COMP 6660 Fall 2020 Assignment 1C

Jordan Sosnowski

[jjs0029@auburn.edu](mailto:jjs0029@auburn.edu)

Table of Contents

[Green 3: 2](#_Toc52111279)

[Performance 5](#_Toc52111280)

[BC1 6](#_Toc52111281)

[BC2 6](#_Toc52111282)

[C1 7](#_Toc52111283)

[C2 7](#_Toc52111284)

[Green 3.1: 1B Evolutionary Algorithm vs 1C Evolutionary Algorithm 8](#_Toc52111285)

[1BC 8](#_Toc52111286)

[2BC 8](#_Toc52111287)

[Green 3.2: Validity Forced Analysis 9](#_Toc52111288)

[1BC 9](#_Toc52111289)

[2BC 10](#_Toc52111290)

[C1 11](#_Toc52111291)

[C2 11](#_Toc52111292)

[Green 3.3: Exploring Penalty Coefficients 12](#_Toc52111293)

[10 vs 5 12](#_Toc52111294)

[50 v 10 13](#_Toc52111295)

[50 v 5 14](#_Toc52111296)

[Yellow 1: Self-adaptive Mutation Rate 15](#_Toc52111297)

[Red 2: Self-adaptive Penalty Coefficient 16](#_Toc52111298)

# Green 3:

For this assignment series, we were asked to implement constraint-satisfaction for our evolutionary algorithm. Specifically, a penalty function for constraints. Instead of immediately invalidating solutions and giving them a fitness of 0 a penalty function would be applied to them. My penalty function was as follows. First, you iterate through a dictionary of black cells that are not satisfied. You then get the difference in what the value of bulbs around the cell should be vs what was and take the absolute value of that. This is added to a penalty sum. After checking unsatisfied black cells, I go on to check the amount of intersecting light rays there were. Each number of intersecting rays are added up to the penalty sum as well. Finally, this penalty sum is multiplied by the penalty coefficient and subtracted from the original fitness.

    def penalty\_function(self, fitness, black\_cells, intersections, penalty\_coef):

        """ Penalizes current individual based on the amount of intersecting rays

        and black cells that are not constrained.

        f' = f - p\*(black\_cells + intersections)

        where f is the original fitness

        where f' is the new fitness

        where p is the penalty coefficient

        """

        penalty = 0

        for expected, actual\_list in black\_cells.items():

            for actual in actual\_list:

                penalty += abs(expected - actual)

        for section in intersections.values():

            for intersection in section:

                penalty += intersection

        fitness -= penalty\*penalty\_coef

        return fitness

Using a penalty function was also easy to integrate as I simply inserted it into the fitness function but with a conditional statement based on a user-defined parameter.

    def calculate\_fitness(self, board, penalty\_coef):

        """ Calculates fitness of current board.

        board - RxC numpy array representing the game board, filled with bulbs,

        black cells, and light

        """

        fitness = lightup.calculate\_completion(board)

        black\_cells = lightup.check\_black\_cells(board, self.ignore\_black\_cells)

        intersections = lightup.check\_intersections(board)

        fitness\_prime = fitness

        if self.constraint\_algorithm in CON\_PENALTY:

            if penalty\_coef is None:

                penalty\_coef = self.penalty\_coefficient

            fitness\_prime = self.penalty\_function(

                fitness, black\_cells, intersections, penalty\_coef)

        penalty = fitness - fitness\_prime

        return fitness\_prime, penalty

We are additionally asked to implement a new survival strategy. Thus far we have been using a “plus” survival strategy or a (μ+λ)-EA. Now we were asked to implement a comma or generational EA [(μ,λ)-EA]. For a generational EA for every generation, the existing generation is discarded and only the children live on. Due to the modularity of my code, this was rather easy to implement. After the children are created, I call the below function to determine which individuals will be passed to survival selection.

    def survival\_strategy\_selection(self, old\_generation, children):

        """ Decide on next generation for survival function.

        old\_generation - list of individuals that make up the older generation

        children - list of individuals that make up children for the new generation

        """

        if self.survival\_strategy\_algorithm in PLUS:

            # (μ + λ)-EA

            return old\_generation + children

        if self.survival\_strategy\_algorithm in COMMA:

            # (μ, λ)-EA

            return children

In addition to the survival strategy, we were asked to implement a new parent selection algorithm and two new survival selection algorithms. First, I implemented a uniform random parent selection algorithm which was trivial.

    def uniform\_random\_parent(self, individuals):

        """ Selects parents randomly with a uniform distribution

        individuals - list of Individual objects

        """

        mating\_pool = random.choices(individuals, k=self.children\*2)

        return mating\_pool

Following this, I implemented the uniform random for survival selection as well. Since for survival, we do not want to sample with the replacement I used random.sample as random.choices use replacement.

    def uniform\_random\_survival(self, individuals):

        """ Selects parents randomly without replacement with a uniform distribution

        individuals - list of Individual objects

        """

        mating\_pool = random.sample(individuals, k=self.parents)

        return mating\_pool

Last was FPS for survival selection. For this I had to use NumPy's random.choice as Python’s standard library’s sample does not allow weight to be passed to it.

    def fitness\_proportional\_selection\_survival(self, individuals):

        """ Selects parents based on fitness probability without replacement.

        individuals - list of Individual objects

        """

        fitness\_sum = sum(individuals)

        if fitness\_sum == 0:

            return random.choices(individuals, k=self.children\*2)

        probs = [individual.fitness /

                 fitness\_sum for individual in individuals]

        return list(np.random.choice(individuals, p=probs, size=self.children\*2, replace=False))

## Performance

For the problems for this assignment, I used the same EA parameters for each one. They are listed below:

{

    "children": 50,

    "parents": 100,

    "parent\_alg": "sus",

    "child\_alg": "one point crossover",

    "survival\_alg": "truncation",

    "termination\_alg": "num of evals",

    "num\_of\_runs": 30,

    "initialization\_alg": "vanilla",

    "survival\_strategy\_alg": "plus",

    "fitness\_evals": 10000,

    "mutation\_rate": 0.40,

    "penalty\_coefficient": 10,

    "constraint": "penalty",

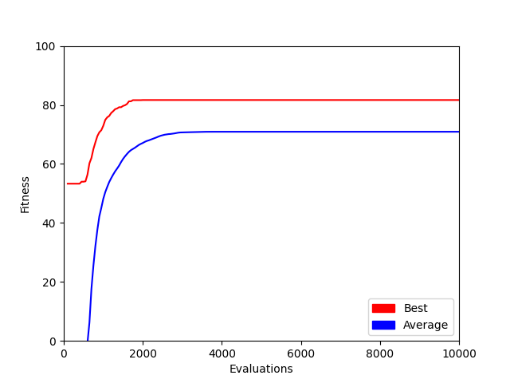
    "ignore\_black\_cells": false,

    "log\_1b": true

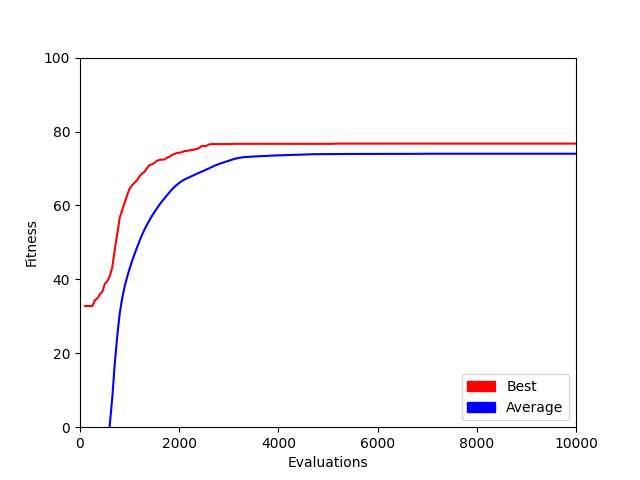
}

As BC# are easier problems the graphs are what I expected. Since C# are more difficult their poor graphs are also not surprising. I think I could achieve higher performance on those but since that is not the focus on these assignments I did not put to much effort into those portions.

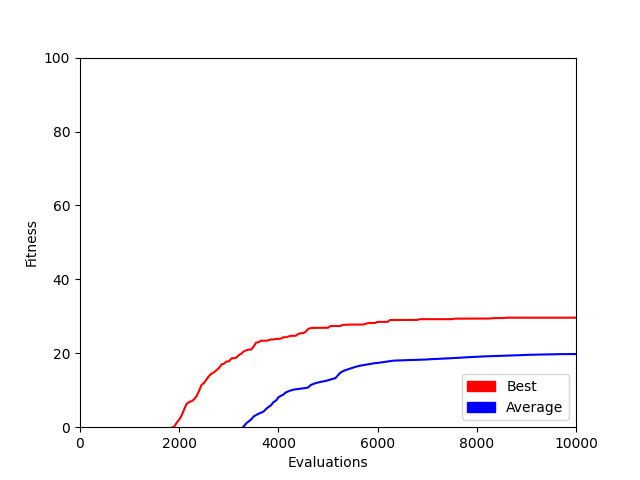
### BC1



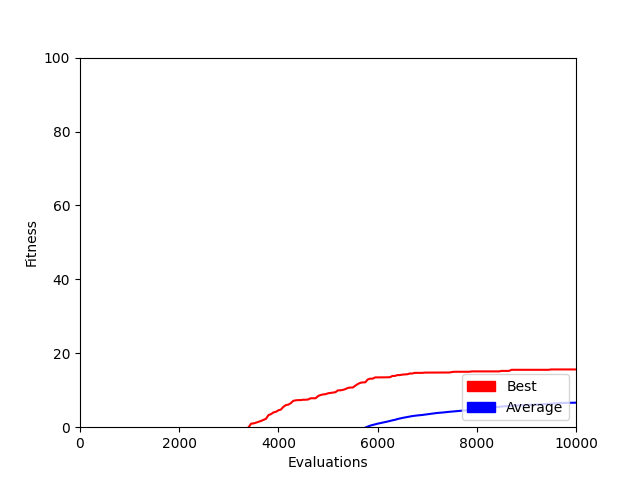
### BC2



### C1



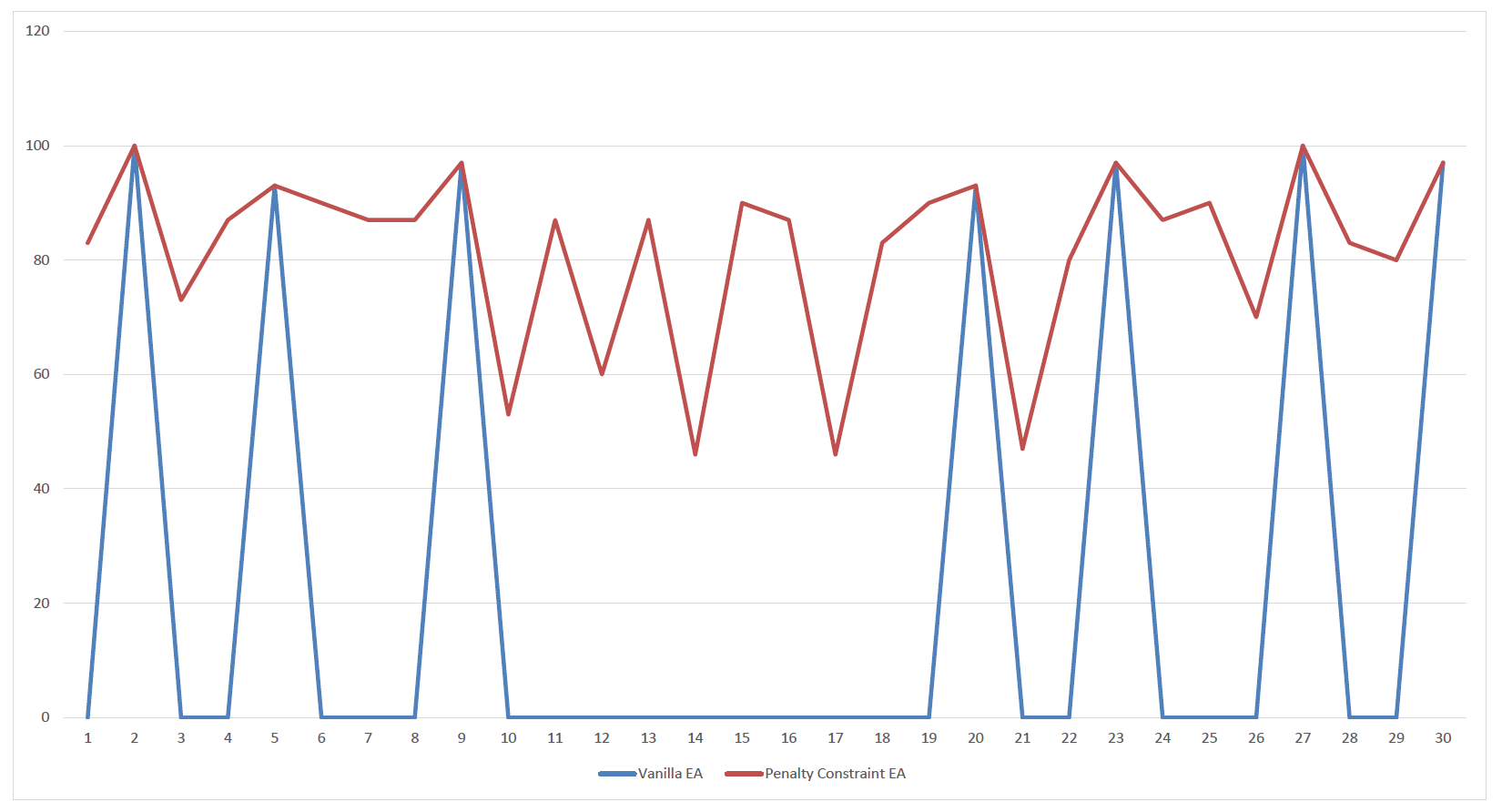
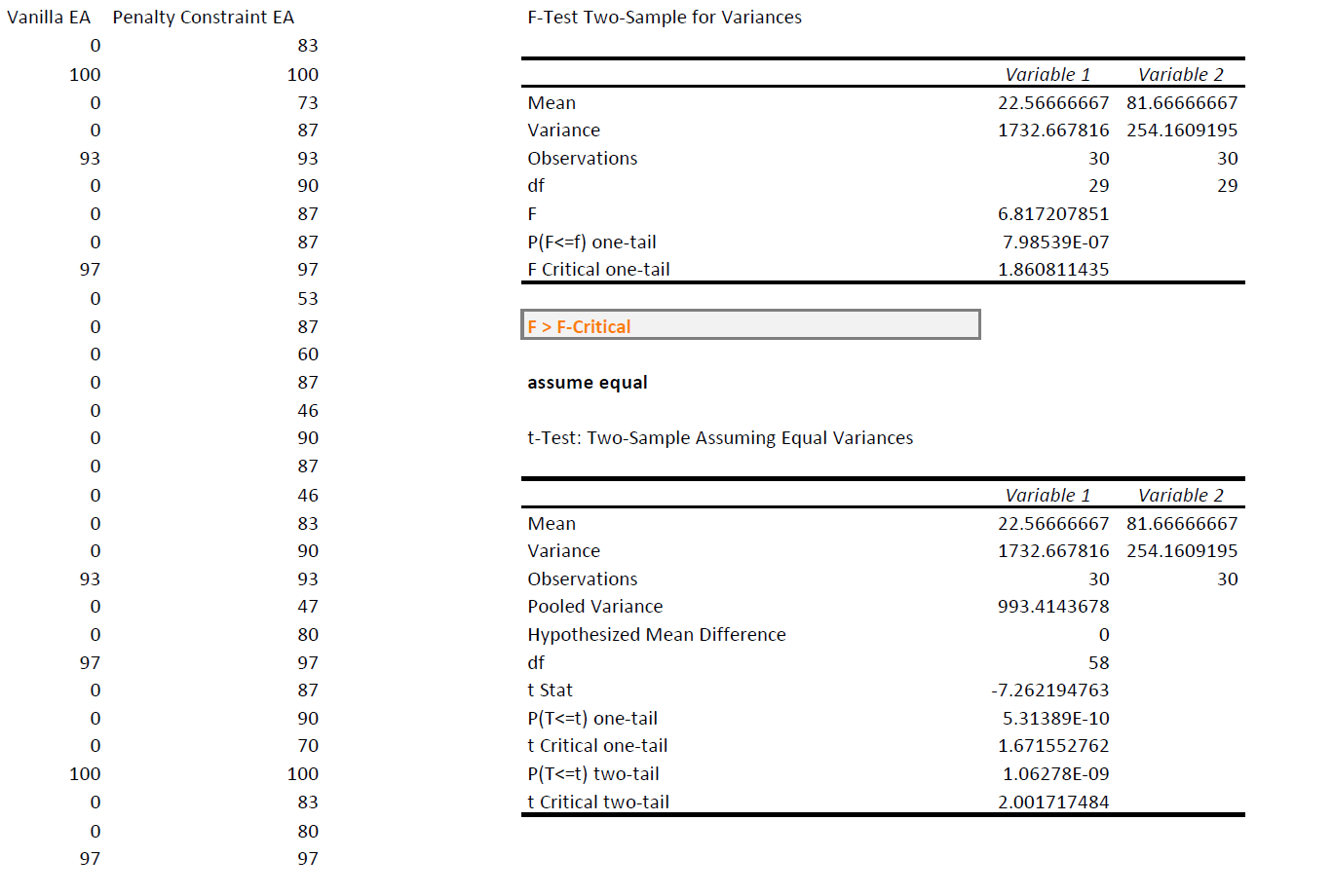
### C2



