Big Ideas in Computing

Sergei Agaronian

Taina Ahola

**Taming Complexity: EA for the knight problem Part 1**

A short description of stepwise refinement used to arrive to our code:

We started making our code by writing down all the functions/methods that were required in the assignment description. This gave us the following methods; main loop, parent selection, recombination, mutation and accepting. Then, we started by extending the main loop with the description given in the website about genetic algorithms. The algorithm starts with initialising a new population, so we made a new function to do that. Then, the initial population needs to be evaluated, so we needed to make a function for that too. Then, the evolutionary operators are called, so we made a function in the module 2 (EA class) to handle one step of all the operators being applied. This function is called step(). The step function just implements the inner part of the main loop, whose description was quite detailed and we had already made the functions for the evolutionary operators (parent selection, recombination, mutation, survivor selection), so we continued on to write what happens inside those evolutionary operators.

With all the evolutionary operators, we started by looking at the specification given in the assignment description. For example, it was given that ‘mutation takes one individual plus the mutation rate and returns one (possibly mutated) individual’, so we started by giving the function two inputs, namely individual and mutation rate, and writing that it returns the individual. Then, we chose which kind of mutation we use (flipping random bits) and wrote a description of that. Then, we translated that description to Python code. The procedure was used with the other evolutionary operators.

Next, we created the function for initialising a population. We started by thinking about what we need the method to output (a new population) and what should be the input (population size) and added those. Then, we looked at the required representation and wrote what we need to do to randomly generate 195 bits, so we just decided to choose every bit randomly.

Next, we looked at the evaluation function. It was more complicated because we were given little information, but that the fitness value is the number of legal moves. So, we need to check which of the 63 moves are legal. We made a new class for the representation of the chess board, so we could easily check whether a move is legal or not. Then, we thought about all the information we needed about the board state, and came up with the possible moves of the knight and updating the position of the knight. To achieve those, we realised we need to translate the binary string representations to a square on the board and split the binary strings to the different transitions. We made methods for all of those procedures. Then, we just needed a function for calling all those board functions. That became the EvaluateGene-function, which adds 1 to the fitness value every time we have a legal move, and outputs the fitness. Next, we added descriptions of what should happen inside all of these functions and then refined that to python syntax.

Then, we just translated/checked that the detailed descriptions of all the methods in both classes are in correct Python syntax.

A short discussion on cohesion and coupling of the two modules in our code:

We have very low coupling because module 1 calls module 2 exactly two times. The first time is to create an EA object, and the second time is to call the step function. The second module never calls the first module, so those two calls are the only connections between the modules.

However, the cohesion might not be as good as it could be, since the class board is in the same module as the main loop, but it is only called twice inside the evaluation function. However, the evaluation-, generate initial population-, and main-functions are very connected together. Also, the functions inside the Board-class are quite interconnected. However, the functions in the EA-class are only connected in the step-function, but they are semantically very connected. All in all, the cohesion could probably be better, but it is quite good already. Something that might help, could be making the board class its own module (as it kind of is already).