The genetic algorithm

Introduction

In this project we are to create a genetic algorithm which can converge a random bit string of 0s and 1s to a bit string of only 1s. Example: if the random string is 01010, the algorithm should converge to a string of 11111. The length of the string should vary in experiments along with other parameters such as population size, mutation rate, or selection scheme. Statistics over at least 30 individual runs with a given parameter set should be retrieved and graphed for later discussion and comparison between different parameter sets. We focus on creating experiments with two selection schemes, along with changing the bit string length, mutation rate and population size. Since the assignment does not mention crossover, we have left it out of the project for focus on the selection schemes comparison.

Overview of the algorithm

In this problem we view an individual as a solution to the bit string problem. An individual is considered better, of higher fitness, if the individuals bit string contains more 1s. Inside the individual we consider the bit string as the genotype, the representation of the individual or the genome of the individual which we gradually change to improve next generation of individuals. The phenotype would describe what the genotype would result in, however, the problem doesn't discuss what this bit string might represent, and we therefore don't put energy into describing the resulting phenotype.

Before we start the algorithm, we decide a set of parameters, population size, mutation rate, bit string size, etc. Having defined the parameters, the algorithm starts by generating the population size of individuals. Each individual contains a bit string, the bit string is created randomly at the start of the algorithm. Then, the fitness score is calculated for each individual in the population. We define our fitness function as the count of 1s in a bit string for an individual. The lowest fitness score is 0 for no 1s in the genome, and max fitness score is the length of the bit string for all bits are 1.

Having the fitness scores of each individual, the population is with their corresponding fitness scores processed through a selection scheme. We use a rank or fitness proportional selection scheme (read more about these in details below). Both selection schemes select half the population size individuals from the current generation to be a part of the next generation. Whether an individual gets selected or not depends very much on its fitness score. For both schemes, higher fitness score means higher chance of getting selected. We also have the option of using elitism. When using elitism, we select a number of elites from the population which automatically are selected to the next generation population. Elites are defined as the best individuals in the current population. Elites are removed from the current generation and can therefore not be selected again for the next generation.

After selection, we stand with half the population size. The remaining half of the population is created by copying each individual of the half population and applying mutation to each of these copies. Mutation is applied by randomly flipping bits in the genome based on a mutation rate. When mutation have been applied to the copies the next generation population has been created.

The whole process of calculating fitness for individuals, selection, and mutation continuous until the max number of generations has been reached or a perfect individual appears (an individual which

only has 1s in its bit string). For biological comparison, we think our algorithm has more in common with asexual reproduction such as budding, then sexual reproduction.

Rank proportional selection scheme

For our rank proportional scheme, we want higher ranking individuals to have better chances of getting selected then lower ranking individuals. First, we rank all individuals of the population from worst to best. The worst individual is ranked 1 and the best is ranked with the population size. Having ranked the entire population, we define a probability of getting selected using the n-th triangular number. First, we calculate the population size n-th triangular number, then each individual is assigned a probability of getting selected by dividing its rank by that n-th triangular number.

For an example given the fitness scores of a population with 6 individuals: [47, 48, 41, 50, 60, 45]. The individuals are ranked from worst to best. Worst individual will be ranked 1 and best will be ranked 6. For this case the 3rd individual has lowest score and is ranked 1, while the 5th which has the best score is ranked 6. The n-th triangular number is defined as: $\sum_{i=1}^{n} i$, where n is the population size. For a population size of 6 individuals, the n-th triangular number becomes: 1+2+3+4+5+6 = 21. The individual's rank is divided by the n-th triangular number to assign selection probabilities in the following order from worst to best: $\frac{1}{21}\ \frac{2}{21}\ \frac{3}{21}\ \frac{4}{21}\ \frac{5}{21}\ \frac{6}{21}$.

Fitness proportional selection scheme

For our fitness proportional scheme, we want the selection chance to be proportional with the fitness of the entire population. We perform this selection by first calculating the entire populations sum fitness. Each individual is assigned a selection chance by dividing its fitness by the sum fitness of the population.

Considering the same fitness scores of a population as in the above example: [47, 48, 41, 50, 60, 45]. The sum population fitness is calculated to: 47+48+41+50+60+45 = 291. The individual's fitness is divided by the populations sum fitness to assign the selection probabilities in the following order of the population fitnesses: $\frac{47}{291}\ \frac{48}{291}\ \frac{41}{291}\ \frac{50}{291}\ \frac{60}{291}\ \frac{45}{291}$.

