COMP 6660 Fall 2020 Assignment 2C

Jordan Sosnowski
jjs0029@auburn.edu

I. Introduction

For this assignment, the author was instructed to build a competitive co-evolutionary search algorithm for Pac-Man. In the prior assignment, 2B, Pac-Man's intelligence was extended by using a genetic programming search algorithm. To build on this the current iteration has the Ghosts' utilizing a genetic programming search algorithm. The two populations will compete against each other, and in theory, get better as the generations go on. However, there can be certain situations that can cause undesirable effects which are discussed later.

The Ghosts are given the same algorithms that Pac-Man has access to. Their choices for parent selection algorithms are fitness proportional selection and over selection and for survival selection: truncation, k-tournament without replacement. Via the configuration file, one can change both λ, μ, k, mutation rates, and parsimony pressure for Pac-Man and the Ghosts separately.

This paper starts with the methodology section which explains the custom parts of the author's EA framework. Additionally, differences relative to assignment 2B are explained as to not re-explain the entire framework to the grader. Following the methodology section is a section explaining the author's experiments and how they were setup and ran. This is useful so the graders can produce the same results as the author if the need arises. The results of these experiments are located in the result section. Here all the experiment's data is listed in graphical formats along with statistical analysis of the data sets. After the results are stated there will be a discussion that contains the author's understanding of the data. Along with this is a brief discussion of the CIAO plots produced by the experiments. Following the discussion section will be the conclusion which will wrap up the paper.

II. Methodology

A. Changes

Not much had to be changed to extend the existing framework to give the Ghosts genetic programming aspects. The function that required the most changes was `_genetic_programming()`. This function is the core of the GP framework, here it loops through each run, creates the initial population, performs the children's creations, individual selection, etc. However, in 2B it is assumed that there was only one population, the Pac-Man population. An example of this change is in the function where parent selection and recombination occurs. One can see that `parent_selection()` needs to be called with both a Pac-Man population and the Ghost population.

```
# parent selection
pacman_parents = self.parent_selection(
    pacman_population, gpac.PACMAN)
ghost_parents = self.parent_selection(
    ghost_population, gpac.GHOST)

# recombination and mutation
pacman_children = self.child_selection(
    pacman_parents, gpac.PACMAN)
ghost_children = self.child_selection(ghost_parents,
     gpac.GHOST)
```

B. Process

a) Creating the Population: For each run the first function the program calls is `_create_initial_populations()`. This function handles creating the initial population for both the Ghosts and Pac-Man. `_create_initial_populations()` is just a wrapper for `_create_initial_population()` which creates a specific population; either a Ghost or Pac-Man.

```
def _create_initial_population(self, unit):
    population = []
    for _ in range(self.pacman_parents):
        # create tree
        if random.randint(0, 1):
            tree_type = 'full'
        else:
            tree_type = 'grow'
        head = node.Node(tree_type=tree_type,
    depth_limit=self.max_depth, unit=unit)
        head.grow()
        population.append(head)
    return population

def _create_initial_populations(self):
    pops = []
    pops.append(self._create_initial_population(gpac
    .PACMAN))
    pops.append(self._create_initial_population(gpac
    .GHOST))
    # play game and return individual
    return self.create_individuals(pops[0], pops[1])
```

After the initial set of nodes are created for each population it is time to pair the controllers against one another, evaluate them, and get their fitnesses. This is handled by `_create_individuals()` which maps each Pac-Man controller to a Ghost controller and plays a game to completion. The return of `calculate_fitness()` is the fitness of the Pac-Man controller and the Ghost.

```
def create_individuals(self, pacman_population,
    ghost_population):
    pacman_ind = []
    ghost_ind = []
    population = list(zip(pacman_population,
    ghost_population))
    with multiprocessing.Pool() as pool:
        for i, res in enumerate(pool.imap_unordered(
    self.calculate_fitness, population)):
            pacman_ind.append(res[0])
            ghost_ind.append(res[1])
    return pacman_ind, ghost_ind
```

b) Playing Pac-Man: `calculate_fitness()` is fairly similar to the last assignment except for this iteration each turn requires the Ghost controller to be passed along with the Pac-Man controller. Additionally, the Ghost's fitness has to be calculated alongside Pac-Man's fitness. The Ghost's fitness is calculated in the following way: $\frac{1}{s-p+1} + b$ where s is the final score of the game, p is the parsimony penalty of the Ghost tree, and b is the bonus the Ghost get if they catch Pac-Man. It is important to note that the assignment's specifications mentioned that the bonus should be relative to how much time was left in the game but the author did not realize that until the day before the due date and there was not enough time to rerun the experiments and perform additional analysis.

