# CT421- Artificial Intelligence

## Assignment-1

-Srinivas Ilancheran, 19280039  
Blake Preston, 19280019

**Introduction:**

This report aims to provide an overview of the design decisions, implementation of described functionalities, and objectives achieved after completion of Assignment-1, which focused on using Genetic Algorithms to represent and solve some posed problems.

**Assignment 1- Part 2:**

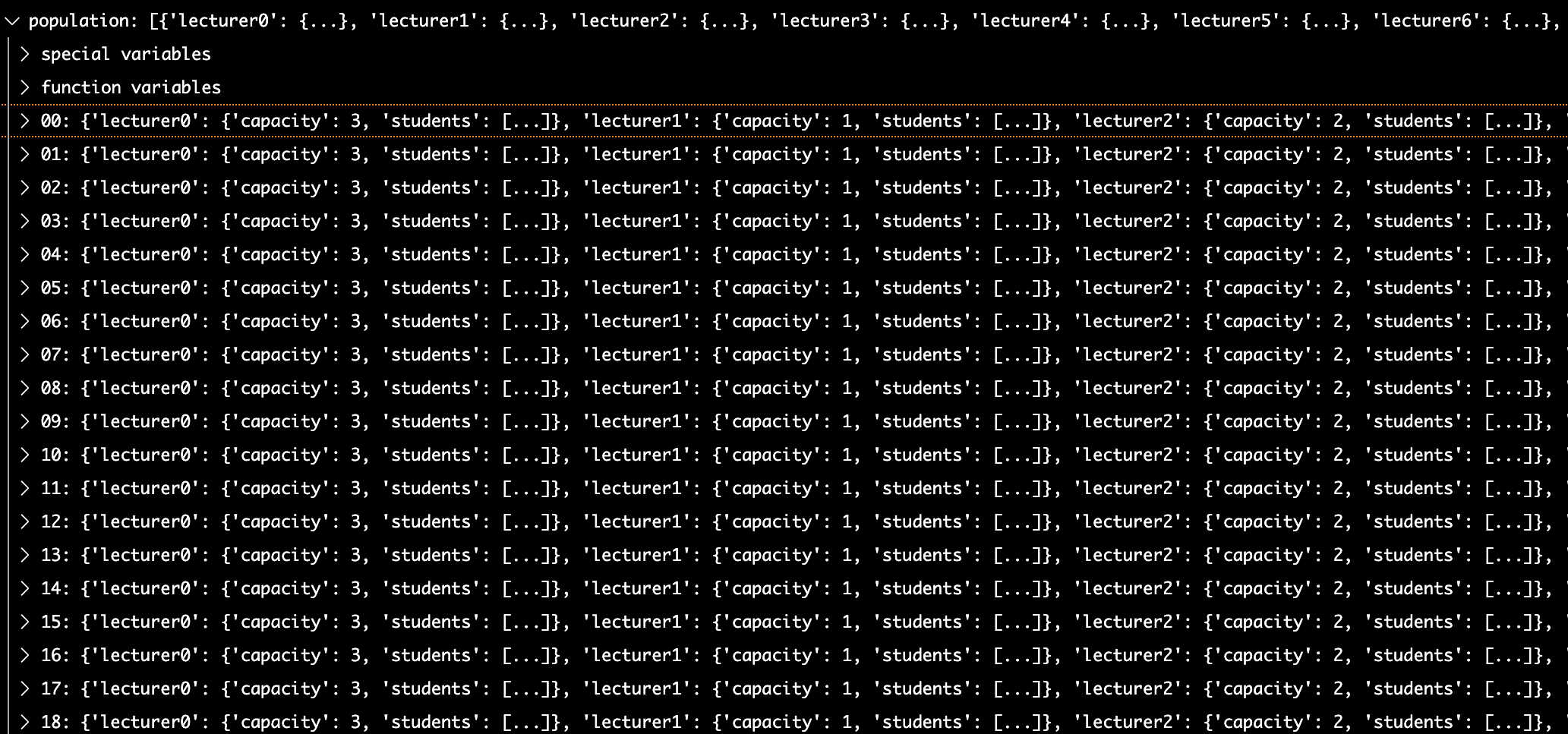
Part 2 of the assignment involved using Genetic Algorithms in order to identify the best possible mapping between students and lecturers for project matching. The provided inputs were a list of lecturers, their capacity to take students, a list of students and a list of the preferences of each student, when it comes to lecturers. The relevant input was provided via multiple csv files, which were read in to the program using appropriate libraries.