Experiments

We perform two types of experiments. The first experiment involves systematically testing different setups of the following parameters population size, mutation rate, elites, and selection scheme on a bit string length of 100 to try and find the fastest converging setup. The second experiment involves testing different setups on bits string lengths of 10 and 1000 based on the best findings in the first experiment. We define a parameter setup as better if it on average can generate the best solution faster. Each setup is run 30 times where the best fitness, the average fitness, and the standard deviation of the fitness per generation is collected across all individual runs. The maximum number of generations for each run is 10 000 generations.

Because the report is limited to at most 5 pages, it gets difficult to show any graphs as they take a lot of space. In the experiments table, each experiment has a label. These labels also refer to Python notebooks in the code project, which ran the experiment and has all of the corresponding graphs and statistics that followed with it. We refer to these graphs from time to time. Please, see notebook with corresponding file name as the experiment label to see graphs.

Experiment 1 results

The parameter setups were created and tested sequentially from top to bottom in the table. We collect the average number of generations taken to find the best solution and the time taken to complete the 30 individual runs. We collect time to complete the 30 runs as a statistic because a higher population size could lead to faster convergence, but bigger population size also means more processing time per generation, and therefore a lower number of generations doesn't necessarily mean faster convergence. All experiments are performed on 100 bit string lengths. If average generations equals -1, none of the runs were able to find the solutions.

Experiment label	Selection scheme	Population size	Mutation rate	Elites	Average generations	Time (min: sec)
Experiment1_1	fitness	10	0.01	0	-1	8:41
Experiment1_2	rank	10	0.01	0	7475	5:16
Experiment1_3	fitness	10	0.01	2	445	0:18
Experiment1_4	rank	10	0.01	2	334	0:13
Experiment1_5	rank	20	0.01	2	268	0:17
Experiment1_6	rank	6	0.01	2	479	0:13
Experiment1_7	Rank	10	0.1	2	-1	6:31
Experiment1_8	Rank	10	0.001	2	1018	0:41

Experiment 1 discussion

First, we evaluate how the fitness selection compares to rank selection using the same parameters for population size mutation rate and elites, see Experiment1_1 and Experiment1_2. As can be seen in the table above, the fitness proportional scheme wasn't able to find the solution in any of the 30 runs. The rank selection was, but it took a lot of generations to achieve. The reason fitness is unable to converge is because of the selection scheme. On random initialization the fitness scores will be very close to each other, and a better solution will therefore often not be that much better than the lowest fitness score. This results in the selection probabilities for the best solution isn't that much higher than the selection probabilities of the worst solution. This means when probabilities of are very even across all individuals, the chance of us picking a worse individual is about the same as picking the better individual. This results in extremely slow convergence or getting stuck in a local optima. Rank selection on the other hand is much more capable of giving better individuals a higher selection rate, even if the fitness is about the same for all individuals. For this case, no matter how much better the best individual is than the worst, it will always have 10 times the chance of getting selected over the worst.

Seeing that the algorithm seems to struggle with exploiting strong individuals, we add 2 elites to the setup of both fitness and rank, see Experiment1_3 and Experiment1_4. This way the 2 best individuals of each generation will always be members of the next generation. The next generations 2 best individuals will always have the same fitness or better if one is discovered. This has of course huge effects on the algorithms ability to converge on a solution. However, rank still seems to be doing a bit better, most likely because of the fact that it will give better selection rate ratio between the best and worst individual, then what fitness selection will.

Elitism can be used to “cheat” this problem. It will always be faster for the algorithm to just use half of the populations best or even just the best individual to create mutational offspring to create the better individuals. However, if we pick half the population as elites, we don’t get to test the selection schemes. Future experiments will as a maximum use 2 elites. Elitism might not be the best for other problems, but here it’s very effective as it secures the best solution stays in the population, and we don’t need much variation in population to get the best solution anyway.

Experiment1_5 and 1_6 alters the population size. A greater population size seems to reduce the number of generations needed to find the solution, but as be seen in time to complete the 30 runs, the algorithm actually runs a bit slower. This is because we need to perform more calculations on each generation, some of which might be unnecessary. Decreasing population size seems to increase the generations needed to find the best solution, but the time to complete doesn’t improve nor worsens over the current best time. The increase in generation probably happens because the algorithm doesn’t get as many new mutations as population with greater sizes. The increase in generations doesn’t mean much though as each generation still takes shorter time to process than that of greater population sizes.

