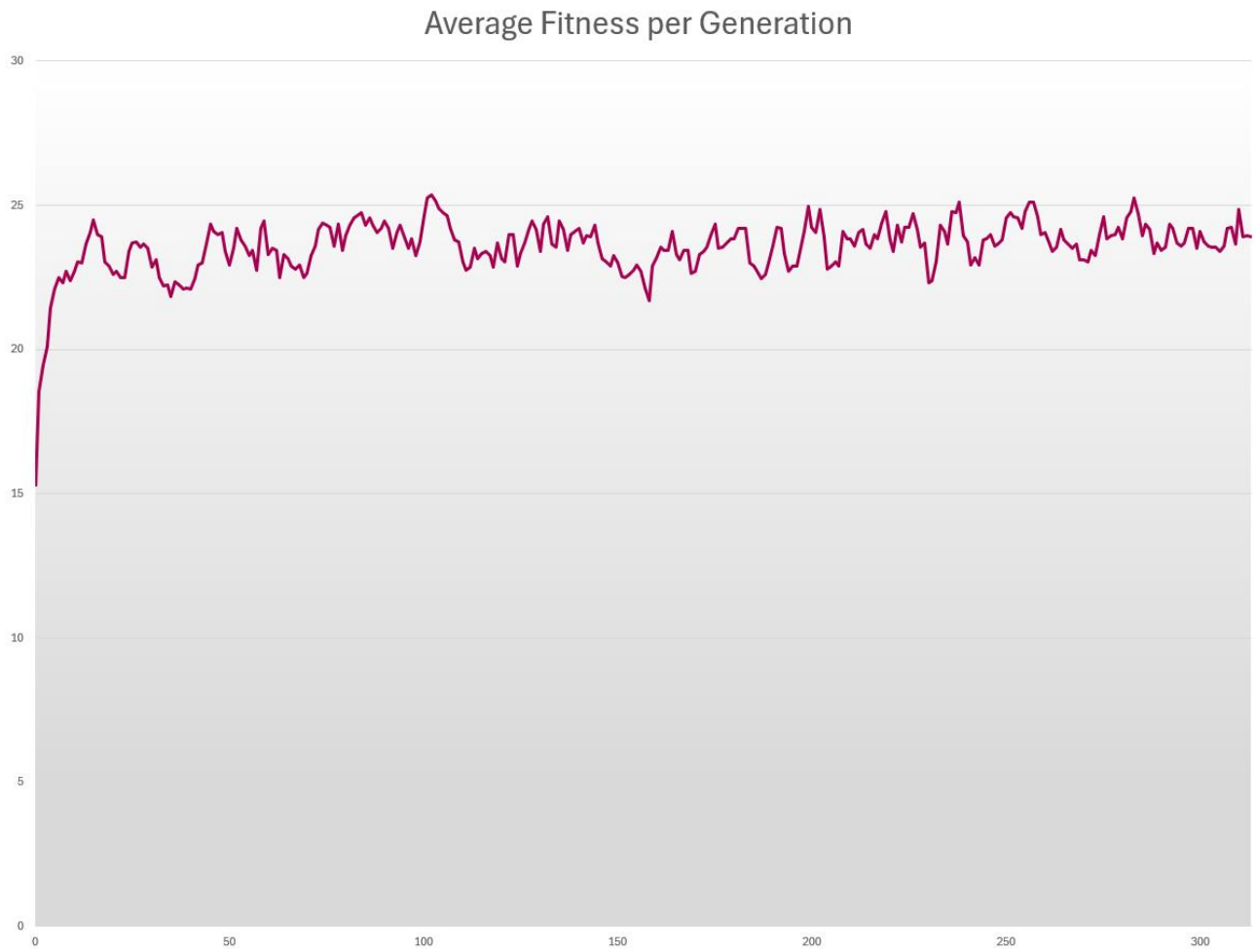
1.1 One-max problem

The goal of the one-max problem is to create a string of a certain length, in this case, 30, in which all characters within the string are equal to 1. We were tasked with generating 20 random strings of 1s and 0s, and modifying these strings, using selection, crossover and mutation techniques.