## Green 3.1: 1B Evolutionary Algorithm vs 1C Evolutionary Algorithm

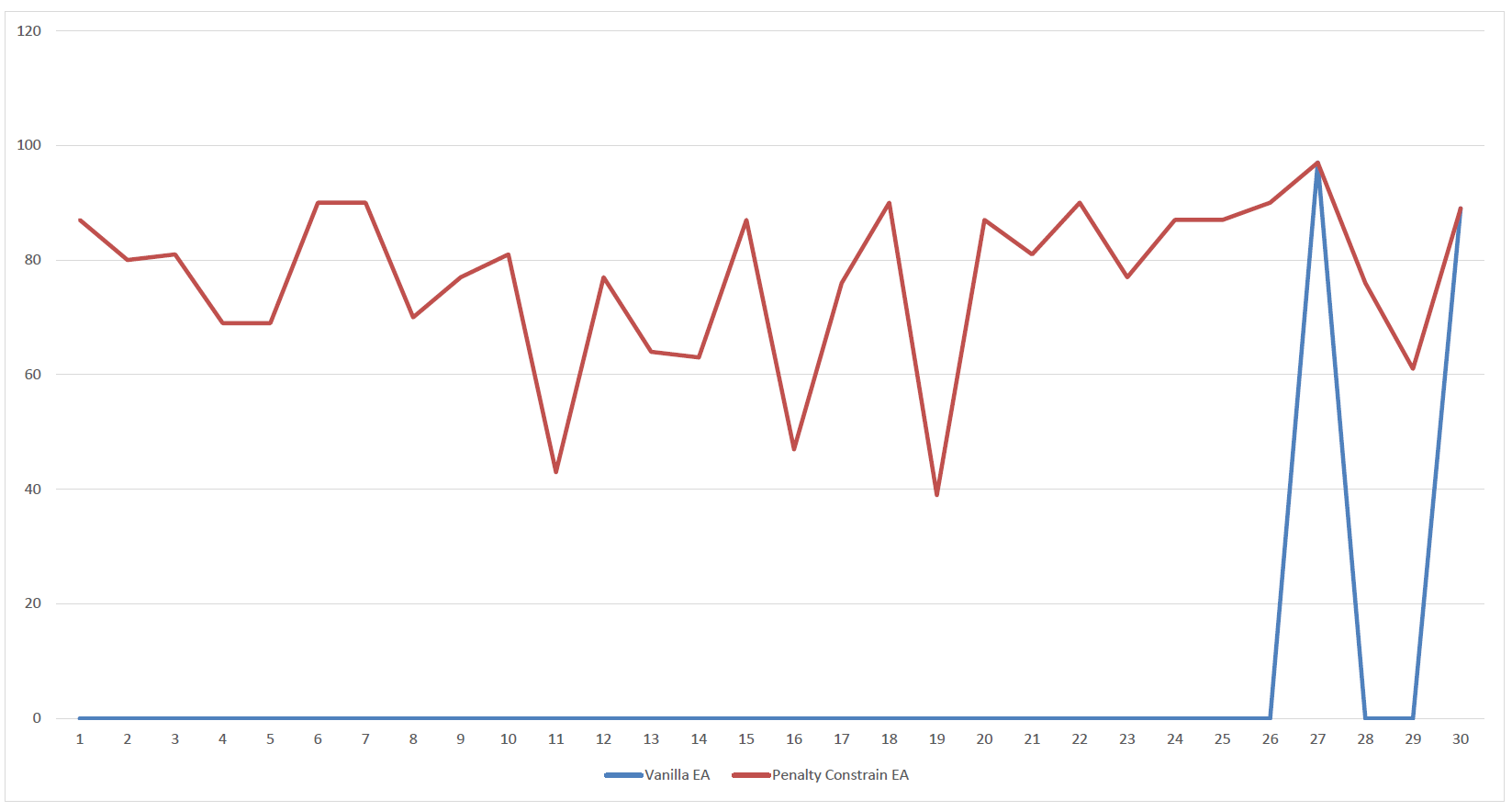
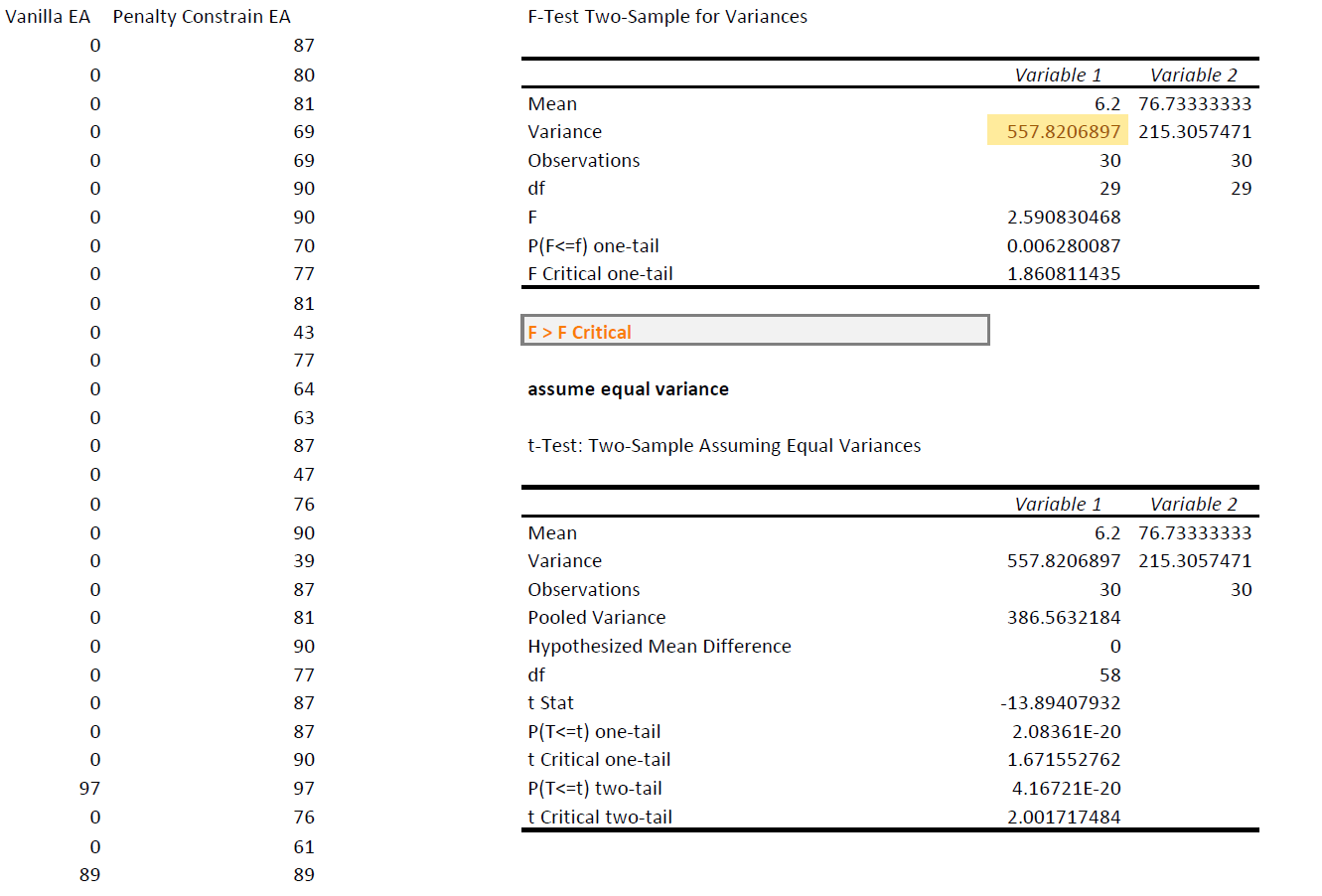
### 1BC

Here we have a comparison against 1B’s EA and 1C’s EA. 1B is using a “vanilla” EA and 1C is using a penalty constraint EA. As you can see 1C has higher finesses where 1C mostly has finesses of 0. Unfortunately, this means that most of the solutions in 1C are technically invalid but if this EA could run longer in theory, we would see these penalized solutions go away and be replaced with “valid” ones.



### 2BC

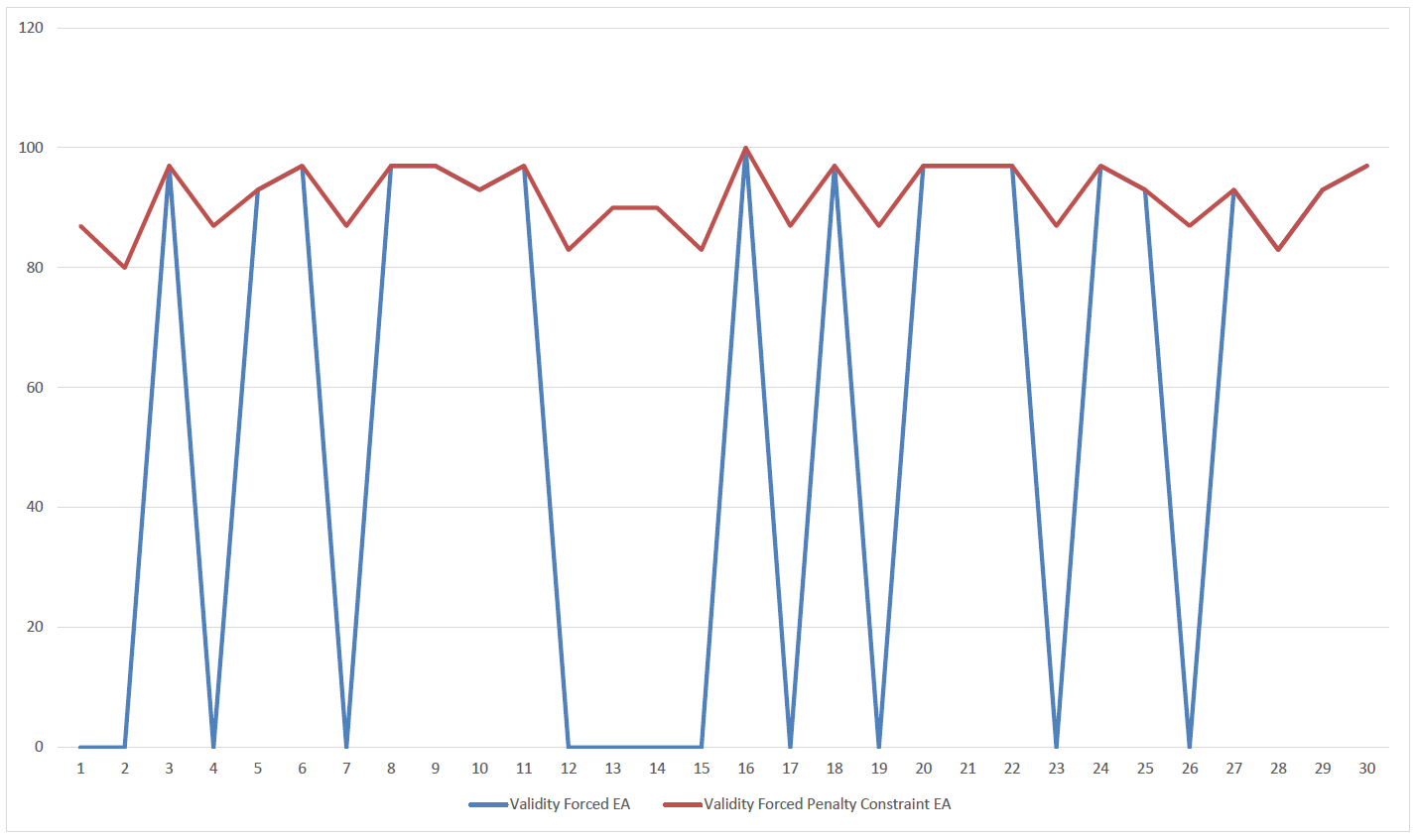
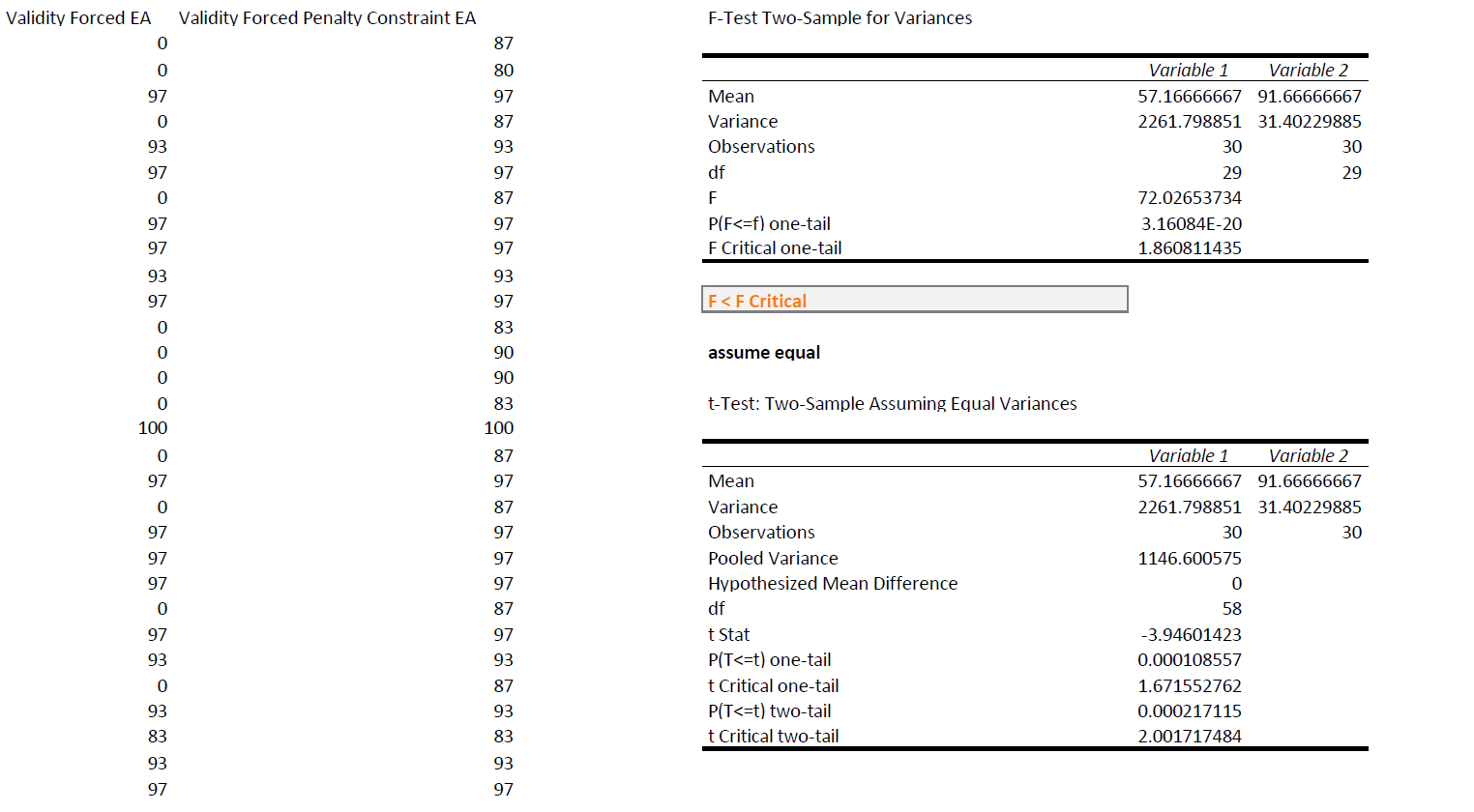
Here we can see the same thing except with the problem 2BC.



