weekly_instances': 2, 'requires_media': True, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 17}, 'intro to psychology': {'type': 'psychology', 'weekly_instances': 2, 'requires_media': False, 'requires_computers': True, 'requires_chemlab': False, 'class_size': 20}, 'principles of sociology': {'type': 'sociology', 'weekly_instances': 2, 'requires_media': False, 'requires_computers': True, 'requires_chemlab': False, 'class_size': 19}, 'introduction to programming': {'type': 'computer science', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': True, 'requires_chemlab': False, 'class_size': 20}, 'AP biology': {'type': 'biology', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': True, 'class_size': 20}, 'chemistry fundamentals': {'type': 'chemistry', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': True, 'class_size': 21}, 'debate and rhetoric': {'type': 'debate', 'weekly_instances': 2, 'requires_media': False, 'requires_computers': False, 'requires_chemlab': False, 'class_size': 18}, 'statistics and probability': {'type': 'statistics', 'weekly_instances': 3, 'requires_media': False, 'requires_computers': True, 'requires_chemlab': False, 'class_size': 23} }And each of those classes needs to be taught in a particular classroom:rooms = { 'Room 101': {'capacity': 30, 'has_media': False, 'has_computers': False, 'is_chemlab': False}, 'Room 102': {'capacity': 25, 'has_media': True, 'has_computers': False, 'is_chemlab': False}, 'Room 105': {'capacity': 30, 'has_media': True, 'has_computers': True, 'is_chemlab': False}, 'Art Studio A': {'capacity': 20, 'has_media': True, 'has_computers': False, 'is_chemlab': False}, 'Drama Stage': {'capacity': 25, 'has_media': True, 'has_computers': False, 'is_chemlab': False}, 'Gym A': {'capacity': 30, 'has_media': False, 'has_computers': False, 'is_chemlab': False}, 'Computer Lab 2': {'capacity': 20, 'has_media': False, 'has_computers': True, 'is_chemlab': False}, 'Chem Lab A': {'capacity': 20, 'has_media': False, 'has_computers': False, 'is_chemlab': True}, 'Lecture Hall 2': {'capacity': 35, 'has_media': True, 'has_computers': False, 'is_chemlab': False}, 'Language Lab': {'capacity': 20, 'has_media': True, 'has_computers': True, 'is_chemlab': False}, 'Philosophy Room': {'capacity': 18, 'has_media': False, 'has_computers': False, 'is_chemlab': False} }The goal is to come up with a good schedule, which is hard. If each room is available for 8 one hour time slots, 5 days a week, that's 440 possible time slots in which one of 15 classes can be taught (including no class at all), and one of 8 teachers (including no teacher at all).Doing some back-of-the-napkin math, that's 120^{440} possible schedules. There are roughly 10^{82} atoms in the observable universe, meaning brute forcing this problem would be much harder than counting every atom in every known galaxy one at a time. Let's solve this problem in a few minutes by making schedules have sex.First, we need to represent this problem in a way that's similar to how DNA is structured. We can think of every possible time slot a room can have as a location in our DNA"""Turning the problem into a structure that's similar to DNA """ import pandas as pd # Parameters num_periods = 8 num_days = 5 # Generate timeslots like "Period 1_Day 1", "Period 2_Day 3", etc. timeslots = [f'Period {p+1}_Day {d+1}' for d in range(num_days) for p in range(num_periods)] # Create list of room-timeslot pairs data = [{'room': room, 'timeslot': timeslot} for room in rooms.keys() for timeslot in timeslots] # Create DataFrame df = pd.DataFrame(data) dfWe can then create random members of our population by simply filling in random classes and professors in each time slot."""Creating a random population member """ import random def create_population_member(): # Pull class names and teacher names class_names = list(classes.keys()) teacher_names = list(teachers.keys()) # Define probability of assigning None vs a valid entry none_prob = 0.2 # 20% chance of being None # Randomly assign class and teacher df['class'] = [random.choice(class_names + [None] * int(len(class_names) * none_prob)) for _ in range(len(df))] df['teacher'] = [random.choice(teacher_names + [None] * int(len(teacher_names) * none_prob)) for _ in range(len(df))] return df.copy(deep=True) create_population_member()Creating an initial population simply consists of calling this function numerous times, thus creating a set of random schedules.population = [create_population_member() for _ in range(100)]Now that we have a random population, we need a way to calculate how "fit" a certain member of that population is. This can be done by simply iterating through all of the time slots and taking away points for things that are bad about the schedule."""Defining the fitness function. Here I'm calling the input "chromosome", but it's simply a particular member of the population. The result is a number, where 0 is a perfect fitness and imperfection