**20320721 - Karol Karwan**
**https://github.com/karol-karwan/CT421_Assignment1**

Part(a)
**Code structure and description of methods.**

## Fitness function:

The fitness function is quite similar for all of the part 1 (part1.1-1.3) solutions with minor tweaks. We go through each generation and we select the best fitness based on the sum. We then save that result into a variable so that we don't regress the fitness and continue using the highest fitness value found so far. This goes for part 1.2 and part 1.3.

We slightly modify the fitness function in part 1.2 so that instead of getting all 1s, we want to incorporate it for 1s and 0s.

We have the most change in 1.3 where we have to make it harder for the AI to find the optimal solution. We do this by returning variations and different values based on the current string for it to become a deceptive and not an easy solution.

## Tournament selection and crossover:

Similar across 1.1 - 1.3 We select 2 parents for the crossover. We select 3 randoms from the string and the one with the highest value is kept. We then splice and merge the gene into the offspring. The first offspring is spliced from the best parts of parent0 and parent1. This is slightly altered in 1.2 as we want to find which parent has the best fitness since we don't know the target string ahead of time.

## Variables:

Generations = 100. We want to go through a hundred generations of learning if possible.
Solutions = 100. There are 100 different solutions in a generation
String_length = 30. How long the target string is
Mutation_rate = 0.01 Rate it which the AI learns

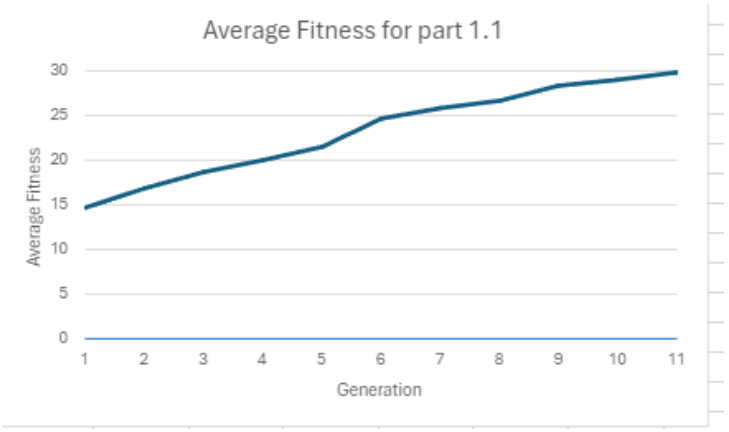
These can all be changed but these are the values I decided to go with.

## Writing to file:

For each of the solutions, I used writer.writerow to write the input at the end of each loop into a csv file which I then opened in excel to create graphs that will be shown later on.

## One-Max Problem Part (A) 1.1

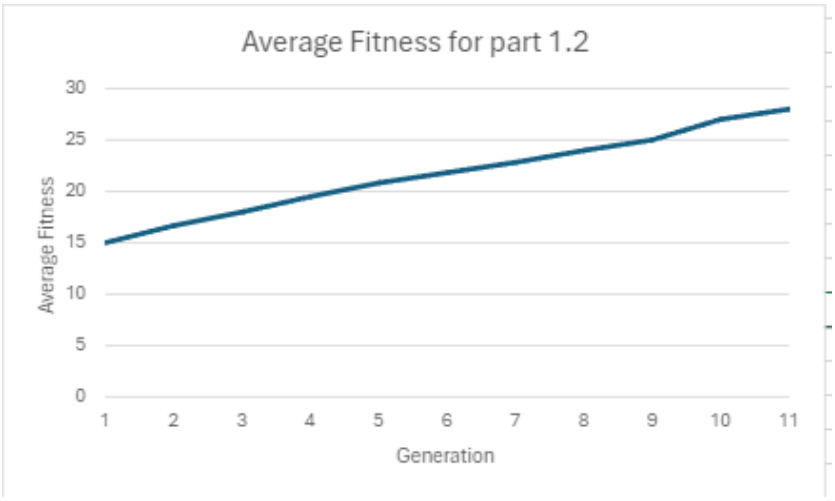
| Generation | Average Fitness |
|---|---|
| 1 | 14.77 |
| 2 | 16.93 |
| 3 | 18.75 |
| 4 | 20 |
| 5 | 21.5 |
| 6 | 24.75 |
| 7 | 25.89 |
| 8 | 26.73 |
| 9 | 28.4 |
| 10 | 29.1 |
| 11 | 29.9 |




**Results**:
At first, we start at an average fitness of 15 which is quite good. We see that each generation the average fitness improves and doesn't go down. This shows the fitness function and tournament selection etc. have been calibrated quite nicely. On average it took 5 generations to go from 15-21 and we hit a nice spike where we got 4 average fitness from generation 5-7. Overall it took the algorithm 11 generations to find the solution which is quite fast.