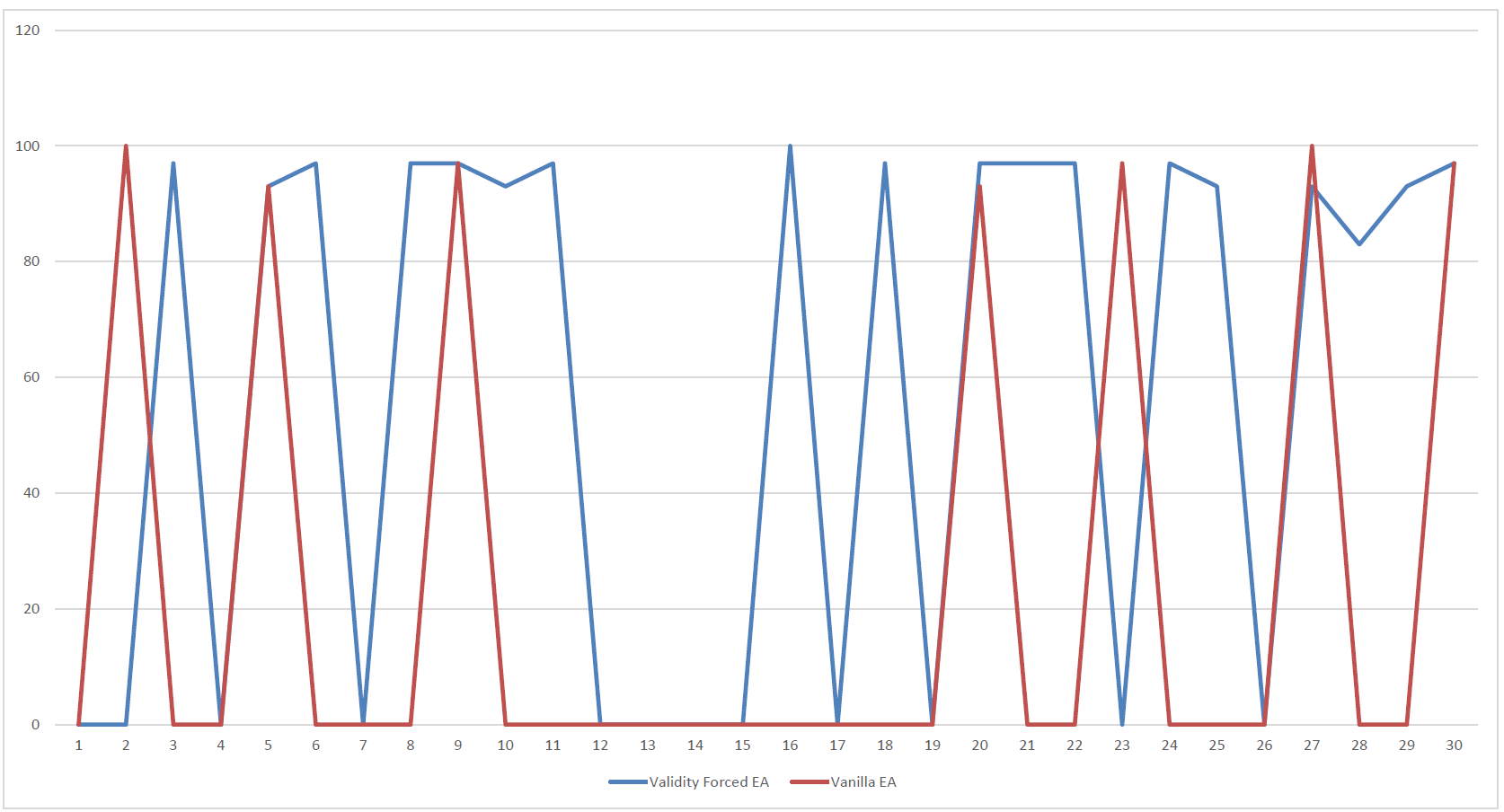
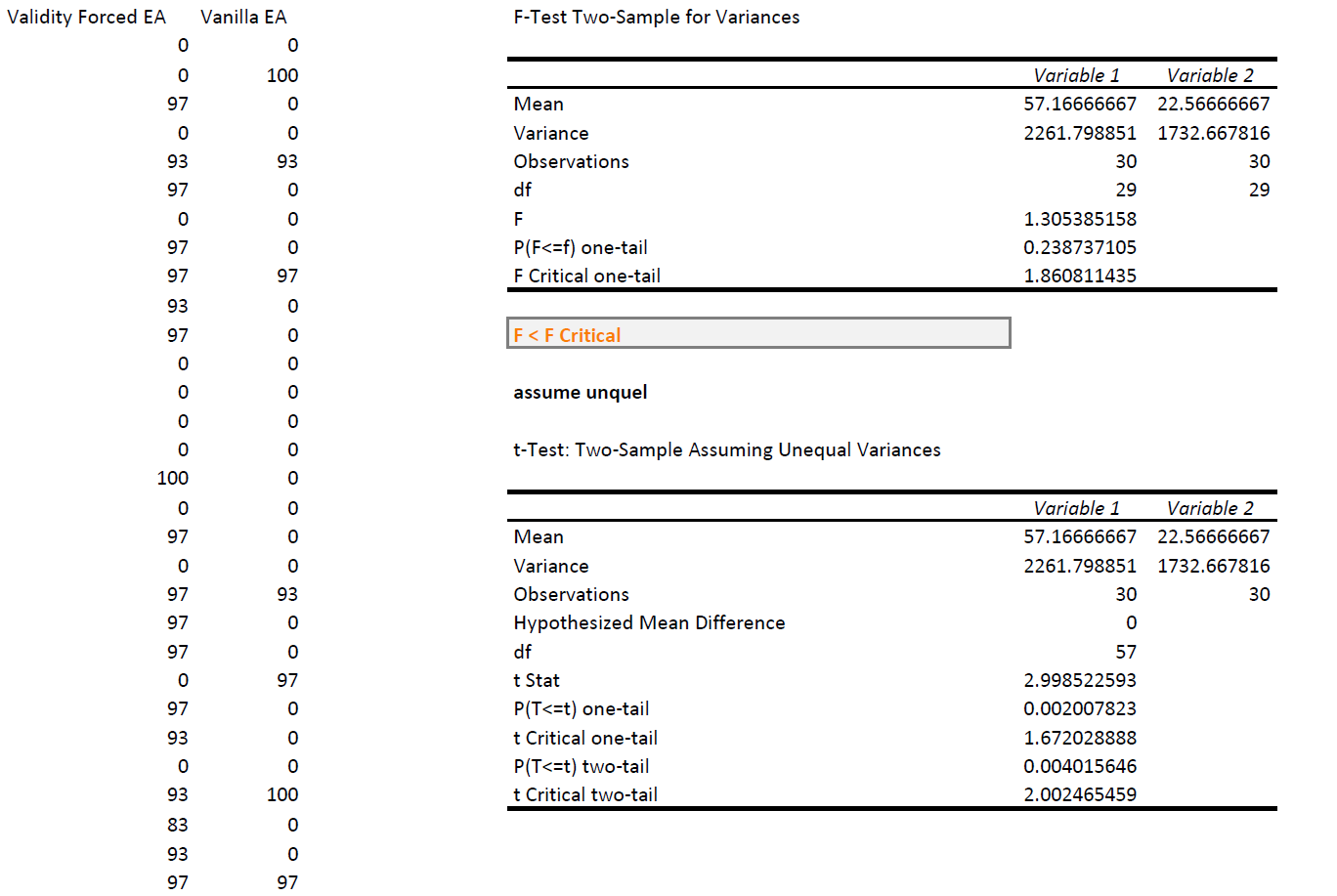
## Green 3.2: Validity of Forced Analysis

### 1BC

For this analysis, we have a Validity Forced Vanilla EA against a Validity Forced Penalty Constraint EA. With the same logic as before we expect to see the penalty constraint to perform better.

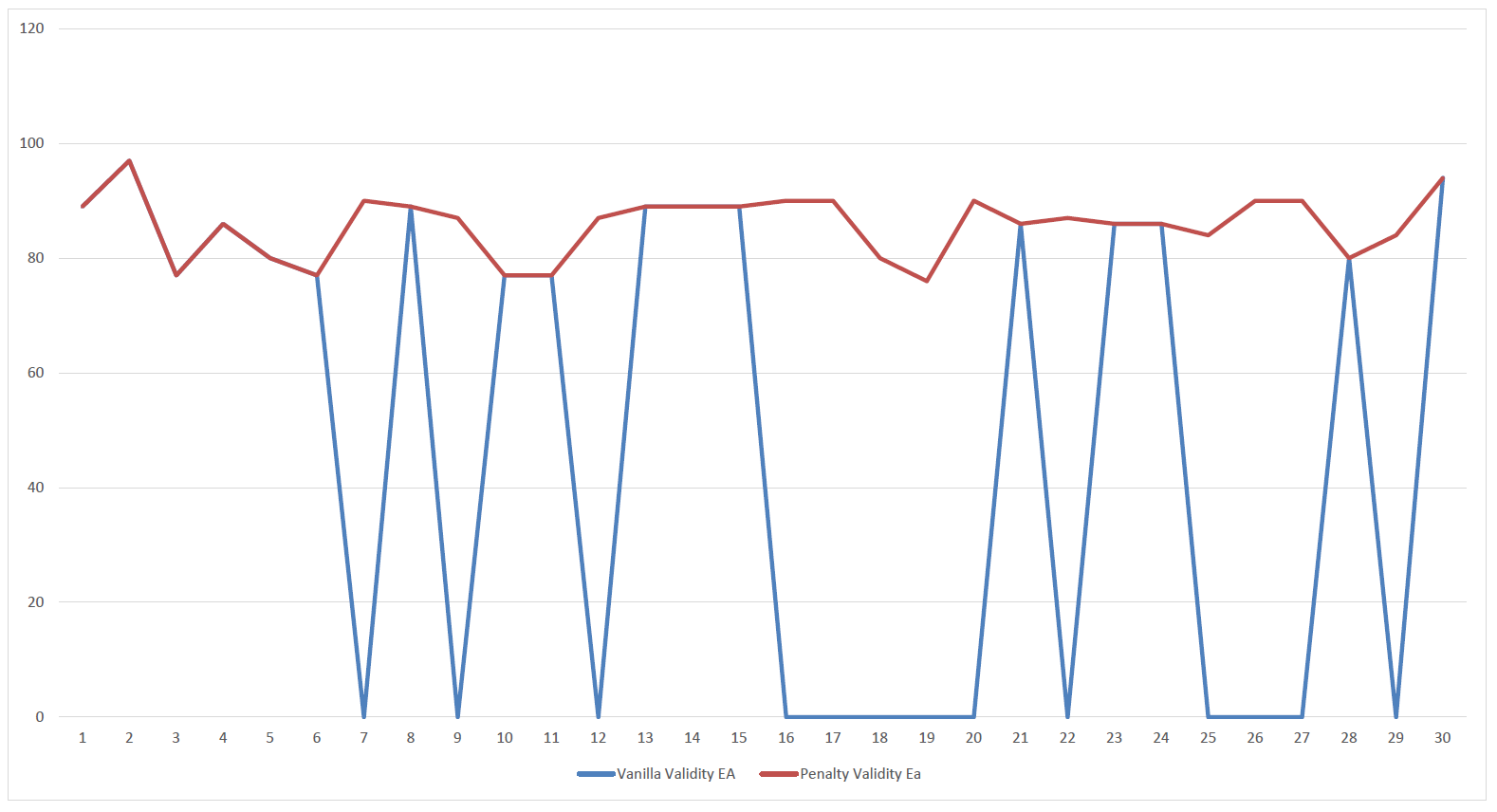
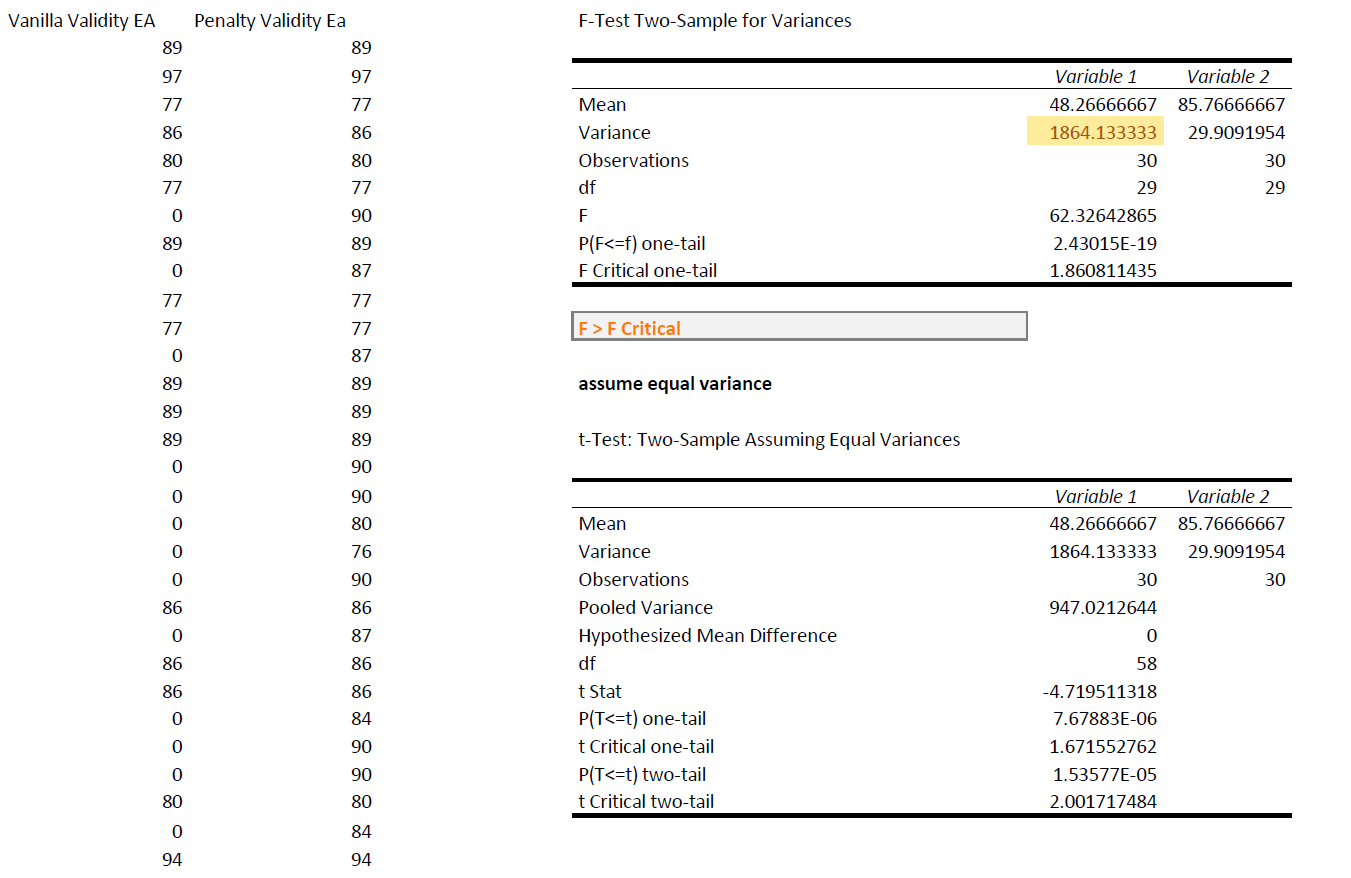