results in larger negative numbers. """ def fitness(chromosome): penalty = 0 teacher_timeslot_set = set() class_instance_counter = {} for idx, row in chromosome.iterrows(): class_name = row['class'] teacher = row['teacher'] room = row['room'] timeslot = row['timeslot'] if class_name is None or teacher is None: continue # No assignment, skip # Track how many times this class is scheduled class_instance_counter[class_name] = class_instance_counter.get(class_name, 0) + 1 # 1. Teacher qualified? class_info = classes[class_name] class_type = class_info['type'] if class_type not in teachers.get(teacher, []): penalty += 10 # 2. Room resources room_info = rooms[room] if class_info['requires_media'] and not room_info['has_media']: penalty += 5 if class_info['requires_computers'] and not room_info['has_computers']: penalty += 5 if class_info['requires_chemlab'] and not room_info['is_chemlab']: penalty += 5 # 3. Room capacity if room_info['capacity'] < class_info['class_size']: penalty += (class_info['class_size'] - room_info['capacity']) # 4. Double-booked teacher key = (teacher, timeslot) if key in teacher_timeslot_set: penalty += 15 else: teacher_timeslot_set.add(key) # 5. Penalize over/under-scheduling of classes for class_name, class_info in classes.items(): required = class_info['weekly_instances'] actual = class_instance_counter.get(class_name, 0) if actual != required: penalty += abs(required - actual) * 5 # Penalty per instance off return -penalty fitness(population[0])Here, I'm penalizing a schedule for the following:if a teacher teaching a class is not qualifiedif a class requires a certain type of resources that the room it's in doesn't haveif the room is smaller than the class size it holdsif a teacher is double-bookedif a class is being offered more or less often than it is supposed to beThe exact schedule we are optimizing for is directly defined in this fitness function, which is why GAs can be so powerful. By modifying these simple rules, we can change the nature of the schedule we get. If teachers complain about having too many classes without a break? Add a rule!Once our fitness function has been defined, we can simply apply it to all of the members of our population (our random schedules) to preserve only the best ones."""Applying survival of the fittest """ def selection(population): # Evaluate fitness for each chromosome scored_population = [(chromosome, fitness(chromosome)) for chromosome in population] # Sort by fitness (higher is better) scored_population.sort(key=lambda x: x[1], reverse=True) # Keep top 50% num_survivors = len(population) // 2 survivors = [chromosome for chromosome, _ in scored_population[:num_survivors]] return survivors survivors = selection(population) len(survivors)The beginning population was of size 100. This particular selection function preserves the top 50% in the population. There are many different types of selection functions to choose from (like probabilistic preservation), but top 50% preservation is my favorite.Now that half of their friends died, our schedules are probably feeling frisky. We can randomly create breeding pairs and have them perform crossover to create new offspring, bringing our population back to its original size."""performing crossover by randomly selecting a breeding pair and creating an offspring. Here, crossover_rate dictates how much swapping of information happens. 0.5 will result in an even division of both parents, while larger or smaller numbers will bias parent 1 or 2. """ import random def crossover(survivors, crossover_rate=0.5): original_population_size = len(survivors) * 2 offspring = [] while len(offspring) < len(survivors): # Pick two distinct parents parent1, parent2 = random.sample(survivors, 2) # Create a new child DataFrame child = parent1.copy() for idx in child.index: if random.random() < crossover_rate: child.at[idx, 'class'] = parent1.at[idx, 'class'] child.at[idx, 'teacher'] = parent1.at[idx, 'teacher'] else: child.at[idx, 'class'] = parent2.at[idx, 'class'] child.at[idx, 'teacher'] = parent2.at[idx, 'teacher'] offspring.append(child) return survivors + offspring new_population = crossover(survivors) len(new_population)Here, we're performing crossover by randomly swapping the class and teacher at a particular time slot of a particular room. The exact way crossover is done can be adjusted based on the application.Now that we've done crossover, we can mutate the population by injecting some random variance in the data. Just like crossover, the way you do this can depend on your application."""Randomly adjusting the classes and teacher for a given timeslot to promote better results, I only choose a teacher that is qualified to teach the new class. """ import random def mutate(population, mutation_rate=0.01): class_names = list(classes.keys()) for chromosome in population: for idx in chromosome.index: # Mutate class and teacher for a given classroom timeslot if