**Description of Representation:**

We decided to represent the population as a list of “allocations”, with a population size of 100 leading to 100 different pseudorandom “allocations” or mappings between lecturers and students being made randomly. Each index in the population list contains a dictionary that represents a mapping between the 46 students and the 22 lecturers. The aforementioned dictionary contains key-value pairs, with the keys being lecturers and values being dictionaries that contain the lecturer’s capacity and a list of students that have been allocated to them.

The student preferences are represented using a list of lists, with each index containing a list that represents a student’s preferences (index 5 of the student preferences list contains a list that represents student 5’s lecturer preferences) .

The representation of the population can be better understood using the screenshots provided below.

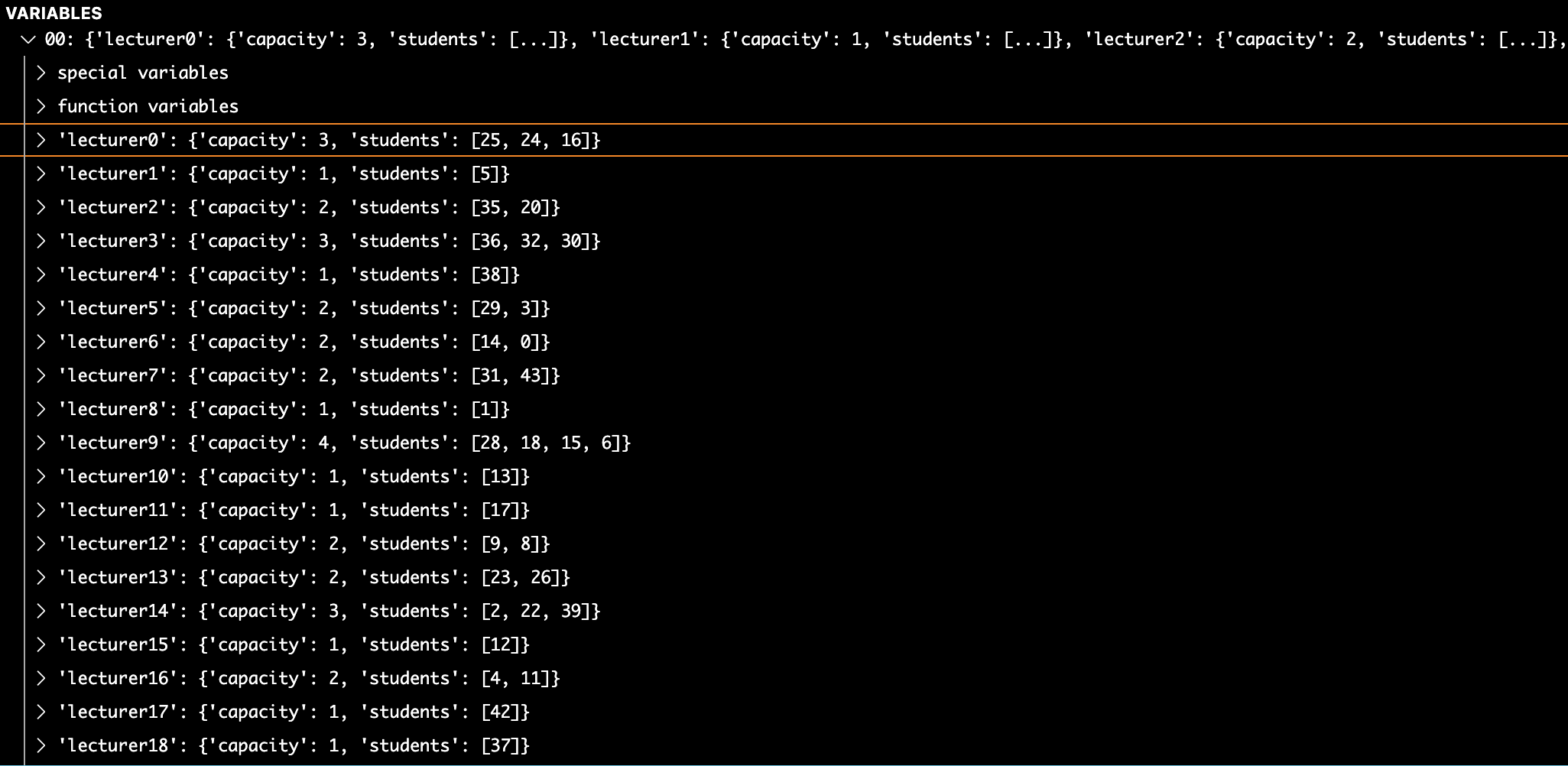


A sample population-> each index in the population list represents a mapping between lecturers and students. (Indexes 0 to 18 are pictured here, with each index containing a dictionary that represents a mapping).

Each of the mappings also contain a fitness “key” that stores the fitness of the mapping. This fitness is calculated using a defined fitness function, which would be described in a further section. The fitness value of the mappings is used in tournament selection to decide the more suitable/fit mapping.

**Description of fitness function:**

We decided to use an inverted approach to defining the fitness function, ie the mapping with lower fitness would be considered more suitable. The fitness function itself accepts a mapping and a list of student preferences (for lecturers). It then iterates through every lecturer in the mapping, gets the list of students assigned for each lecturer, and sums the index of the lecturer in each of the assigned student’s list of lecturer preferences. With this fitness function, the ideal mapping would have a fitness of 0 (every student gets their first preference (stored in the 0th index of the student’s preferences list). .



An index in the population list expanded:-> A dictionary of dictionaries that represent a mapping between lecturers and students (not all 46 lecturers in the mapping are pictured).

**Description of operators:**

The functions that are primarily of interest are the functions used to create the initial population, to organise tournament\_selection, to induce mutation within mappings, to facilitate crossover between mappings and to deal with potential data integrity conflicts caused as a result of crossover between mappings . These functions will be explained in detail in subsequent paragraphs.

