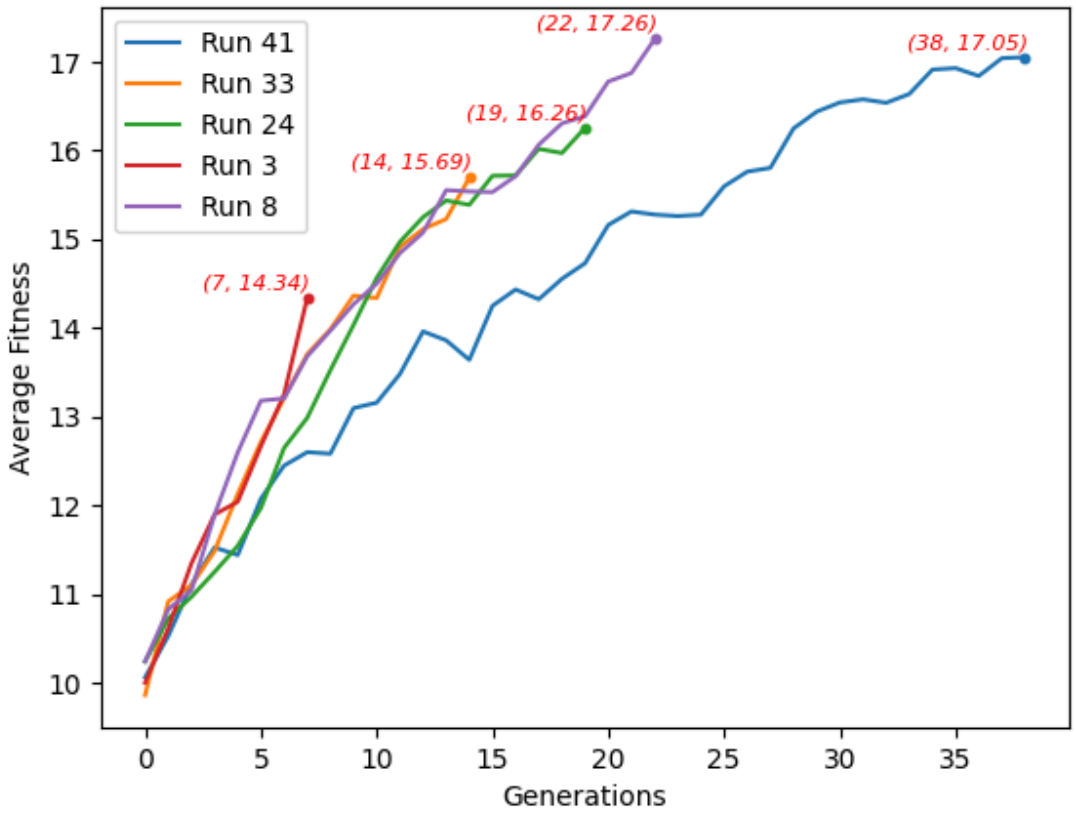
Kragen Wild
Part 1 b

i.
After doing 50 runs with a population size of 100, single-point crossover rate of 0.7, and a bitwise mutation rate of 0.001, these are my results.