random.random() < mutation_rate: new_class = random.choice(class_names + [None]) chromosome.at[idx, 'class'] = new_class if new_class is None: possible_teachers = list(teachers.keys()) + [None] else: class_type = classes[new_class]['type'] possible_teachers = [t for t, qualified_types in teachers.items() if class_type in qualified_types] + [None] if possible_teachers: chromosome.at[idx, 'teacher'] = random.choice(possible_teachers) return population mutated_population = mutate(new_population) len(mutated_population)I chose to simply iterate through each time slot for every room and occasionally change both the class and the teacher teaching that class. Notice that None is a possible selection of both the teacher and the classroom. The way the fitness function is defined, if either the class or the teacher is None, then that particular timeslot for the room is treated as empty."""From the fitness function """ if class_name is None or teacher is None: continue # No assignment, skipSo, this mutation function can add new classes and teachers to classroom timeslots, and can also make classroom time slots empty. The mutation_rate defines how likely it is for a particular time slot to be modified.So, we can create an initial population of random schedules, calculate their fitness, perform selection, crossover, and mutation. That's everything that's needed in a GA, so let's hook it all up."""Optimizing schedules by creating an initial population and iteratively performing selection, crossover, and mutation """ from tqdm import tqdm # --- Set up GA loop --- population_size = 40 generations = 300 #creating initial random population population = [create_population_member() for _ in range(population_size)] # for plotting, later avg_fitness = [] min_fitness = [] max_fitness = []for gen in tqdm(range(generations), desc="Evolving", dynamic_ncols=True): # calculating fitness of population scores = [fitness(chrom) for chrom in population] # keeping track of key data, for plotting avg = sum(scores) / len(scores) mn = min(scores) mx = max(scores) avg_fitness.append(avg) min_fitness.append(mn) max_fitness.append(mx) # making pretty output tqdm.write(f"Gen {gen} — avg: {avg:.1f}, min: {mn:.1f}, max: {mx:.1f}") # performing selection, crossover, and mutation to create a new population survivors = selection(population) population = crossover(survivors, crossover_rate=0.7) population = mutate(population, mutation_rate=0.005)The initial population starts off performing poorly, which is expected because they're completely random, but after a while, the population converges on near-perfect schedules.If we plot out the history of fitness over time, we can see a consistent and steady improvement of the GA.import matplotlib.pyplot as plt # --- Plot results --- plt.plot(avg_fitness, label='Average Fitness') plt.plot(min_fitness, label='Min Fitness') plt.plot(max_fitness, label='Max Fitness') plt.xlabel('Generation') plt.ylabel('Fitness') plt.title('GA Fitness over Generations') plt.legend() plt.grid(True) plt.show()Recall that a fitness of 0 is a perfect schedule, so our GA did a pretty good job! You could take any schedule from the final population and it would be pretty solid. Of course, the "quality" of a schedule is defined by our fitness function. If we found that teachers were being forced to run back and forth across the school, we would want to make revisions to our fitness function.You might notice that the fitness never quite reaches zero. That's because, as a schedule becomes more perfect, it's harder to find mutations that result in an improvement; most mutations harm the quality of a schedule in some way. If you let this run for a while you might find a good solution, or you could play around with the mutation rate and population size to perhaps converge on a perfect solution more quickly. GAs tend to run pretty quickly, so that type of experimentation is easy to do.For now though, let's explore how GAs can be applied in another type of problem.The "traveling salesman" is a famous problem in graph theory. The core idea is that you might have many cities connected to each other, and you need to figure out which route you should take to visit all cities in as little time as possible.In this example, we'll imagine that traffic patterns between the cities change over time. As our salesman is traveling from city to city, these traffic patterns will change. As we plan our route between cities, we ought to account for those traffic patterns.I'm sure there's some clever way to optimize for this, but I don't know it. One of the cool things about genetic algorithms is how easy they are to apply to problems, so let's just throw a GA at it and see what happens!First, we need to define the data we're playing with. The whole idea is that there's a route between each city and the time it takes to traverse from one city to another changes over time. I'm actually storing that information as a directed