CMPT 417/827 X100

Group Project - Final Report

Genetic Programming for Predicting COVID-19 Infection Rates

By

Jovanovic, Marko

Kuppers, Lukas

Bennett, Matthew

To

Professor Hang Ma (Instructor)

Table of Contents

Introduction

Algorithms that imitate natural processes create an interesting point of view for computing scientists because the algorithms in question must be able to adapt and optimise by mimicking both mutation and natural selection. The aim of this project is to obtain a characteristic function similar to what is represented by a given input dataset, so as to gain an understanding of the underlying mechanics driving the aforementioned input.

Each round of evolution, defined by depth in the tree, competes with a cost function. One of our primary tasks was to ensure that successful strategies were passed on to the next generation (or depth in the tree) of the evolutionary processes. With many tasks requiring large search spaces, evolutionary algorithms pave the groundwork for effective search strategies to find the most optimal solutions. This group explores genetic algorithms, simulating the process of evolution, natural selection and randomness of mutation to find and optimise parameters of an input-output map.

As nodes of a tree are generated, they are evaluated and given a score based on the cost function. The better a node scores, the more likely its function, or traits thereof, will be passed down to the next level of the tree known as that node's children. Mathematically, the problem boils down to three main problems. Firstly, the algorithm must generate nodes with random functions. Secondly, the algorithm must generate a reasonable means of representing mutation and crossover behaviours seen in nature.. Lastly, the algorithm should create an output characteristic function that is similar to the input dataset.

$$J = \alpha \sum_{t=0}^{t_f} [y(t) - h(t)]^2 + \beta L^2$$

Equation 1: Example Cost Function

An example of our cost function is shown in equation 1, where J is the cost for a given hypothesis function h(t) over the time period 0 to tf, y(t) is the target input curve, L is the length parameter of a given function hypothesis, and alpha and beta are tunable

parameters. Other variables implemented in this project include the number of initial candidates, how long a typical crossover branch would be, and how heavily the algorithm should penalise function length.

$$P_{crossover} + P_{replication} = 1 - P_{mutation}$$

Equation 2: Probabilities of Crossover, Mutation and Replication sum to 1

The genetic operations of crossover, replication and mutation are performed based on a probabilistic chance of each function having this operation performed on it. Equation 2 is our rule of how the ratio of the probabilities is guaranteed to sum up to one so long as the sum of the probabilities of crossover and replication are less than one. In code, we enforce this rule by having our crossover and replication probabilities as function input parameters to our *evolve* function, and the probability of mutation is whatever is left over.

Using the above formulae, this project aims to use machine learning to test hypotheses against input datasets to allow a better understanding of the forces at work behind mother nature.

Implementation Details and Decisions

Overview:

Three main components were required in order to implement genetic programming. Firstly, a dynamic representation for functions was required - the group decided to use a recursive tree structure, where operators are represented by internal nodes, and variables/constants are represented by leaf nodes. A high level implementation of the genetic algorithm was also required, utilising the specific low level details of genetic programming. Finally, utilities for reading and structuring data were required, in order to apply the genetic programming algorithm to a read dataset.

1. Implement Function Trees and Random Generation (function_tree.py)

As functions are represented as a recursive tree, we require some data structure to represent each node. We settled on the following object to represent nodes:

```
{
    name: <value name or operator name>,
    children: <list of children nodes>,
    value: <constant value or null if node is a variable or operator>
}
```

This flexible structure allows nodes to represent arithmetic operators (which will have children nodes), constant values (which will have the value field set), or a variable, whose value may change over every observation. Using instances of these nodes, we are able to assemble virtually any arithmetic expression using operators we can specify externally (in operators.py).