To handle each turn of the game the function `_turn()` is used which is similar to 2B. Except now it takes two parameters, the first being the Pac-Man controller and the second being the Ghost controller. Instead of randomly picking each Ghosts' move like in 2B the controller evaluates the Ghost's controller tree to determine the optimal move in the same way that Pac-Man operates. Although similar to the Pac-Man controller the Ghost controller has different sensors. The required sensors for the Ghost controllers are Manhattan distance to Pac-Man and Manhattan distance to the nearest Ghost. However, the author decided to implement a third sensor after discussing it with the TAs. The last sensor implemented was the shortest path to Pac-Man which seems to massively improve the Ghosts performance especially when paired with the bonus the Ghosts can get for catching Pac-Man. A new sensor was added to the existing sensors for Pac-Man's controller as well, this sensor is the shortest path to the nearest Ghost. The shortest path code used by the authored was a modified version found online [2].

c) Evolution Loop: After the fitness is generated for each Individual execution returns to the main block; `_genetic_programming()`. Following this, the best individual for each population is added to a running list of individuals; which is used later for logging purposes. After that, the evolution loop happens. Parents are selected from each population and using those parents children are created from each population. The parent selection algorithms have only changed to ensure that both Pac-Man and the Ghost populations use their respective algorithms specified in the configuration file and to ensure that if they have differing parent sizes that those are used correctly. However, at this point, they are not evaluated as they will be evaluated together with the existing population. Since the opponent is evolving with Pac-Man you cannot keep the existing individuals like in 2B; you will need to reevaluate them. Therefore when the children are returned `reevaluate()` is called passing the children and the existing populations. Since the existing population already exists as Individuals the author pulls out the trees from them and adds it to a list of individuals and recalls `create_individuals`. This is done just to be more efficient and to reuse code.

```
    def reevaluate(self, pacman_population,
    ghost_population, pacman_children,
    ghost_children):
        pacman_nodes = [pacman.head_node for pacman
    in pacman_population]
        ghost_nodes = [ghost.head_node for ghost in
    ghost_population]

        pacman_nodes += pacman_children
        ghost_nodes += ghost_children

        return self.create_individuals(pacman_nodes,
     ghost_nodes)
```

Once the individuals in both populations are re-evaluated survival selection occurs on both populations. Similar changes to the parent selection algorithms were made to the survival selection ones. After survival occurs some logging variables are updated and then the termination criteria are checked. If the loop should continue the process described above happens again until the loop needs to end. Once the evolution loop ends the process moves forward to the next run and runs the whole process again just like in all the prior assignments. After all the runs are completed the logging functions are run. In this assignment, the author was asked to log both Pac-Man's solution and the Ghosts' solution. Additionally, an exhibition game needs to be played against the best Pac-Man controller and the best Ghost controller with the world file of this game should be logged.

d) CIAO Plots: An additional requirement for graduate students was to investigate co-evolutionary cycling. Cycling, along with disengagement, and mediocre stability are problems that sometimes occur in co-evolution. With *cycling* both 'populations circle around to previous solutions over and over again' [1]. This can be seen in CIAO plots which are plots that compare the current individual against ancestral opponents. With a CIAO plot cycling appears as strips of black and white which shows that one population dominating one and then swapping off with the other over and over. *Disengagement* is when 'one side will have consistently high fitness while the other is low' [1] and visually presents itself with either large portions of black or white. Finally, *mediocre stability* is when 'neither side changes much, and neither seems to be much stronger' [1]. On a CIAO plot, the graph can present as being gray or look like static; random selections of black and white pixels but no real growth.

III. Experimental Setup

Three different experiments were run for this assignment.