graph, where each edge has a list of traversal times for every hour in the day. We can get the graph of traversal times at a particular time of day by sampling this graph."""Defining a graph with changing traffic patterns over time. """ import random import networkx as nx import matplotlib.pyplot as plt def generate_time_dependent_graph(cities): graph = {} for from_city in cities: graph[from_city] = {} for to_city in cities: if from_city != to_city: graph[from_city][to_city] = [random.randint(1, 4) for _ in range(24)] return graph def get_graph_at_hour(graph, hour): hourly_graph = nx.DiGraph() for from_node, edges in graph.items(): for to_node, weights in edges.items(): hourly_graph.add_edge(from_node, to_node, weight=weights[hour]) return hourly_graph def draw_graphs_for_hours(graph, hours): fig, axes = plt.subplots(1, len(hours), figsize=(16, 4)) pos = nx.spring_layout(nx.DiGraph(graph)) # fixed layout for all graphs for idx, hour in enumerate(hours): G_hour = get_graph_at_hour(graph, hour) ax = axes[idx] ax.set_title(f'Hour {hour}') nx.draw(G_hour, pos, ax=ax, with_labels=True, node_color='lightblue', node_size=800, arrows=True) labels = nx.get_edge_attributes(G_hour, 'weight') nx.draw_networkx_edge_labels(G_hour, pos, edge_labels=labels, ax=ax, font_size=8) plt.tight_layout() plt.show() # Create the graph and visualize it cities = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M'] time_dependent_graph = generate_time_dependent_graph(cities) # Render for 4 sample hours draw_graphs_for_hours(time_dependent_graph, hours=[0, 6, 12, 18]]If this looks like french to you, check out my article on graphs:And, that's really the most difficult part. Defining a GA to solve this problem is virtually identical to the previous example we discussed.Just like in the previous example, we'll need to format the structure into something resembling DNA, define a fitness function, do selection, crossover, and then mutation.Defining the DNA is pretty straightforward, it can just be a list of cities in some order. Here I'm calling it a genome rather than DNA, the analogies between GAs and their biological inspiration can get a bit handwavy."""Defining initial population """ genome_length = 2 * len(cities) population = [[random.choice(cities) for _ in range(genome_length)] for _ in range(pop_size)]Here, I decided to make the genome long because, in theory, it might be optimal to travel through a city than to drive directly from one city to another. I wanted to allow for some room in a population member to have long lists and be able to explore those types of solutions.The fitness function, evaluate_genome takes a particular genome (aka population member, aka DNA, aka solution… Whatever). and assigns a fitness score to it.def evaluate_genome(genome, graph_data, start_city): current_time = 0 current_city = start_city visited = set([current_city]) for next_city in genome: if next_city == current_city: continue graph_at_time = get_graph_at_hour(graph_data, current_time % 24) if not graph_at_time.has_edge(current_city, next_city): continue travel_time = graph_at_time[current_city][next_city]['weight'] current_time += travel_time current_city = next_city visited.add(current_city) if len(visited) == len(graph_data): break return -current_time if len(visited) == len(graph_data) else -float('inf') def mutate(genome, cities, mutation_rate=0.1): new_genome = genome[:] for i in range(len(new_genome)): if random.random() < mutation_rate: new_genome[i] = random.choice(cities) return new_genome def crossover(parent1, parent2): return [random.choice([g1, g2]) for g1, g2 in zip(parent1, parent2)] def run_generation(population, graph, cities, start_city, mutation_rate=0.1): scored_population = [(genome, evaluate_genome(genome, graph, start_city)) for genome in population] scored_population.sort(key=lambda x: x[1], reverse=True) next_generation = [genome for genome, _ in scored_population[:2]] # elitism while len(next_generation) < len(population): parent1, parent2 = random.choices(scored_population[:5], k=2) child = crossover(parent1[0], parent2[0]) mutated_child = mutate(child, cities, mutation_rate) next_generation.append(mutated_child) return next_generation, scored_population[0] def genetic_algorithm(graph, cities, start_city='A', generations=10, pop_size=500, mutation_rate=0.1): genome_length = 2 * len(cities) population = [[random.choice(cities) for _ in range(genome_length)] for _ in range(pop_size)] best_overall = (None, -float('inf')) for gen in range(generations): population, best = run_generation(population, graph, cities, start_city, mutation_rate) if best[1] > best_overall[1]: best_overall = best print(f"Generation {gen+1}: Best fitness = {best[1]}") print("Best path found:", best_overall[0]) print("Final fitness:", best_overall[1]) return best_overall # --- Run the GA on your existing graph --- best_path, best_fitness = genetic_algorithm(time_dependent_graph, cities)Notice that