The function 'get_random_func()' in function_tree.py returns the root of a randomly generated function, given possible variables and a maximum depth. The function is recursive, generating a random node, and then recursively generating its children. In each function call, the node generated is randomly chosen to be either a constant, variable, or operator. An example of such a randomly generated tree can be seen in figure 4.1.1. The pseudocode for the random generation function can be seen below:

```
get_random_func(variables, max_depth):
       Random_choice = random(variable, constant, operator)
       If random_choice = variable or max_depth = 1
              Return variable node using one variable from 'variables'
       If random_choice = constant
              Return constant node using randomly generated constant value
       Else
              Operator = randomly choose operator
              Children = []
              For i = 1 to operator.number_of_arguments
                     Children[i] = get_random_func(variables, max_depth - 1)
              Return operator node with children list as arguments
```

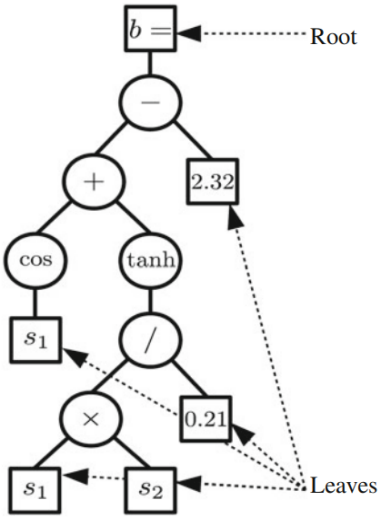


Figure 4.1.1: A visual representation of the function tree data structure. Arithmetic operators are internal nodes, and constants or variables are leaf nodes. In this example, s_i and s_2 are variables.

The following arithmetic operators were used for the generation of new functions: Addition, Subtraction, Multiplication, Division, Logarithm, Natural Exponential, Cosine, and Heaviside Step Function. Since it was possible to generate undefined functions (i.e. division by zero, or logarithm of a negative number) we decided to create edge cases in which if an inadmissible function node were generated, it would evaluate to a similarly performing value. For example in the case of division, if the denominator was found to be less than 10^{-10}, then the denominator would instead be set to exactly 10^{-10}.

2. Implement a cost function

As with any genetic algorithm, some method by which we can compare the performance of individuals in a generation is required. In the case of genetic programming, (and regression), we seek functions that most closely approximate the input dataset. Simply then, our cost function can be the sum of squared differences (SSD) between the fitted values from a function, and the actual data. Clearly, a function with a smaller SSD fits the data better than a function with a larger SSD. In favour of parsimonious functions, we also included a length penalty, that would give larger functions a proportionally larger cost. The cost function can be seen below:

$$Cost(f(X)) = \frac{1}{N} * \sum_{i=1}^{N} (y_i - f(X_i))^2 + Length\ Penalty$$

In reality, length was penalised by converting the function into a LISP string, and then using a scaled version of the string length as the penalty quantity. LISP strings are simply an alternative method of representing functions using prefix notation. For example, the LISP representation of *x + log(y * z)* is *(+ x (log (* y z)))*. LISP strings were preferred, as they are easily obtained from the function tree structure.

3. Implement selection, mutation and crossover (genetic_operators.py)

In order to evolve the function population towards lower cost function values, variability must be introduced in each generation. Genetic algorithms introduce variability by randomly mutating some chosen individuals from the population. Genetic programming uses four different mutation operators, which were implemented in

genetic_operators.py. The mutations used are unique to genetic programming, as they specifically act on the function tree representation of the individuals in the population.

Mutation 1: Cut and grow

The cut and grow mutation randomly selects a subtree in the function, and then replaces this subtree with a newly generated function. Cut and grow generates the new subtree with the same random generation function described previously, adjusting the max depth parameter to ensure that the newly mutated function is not too large. A visual representation of cut and grow is seen in figure 4.3.1, and the pseudocode is shown below:

```
Cut_and_grow(root, variables, max_depth):
    If root is a leaf node, or random(0, 1) < cut_probability
        return get_random_func(variables)           //cut at current node
    rand_child = random choice from root[children]
    rand_child = cut_and_grow(rand_child, variables, max_depth -1)
    return root
```