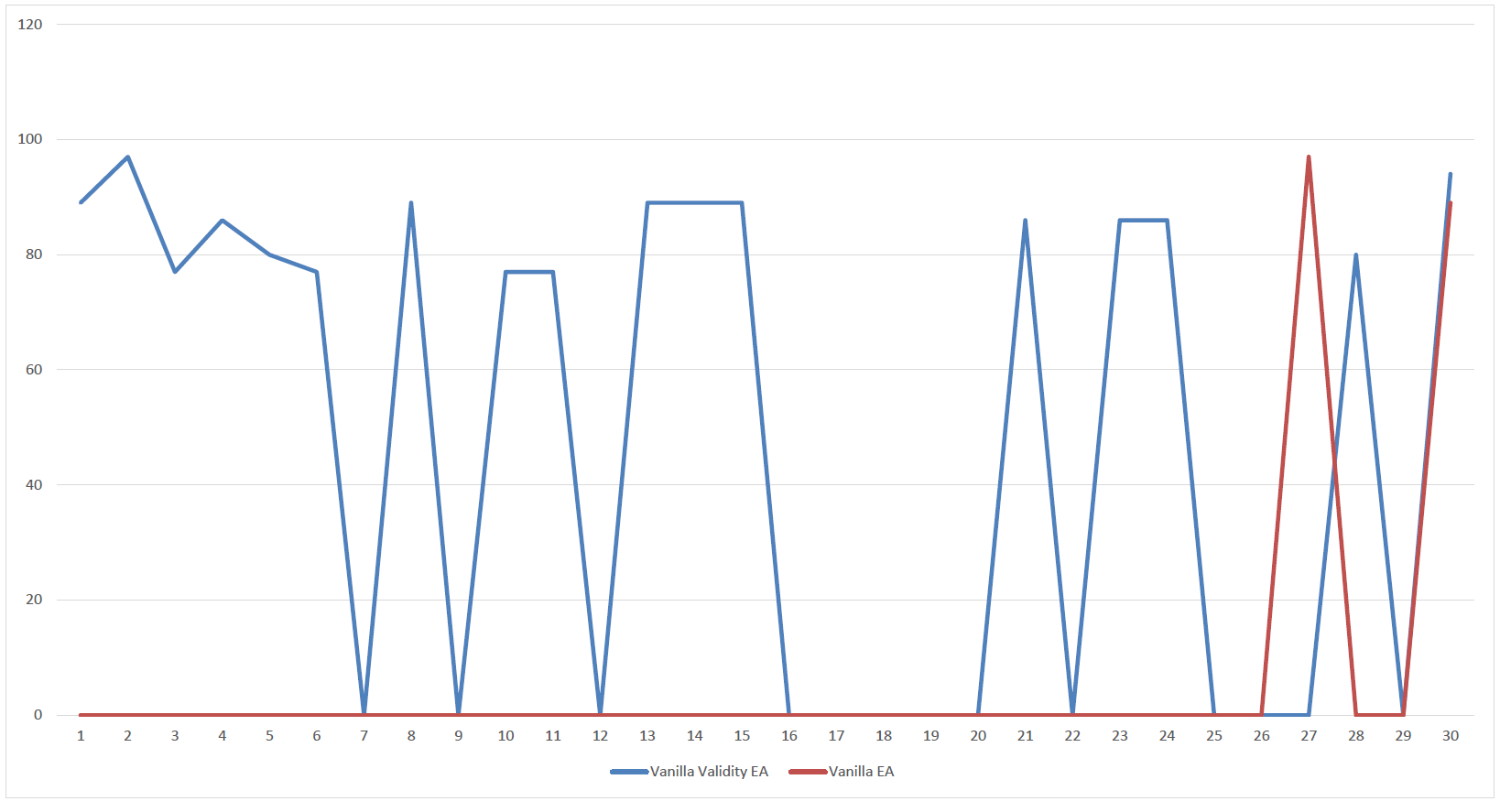
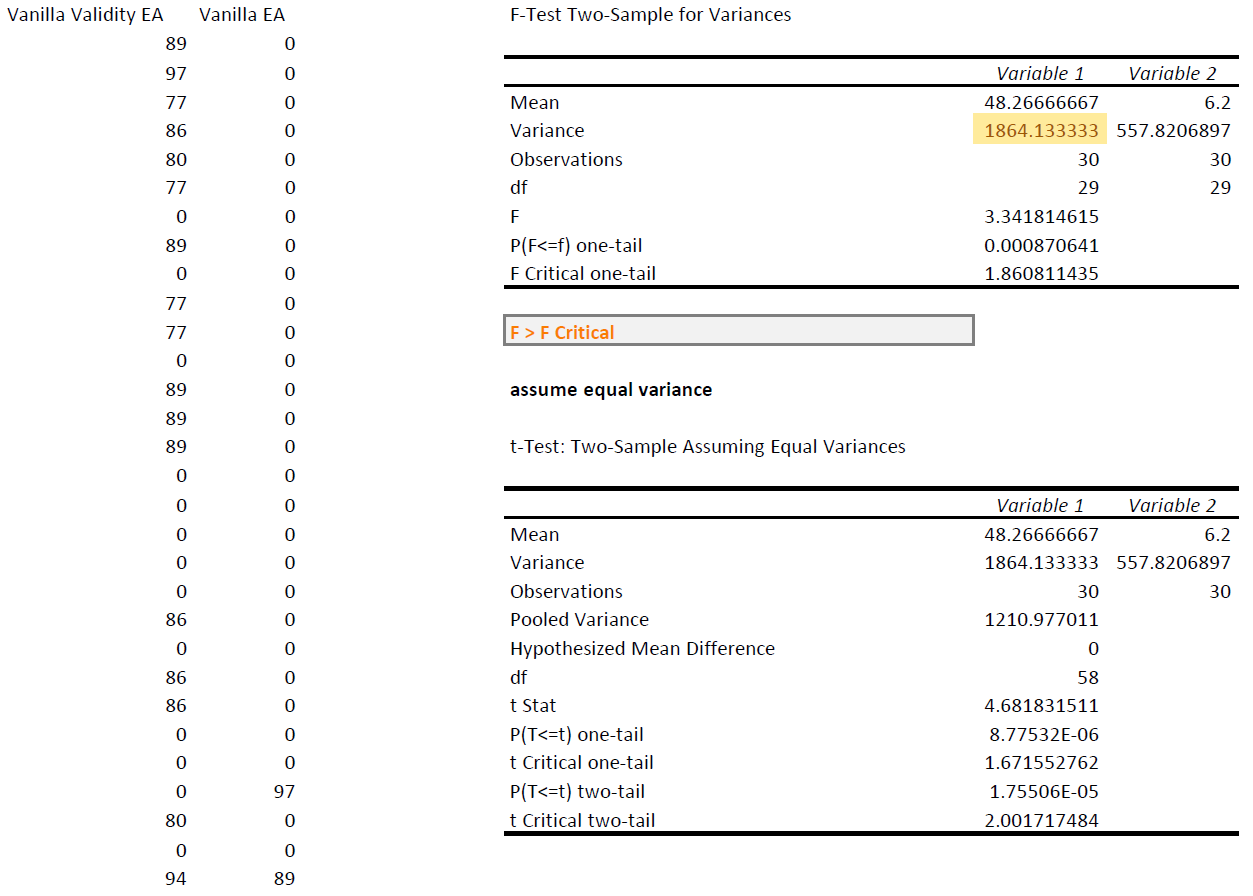
For the next analysis, we compare validity forced vs a plain vanilla EA. Since the validity is enforced, we see that the Validity Forced EA is performing slightly better than the plain Vanilla EA which is to be expected.



### 2BC

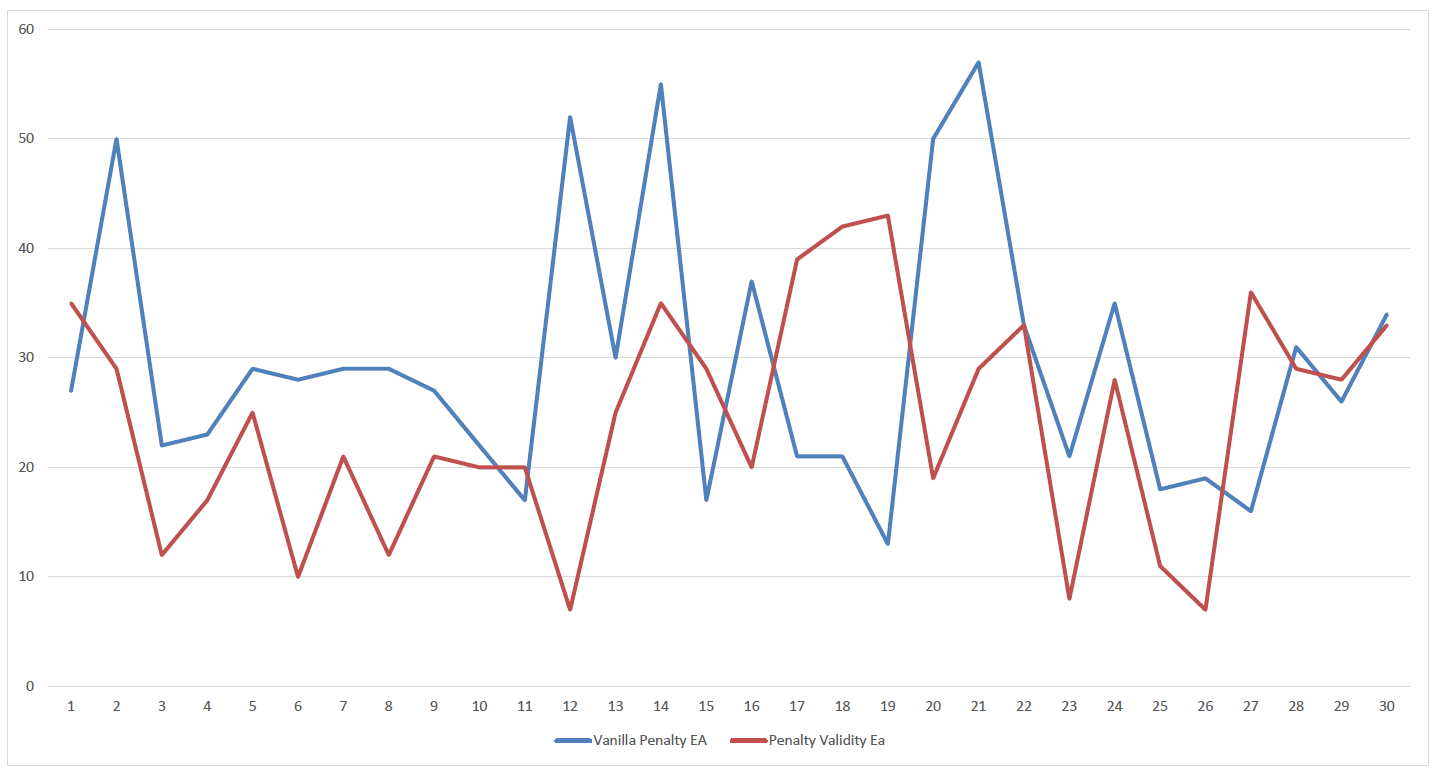
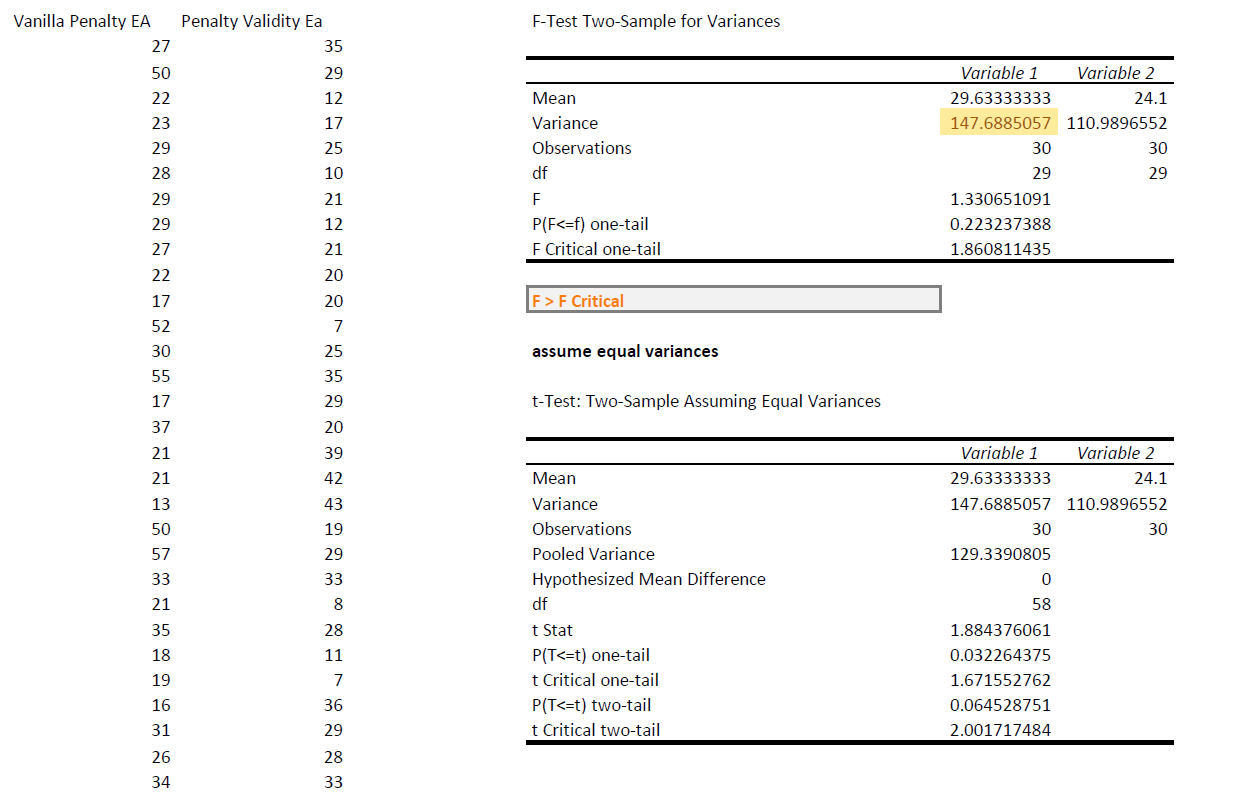
Here we can see the same thing except with the problem 2BC.



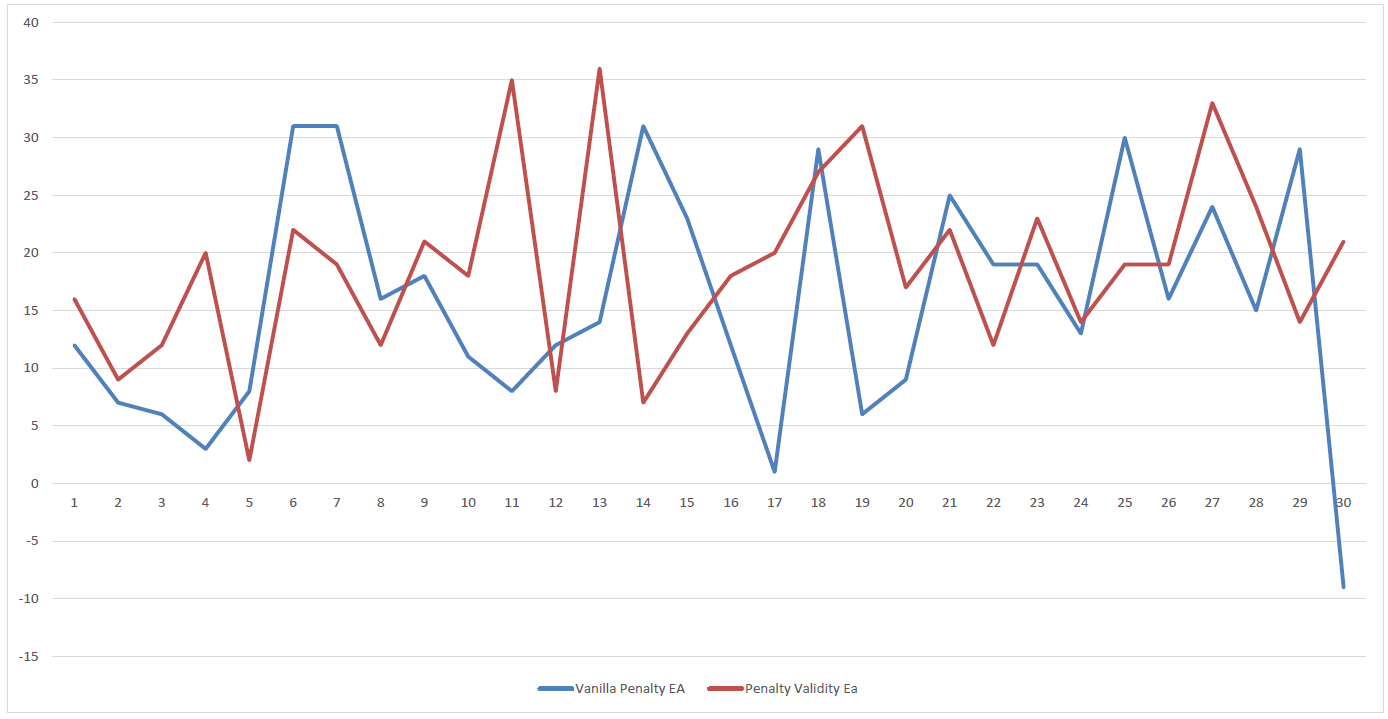
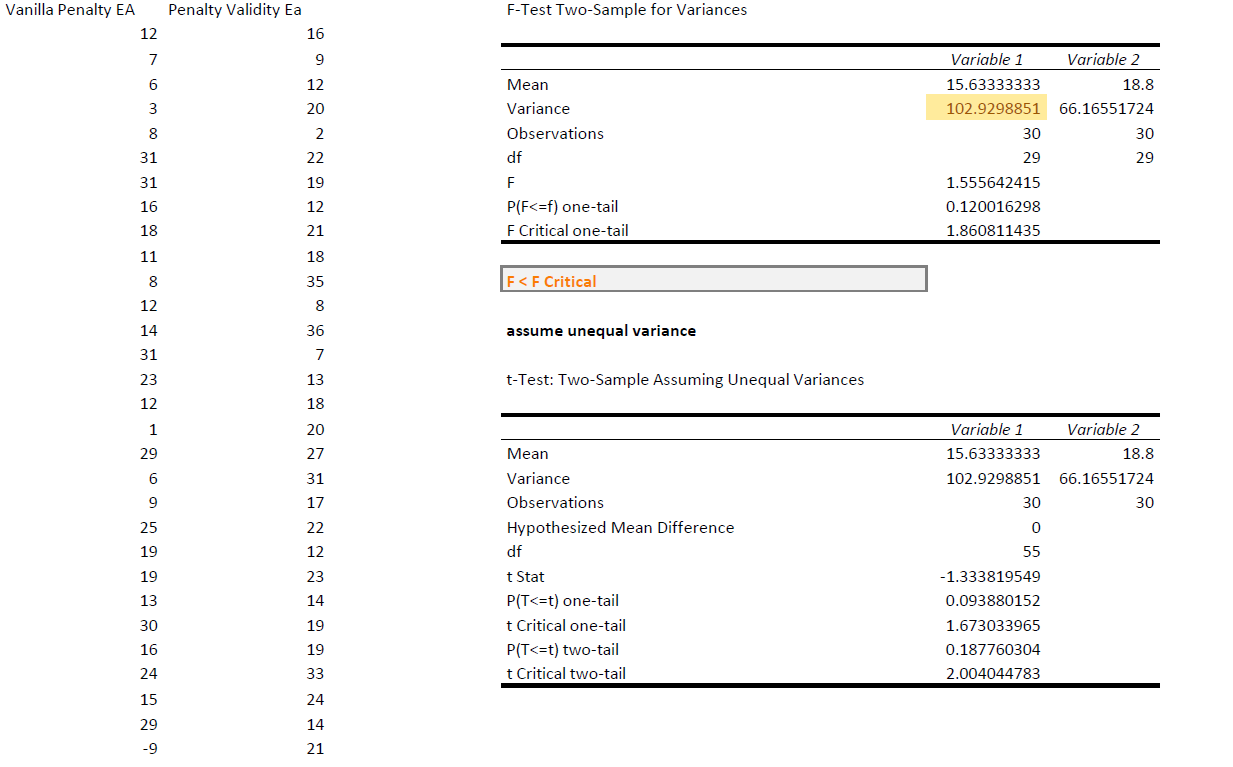