there are some repeated characters in the best solution found. Because of the way we define our fitness function, these are simply skipped. Also, we stop counting once all the cities have been accessed. So, the real result is:L, F, D, K, B, M, C, E, J, G, I, H, F, K, J, AWhere each letter represents a city. That's 21 hours to go through all 16 cities. One drawback of GAs is that, often, we have no idea if the solution we end up with is optimal. That said, if you're working in a use case where "good enough" is good enough, then GAs are a fantastic choice. Considering the average time to traverse from one city to another is two hours, that result is much better than average!In this article we watched schedules have sex like a bunch of creeps.Haven't you people ever heard of, closing the god damn door, no? Jokes aside, we actually learned about a very powerful approach to solving arbitrary problems. If you can turn your problem into a representation that is similar to DNA, and define a fitness function that makes some amount of sense, you can use genetic algorithms to quickly find compelling solutions! In this tutorial, we'll first define some fundamental properties of genetic algorithms. Secondly, we'll review how they are constructed. Then we'll discuss how they work. Lastly, we'll review some real-life applications of genetic algorithms. Genetic algorithms are mostly applicable in optimization problems. This is because they are designed to search for solutions in a search space until an optimal solution is found. In particular, genetic algorithms are capable of iteratively making improvements on solutions generated until optimal solutions are generated. Now let's look at the formal definition of genetic algorithms. 2. Definitions Genetic algorithms are heuristic algorithms inspired by the natural process of evolution. This theory of evolution was first proposed by Charles Darwin in the mid 19th century. Evolution describes the change in the biological characteristics of species over a generation through natural selection. Consequently, genetic algorithms are based on natural selection. Where only the fittest individuals in a population are selected to reproduce and generate offspring. 3. How Do They Work? Genetic algorithms follow the natural process of evolution, and are described as follows: Now let's look at the steps in a basic genetic algorithm. 3.1. Algorithm The first step is to initialize the population. In the case of problem-solving, a set of solutions to the problem at hand is the initial population. Secondly, we evaluate the optimality of the population using a fitness function. The fitness function is an objective function that evaluates the quality of the solutions. The ultimate goal is to find the maxima or minima of this function . Take, for instance, a fitness function that assigns a score to the solutions in the population. Then, our goal here is to reach a global maximum for the function, . If the solutions from the population are satisfactory, we usually stop the algorithm and crown these as the best individuals. Now, if the solutions are not satisfactory, we perform the process of selection and pick the fittest individuals to reproduce and generate new solutions. This is done iteratively. Next, after selection, the fittest individuals reproduce through crossover, to generate offspring. The word offspring here refers to a new generation of solutions. During crossover, values are exchanged to generate new individuals. Then, certain offspring produced will undergo mutation. This is a process of randomly changing the values or characteristics in the offspring to introduce diversity. Specifically, an iterative application of the mutation operation will allow the algorithm to get out of a local minimum during the search. It is important to note that crossover and mutation are the two main methods used to generate offspring in genetic algorithms. Lastly, if the solutions or offsprings are satisfactory and there are no better offsprings to produce, the algorithm terminates and presents the best individuals as the optimal solutions. 4. Applications Genetic algorithms are one of the most fundamental algorithms in computer science. Consequently, they have found many applications in the real world in different industries and for different tasks. However, we'll only discuss a few in this article. 4.1. Robotics Robotics encompasses the design, construction, and operation of autonomous robots. In robotics, genetic algorithms are used to provide insight into the decisions a robot has to make. For instance, given an environment, suppose a robot has to get to a specific position using the least amount of resources. Genetic algorithms are used to generate optimal routes the robot could use to get to the desired position. 