With a mutation rate of 0.1 we are on average bound to flip 10 bits per generation in an individual, with a mutation rate of 0.001 we should on average only flip 0.1 bits per generations in an individual, see Experiment1_7 and 1_8. 0.1 mutation rate leads to us flipping to many bits, and it therefore gets increasingly more difficult for the algorithm to flips the right bits as the number of 1s increase in the genome. The algorithm seems to get stuck in a local optima around 90 1 bits from what we see in the plots. 0.001 mutation rate flips a lower amounts of bits per generation and therefore progresses slower which is the reason behind the increase in generations and time to complete.

Experiment 2 results

In the previous experiment we found that Experiment1_4 had the best setup for low number of average generations and short time to complete. We test this parameter set with a bit string length of 10 and 1000. However, we found that if the mutation rate is too great the solution won’t converge (we also testet this and the solution can’t converge using the same mutation rate). Therefore, we change the mutation rate to on average flip one bit per generation. So, 0.1 mutation rate for length 10, and 0.001 for length 1000. We run the algorithm for 30 runs.

Experiment label	Bit string length	Average generations	Time (min: sec)
Experiment2_1	1000	5593	20:22
Experiment2_2	10	14	0:01 >

Experiment 2 discussion

As the problem becomes bigger/harder it is to be expected that the algorithm is going to take a larger amount of time to complete it. Increasing the bit string length will make the problem harder but not unsolvable to algorithm as long as the mutation rate is right. The same goes for a bit string length of 10, which is much easier to solve. In previous experiments we figured that as long as we just flip one bit, the algorithm can solve the problem. When we consider a bit string of length 10, it doesn’t take much trial and error for the algorithm to hit the right bits. This is because one bit is

essentially 10% of the entire length. However, for the bit string length of 1000 one bit is 0.1 % of the entire length. The algorithm will need to flip a lot more bits on a length of 1000.

Adaptive mutation rate

Because of the increased time when increasing the bit string length, it was hypothesized that a higher mutation rate might be more appropriate in the early stages of the algorithm as possible more bits could be flipped into 1. A mutation rate method was developed which would flip more bits based on how low the individual's fitness was. The mutation rate was calculated such that in mutation the individual would flip on average the same number of bits as 0s in its genome. The mutation rate was calculated as such: $1 - (fitness \ / \ bit\_string\_length)$

This strategy turned out to be very inefficient, only converging +5000 generations or not converging at all, and only for the bit string length of 100. When we increase the number of bits to flip in one generation it becomes statistically unstable that we will make much progress. This happens because as the number of 1s grows in the bit string, the chance of us flipping a 1 to 0 increases, effectively setting the individual back in terms of fitness. When we only flip one bit it much more stable to make improvements. If it flips the wrong bit, the individuals will have worse chances of getting selected, if it flips the right bit it becomes a strong contender for the next selection, especially if it evolved from an elite. You can see the result of the adaptive mutation in other_notebooks\adaptive_mutation_rate.ipynb.

Conclusion

In this project we created a genetic algorithm which can evolve a random bit string into a bit string only containing 1s. The algorithm has different options for selection scheme and mutation scheme, although we do find in experiments that some options function better than other. Our experiments test different parameter setups of the algorithm and on different bit string lengths.

In general, we find that rank proportional selection works better than fitness proportional because ranked selection is better at giving the best individual better selection chances despite the fitness scores between individuals being close. Elitism works very well for keeping the progress we have gained in the population; however, we limit the number of elites because we still want to evaluate based on selection schemes.

Mutation rate seems to be one of the most important parameters that decide how fast and if at all the algorithm will converge. We find it best to have mutation rate that secures on average that 1 bit is flipped. This secures stable and fast progress. A to high mutation rate can lead to the model getting stuck in a local optimum and to low will lead to the algorithm converging slower. We tested a mutation scheme, which uses higher mutation rate at lower fitness levels, however, it didn't work well because of unstable progress. We conclude using higher mutation rate, even just in the start leaves nothing to be gained

At last, we test our best algorithm on different bit string lengths. As expected, the algorithm is able to converge faster when the problem is smaller and converges slower when the problem is bigger.

Code

https://github.com/Jollokim/evolution (please read README.md before running)