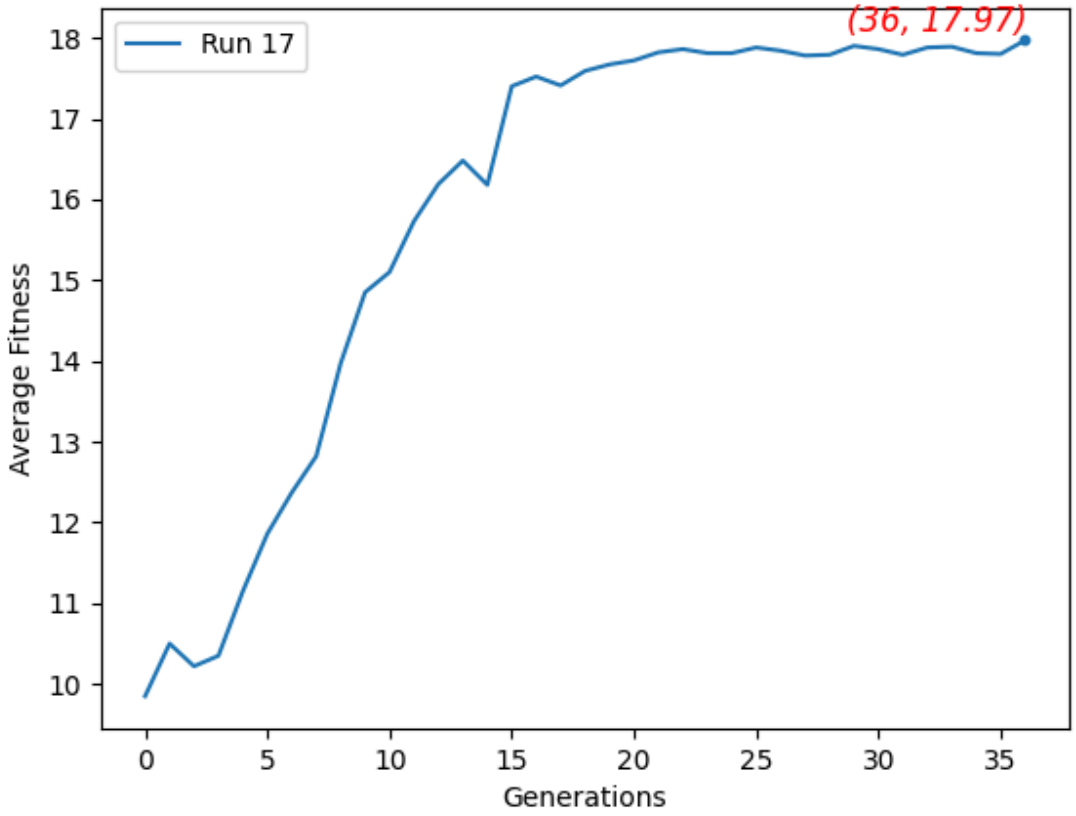
Max Generation: 49
Min Generation: 6
Average Generation: 25.49



I notice that they all have a similar shape, especially towards the beginning. They all start off by increasing from a fitness of ~10 very rapidly, but then the rate of growth slows down, and they level off towards the end. Runs that find the goal more quickly don't have as much of a chance to level out, while the ones that take much longer do. I assume that this is because as you have more generations, improvements are only marginal. All runs except run 41 have more or less the same exact path. The only difference with them is what generation they found the goal. Some found the goal very quickly, like run 3, which found the goal after only 7 generations, with an average fitness of only 14.34. Run 41 wasn't as lucky, and took much longer, with a much

slower rate of increase for its average fitness. I assume that this is because run 41 had some unlucky mutations and crossovers, which slowed its progress.

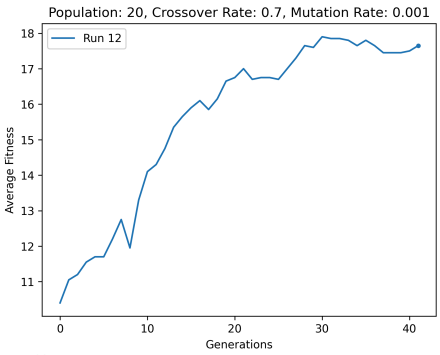
ii.
When dropping the single-point crossover rate to 0, the algorithm almost always fails. I was able to get one run that found the goal, but it took many many tries of running 50 populations through 50 generations.



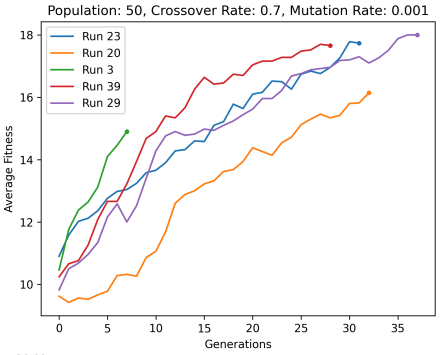
The population rarely reaches the goal because individuals can no longer share useful traits. Instead, the algorithm relies solely on random mutation, which makes exploration of the solution space inefficient, as each member of the population is more isolated. While selection still allows slightly better individuals to survive, progress is extremely slow, since improvements cannot be combined. As a result, finding a solution becomes more of a random process than an evolutionary one, and successful runs are rare compared to when crossover is enabled.

iii.
I systematically varied population size, crossover rate, and mutation rate to examine how each parameter affects the time required for the genetic algorithm to discover the optimal string of all ones. Each experiment consisted of 50 independent runs, and I recorded the average, minimum, and maximum generations to solution, as well as the success rate.