**The initialise population function:**

The initial population is randomly created by invoking the initialise\_population() function which takes an integer that represents the expected population size, and the list of student preferences and lecturer capacities data from the csv files. This function creates multiple variations of mappings and returns the generated population as a list.

**The tournament\_selection function:**

The tournament\_selection function takes in an integer that represents the tournament\_size(whether the fights have 2 participants, or 3 participants or n participants) and the population of mappings. It then ensures that all mappings in the population end up in a fight, by randomly sampling mappings from the population, getting them to fight using the ***fight()*** function(which returns the most suitable mapping, in terms of fitness), storing the winner of the fights in a list and then removing the fighters from the population (the population is then set to be the list of winning mappings).

**The Mutation function :**

We decided to implement a directed mutation, where the mutation would have a decent chance at making the fitness of a mapping better(lower). The mutation function takes in the entire population, a variable that represents the chance for mutation to happen and the list of student preferences as parameters. The following process is then applied:

1. For every mapping in the population:

1.1.For every lecturer in a mapping:

If mutation chance is successful:

1. Obtain the list of students assigned to the lecturer

2. Identify the most unhappy student (lecturer is farthest down their preferences list, when compared to the other assigned students)

3. Randomly choose another lecturer in the mapping.

4. Randomly swap a student assigned to the chosen lecturer with the unhappy student of the current lecturer being mutated.

**The Cross-over function:**

Cross-over proved to be one of the more challenging aspects of the assignment to implement. In a broad sense, this would involve lecturers from a section of a mapping being concatenated with lecturers from another section of another mapping (lecturers 1-10 from mapping A would be concatenated with lecturers 11-22 of mapping B, and lecturers 11-22 of mapping A would be concatenated with lecturers 1-10 of mapping B for example).

