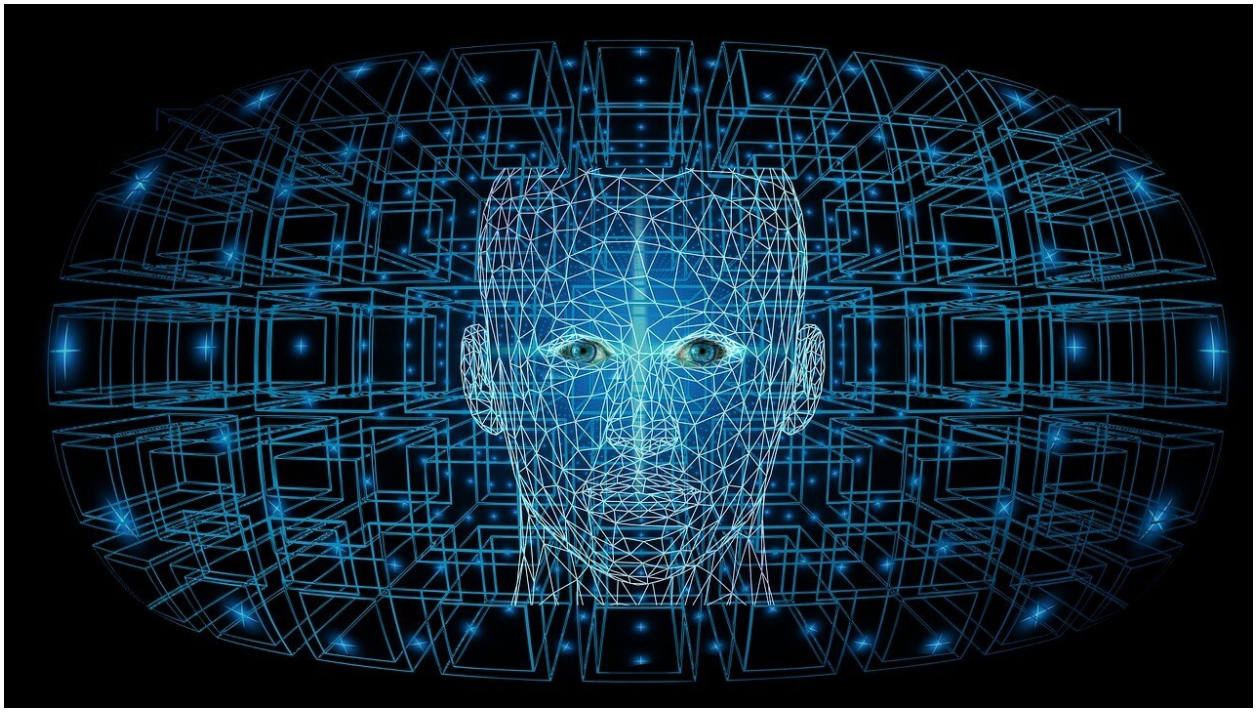


CI HW4

Sohrab Namazinia

97522085

A1)

I coded for this question very cleanly. Every variable naming is like a comment (so clean). But I do explain it here too section by section as always:

Section #1:

It is a simple graph structure that takes the input 1-based, converts it to 0-based for further use, but the final result is again printed 1 based.

Section #2:

Firstly there is an individual class, that is a member of the population containing fitness value and pattern.

All the genetic algorithm structure, core of my code is here. Initialization of the genetic algorithm structure in the constructor consists of setting the parameters of the genetic algorithm. (like population count, etc). Here are what other methods do:

Init_population: sets a random population for starting the algorithm.

Compute_parents_and_random_population: make the necessary inputs prepared for crossover. It means to choose some of the population as parents and use some others of populations to grow randomness of the algorithm for having more exploration.

Print_generation: prints the fitness and pattern of each individual of the population.

Flip_coin: returns true or false for a simple possibility distribution.

Fitness_function: computes the fitness of an individual. The fitness is -[length of the total path]. And the pattern is the order of the traveling nodes. So the more fitness, the better the pattern is.

Order1_crossover: implements this algorithm for 2 parents to create a child.

Crossover: does order1_crossover on the whole population with some probabilities.

Compute_average_fitness: never used, but computes the average fitness of the whole population.

Get_best_pattern_and_fitness: gets the best pattern from the current population.

Mutation: does mutation with a probability.

It does the algorithm in a while loop with 2 finishing conditions:

1- max generation count

2- getting the same answer for many generations in a row

Run_mutiple_times:

It runs the algorithm multiple times and gets multiple answers (like slides), prints the best one (least distance).