For our solution, we used all three, using a "*crossover rate*" of 0.6, and a "*mutation rate*" of 0.1 to dictate how often a crossover or mutation was performed on each individual string. We used *truncation* to decide which strings to perform crossover on, meaning crossover was only performed on "*elite*" strings, which were the 5 strings with the highest fitness ranking. Truncation would make it so that the strings with the best fitness would have the most control over the algorithm, which we felt would benefit us, especially with the later tasks in mind. We sorted every string based on fitness, using python's `.sort()` method, and returning the modified population. Crossover was used on these strings as we wanted the strings closest to One-max to have the greatest effect on the evolution of the strings. Fitness was calculated, for this version of the problem, by simply counting the number of 1s in each string, as the more 1s within the string, the closer to the "one-max".

We then used a "*Generate New Population*" method to carry out these modifications to the population, using a loop to work out the five Elite Strings, then performing crossover, at a rate of 0.6 on these strings, before performing mutation on the new population of elite strings, at a rate of 0.1.

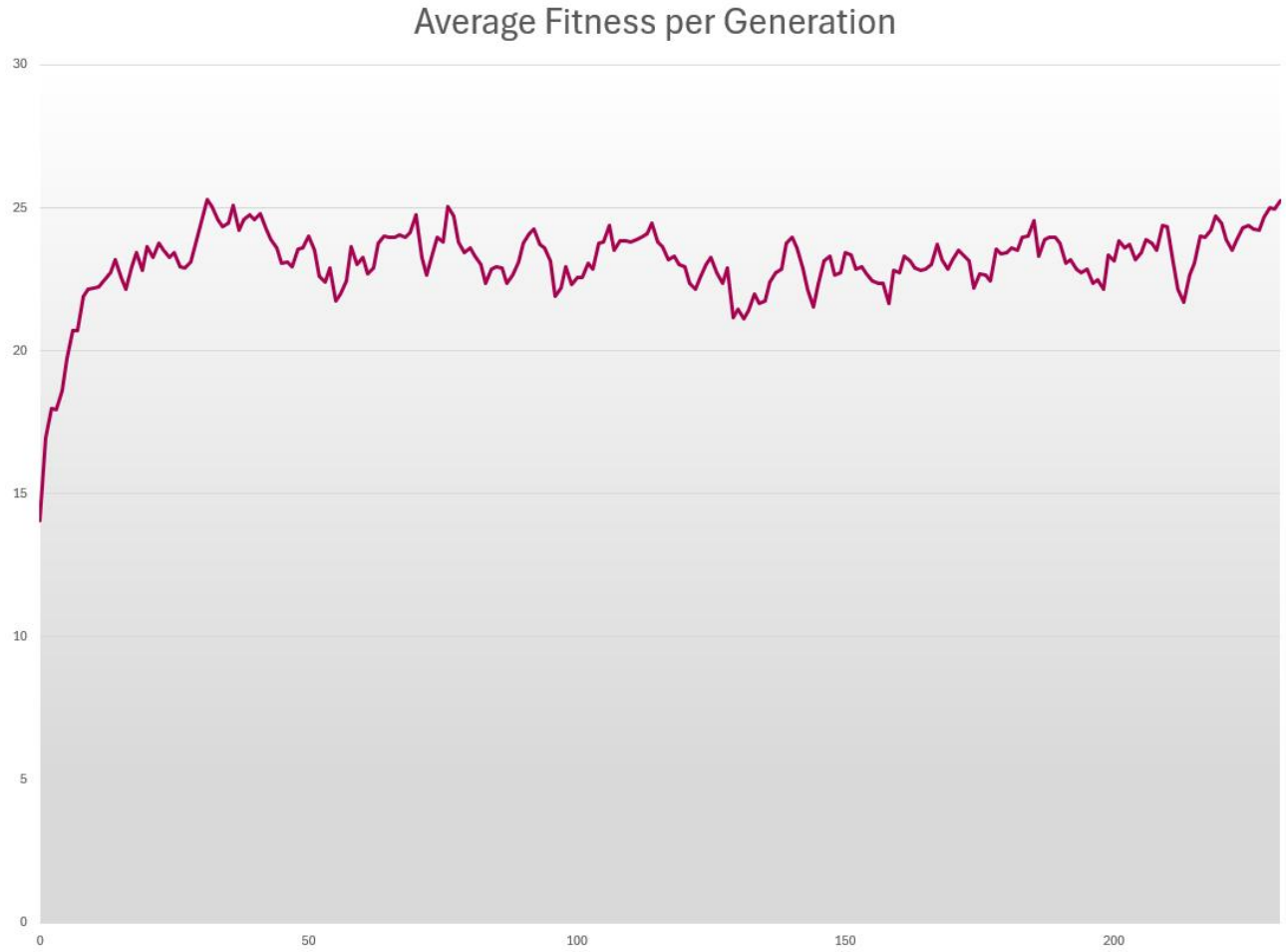
Once a population was generated, we then looped through each evolution of the strings, printing each one, until a solution was found, i.e. One-max was reached.