4.2. Economics Economics is the science of the use of resources in the production, distribution, and overall consumption of goods and services. In economics, genetic algorithms are used to create models of supply and demand over periods of time. Additionally, genetic models are also used to derive game theory and asset pricing, models. 4.3. Automated Design Automated design constitutes the design and production of automobiles such as cars. In particular, a car manufacturing company may have specifications on how a car should operate. As an example, minimum fuel consumption is a desirable specification in the design of vehicles. As a result, genetic algorithms are used to derive designs of automobiles that satisfy constraints such as low fuel consumption. 4.4. Scheduling Tasks Suppose, we have a task to schedule the timetable for a university. The goal of a scheduling task is to find an optimal solution that satisfies certain constraints. Genetic algorithms are used in this aspect to derive optimal schedules considering the courses, number of students, and lecture rooms that the university has. 4.5. Vehicle Routing In most logistics companies, one main task is vehicle routing. Here, we have to figure out how to route and transport goods to different clients using a fleet of vehicles. Consequently, genetic algorithms are used to derive cost-effective routes to transport these goods to the rightful client at the right time. 4.6. Marketing In marketing, the main idea is to promote goods and services in such a way that it will gain a lot of customers or clients. Genetic algorithms are used here to derive the best combinations of product specifications and attributes that will attract the most customers. 4.7. Medicine In medicine and bioinformatics, genetic algorithms have been used to identify benign and malignant tumours in ultrasounds and mammograms. 5. Conclusions In this article, we first defined evolution, then genetic algorithms. We've discussed the basic components and how they work. Lastly, we reviewed some real-life applications of these algorithms. Genetic algorithms are a part of a family of algorithms for global optimization called Evolutionary Computation, which is comprised of artificial intelligence metaheuristics with randomization inspired by biology. Wow, words can really be arranged in any order! But hang in there, we'll break this down: Global optimization is a branch of applied mathematics used to find global minimums or maximums of functions. To find these values in a reasonable time efficiency, we use artificial intelligence optimizations. Many things can be expressed as functions, which allows us to solve a variety of problems with optimizations. Evolutionary Computation is a family of algorithms for optimization, which are specifically inspired by biology. Genetic algorithms are designed to simulate mutation and natural selection, but other kinds of algorithms simulate behaviors of ants, bees, wolves, and the like, as well as many different variations and implementations of each of those. Artificial intelligence is, most commonly, a branch of computer science and a designation for algorithms which deal with problems where there is combinatorial explosion. Those problems cannot be solved in a reasonable time with classical algorithms, so artificial intelligence is all about devising correct solutions based on some unusual mathematically provable properties of our algorithms, or approximating solutions using metaheuristics. A metaheuristic is a higher order heuristic, designed to be a pattern for creation of heuristics. Heuristics are techniques for approximating a solution of a problem with a much better time complexity than if you were to solve for the exact solution. So we use a metaheuristic to create heuristics for all sorts of different problems. Sheesh, that's a lot to take in! The good news is you won't really need it to understand the meat of the article, but it was included to give you a wider picture of the context in which these sorts of algorithms exist, and give you an appreciation for the vastness of the field of artificial intelligence. Basic Concepts Genetic algorithms, as mentioned, were inspired by evolution and natural selection, and aim to emulate it. The basic idea is to represent the domain of possible solutions as a discrete genome - a finite array of genes - and then figure out which of those possible solutions is the correct one. You figure this out by creating a random population of solutions and 'rating' those solutions in some manner, and then combining the best solutions into a new one to create an even better generation of solutions, until the 'rating' is satisfactory. Said rating is referred to as fitness, while combining solutions is called reproduction or crossover. Because the algorithm is based on randomness, it's possible for it to accidentally converge on a wrong solution. To avoid that, we randomly perform mutation on a small percentage of our genomes to increase the likelihood that we'll find the right solution. Genetic algorithms can be