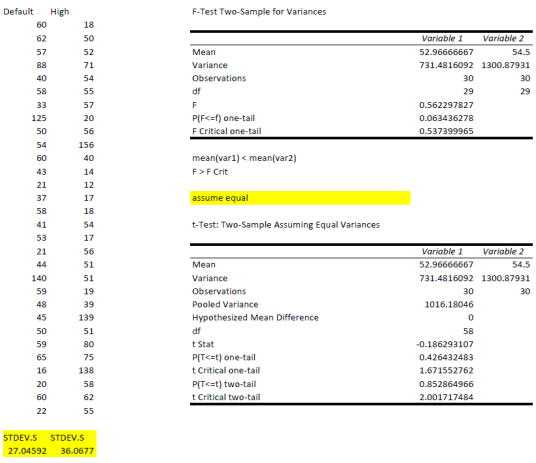
a) Base experiment: This configuration had a pill density of 50%, a fruit spawn probability of 1%, a fruit score of 10, and a time multiplier of 2 for the Pac-Man game. Additionally, for the GP configuration parameters $\mu_{Pac-Man}, \mu_{Ghost}$ were both set to 200 and $\lambda_{Pac-Man}, \lambda_{Ghost}$ were set to 100. Both the Ghost population and the Pac-Man population used truncation for survival selection and fitness proportional selection for parent selection. $p_{Pac-Man}, p_{Ghost}$ were both set to 100% and the number of evals till termination is set to 2000.

b) High Pill Density: For this configuration, the only change was to the pill density which was 80%. As seen in 2B a higher pill density usually correlates to higher fitness. This does not entirely mean a smarter Pac-Man as there are more pills for him to accidentally step on the higher the density is. Regardless, the author thought it would be interesting to look at.

c) Max Pill Density: Using the same logic here we set the pill density to 100% to see if it further improves anything.

IV. Results

One can see the statistical results of the experiments in Figs. 1, 2, and 3. The graphs can be seen in Figs. 4, 5, and 6. To run these experiments one simply needs to run either the `run.sh` shell script or the Python driver located in the source directory called `driver.py`. The shell script just wraps the Python script and is there simply because it is required in the specifications.



Default	High
60	18
62	50
57	52
88	71
40	54
58	55
33	57
125	20
50	56
54	156
60	40
43	14
21	12
37	17
58	18
41	54
53	17
21	56
44	51
140	51
59	19
48	39
45	139
50	51
59	80
65	75
16	138
20	58
60	62
22	55

STDEV.S	STDEV.S
27.04592	36.0677

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	52.96666667	54.5
Variance	731.4816092	1300.87931
Observations	30	30
df	29	29
F	0.562297827	
P(F<=f) one-tail	0.063436278	
F Critical one-tail	0.537399965	

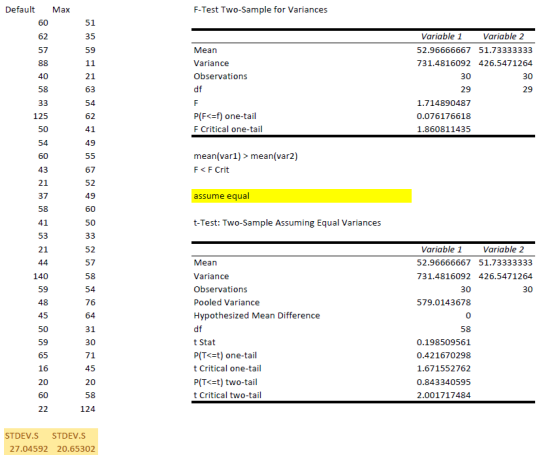
mean(var1) < mean(var2)
F > F Crit

assume equal

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	52.96666667	54.5
Variance	731.4816092	1300.87931
Observations	30	30
Pooled Variance	1016.18046	
Hypothesized Mean Difference	0	
df	58	
t Stat	-0.186293107	
P(T<=t) one-tail	0.426432483	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.852864966	
t Critical two-tail	2.001717484	

Fig. 1: Statistical Analysis for the Default Configuration versus the High Configuration



Default	Max
60	51
62	35
57	59
88	11
40	21
58	63
33	54
125	62
50	41
54	49
60	55
43	67
21	52
37	49
58	60
41	50
53	33
21	52
44	57
140	58
59	54
48	76
45	64
50	31
59	30
65	71
16	45
20	20
60	58
22	124

STDEV.S	STDEV.S
27.04592	20.65302

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	52.96666667	51.73333333
Variance	731.4816092	426.5471264
Observations	30	30
df	29	29
F	1.714890487	
P(F<=f) one-tail	0.076176618	
F Critical one-tail	1.860811435	