## Problem Part (A) 1.2 Evolving to a target string

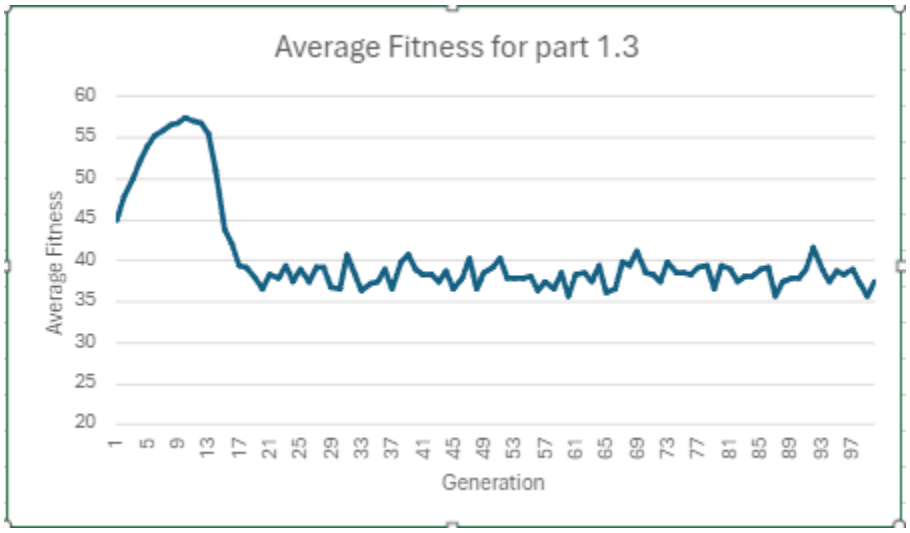
| Generation | Average Fitness |
|---|---|
| 1 | 15.04 |
| 2 | 16.73 |
| 3 | 18.06 |
| 4 | 19.52 |
| 5 | 20.82 |
| 6 | 21.9 |
| 7 | 22.94 |
| 8 | 24 |
| 9 | 25.12 |
| 10 | 27.1 |
| 11 | 28.05 |
| 12 | 29.05 |




**Results:**
We see here from the graph and the table that it took the same amount of generations to get the target string of random 1s and 0s. On average it took 12 -15 generations to find it in my testing so I chose the best result we had. This makes sense as the target string is not all 1s so it would be logical that it would take more tries for the algorithm to find the correct solution. The graph is quite similar to part 1.1 Where we start at 15 and increase mostly linearly until we find the solution.

## Problem Part(A) 1.3 Deceptive landscape

| Generation | Average Fitness |
|---|---|
| 1 | 44.87 |
| 2 | 47.68 |
| 3 | 49.97 |
| 4 | 52.16 |
| 5 | 53.96 |
| 6 | 55.23 |
| 7 | 55.92 |
| 8 | 56.5 |
| 9 | 56.88 |
| 10 | 57.42 |
| 11 | 57 |
| 12 | 56.76 |
| 13 | 55.56 |
| 14 | 51.18 |
| 15 | 43.79 |
| 16 | 42.06 |
| 17 | 39.51 |




**Results:**

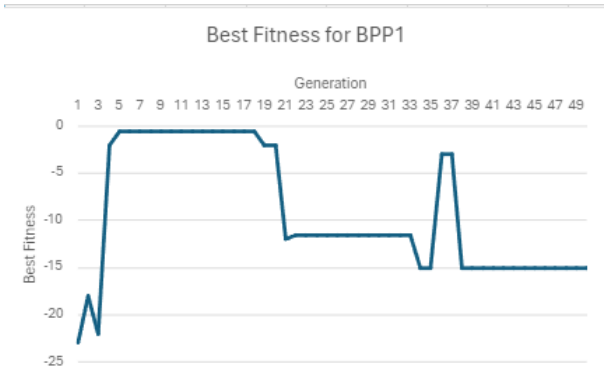
In deceptive landscapes we want to create a solution that is extremely hard for the AI to find. Initially it looks like the AI is close to finding the solution but the average fitness drastically falls off at around generation 10-13. This is due to the fitness function that was used to ‘trick’ the algorithm and to make it tougher for it to find the solution. This is more evident since the algorithm doesn't really recover its average fitness and fluctuates between 43 and 35 for the remainder of the generations. The clear difference between 1.3 and 1.2/1.1 is that obviously there is no solution ever found in the 100 generations. For example it only took 11 and 15 respectability between 1.1 and 1.2 to find the solution. It also isn't linear with quite extreme digs and gains in average fitness.

**Part B**

Code Structure:

Code is quite similar to part As code. Except we have an array called BPP_Instances which holds all the bin packing values in an array. The fitness function is slightly changed to allow for penalties similar to part A 1.3 if the bin is overflowed and by how much space an item is using in the BPP. The rest of the code like tournament selection and crossover etc is practically the same as part A.

NOTE: graph is flipped. For example -25 means that we are using 25 bins. We want to be as close to 0 as possible for an optimal solution.