applied on virtually any search problem, but it's often said that genetic algorithms are the second best solution to every problem. What this adage is getting at is that genetic algorithms are fairly easy to implement, but may not be as efficient as an algorithm hand-crafted for a particular problem. However, when hard problems are concerned, it can take quite a long time to actually create a perfect solution. Sometimes we prefer to make a genetic algorithm in an hour or two and let it run for half an hour, than to spend days or weeks analyzing math properties of a particular problem to design an efficient algorithm, to then have it still take ten minutes or something of runtime. Of course, if a particular problem has an already known solution, or if the runtime of the algorithm is vitally important, genetic algorithms may not be your ideal solution. They're mostly used in problems with huge computational needs where the solution can be good enough, and doesn't need to be perfect. As an example of where you may apply a genetic algorithm, look at the following graph representing a 2D height map of a cliff top: Let's say we want to find the maximum of the function f on the given segment. However, checking every point in the segment is impossible because there's uncountably infinite real numbers between any two different real numbers. Even if we say we'll be happy with an approximate answer, and we may just check the value of f(x) for a million values of x and take the maximum, that could in some scenarios be a very expensive operation. For example, if each point of the mountain had to be scaled and its height measured by hand, let's just say your assistant would grow tired of you a few measurements short of a million. So what would be a good way to guess some nice values of x to measure so that we don't have to climb quite so many times, but can still arrive at a pretty good solution? Genetic Representation In order to be able to use the genetic algorithm, we need to represent it in some way. Different species have a different number of chromosomes, each containing vital information about the construction of the specimen. In our case, we typically won't need more than a single chromosome to encode our candidate solution. Another term used for the candidate solution is the genome. The genome needs to be represented in a way which allows us to easily generate a valid genome randomly, calculate its fitness quickly, and reproduce and mutate specific genes. Of course, you could technically let your algorithm run with invalid solutions in the population and hope that they'll be weeded out, but it's simply inefficient and usually unnecessary. A common way to represent a genome is an array of binary digits. This representation is great because then we can use fast binary operations to work with it, and it's very intuitive to envision how it evolves. For example, given a segment [a,b] and a function f(x) defined on that segment, we could define the leftmost point of the function, which is a, to be represented as 0000000000 (ten zeros), and we could say the rightmost point b is 1111111111 (ten ones). There's $2^{10}=1024$ points that we can denote with these arrays of length 10. Let's say length([a,b])/1024 = l. Then we could represent a+l as 0000000001, a+2l as 0000000010, and so on. If p is the value of a binary number, we can calculate the corresponding real value of x with the following formula: $$ x=a+\frac{p}{2^n-1}(b-a) $$ On the other hand, to assign a binary representation to a number from the interval [a,b], we'd use the following equation: $$ p=\Bigg[\frac{x-a}{b-a}(2^n-1)\Bigg] $$ There are many possible ways to represent a genome, and the convenient one to use will depend on the specific problem you are faced with. It's important to remember that a genetic algorithm isn't just one algorithm, but a metaheuristic, which means that the point of this article is for you to understand the way of thinking behind it, not the particular examples. For example, let's say your algorithm was supposed to guess a 5 letter word and it can know how many letters it got correct. It'd be pretty natural to use a string as your genome in that case. If you were trying to teach it to jump over holes in a game, you may use an array of booleans, where true means jump and false means run, though again, you could map it so 1 means jump and 0 means run. Population Each generation is a collection of usually an equal number of genomes. This collection is typically called a population of candidate solutions - or population and individuals. The initial generation is populated with completely randomly generated individuals, and uniformly distributed across the search space. Sometimes we can more precisely guess where the solution will be, so we can create more adequate genomes from the get go. Sometimes, we have additional conditions that a valid specimen has to fulfill. It is preferred to