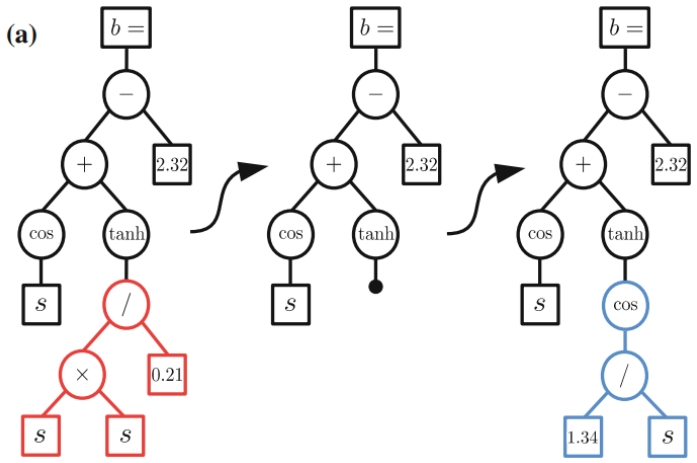


Figure 4.3.1: Visual representation of cut and grow mutation

Mutation 2: Shrink

The shrink mutation randomly selects a subtree in the function tree, and replaces it with a randomly chosen constant value. The method of randomly choosing the subtree is the same used in cut and grow. Similarly, the method of randomly generating a constant is

the same as in the random function generation routine. A visual representation of shrink is seen in figure 4.3.2, and the pseudocode is shown below:

```
Shrink(root):
	If root is a leaf node, return root
	If random(0, 1) < shrink_probability
		Return random constant node
	Rand_child = random choice from root[children]
	Rand_child = shrink(rand_child)
	 Return root
```

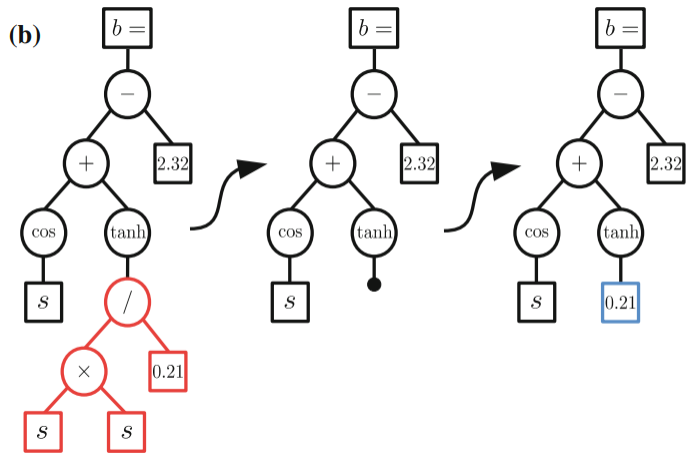


Figure 4.3.2: Visual representation of shrink mutation

Mutation 3: Hoist

The hoist mutation randomly selects a subtree (the same way as before) and makes this subtree the new function tree (discarding all other parts of the tree). A visual representation of hoist can be seen in figure 4.3.3, and its pseudocode is shown below:

```
Hoist(root):
	If root is a leaf node, or random(0, 1) < hoist_probability:
		Return root
	Rand_child = random choice of root[children]
	Return hoist(rand_child)
```

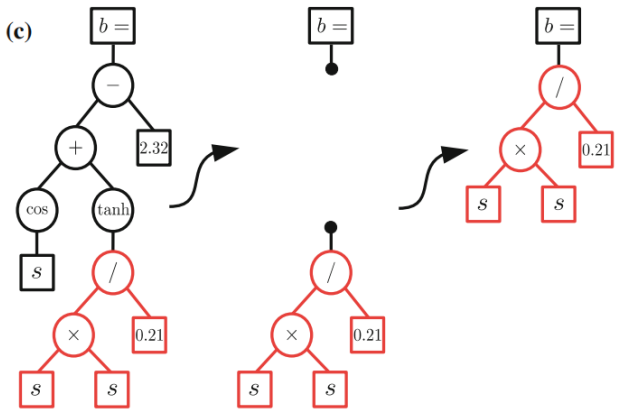


Figure 4.3.3: Visual representation of hoist mutation

Mutation 4: Reparameterization

The reparameterization mutation gives a random chance for each constant leaf node in the tree to take on a new random value. A visual representation of reparameterization can be seen in figure 4.3.4, and its pseudocode is shown below:

```
Reparameterization(root):
      If root is a constant leaf node and random(0, 1) < reparam_propbability
            Root[value] = random_constant()
            Return root
      For each child in root[children]
            Child = reparameterization(child)
      Return root
```