Our cross-over function involved the use of a wrapper function- cross\_over() which takes in the population, population size, number of lecturers, number of students and the list of student preferences as parameters. In order to keep the population stable (population size is maintained at original size, even after pruning losers of tournament selection) cross-over is performed until the number of new mappings generated is enough to fill the void left by eliminated allocations. Pairs of mappings in the population are taken and are passed to a function- cross\_over\_two\_parents() which actually performs the cross-over and returns new mappings. These new mappings are added to a list and the entire list is returned to be added to the population once the satisfactory number of new mappings have been generated via cross\_over.

The cross\_over\_two\_parents function takes in the 2 mappings that represent parents, the list of student preferences and the number of students and lecturers. It then selects a random number, bounded by the number of lecturers, which is then used as a “pivot point” to divide the lecturers. Cross-over between the 2 mappings is then performed by concatenating alternating sections of the mappings (lectures are crossed over between mappings).

As our primary keys/reference points in the mappings were lecturers, we faced the possibility of students being duplicated or removed when lecturers belonging to different mappings were crossed over with each other. This possibility was almost guaranteed to occur due to the randomness involved in the crossover, and as a result we had to develop helper functions- conflict\_resolution() and quality check() to ensure that the mappings in the population were valid (all students were assigned to a lecturer and no student was assigned to more than one lecturer).

***The helper functions to ensure valid cross\_over:***

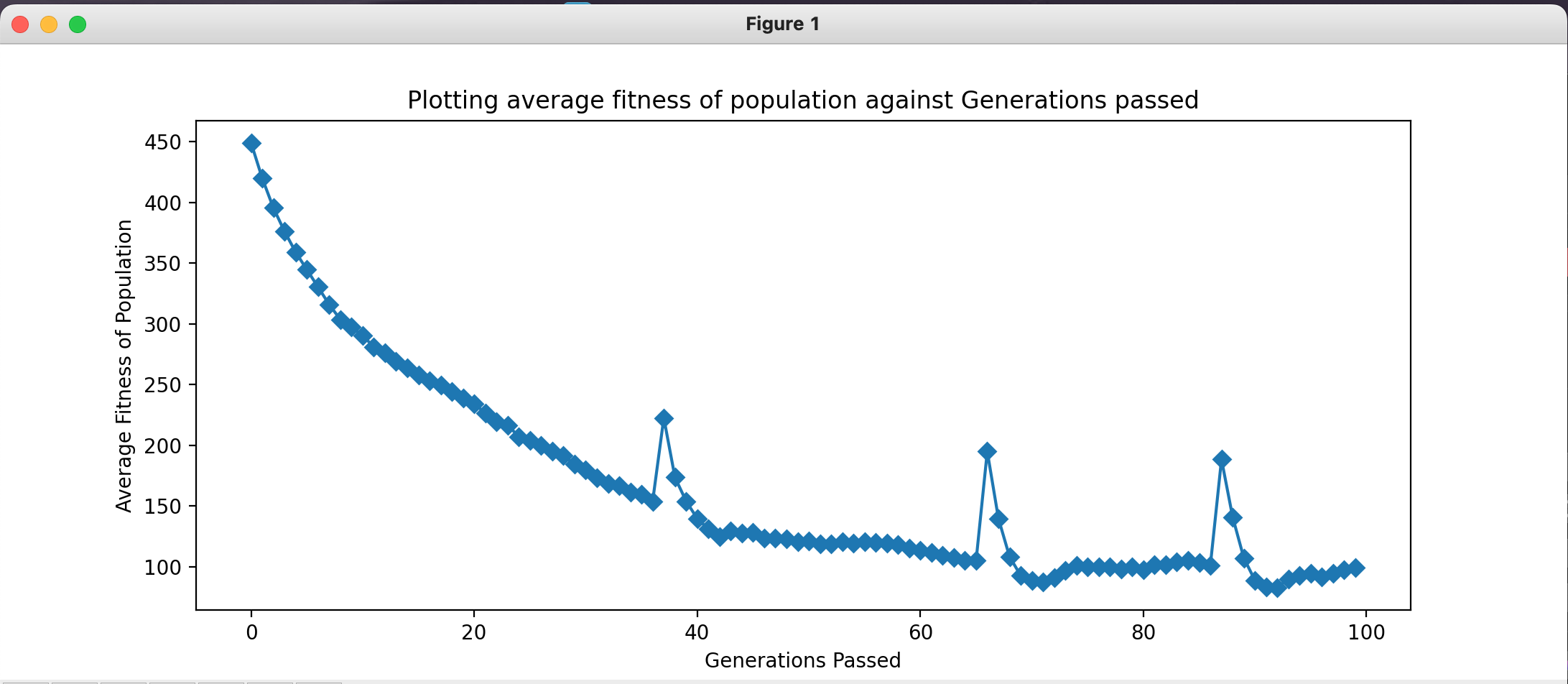
***conflict\_resolution()*** is a function that takes in a newly generated mapping (as a result of cross-over), the list of lecturers and student preferences and the number of students as input parameters. It then makes a list of all the students that are found in the provided mapping, and then identifies the students that are missing from this list (this is done by subtracting a set of the students in the mapping from a set that has numbers in the range 0- num\_students ). Duplicate students and the lecturers that have them are then identified and a conflict dictionary is made with the student being the key and the value being a list of lecturers that have them.