The algorithm's average fitness started at the lowest it would reach, at ~15, after a few generations this then levelled out at ~between 22 and 25. The average fitness of the strings, in this case because of our use of truncation, would not be expected to fluctuate majorly, as the algorithm will mostly be focusing on the few elite strings, and only occurring 60% of the time for crossover, and 10% of the time for mutation.

1.2 Evolving to a target string

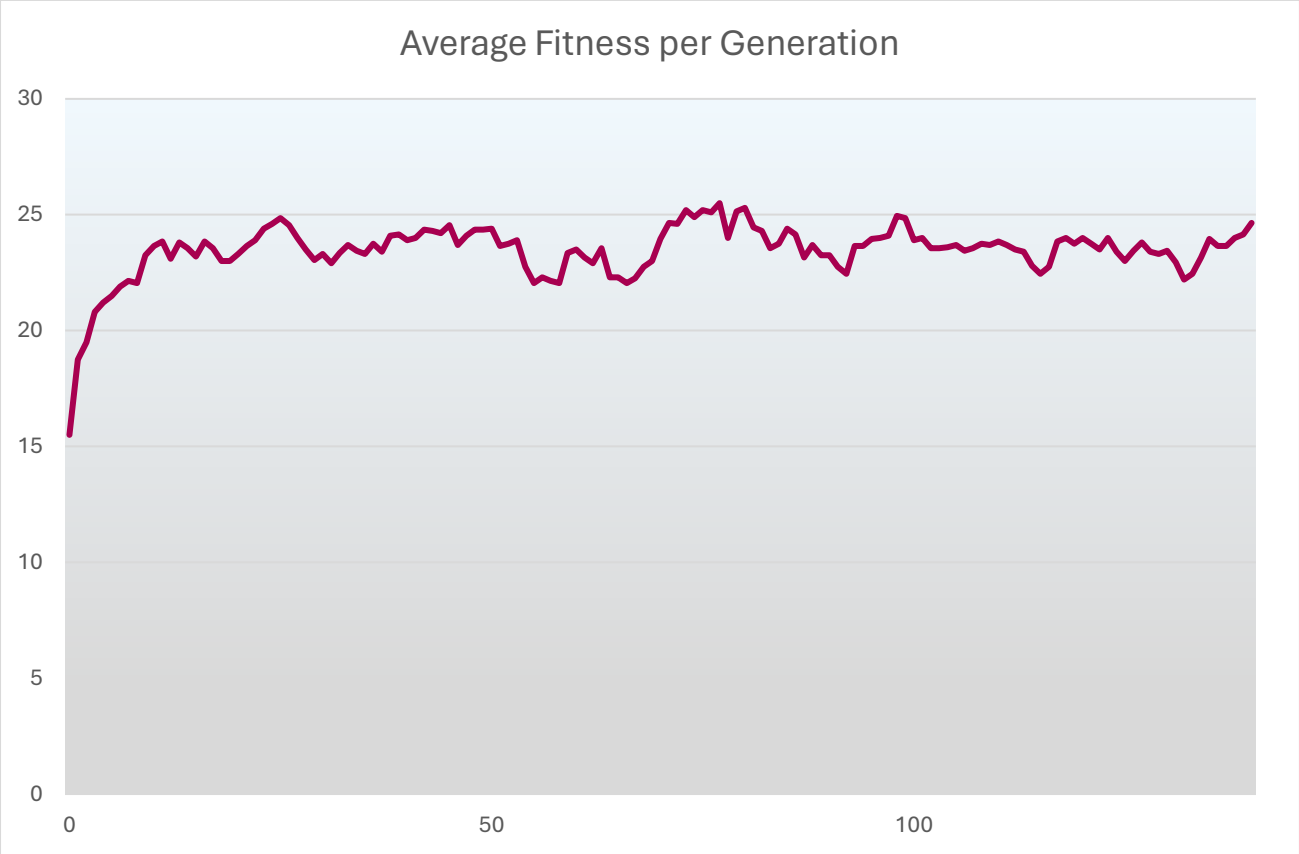
The *evolving to a string* algorithm worked much the same as the One-max algorithm, the main difference being how the fitness was calculated. Fitness was calculated by taking each character of the string, being either 1 or 0. If this character was the same as that of the target string, the fitness would be incremented by 1, hence, more similar strings would have a higher fitness. Everything besides the fitness in the process, worked identical.