### C1

Here we can see the same thing except with the problem C1 which is meant to be harder than the BC problems.



### C2



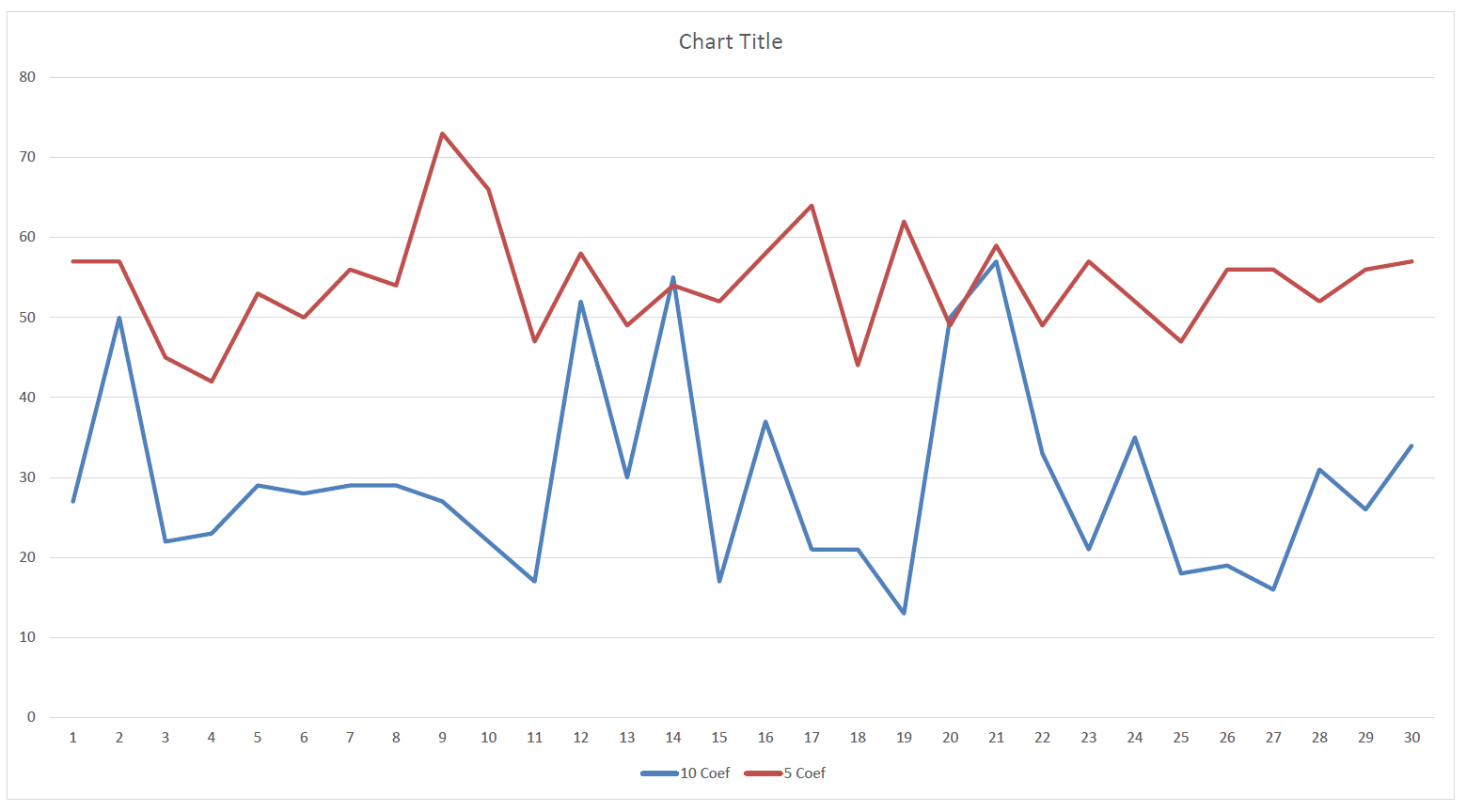
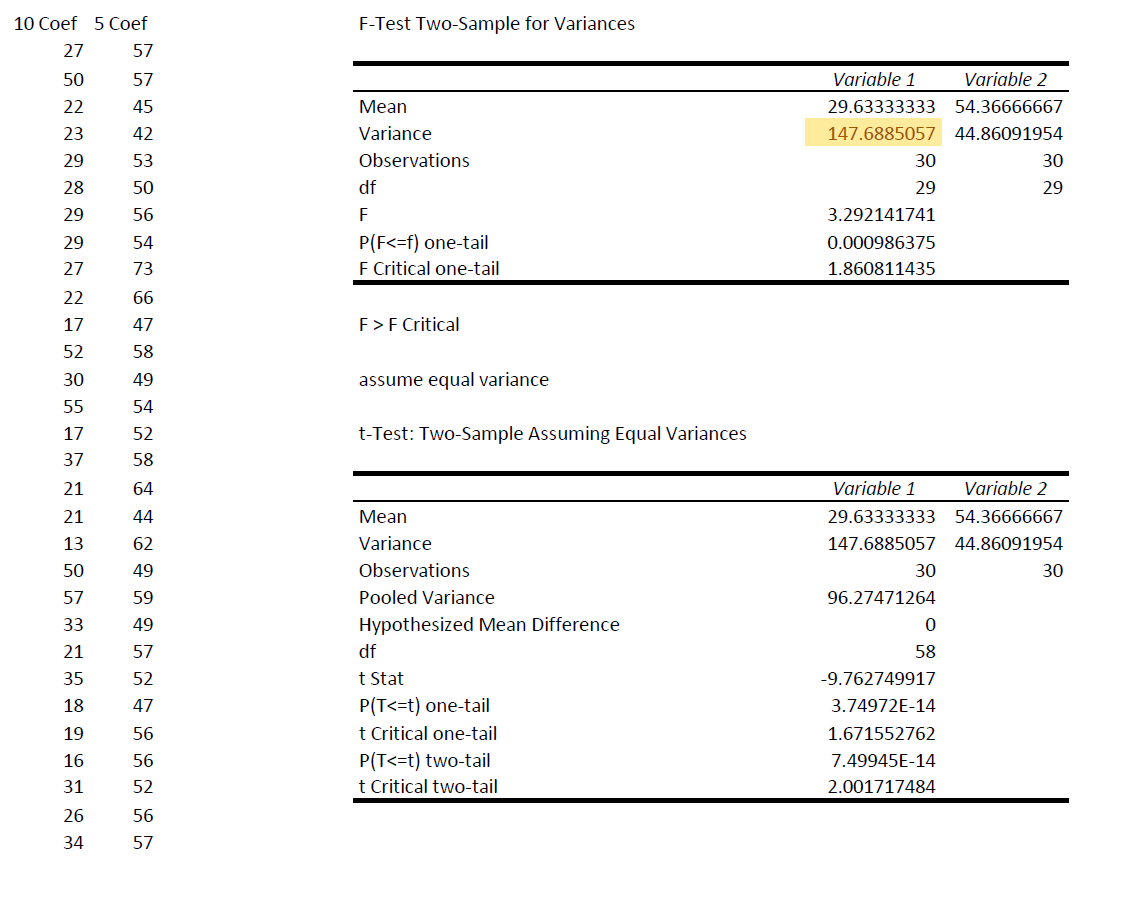
## Green 3.3: Exploring Penalty Coefficients

For the last green objective, we were asked to explore penalty coefficients. When I first implemented the penalty coefficient, I picked 10 as that seemed like a good number, and the results, I got were not awful, so I stuck with it. For this objective, I picked a coefficient that was lower and was significantly higher; 5 and 50. My main question after doing this was what is the best way of finding a good penalty coefficient? As if you make it too high it will just kill everything, but if its too low the invalid solutions will look “good”. Additionally, what is the best way of checking that information? In our logs we only check fitness but if we have a low penalty it will look like everything has a “good” fitness.

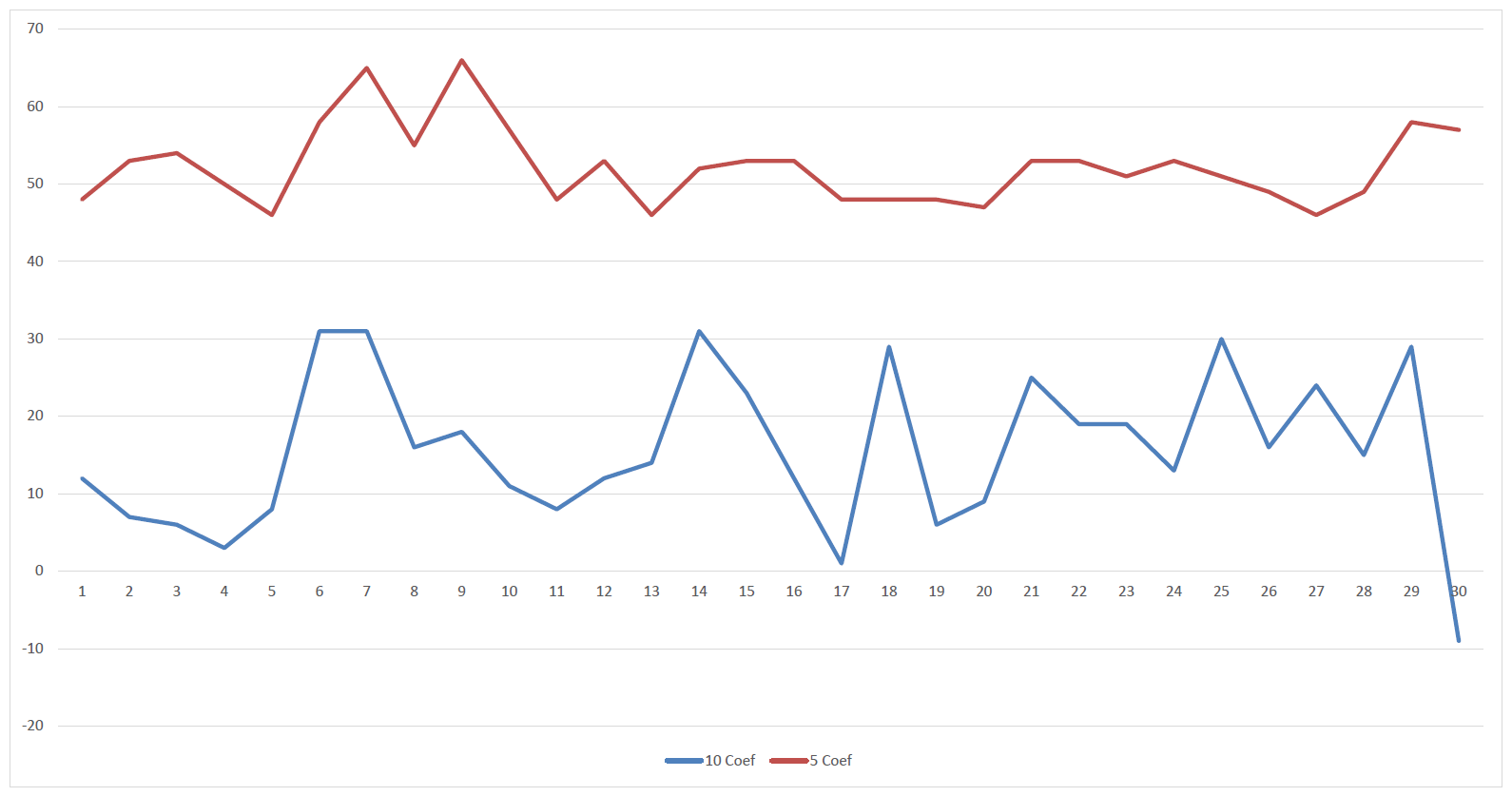
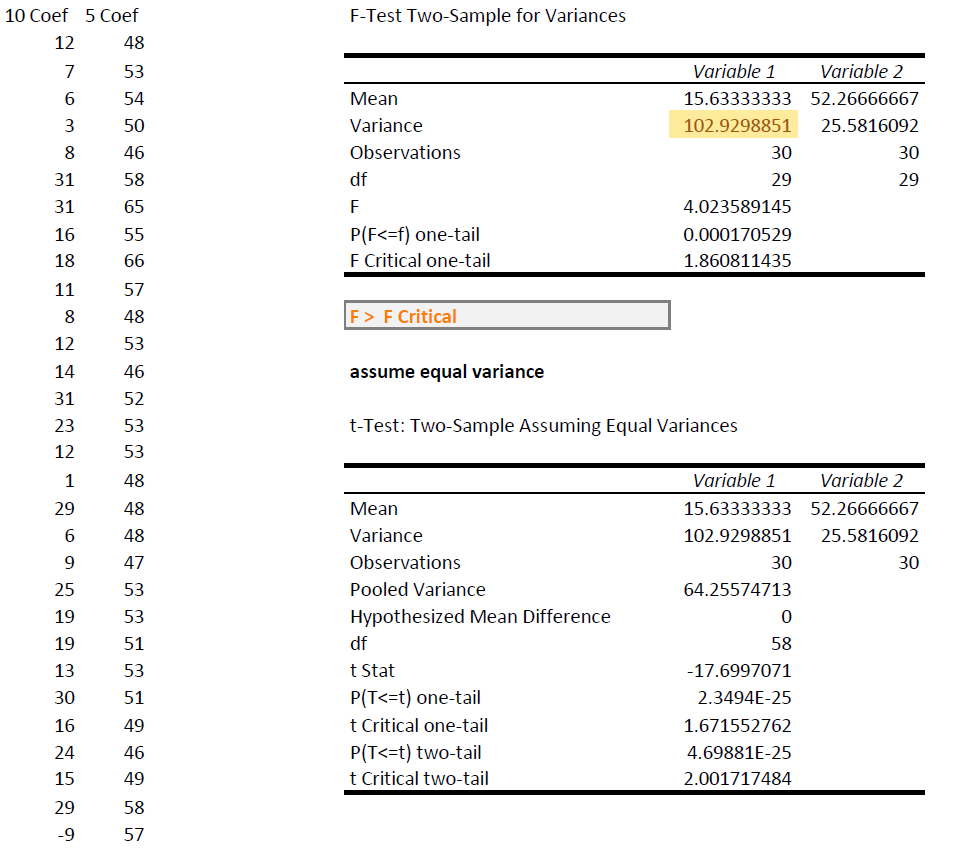
### 10 vs 5

We can see that a penalty of 5 outperforms a penalty coefficient of 10. However, as discussed later a lower penalty coefficient can allow seriously “invalid” solutions to flourish. So I think it would most likely be best to be a little higher but with lower performance as the solutions are most likely similar.