The duplicate students are then iterated through and the lecturer that they prefer less is assigned a missing student in place of the duplicate student they have been assigned (if student 4 is assigned to lecturers 7 and 10, and student 4 prefers lecturer 7 more, then lecturer 10 is assigned one of the missing students).

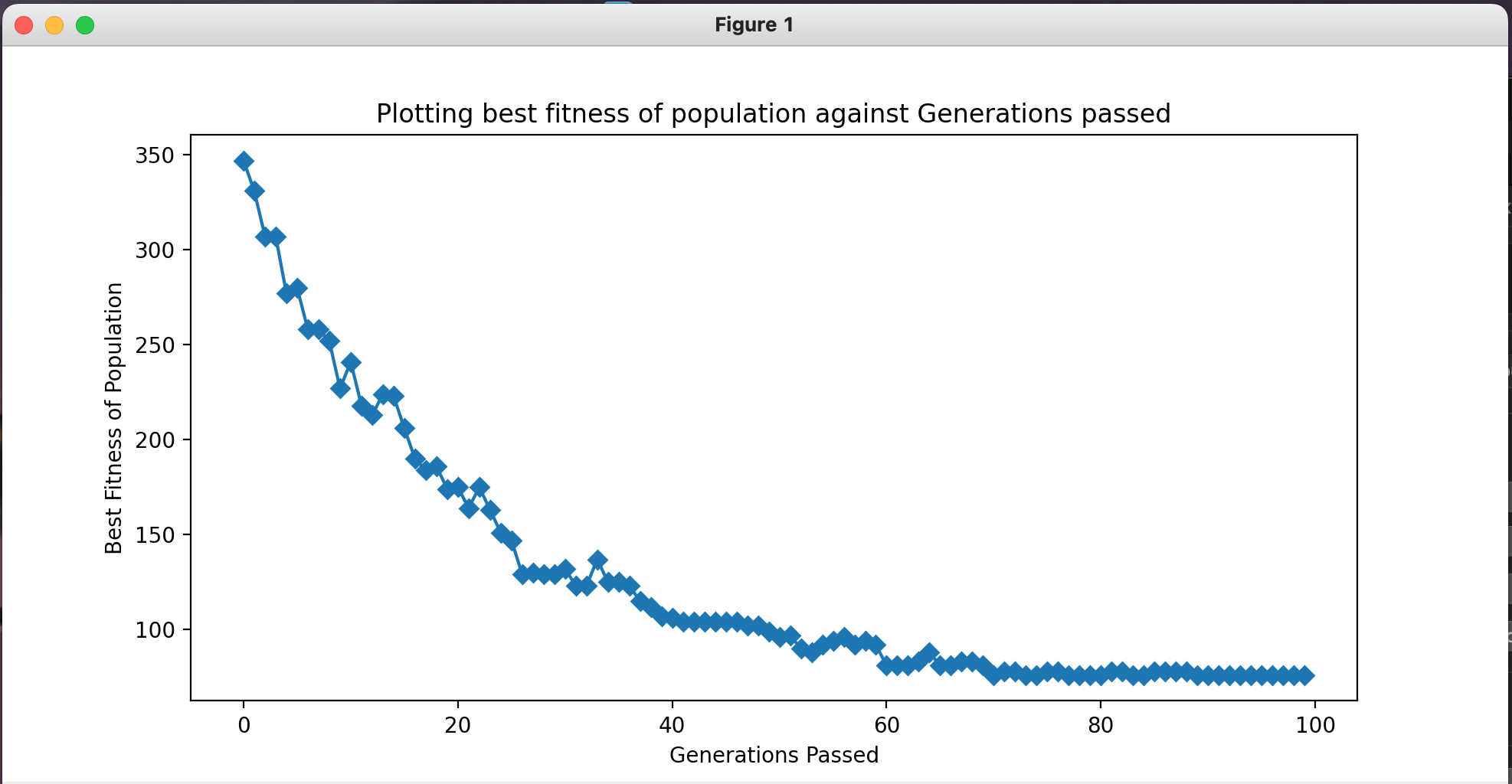
***quality\_check()*** is a function that check if a population is valid by iterating through every mapping and checking if a set of allocated students has a length equal to number of students.