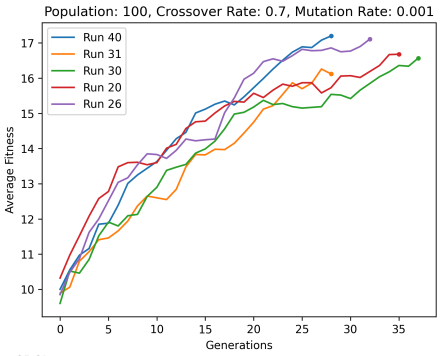
Population size.
Smaller populations (20–50) performed poorly, with success rates below 50%. A population of 20 succeeded only once in 50 runs (2% success). Performance improved dramatically as population size increased: with 100 individuals, success reached 92% at an average of 25.6 generations, while at 200 and 500 individuals, success was 100% and the average discovery time dropped to 16.8 and 14.4 generations respectively. In addition, the shapes of the fitness-over-time curves became more consistent as population size grew. At higher populations, the curves resembled a near-linear improvement (closer to y = x) instead of a square-root-like curve (y = √x). This means larger populations progress more steadily: they start less explosively, but they also avoid the rapid tapering off seen in smaller populations, leading to more reliable convergence.



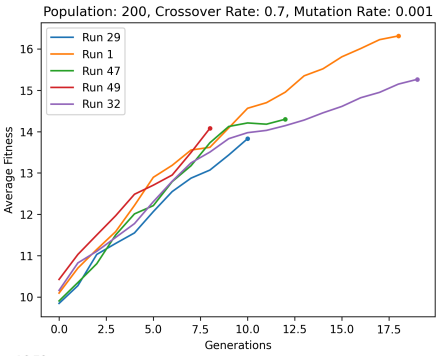
Avg Generation: 41.00
Max Generation: 41
Min Generation: 41
Success Rate: 2.00%



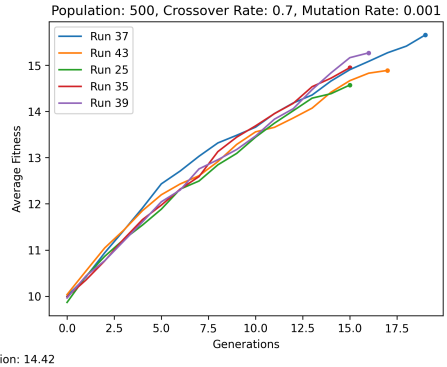
Avg Generation: 30.92
Max Generation: 49
Min Generation: 7
Success Rate: 48.00%