#### C1



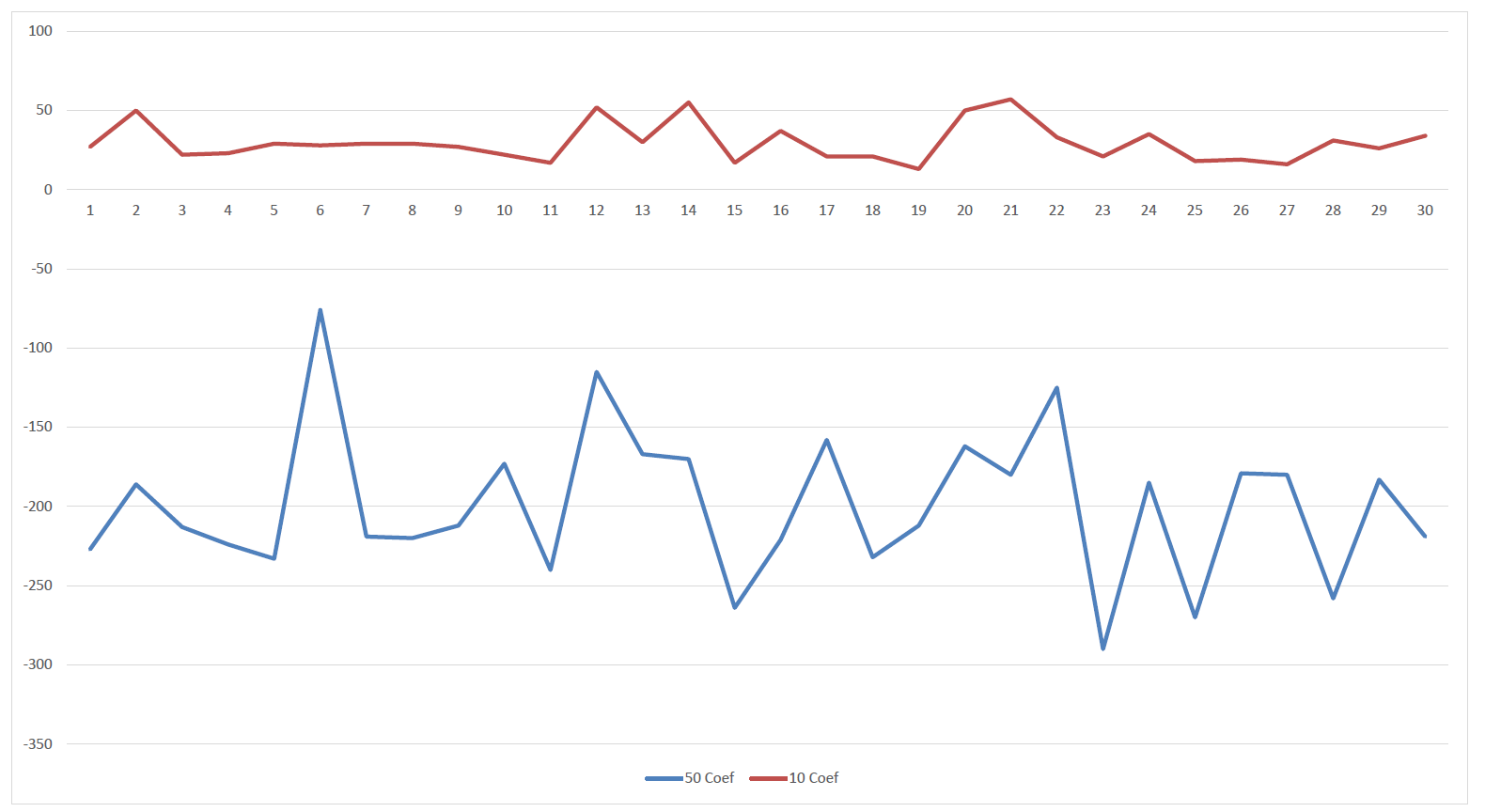
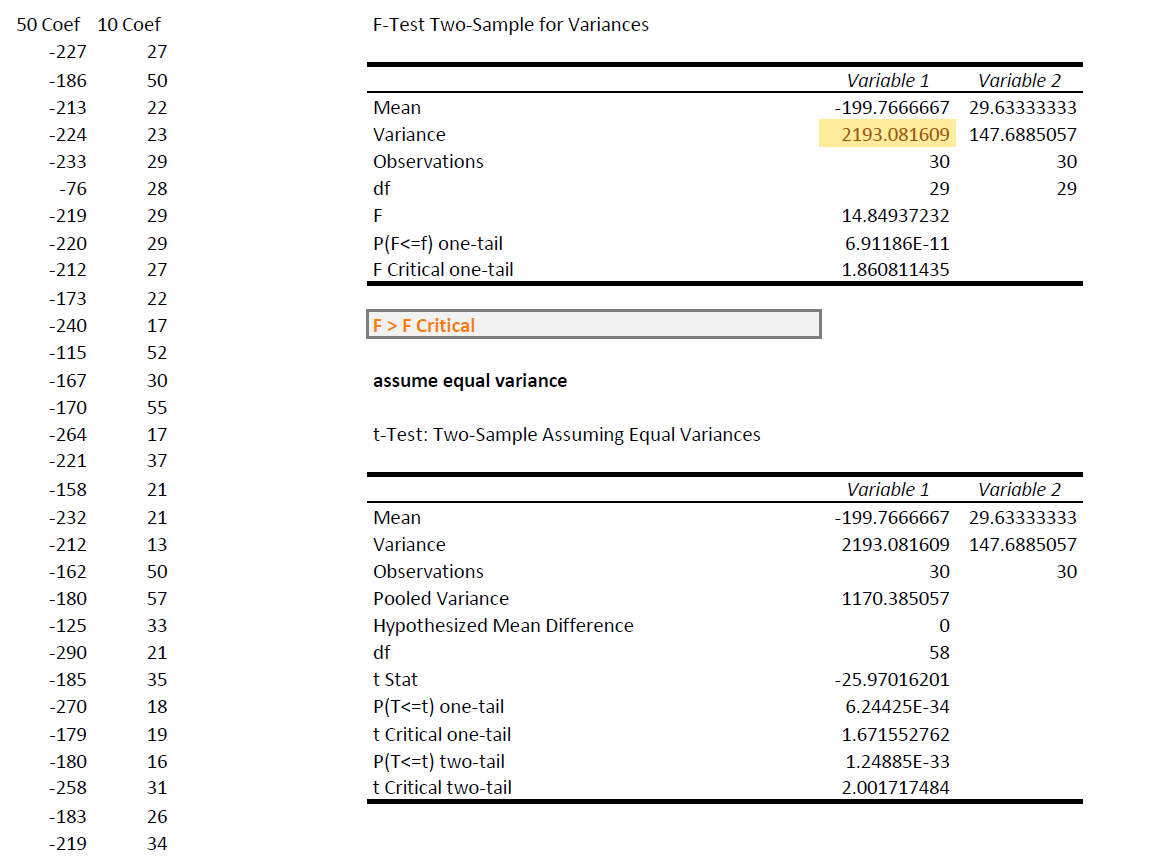
#### C2



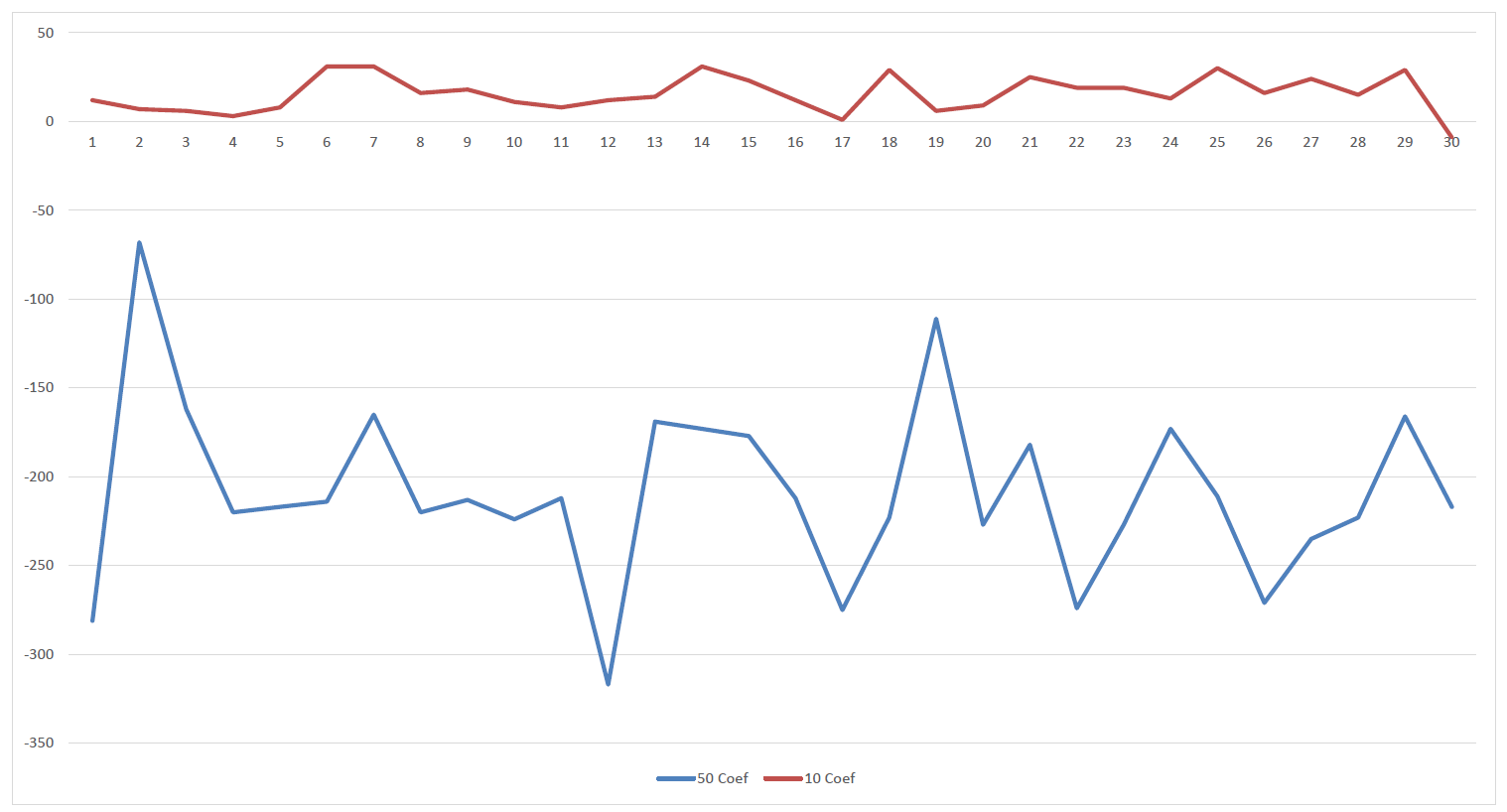
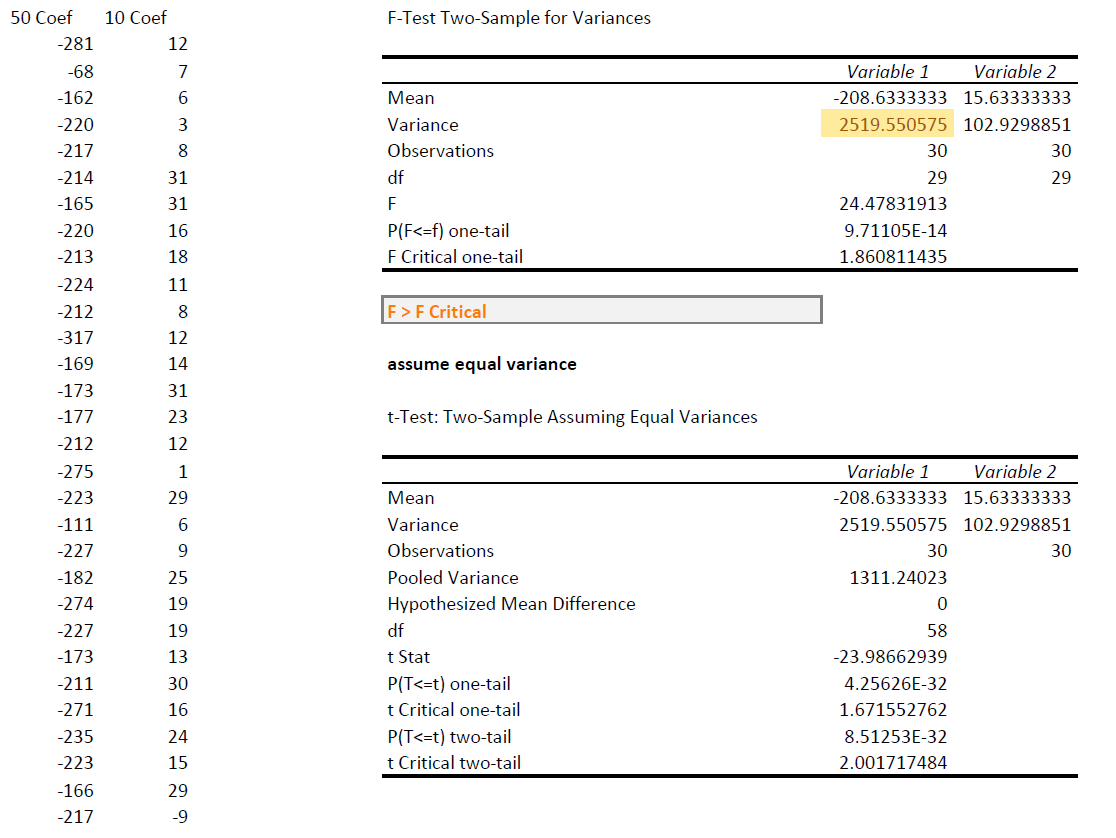
### 50 v 10

Here we have a coefficient of 50 running against 10. Logically it makes sense that the coefficient of 50 is seriously underperforming. As one mistake would have a fitness of 50 and two mistakes would have a fitness of 0 (assuming the base fitness was 100).