The algorithm behaved more or less the same as the One-Max algorithm, as, at heart, the algorithms are more or less identical, just with a different parameter used to calculated fitness. As the strings as random, it will be just as easy for these strings to get the a One-max string, as it will a string of random 1s and 0s.

1.3 Deceptive Landscape

The *deceptive landscape* added a parameter that meant that a string of all 0s would be given a fitness of double the string length, i.e. 60. Besides this the algorithm is the same as the One-max solution. In order to test the effectiveness of the algorithm dealing with the deceptive landscape, a string of all 0s was planted into the population each run.



When running the algorithm, we found that on average, the algorithm with an all-zero string planted seemed to have little effect on the algorithm's efficiency. A reason for this may be that due to our use of truncation. The operations being made on the strings with the highest fitness, which an all zero string would always be included in, the operations would modify the string with either mutation or crossover, which would immediately negate the deceptive string, making the algorithm behave as normal.

1.4 Contribution

For this project, we used the pair programming technique. We found it easier for collaboration and bouncing ideas of each other and for general sanity checks through the coding process. The project was also uploaded to GitHub, to allow us to independently work on the assignment if required.

Dion Collins

I spent the majority of the project as the navigator, whereas Casey acted as an interpreter. Being able to review the project as it was written allowed me to take on a more analytical role, and give pointers when needed to the solution, and bringing up any divergence or compromises that needed to be made to the original plan, or to improve the original plan.

While I contributed less code to the project, I took a much more active role analyzing the problem and compiling our work into the report, confirming why each version of the algorithm behaved the way it did, and how best to visualize the information produced by our experimentation.

Casey Bracken

My primary role in the project was taking our proposed solution structure and finding the best way to represent it in our language of choice, Python. I would discuss with Dion how a specific part of the algorithm would be structured, then begin working on an implementation. If I was confused about any part of the algorithm I would consult Dion for clarification, which would result in a change to our specification or an adjustment to the code to fit our proposed system.

Due to my strong experience with Python, parts of the code initially put aside to implement after the bulk of the project was completed could be implemented cleanly with minor changes to the existing code. An example of this is the code required for writing algorithm data out to a file, which could then be imported into Excel for analysis.

2 Bin-Packing

For the bin-packing problem, we stored solutions as a list of all the items of varying weights provided by the task. The starting population is a randomization of these items, generated from the frequencies provided by the file. Each solution is a different ordering of these items. Bins do not exist as a structure but are filled sequentially with items in the solution that don't exceed bin capacity. For example, a bin might fit the first 5 elements of a solution, another bin fits the next 6, and so on.

Our crossover function is an implementation of one-point crossover, using 5 elite items to perform crossover on, to maximize packing, however multiple functions for checking solution correctness and correcting incorrect solutions had to be implemented. This was necessary as when performing one-point crossover, weights in a solution may be overwritten by other weights, producing an excess of some weights and a lack of the weights overwritten. Therefore, it is necessary to correct each new solution before including it in the new population.