Avg Generation: 25.61
Max Generation: 44
Min Generation: 11
Success Rate: 92.00%

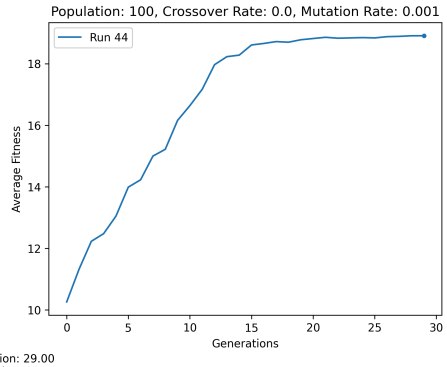


Avg Generation: 16.78
Max Generation: 26
Min Generation: 7
Success Rate: 100.00%

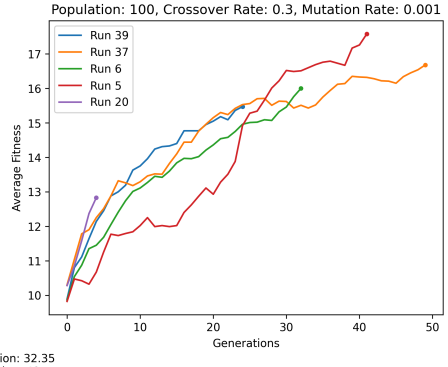


Avg Generation: 14.42
Max Generation: 20
Min Generation: 6
Success Rate: 100.00%

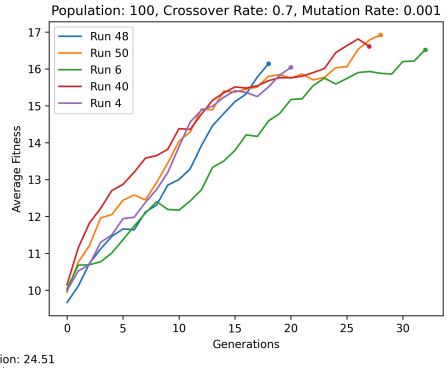
Crossover rate.
With crossover disabled (0.0), the GA essentially failed, with only one successful run out of 50. Low crossover (0.3) achieved a 52% success rate, but at a relatively high average discovery time of 32.4 generations. Increasing crossover to 0.7 and 1.0 significantly improved both reliability and speed: success rates rose to 90–98%, and the average discovery time decreased to 24.5 and 21.2 generations respectively. This confirms the importance of crossover in combining building blocks of good solutions. Runs with higher crossover also showed smoother convergence curves, reflecting more consistent exploitation of good traits across the population.



Avg Generation: 29.00
Max Generation: 29
Min Generation: 29
Success Rate: 2.00%



Avg Generation: 32.35
Max Generation: 49
Min Generation: 4
Success Rate: 52.00%

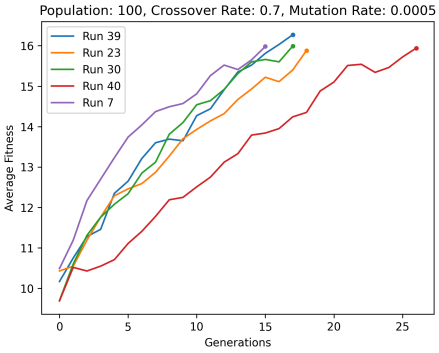


Avg Generation: 24.51
Max Generation: 41
Min Generation: 10
Success Rate: 90.00%

Avg Generation: 21.24
Max Generation: 41
Min Generation: 5
Success Rate: 98.00%

Mutation rate.
Mutation was more robust to variation. Across tested rates from 0.0001 to 0.01, success rates stayed high (90–98%), and average generations to solution ranged only from about 24 to 28. The best performance occurred around 0.001 (the default) with 24.4 generations on average. Extremely low mutation (0.0001) slightly slowed convergence, and very high mutation (0.01) also increased average generations, suggesting that too little or too much mutation hinders efficiency. However, the shape of the fitness curves remained broadly similar across mutation settings, with no dramatic structural changes like those seen in the population-size experiments. At mutation rates of 0.001 and 0.0001, the lines of the plot are closer together, which could signify that values in this area create a more uniform output.

Avg Generation: 26.37
Max Generation: 43
Min Generation: 11
Success Rate: 92.00%



Avg Generation: 25.24
Max Generation: 48
Min Generation: 6
Success Rate: 92.00%

Avg Generation: 24.42
Max Generation: 43
Min Generation: 11
Success Rate: 90.00%

Avg Generation: 26.27
Max Generation: 48
Min Generation: 9
Success Rate: 98.00%

Avg Generation: 28.33
Max Generation: 47
Min Generation: 5
Success Rate: 96.00%

Summary.
Overall, the experiments show that the GA performs best with large population sizes (200–500) and high crossover rates (close to 1.0). Mutation rate matters less within the tested range, though moderate values (around 0.001) are slightly more efficient. The key takeaway is that population size and crossover strongly determine whether the GA reliably finds the optimal solution, while mutation primarily fine-tunes the balance between exploration and stability. I think this makes sense because mutation mostly just shakes things up a little, while population size and crossover really control how much “good DNA” sticks around. Mutation only really matters if the population gets stuck, and since my runs usually had plenty of variation anyway, it didn’t seem to make a big difference. The convergence curves reinforce this conclusion: larger populations yield steadier, more linear progress toward the solution, whereas smaller populations show bursty early improvements followed by stagnation. Overall, Part 1 showed me that the GA works consistently well if the population is big enough and there’s lots of crossover. Mutation doesn’t hurt, but it’s not the main driver here. The algorithm is definitely stochastic, but with enough generations it usually gets the job done.