**NOTE: When reading in the data from the spreadsheet that represents student preferences, we decided to represent the data as a list of preferences for each student, ordered from most preferred to least preferred. This was done by mapping the lecturer to the [score-1] index of that student’s preferences (where score is the score given to a lecturer by a student to a lecturer). For example, this would mean that the lecturer to whom the student gave a score of 1 would be stored in the 0th index, and the lecturer to whom the student gave a score of 17 would be stored in the 16th index.**

**Graphs:**



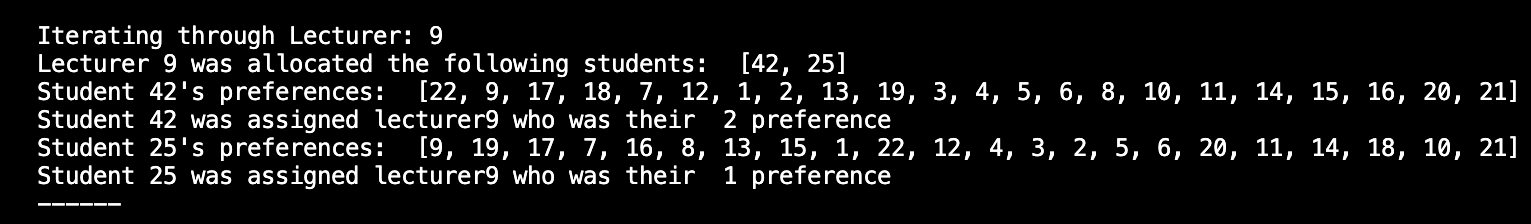
Plotting the average fitness of the population against the number of generations that the GA ran for.



Plotting the best fitness identified in each generation, against the generations that the GA ran for.

**Discussion of results of part b:**

We decided to run the GA on a population size of 100, over 100 generations with a tournament size of 2 (1 vs 1 fights) and a mutate chance of 2% per lecturer. Over multiple runs it was noted that the best mapping obtained after the GA was run was the students getting between their 2.5th to 2.9th choice lecturer, on average. It was also noted that on average, over multiple runs of the GA, that around 40% of the students got their first choice lecturer.



Screenshot of a sample mapping for a lecturer. Note our re-arranging of the student-preferences data from the excel sheet into a list of preferences, ordered from most-preferred lecturer to least preferred lecturer.



Screenshot of the best mapping identified and the number of students that got their first choice.