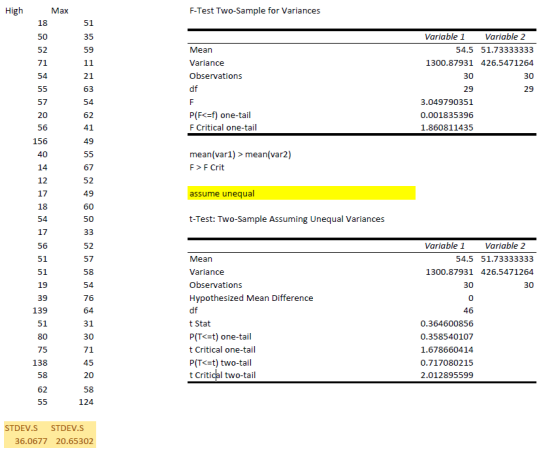
mean(var1) > mean(var2)
F < F Crit

assume equal

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	52.96666667	51.73333333
Variance	731.4816092	426.5471264
Observations	30	30
Pooled Variance	579.0143678	
Hypothesized Mean Difference	0	
df	58	
t Stat	0.198509561	
P(T<=t) one-tail	0.421670298	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.843340595	
t Critical two-tail	2.001717484	

Fig. 2: Statistical Analysis for the Default Configuration versus the Max Configuration



High	Max
18	51
50	35
52	59
71	11
54	21
55	63
57	54
20	62
56	41
156	49
40	55
14	67
12	52
17	49
18	60
54	50
17	33
56	52
51	57
51	58
19	54
39	76
139	64
51	31
80	30
75	71
138	45
58	20
62	58
55	124

STDEV.S	STDEV.S
36.0677	20.65302

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	54.5	51.73333333
Variance	1300.87931	426.5471264
Observations	30	30
df	29	29
F	3.049790351	
P(F<=f) one-tail	0.001835396	
F Critical one-tail	1.860811435	

mean(var1) > mean(var2)
F > F Crit

assume unequal

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	54.5	51.73333333
Variance	1300.87931	426.5471264
Observations	30	30
Hypothesized Mean Difference	0	
df	46	
t Stat	0.364600856	
P(T<=t) one-tail	0.358540107	
t Critical one-tail	1.678660414	
P(T<=t) two-tail	0.717080215	
t Critical two-tail	2.012895599	

Fig. 3: Statistical Analysis for the High Configuration versus the Max Configuration

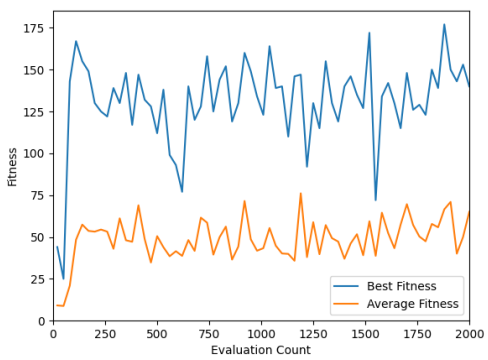


Fig. 4: Graph of Default Configuration best fitness per generation versus average fitness per generation

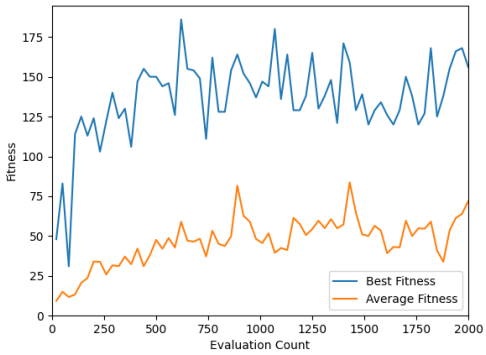


Fig. 5: Graph of High Configuration best fitness per generation versus average fitness per generation

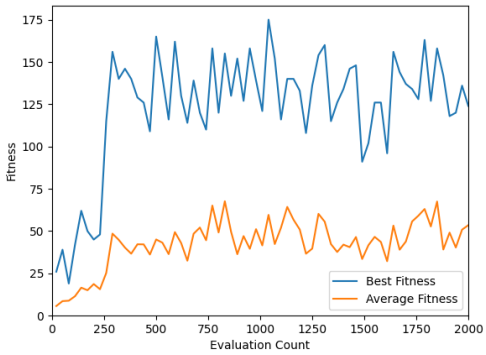


Fig. 6: Graph of Max Configuration best fitness per generation versus average fitness per generation