Part 2 b

I started with these settings:

Population size: 100
Crossover rate: 1.0
Mutation rate: 0.005
200 actions per cleaning session
Number of generations: 300

The best solution I came up with had a fitness of 114.96, which is okay, but I wanted to find a solution with a score comparable to rw.strategyM, about 360. So, I started changing up settings, however, it was a very strenuous process because of how slow each GA takes to run. So, I spent a lot of time making the code more efficient, which sped things up a bit, but it was still painful. The slowness also really shows just how inefficient the GA is. Each fitness test is 25 full simulations of 200 steps each, and that multiplies by the entire population every generation. Even a medium-sized run ends up taking forever, which made it tough to explore parameter sweeps as much as I wanted.

Part 1 taught me that a higher population is very helpful, so I increased it from 100 to 200, which only slowed things down further. In a fit of exhaustion and frustration, I decided to bump up the generation count to 3000, and let it run overnight. I used the settings that got me the best results in part 1:

Population size: 200
Crossover rate: 1.0
Mutation rate: 0.001
200 actions per cleaning session
Number of cleaning sessions to calculate fitness: 25
Number of generations: 3000

When I woke up, I was disappointed to find that after only 150 generations, the GA found its local maxima of about 60, and did not improve for 2850 generations. What this really highlighted for me is how random the GA can be. Some runs just lock onto a mediocre strategy and never escape, while others stumble onto a great combination early and climb much higher. The difference between plateauing at 60 and hitting 400+ shows how dependent the results are on chance.

I knew from part 1 that some GA runs are just unlucky, but I also knew that with a high enough mutation rate, the GA could break past a local maxima. So, I set the whole thing up again, and let it run while I was out doing errands. The settings were:

Population size: 200
Crossover rate: 1.0
Mutation rate: 0.01
200 actions per cleaning session
Number of cleaning sessions to calculate fitness: 25
Number of generations: 3000

I came back before the function finished, but I could already see that I was getting scores right around 400. The results for this file are in 3000_high_mut.txt. What's cool, though, is that my best evolved strategy actually beats the hand-coded strategyM (which has an average fitness around 360). So even though a lot of runs stalled out, the GA is capable of discovering strategies that outperform a carefully designed one, which is kind of the whole point. However, I noticed that while the function ran for about 1600 generations, a score above 400 was reached at generation 640, and again, the GA plateaued. I'd be curious to see if this is just another local maxima, and what it would take to break past it. In the end, the best genome I found had a score of 413, and it occurred at generation 870.

After this, I tried to see how high of a score I could get in just 300 generations. To see which settings were the best, I tried a similar approach as to part 1, where I systematically tested different mutation and crossover rates, as well as different populations.

The base settings were population of 100, crossover of 0.7, and mutation of 0.001.
A crossover of 1 is better than 0.7, resulting in a lower average (14.2 vs 6.44), but higher max score (23.0 vs 37.6)
A mutation rate of 0.01 is the best, with an average of 31.28 and max score of 65.2, compared to the second best rate of 0.005, with an average of 19.63 and a max score of 50.2.

Higher population is usually better, 1000 had my best average of 245.33 and max of 306.0. This isn't always the case, a population of 500 did very poorly, even worse than a population of 100. The full results are in the folder robby_testing.

From this, I found the optimal setting is a crossover rate of 1, mutation rate of 0.01, and a population of 1000. So, I tried a run with all of these settings together, and was able to get a high score of 306.0. The full results are in the file Optimized.txt.

In the end, the settings that worked best for me were larger populations (1000), fairly high mutation rates (0.01), crossovers every time (1.0), and lots of generations. Those gave the best shot at escaping local maxima and finding strong strategies. I think if I kept increasing the population and the number of generations, I'd still see some improvement, but the trade-off is the huge runtime cost. A smarter approach might be to use something like adaptive mutation that kicks in when progress stalls. That could help push past those frustrating plateaus without blowing up the runtime even more.