#### C1



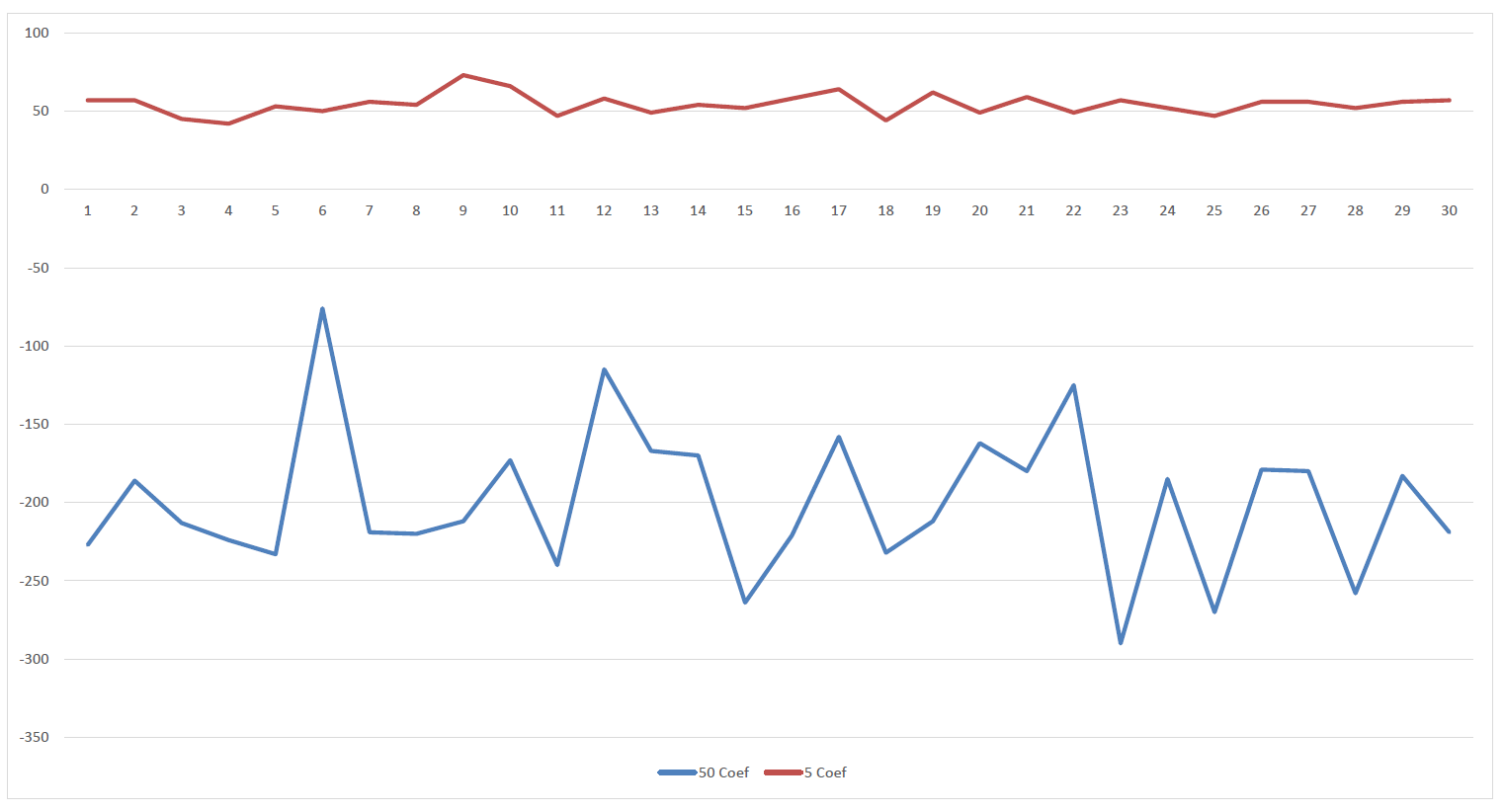
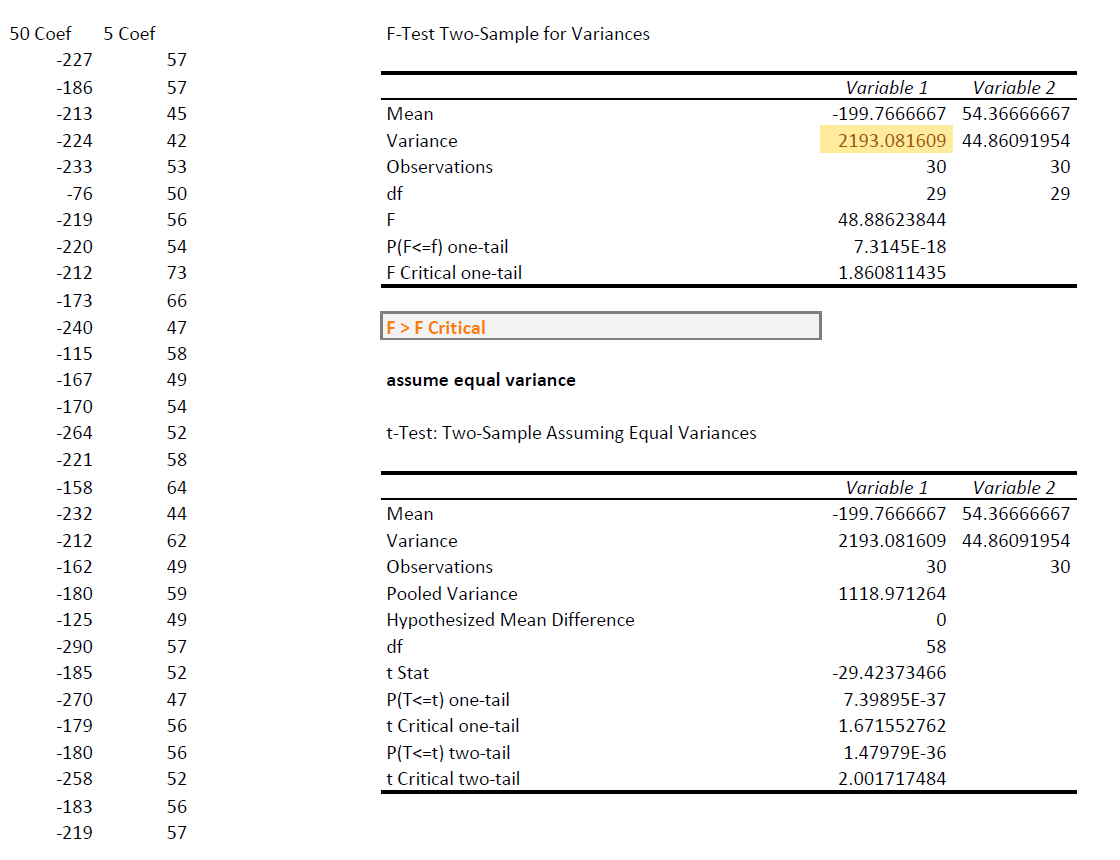
#### C2



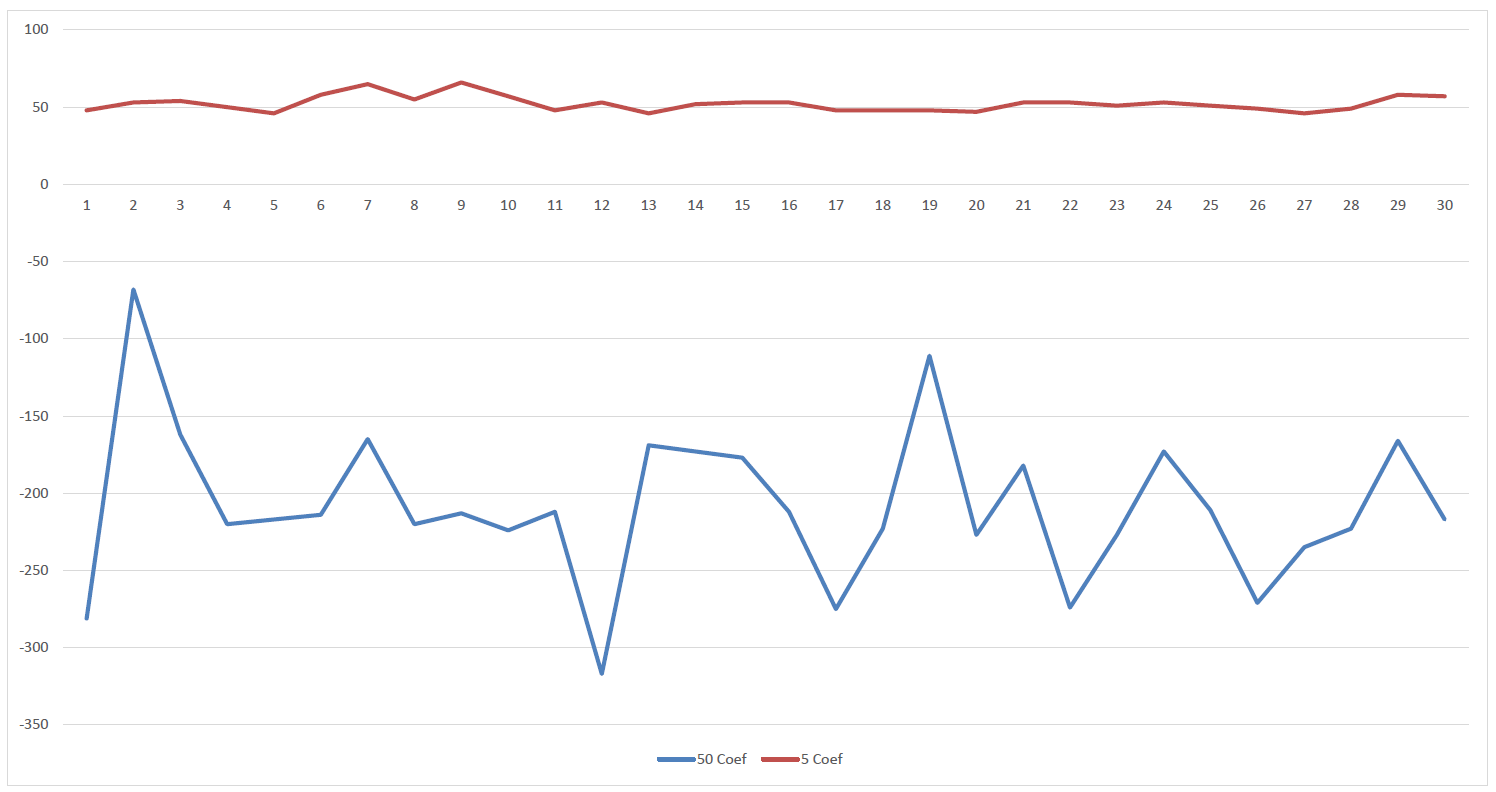
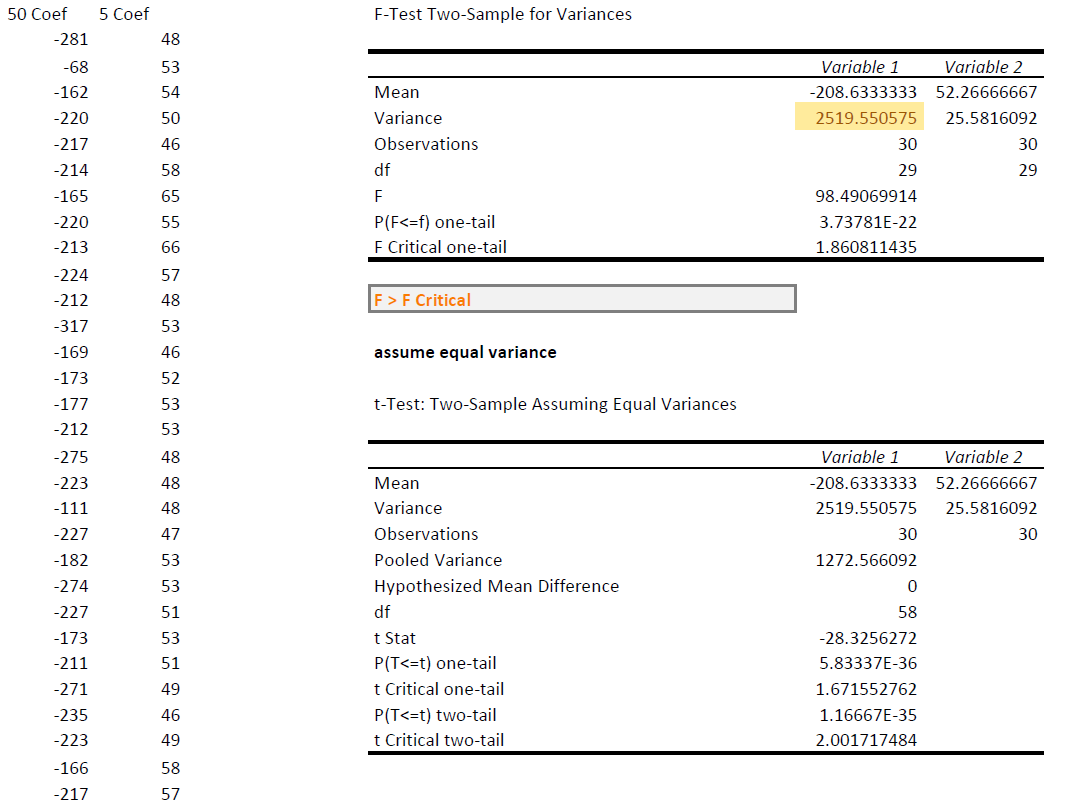
### 50 v 5

As seen earlier 5 performs the “best” so it is no surprise that it outperforms 50.

#### C1



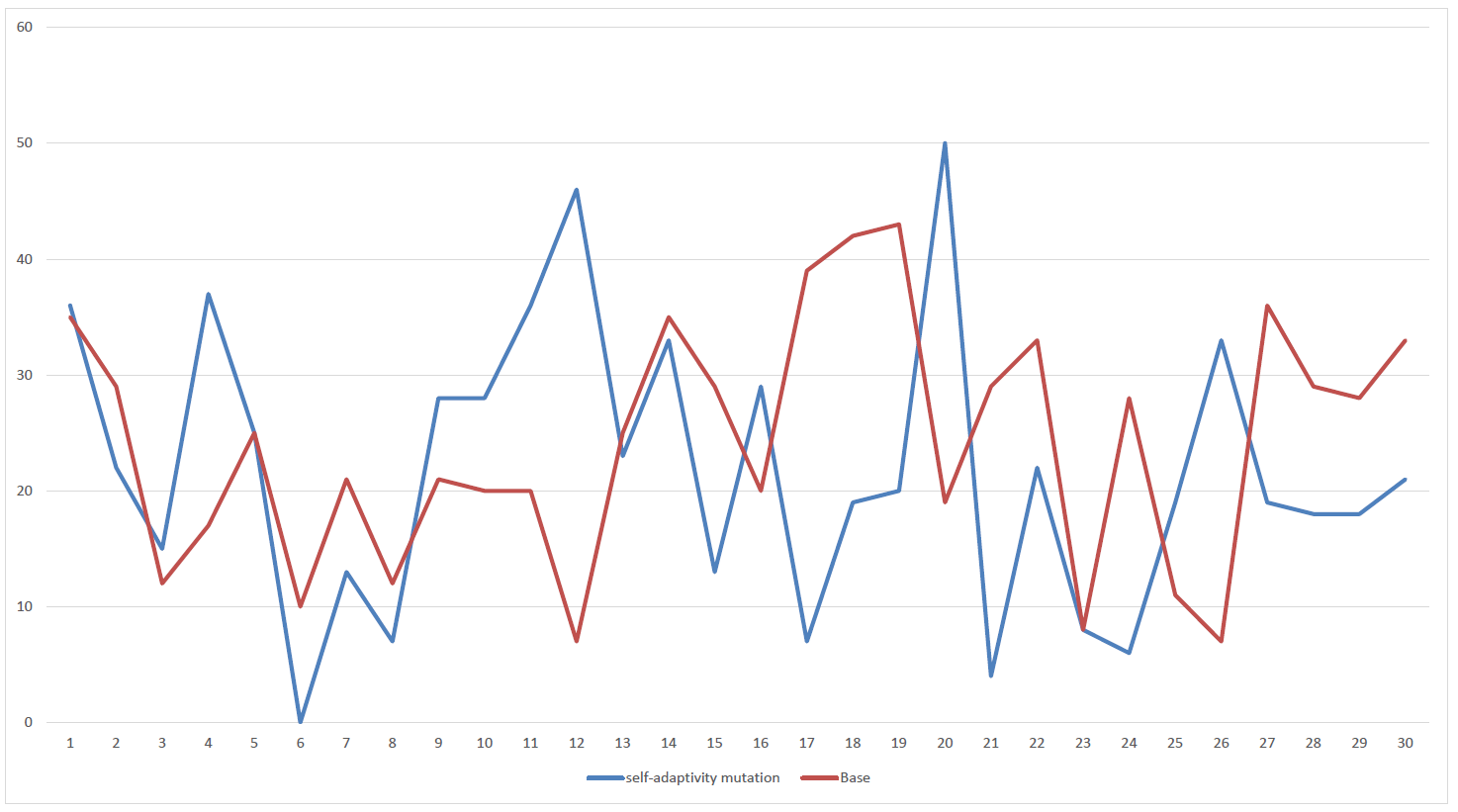
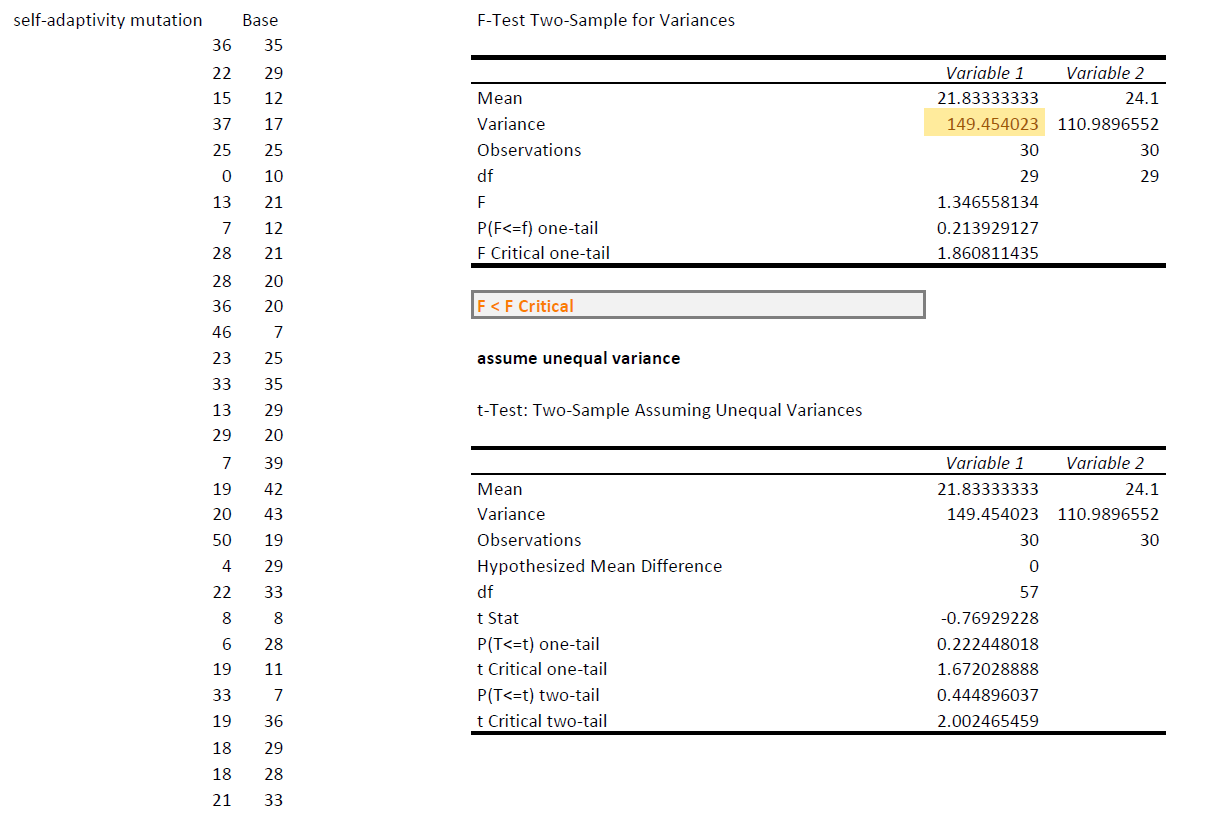
#### C2