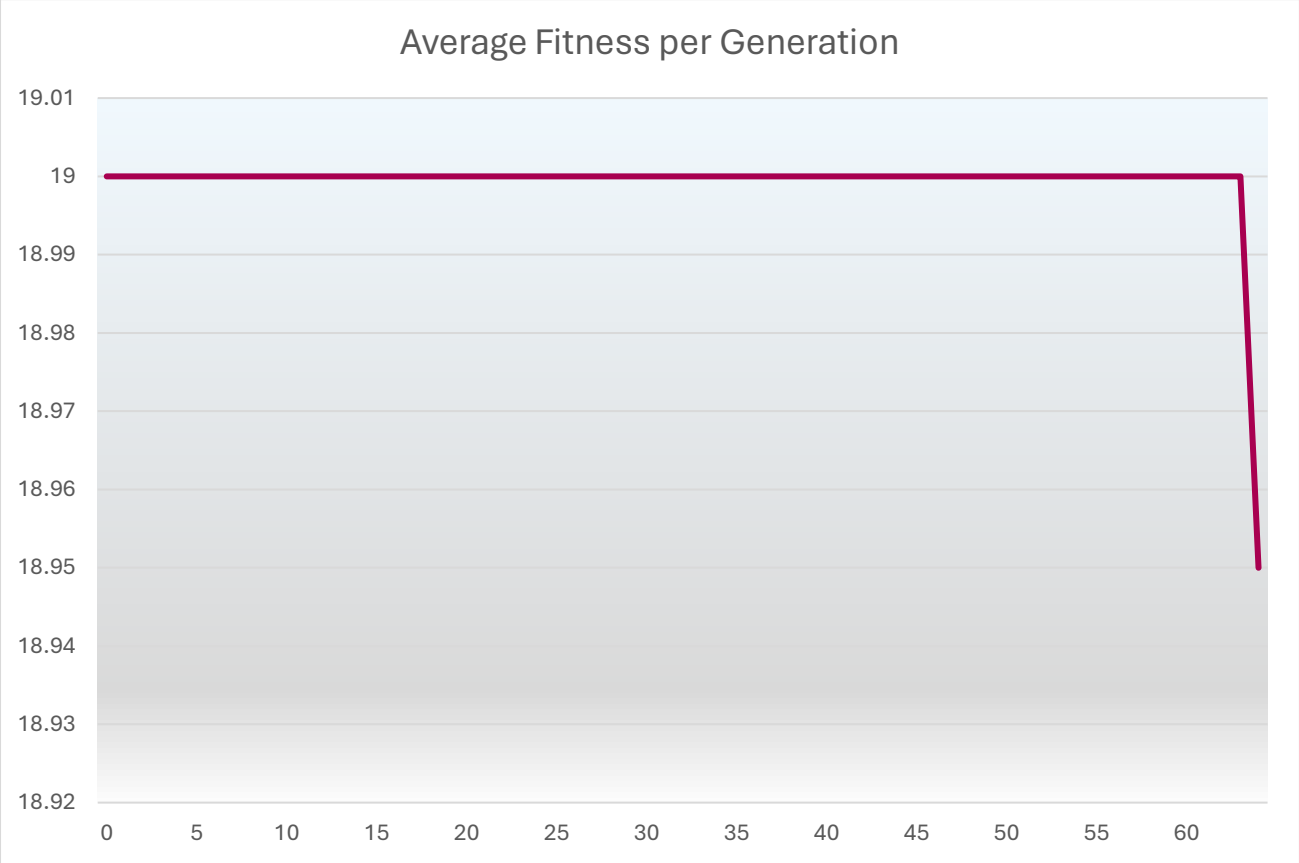
Thankfully, correcting weights is trivial. After performing crossover, if a weight is overwritten by another weight, it must exist in the other solution. Due to this, any weight missing from the first child will be excess in the second child, and any weight missing from the second child will be excess in the first child. By calculating the difference in weight frequencies between one child and the task's given weights, the excess and lacking weights of both children is known. Excess weights in each solution may be iterated through and replaced with weights missing from that solution. There is always the same amount of excess weights and missing weights in a solution

After performing crossover on the best solutions, we would mutate each child produced. This was performed by taking two random weights in a solution at a given rate, taking the first and swapping it with the second. It is possible, but exceedingly rare, that an item would be swapped with itself or swapped with an item, only for that item to swap back with it. This action is unlikely to occur often, however if it does occur the explored space of the algorithm is lower than if a proper mutation occurred.

Fitness was calculated as the number of bins required by the solution. For this problem, a lower fitness score indicates a more desirable solution. Since a fraction of a bin cannot be used, the fitness is rounded up to the nearest whole number. Truncation selection was utilized to select elite members of the population, the top 5 solutions with the lowest fitness, which would be crossed-over and mutated to generate the next population.

Due to the structure of our algorithm and low variance in the weights supplied for each problem, our algorithm could quickly find an optimal solution for each problem. For each task a target number of bins was calculated, being the total weight of all the items from the task divided by the bin capacity. Once a solution was found with this target number of bins, the algorithm terminates and prints this solution to the console. While there may be a way to more efficiently pack different weights into a minimum number of bins, there is no possible solution where the algorithm would be able to pack items into a number of bins smaller than the target.

Many solutions exist for the problems provided. For each item, other items with the same or similar weights could easily be substituted into other bins while maintaining an optimal solution. Still, swapping 2 weights could increase the number of bins required for a solution that push it out of the set of optimal solutions. For this reason, we believe the problem landscape is composed of many small peaks, or rather valleys when using our fitness calculations. Suboptimal solutions can easily become optimal, and optimal solutions can quickly be pushed into suboptimal territory.



Average fitness per generation for Task 4