—————————————

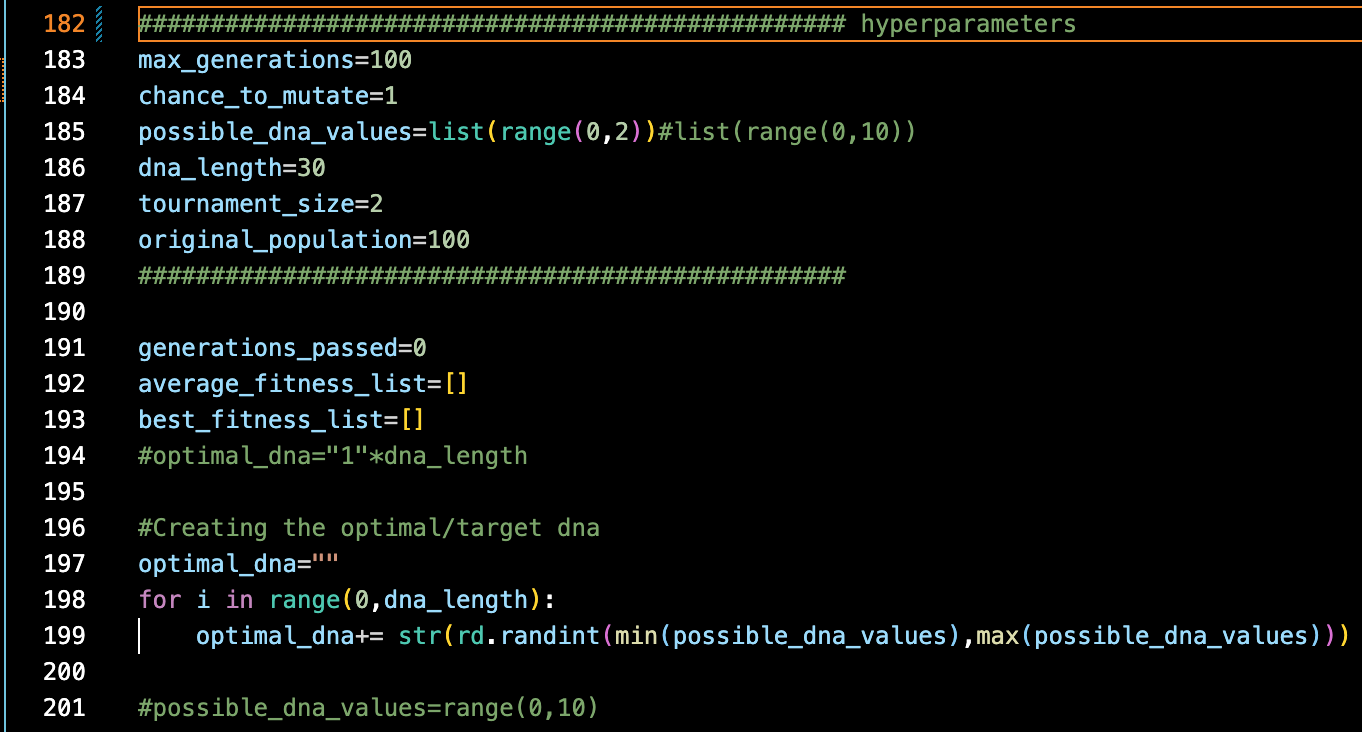
**Explanation of part a:**

We decided to do all 4 sections of part a within the one codebase, and as a result made our codebase as modular as possible. The different sections of part a (one-max, evolving to a target, deceptive landscape and evolving to a target string with a larger alphabet ) can all be solved using the same codebase. These hyper-parameters are displayed in the screenshot below.

The possible\_dna\_values variable is a list of possible dna values in the representation. So for the one-max problem, it can be set to a range of (0,2) which would produce values of 0 and 1.

For the evolving to a target string problem, the optimal\_dna variable can be set to a specified range of values of a specified length.

For the evolving to a specified target string with a larger alphabet problem, the possible\_dna\_values variable can be set to a wider range that covers the alphanumerical values that are specified.



Screenshot of hyper-parameters for part a of the assignment.

**The code made for part a of the assignment can be found on the GitHub repo in the code file named “Submission\_partA.py”**

**The code made for part b of the assignment can be found on the GitHub repo in the code file named “Submission\_partB.ipynb”**