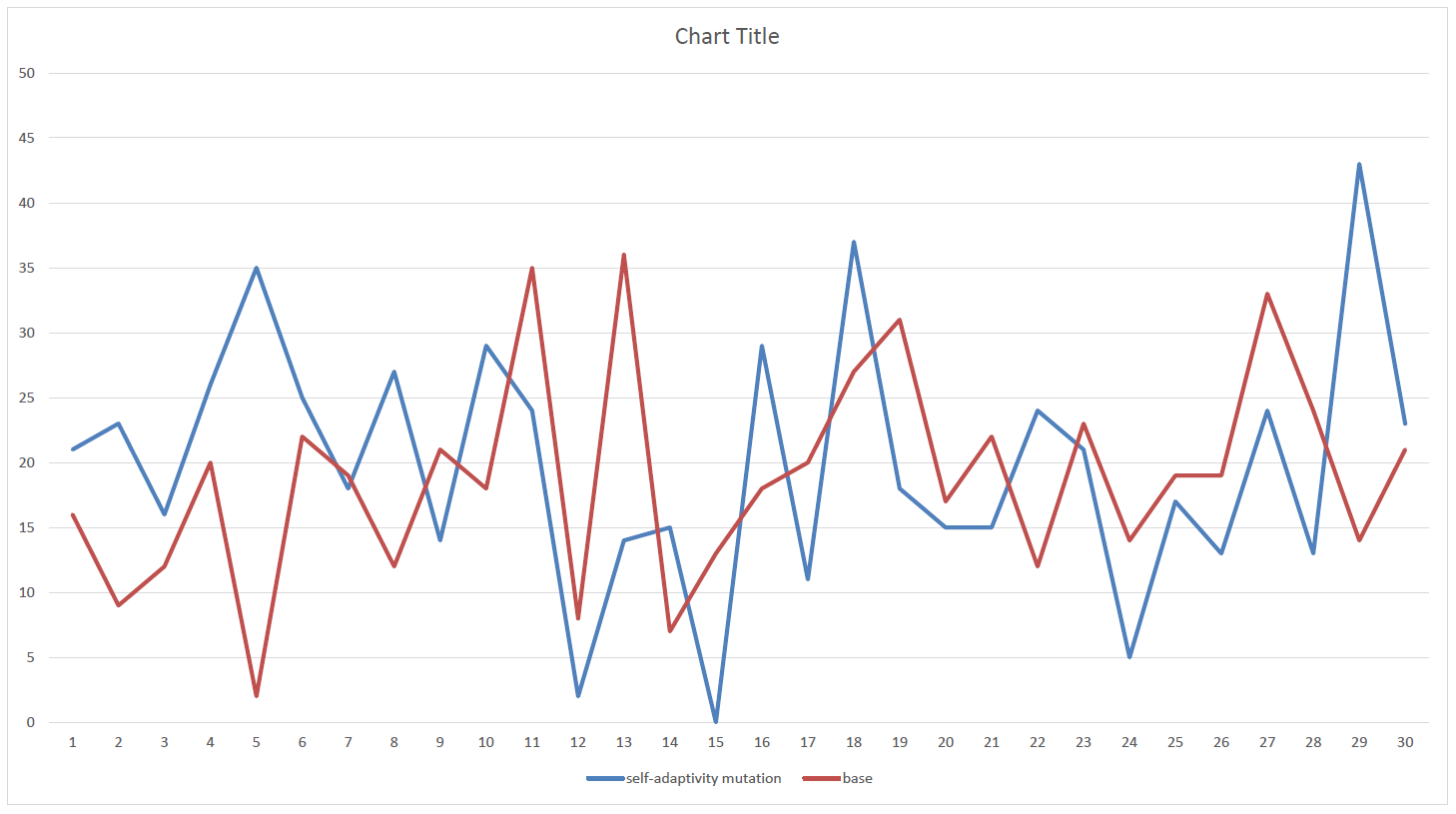
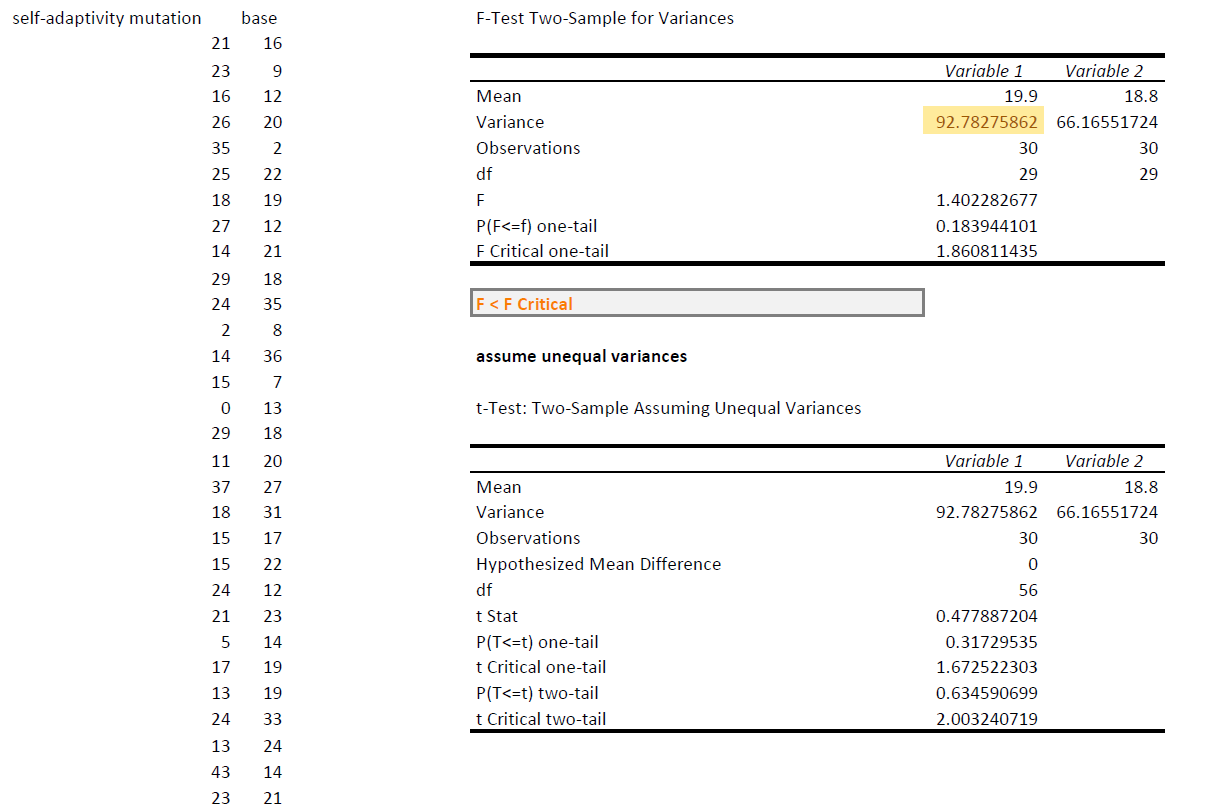
# Yellow 1: Self-adaptive Mutation Rate

A self-adaptive mutation rate, as far as I understand, is having the mutation rate being defined as a gene that can be passed between parents instead of something that is defined to be static for the entire experiment. As I used a class-based approach for my Individuals all I had to do was add a new instance variable to it. Then during recombination, mutation, and initialization I had to ensure to generate/pick/mutate/use that new gene. I did expect the self-adaptive mutation to do better but they perform very similar.

#### C1



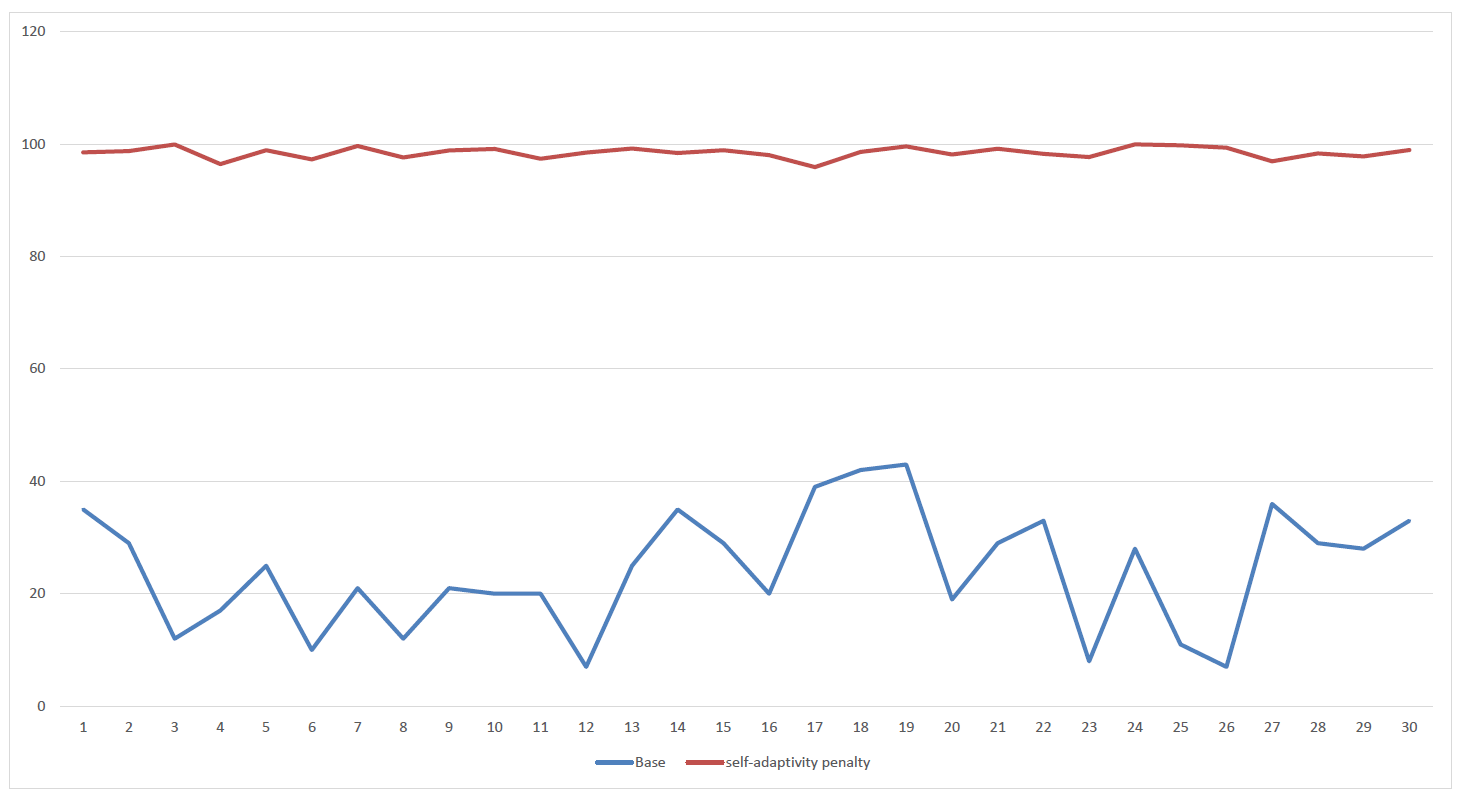
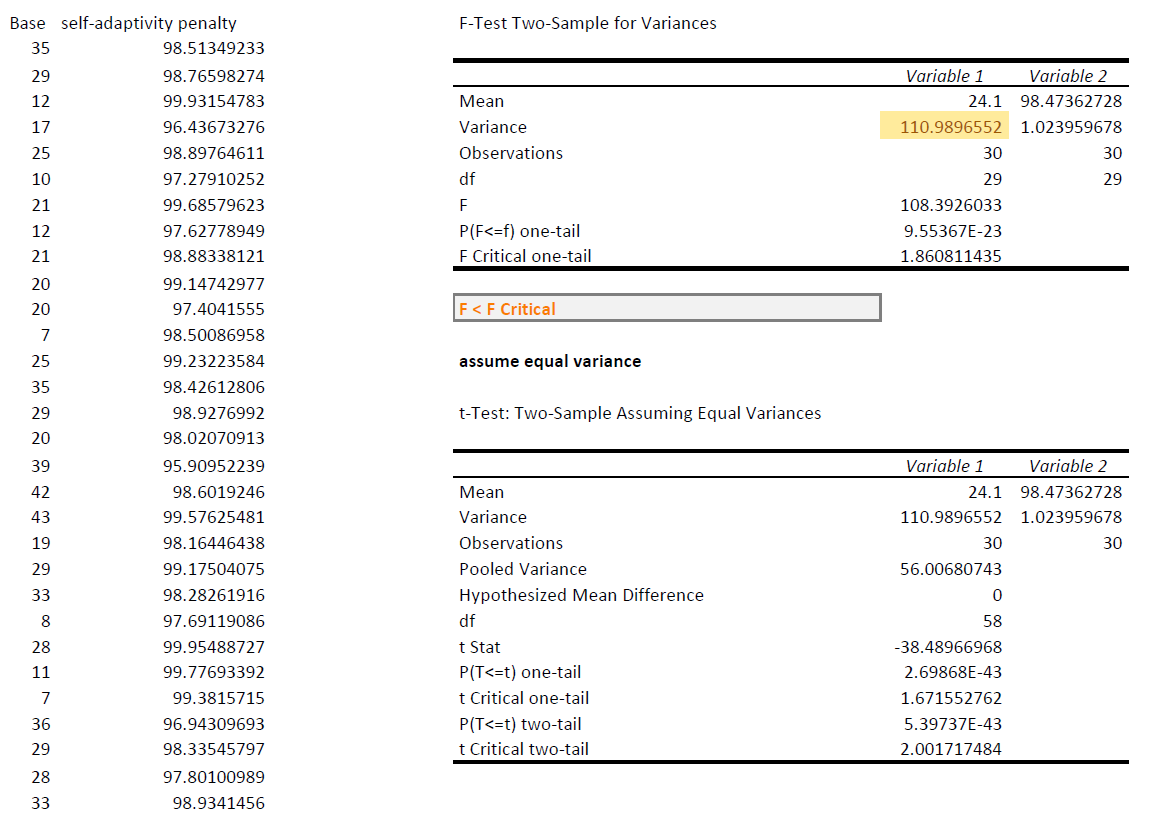
#### C2

****

# Red 2: Self-adaptive Penalty Coefficient

For the self-adaptive penalty coefficient, it was essentially the same as the self-adaptive mutation rate. A new gene was added to allow the penalty coefficient to be defined and to be evolved per individual. I do not think a self-adaptive penalty coefficient sounds like a good idea, however. Since we no longer throw away invalid solutions, invalid solutions can thrive with a low penalty coefficient as they will look, as far as the fitness algorithm is concerned, as valid as “valid” solutions. Therefore, the most efficient penalty coefficient would be 0, allowing extremely invalid solutions to have high fitness values. Since our logging only records overall and average fitness one cannot see the actual board of these individuals but I would assume that most of them are producing extremely invalid solutions; such as intersecting rays or unsatisfied black cells.

#### C1



#### C2