Section #3:

It creates the graph (u need to just add existing edges and specify node counts. Then there are some parameters for the algorithm that depends on the input. Here are what has been gotten by a bit of trial and error. Then It runs the algorithm multiple times, prints the result of each one, finally prints the final (best) result.

You can also uncomment the following line in the run method to get more details in the algorithm steps:

```
while not finish_algorithm:
  #if (generation_index % int(max_generation_count / 50) == 0):
    #self.print_generation(generation_index)
  current_best_pattern, current_best_fitness = self.get_best_pattern_and_fitness()
  parents, random_pop = self.compute_parents_and_random_subpopulation()
  self.crossover(parents, random_pop)
  self.mutation()
  new_best_pattern, new_best_fitness = self.get_best_pattern_and_fitness()
  if (current_best_pattern == new_best_pattern):
    best_equal_in_a_row_count += 1
  else:
    best_equal_in_a_row_count = 0
  if (best_equal_in_a_row_count == finish_criteria_sucessive_equal_count):
    finish_algorithm = True
  generation_index += 1
  if generation_index == self.max_generation_count:
    finish_algorithm = True
self.print_generation(generation_index, index)
```

A2)

Exactly like the last question, I have solved this one in object-oriented form with the same structure and variable naming conventions. I have some comments on my code but I do explain it here too:

Section #1:

Here is all the genetic algorithm structure of my code, in the constructor like the first question I have set the appropriate variables and ratios that I need in the algorithm. Also, it has the following methods:

Init_population: seeds the population with some simple small numbers.

Print_generation: prints the population values and fitnesses.

F(X): computes the question function for input x

Crossover_avg: does cross-over for 2 parents to create a child. Child pattern is the average of parent's pattern. **The pattern is actually the same as X.**

Flip_coin: returns true or false for a simple possibility distribution.

Fitness_function: computes the fitness of an individual. The fitness is -[F value for the pattern (x)] so, the less the F value, the better the fitness, much more possible to become root.

Compute_parents_and_random_population: make the necessary inputs prepared for crossover. It means to choose some of the population as parents and use some others of populations to grow randomness of the algorithm for having more exploration.

Crossover: does crossover_avg on the whole population with some probabilities.

Compute_best_pattern_and_fitness: gets the best (pattern, fitness) of the whole population.

Mutation: does mutation in such a way that it changes the current value of an individual a little bit. It can be increased or decreased which is chosen randomly.

Run: it runs all steps of the algorithm with 2 finishing conditions in a while loop:

1- max generation count

2- getting a worse answer (less fitness in the best fitness) for some generations in a row

multiple_run: that is what we should run with a run_count number. It runs the algorithm multiple times, prints the result of each one, finally prints the best ever solution. It means it prints the best X matching as the root with its F value that is near to 0.

Section #2:

It actually sets the necessary parameters that some are gained by trial and error. Then it runs multiple_run to get the best answer. Details of generations are printed in the output.

You can see more details of each run (for example population details of all generations in a run) by uncommenting this line:

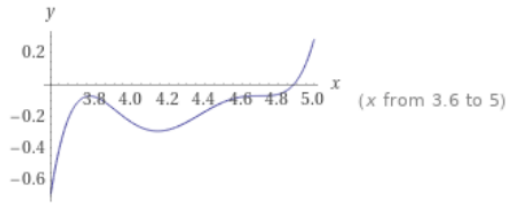
```
def run(self, run_index):
  self.init_population()
  finish_algorithm = False
  generation_index = 0
  worse_in_a_row_count = 0
  while (not finish_algorithm):
    current_best_pattern, current_best_fitness = self.compute_best_pattern_and_fitness()
    parents, random_pop = self.compute_parents_and_random_pop()
    self.crossover(parents, random_pop)
    self.mutation()
    #self.print_generation(generation_index, run_index)
    generation_index += 1
    if (generation_index == self.max_generation_count):
      finish_algorithm = True
    new_best_pattern, new_best_fitness = self.compute_best_pattern_and_fitness()
    if current_best_fitness > new_best_fitness:
     worse_in_a_row_count += 1
    if worse_in_a_row_count == self.max_worse_in_a_row_count:
      finish_algorithm = True
  self.print_generation(generation_index, run_index)
  best_pattern, best_fitness = self.compute_best_pattern_and_fitness()
  return best_pattern, best_fitness
```

Finally, my answer is almost the root as I tested it in an online equation solver:

Input interpretation:

$$9\,x^5 - 194.7\,x^4 + 1680.1\,x^3 - 7227.94\,x^2 + 15\,501.2\,x - 13\,257.2$$

Plots:



“THANKS”