V. Discussion

a) Graphs: The first thing that stood out to the author was the difference in the graphs compared to assignment 2B. In 2B the graphs only increased never decreased, however, in this assignment they would fluctuate. The author's assumption behind this is due to the re-evaluation portion of the code. Where before an individual's fitness would never change and due to the elitist survival selection algorithm that was chosen to be used fitness would only increase. Now an individual's fitness can decrease during re-evaluation and cause graphs that do not solely increase. Graphs comparing all the configurations best fitnesses and one comparing the averages can be seen in the appendix in Figs. 13 and 14.

b) Statistical Analysis: Changing the pill density, unlike in 2B did not seem to affect the results that much as all the graphs are relatively similar. The only difference the author sees in the graphs, at least, is that the default and max configuration achieved a higher fitness quicker than the high configuration, however, this could completely occur due to the stochastic nature of the algorithms. The max configuration file seems to provide the lowest variance as seen in Figs. 2 and 3. All of the configurations seem to have a similar mean hovering around 53 as seen in Figs. 1, 2, 3.

c) CIAO Plots: As discussed earlier CIAO plots are a way to see how individuals perform against ancestral opponents which can be used to see how well individuals are retaining prior knowledge. A 'good' CIAO plot has a gradient that darkens in the top left corner and lightens as it spreads out. There was only one plot that the author saw that semi-fit that description, except it the opposite. It instead was light in the top left and dark in the other corners which can be seen in Fig. 7.

Fig. 7: Example of odd CIAO Plot

Most of the CIAO plots produced throughout all the configurations tended to show *mediocre stability*; which as discussed earlier is just when the two populations no longer evolve. An example of this plot can be seen in Fig. 8 and more examples can be seen in the appendix in Fig. 12.

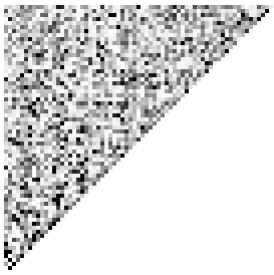


Fig. 8: Example of Mediocre Stability

Another issue co-evolution algorithms can exhibit is *cycling*; which did not happen much in these experiments but did happen a few times. It happened more in the high configuration, which had a pill density of 80%. The author is unsure as to why it occur ed in the high configuration more so than the max configuration which had a 100% pill density. Regardless, an example can be seen in Fig. 9 with more examples in the appendix in Fig. 10.

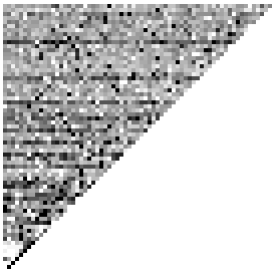


Fig. 9: Example of Cycling

VI. Conclusion

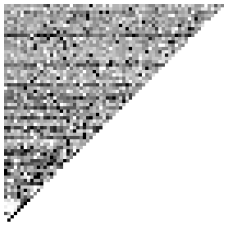
Utilizing competitive co-evaluation one can create smart and more diverse solutions to problems. Before when it was a GP controller against a random agent, not a lot of interesting things occurred. However, when both agents can learn and play off of each other interesting strategies can occur. For instance, in one game the author saw a Ghost evolve with a tree that simply added together with the shortest path to Pac-Man to the shortest path to Pac-Man. Utilizing this the Ghost ensured that they would always move closer to Pac-Man acting like a homing missing and eating him. However, this strategy only seemed to work against 'dumb' Pac-Man, as the author saw one game where the Ghost utilized this strategy, however, Pac-Man simply kept moving away from the Ghost and kept consuming pills so he was able to finish the game with the Ghost one tile away from him. The issue with that Ghost strategy is that *all* of the Ghosts take the same move, however, if they had split off they could have potentially boxed Pac-Man in. Regardless it was very interesting to watch.

There were some areas the author wished to look into but ran out of time and motivation (mainly due to the end of the semester burnout). An example of an area that the author wished to implement was to have paired the controllers up multiple times instead of doing a 1-1 evaluation as doing 1-1 under samples both populations. For instance, a Pac-Man controller that is just adequate could get paired against a really bad Ghost controller. In this pairing, the Pac-Man controller could achieve a high score simply because the Ghost controller is not intelligent enough to do anything. If this happens then the Pac-Man controller would achieve high fitness even though it is not that great. Another area that needs improvement is the optimization of the codebase. Run-time was a reoccurring issue throughout the entire project and the author understands that it is usually an issue with AI-like projects. However, running a debugger and waiting for the program to execute for 30 minutes just for it to break was unfortunate. This is not of course the fault of the TA's or the professor but the student himself.