generate the genome so that it necessarily fulfills those conditions, over performing checks and fixes after generating it, because that wastes a lot of time and generation sizes are usually huge. Fitness Function and Objective Function In order to assess which of our genomes should go on into the next generation through reproduction or another means, we need a function to calculate their value in a way that allows us to compare values of two different genomes. This function is called a fitness function and we can denote it as f(x). Although it's not quite our f(x) from the cliff top picture, it's meant to approximate it. It's usually always positive, and the larger the number the better the genome. When we use such a fitness function, we're performing maximization on the search space - looking for maximum value of fitness. The objective function is quite similar to fitness function, and in a lot of cases they're the same, but sometimes the distinction is important. The objective function is used to calculate the fitness of the best genome in each generation (the one with the maximum fitness function value) in order to check whether it satisfies a predetermined condition. Why use two different functions? Well, because the fitness function is performed on every genome in every generation, it's very important for it to be fast. It doesn't have to be very precise, as long as it more or less sorts the genomes by quality reasonably well. On the other hand, the objective function is called only once per generation, so we can afford to use a more costly and more precise function, so we'd know for sure how good our result is. The objective function would be our f(x) on the cliff top picture, while the fitness function would be its close approximation. Selection Selection is a method used to determine and transfer the good attributes of one generation to the next one. Not all individuals in a population are allowed to reproduce, and we have to be mindful of various things when picking which ones will carry over their genes into the next generation. The first idea would, of course, be to just take the top, let's say 25%, and have them reproduce. The problem with this method is that it very often causes what's called early convergence. For example, look at the picture below: Check out our hands-on, practical guide to learning Git, with best-practices, industry-accepted standards, and included cheat sheet. Stop Googling Git commands and actually learn it!If all of the solutions in the current generation are in the blue area and we just pick the highest fitness ones, we're going to end up picking the ones on the local maximum. The ones to the left, which are a bit worse when it comes to fitness, but are approaching the real solution, are going to be left out of the next generation. With each generation the blue area is going to get more and more narrow because we'll be combining solutions that are within it, until we eventually stall at the local maximum. We're trying to find the global maximum (labeled 'actual solution'), so this is undesirable. In order to avoid this, we use special selection methods. Roulette Selection A good way to select the fittest genomes would be to select them with the probability proportional to their fitness. This way, even less fit genomes will have a chance to be selected, but it will be a smaller chance. This is akin to a roulette where the pie slices are not equal. In the picture above, the genome labeled c has the greatest fitness, and therefore it takes up the greatest part of the roulette. The probability that each genome i will participate in reproduction (that it will win the roulette) is: $$ p=\frac{f(i)}{\sum_j^N f(j)} $$ In other words, it's the fitness of said genome, divided by the summed up fitness of the entire generation. Because a fitness function is always positive, this number will be between 0 and 1. The way we achieve this in the code is to generate a random positive number n, smaller than the sum total fitness of the generation. We then go through our generation and add their fitness one by one to another sum. When that sum reaches or surpasses n, we take the current genome as the winner. Tournament Selection In tournament selection, we pick k random genomes to participate in a tournament, and select the winner. The higher the fitness of a genome, the more likely it is that it will win (or less likely, if we're doing minimization). There are different types of tournaments: Deterministic tournament always selects the best genome in a tournament. This is essentially just looking for a genome with maximum or minimum fitness. 1-way tournament is a tournament with only one competitor, and it's equivalent to stochastic (random) selection. Fitness proportionate tournament sorts the genomes according to fitness and indexes them. The ith genome is then chosen with the probability: $$ p(1-p)^{i-1} $$ When deciding on tournament size,