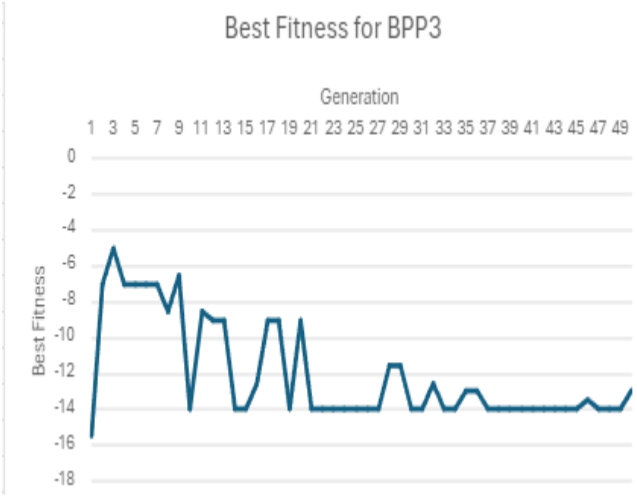



**Results for BPP1:**

As seen from the graph, the results are quite analog. Meaning that the fitness function could be too extreme or that the data is quite unique itself. For the other BPP2-5 the dips and gains aren't as extreme as this one so I presume that it could be the data that was provided that caused the algorithm some trouble. The graph stabilizes from generation 39 onwards and never finds the optimal solution. It didn't matter if the generations were increased, we should have around 15 bins being used. The fitness function could be tweaked more to find better solutions but I decided to run with this one as it worked quite nicely for the other BPPs. We had a min and max of -23 and -1

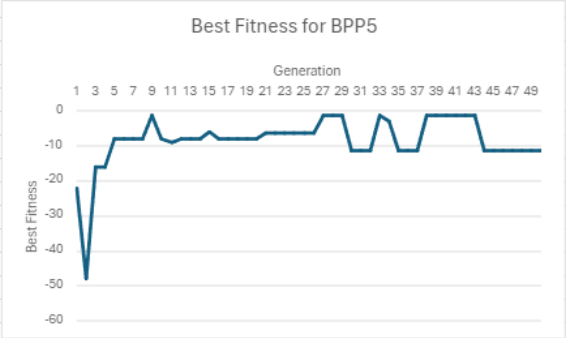

**Results for BPP2:**

Compared to BPP1, we can see a lot better results. Instead of regressing quite drastically, this one slowly increases and decreases fitness as we continue. We start at around -15 dipping to 30 bins used then drastically returning to -3 and having minor fluctuations at the -2 to -10 range. Similarly to BPP1 , no optimal solution is ever found. The fluctuations near the end suggests that the algorithm has reached a point where improvement is hard to find.




**Results for BPP3:**
The increase at the start shows rapid development in the earlier stages of the algorithm. Compared to BPP1 and BPP2 the values here are more in the middle of the road in terms of performance. The data plateau shows at generation 35 onwards that the optimal solution for this dataset could have been found. The dips are quite exponential and quickly rise and fall until about generation 20 where they subside. The maximum and minimum values are -16 and -6. Meaning the number of bins used was in range 16 to 6.


**Results for BPP4:**
In the beginning we see quite high highs and low lows. This could mean that the algorithm is struggling quite hard initially to come up with a good solution. It takes it around 15 generations to plateau and to find a middle ground and work from there. We see a less drastic gain and loss from around gen 25 onwards and its quite evident that we are seeing improvements. Similarly to other BPPs we dont find the optimal solution and stay in the center of the graph for the majority. The minimum and maximum values are -22 and - 2




**Results for BPP5:**
We see nice improvement early on which suggests that the algorithm found a solution that it likes and works from there onwards. The algorithm is quite stable from generation 5 onwards till 50 with only minor fluctuations and mostly we see a plateau in the data. We have a max and min of -50 and -2. We see major improvements here on this graph as we go to -50 instead of the -30 BPP1 - BPP4 have gone to which suggests poor solutions at the beginning of the run. As with the other BPP solutions, no best solution is found.

**Key findings:**

- All of the Instances have rapid improvement near the beginning and fall off or plateau near the end of the generations
- After the initial improvement, all the BPP instances plateau or stabilize quite rapidly also with minor fluctuations in their fitness.
- BPP5,BPP4 and BPP2 have some major Dips in their graphs but recover quite nicely afterwards.
- None of the BPP instances found the perfect solution.
- BPP2 had the best overall performance.

Sources:


- https://www.geeksforgeeks.org/bin-packing-problem-minimize-number-of-used-bins/
- https://www.geeksforgeeks.org/genetic-algorithms/
- https://www.youtube.com/watch?v=uQj5UNhCPuo - Genetic Algorithms Explained By Example
- https://www.youtube.com/watch?v=nhT56blfRpE - Genetic Algorithm from Scratch in Python