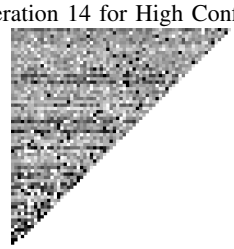
The projects were all very interesting throughout the semester and the author very much enjoyed working on them. Towards the end, some slacking did occur and they may have not been in the best state. The TA's also did a great job of answering any pending questions on the projects and for just general EC stuff!

VII. Appendix

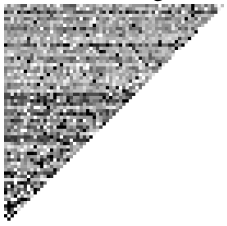
A. CIAO Plots



(a) Generation 14 for High Configuration

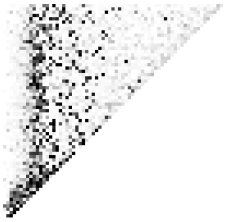


(b) Generation 16 for High Configuration

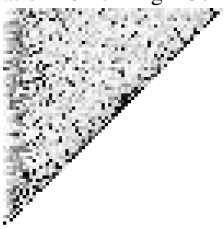


(c) Generation 7 for High Configuration

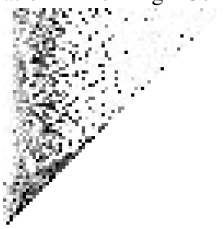
Fig. 10: Examples of Cycling



(a) Generation 13 for High Configuration

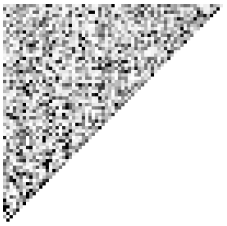


(b) Generation 22 for High Configuration

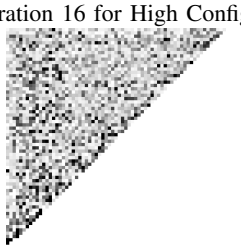


(c) Generation 3 for High Configuration

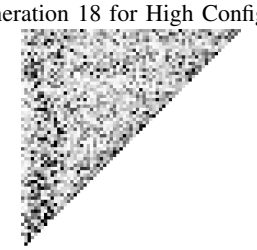
Fig. 11: Examples of Disengagement



(a) Generation 16 for High Configuration



(b) Generation 18 for High Configuration



(c) Generation 2 for High Configuration

Fig. 12: Examples of Mediocre Stability

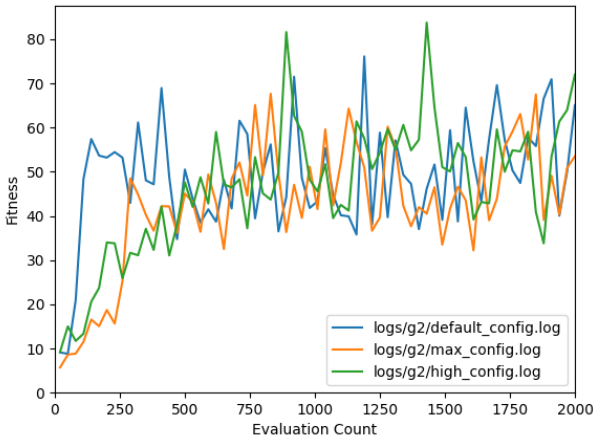


Fig. 13: Graph of all Configurations comparing their average fitness's

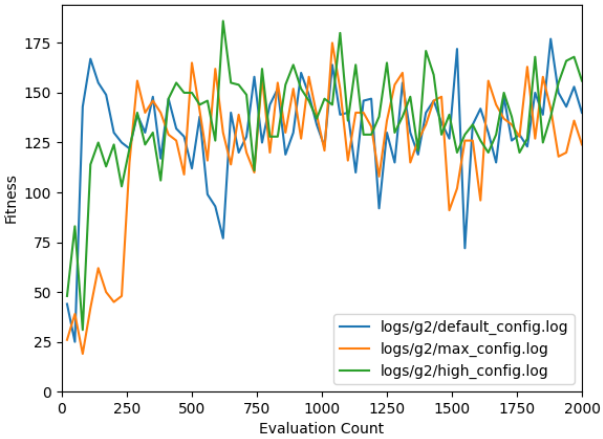


Fig. 14: Graph of all Configurations comparing their best fitness's

References

[1] George Rush and Daniel Tauritz. CIAO Plots, 2020.

[2] stutipathak31jan. Shortest path in a Binary Maze, Oct 2020. Avaialable at https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/.