# CT421 – Assignment 1

Cian Mc Sweeney – 17391321

## Github Repo: https://github.com/ciano1000/CollegeArtificialIntelligenceGeneticAlgorithms-

## Part A

### Description of Algorithm:

Fairly standard GA implementation, we have a population of size N, each member of the population is simply a string with 20 characters (all 1’s & 0’s). We start by randomly populating this with its initial values and computing the fitness of each candidate, we then we begin the algorithm. Currently we run for a user defined number of iterations, this could be changed to stop when the algorithm converges on a reasonable answer. The algorithm works as follows. Firstly we perform Tournament Selection, we have a list of size N called our “mating\_pool” and to select candidate solutions to be part of this we randomly select K from our population and pick the solution with the highest fitness, this process is repeated till our mating pool is full. Next, we perform crossover and mutation by randomly selecting 2 parents from the mating pool, randomly selecting a midpoint, and performing crossover based on that, e.g. midpoint is 4 means we take the first 4 from the first parent and then the last 16 from the second parent. We also have a mutation rate of 1%, which is applied to every character in the solution string which will randomly cause a character’s value to flip from 1 to 0 and vice versa to add variety to the algorithm. Finally, we compute the new fitness of the new solution and add it back to the population, this process is repeated until the population is replaced, now we begin another iteration/generation.

The “Best Fitness” converges almost instantly to the optimal value while the average fitness rises exponentially until it plateaus slightly below the maximum fitness value of 20, this is likely due to our small amount of mutation causing edge cases where solutions aren’t entirely comprising of 1’s.

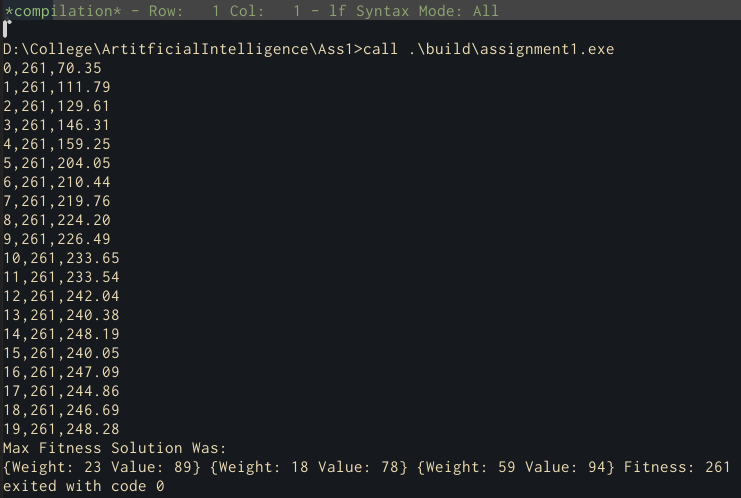
Follows an almost identical pattern to above which is a good sign as it shows that our algorithm can be easily modified to different problems simply by altering the fitness function.

1. In the above test, 50 all 0 solutions were added to the population of 1000. Even with a relatively small proportion of these “deceptive” solutions, the algorithm is quickly overwhelmed with these false positives, indicated by the average fitness quickly rising over, and holding, above the theoretical max fitness of 20.

## Part B – Knapsack problem

1. Representation: Candidate solutions are represented by an array of indices into the weight & value vectors, and a count that keeps track of how many items the candidate solution utilises. E.g. 
2. The fitness function is relatively simple, by default it is simply the sum of the values of items the candidate solution holds. However, if the sum of the item weights exceeds the knapsack capacity, 0 is returned as the fitness.
3. Mutation is slightly different compared to the previous solution since we aren’t operating with binary values, now if a mutation is triggered, we simply pick a random item from the list and add it to the solution. Crossover also works slightly differently since solutions can have different item counts; we need to do some work to prevent the midpoint from going out of bounds. Finally, we also need to ensure that no two items in a solution are the same as each item in the knapsack should be unique.

## Problem 1: Knapsack Size of 103



Linear rise in average fitness until it plateaus directly beneath the best fitness value, from the “seesaw” pattern observed it is unlikely that the algorithm would converge to anything more accurate than this.

## Problem 1: Knapsack Size of 156



Compare to above the algorithm takes far longer to converge, and when it does it is far less stable, this can be presumed to be due to the higher max weight allowing a much larger range of possible solutions leading to the algorithm taking longer to converge.