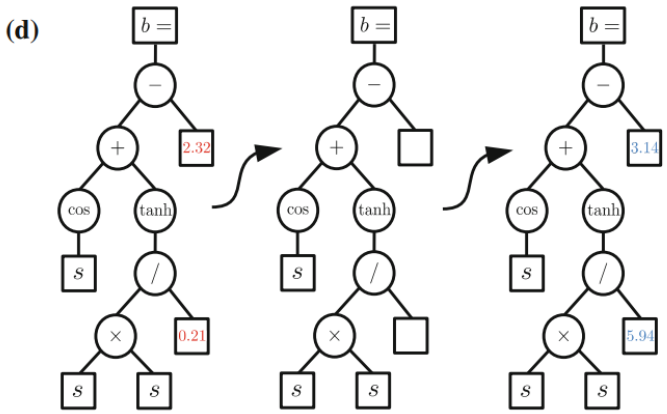


Figure 4.3.4: Visual representation of reparameterization mutation

Combining Individuals: Crossover

In addition to the mutation operations, we also wish to periodically combine the features of two well performing individuals. In general, this process is known as crossover, which takes two input individuals, producing two output individuals which are combinations of the inputs. For genetic programming, the crossover function randomly selects subtrees in each input function, and then produces the output functions by swapping the subtrees. A visual representation of crossover can be seen in figure 4.3.5, and its pseudocode is shown below:

```
Crossover(rootA, rootB):
       subtreeA = select random subtree in rootA
       Subtree B = select random subtree in rootB
       Temp = subtreeA
       subtreeA = subtreeB
       subtreeB = temp
       Return (rootA, rootB)
```

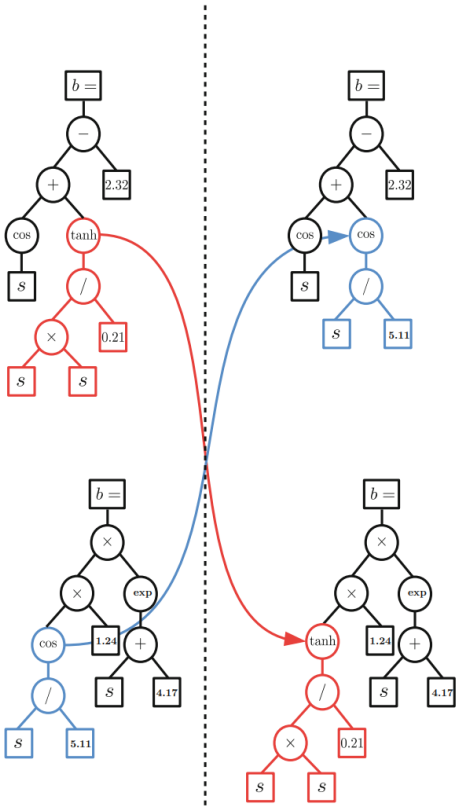


Figure 4.3.5: A visual representation of the crossover operation. The red and blue subtrees are randomly chosen, and swapped, producing two new subtrees.

4. Implementing the Generation Class and Methods (generation_functions.py)

The prototype function trees were batched into a generation class object, given generation size, function variables, and a target values array as arguments. This object would importantly consist of a list of tuples that stored each of the functions. The tuples in this list were of the form of (fitness, unique insertion ID, function tree) so as to be able to sort the list and get an ordered set of the function trees from most fit (low cost) to least fit (high cost). A unique ID was used to be a tiebreaker between functions of equal fitness. This class would also contain methods to perform the evolution of these generations, along with the corresponding helper methods that allowed them to do so. We shall elaborate on the pseudo-code of the important non-helper functions.

Getting the Best Candidate

In order to extract the lowest-cost or highest-fitness candidate function from a given generation, we can simply return the first item in the sorted function list. This is due to the list being stored as tuples sortable by score. The pseudocode is below:

```
get_best_candidate(self):
       Return sorted(self.functions list)[0]
```

Evolving the Generations

Evolution of the generations was performed in a batch using the *evolve* function, This, after passing the elite candidates through to the next generation, went through the previous generation and performed either replication. Crossover, or mutation operations on each of the candidates at a fixed probability ratio for each generation.

Crossovers are performed by, once a given function in a generation is selected to be crossed-over, the secondary function is selected at random from the remaining functions not yet operated on in the generation, this allows us to have a high variance in output function, and can ideally quickly traverse the state space towards a good candidate. A visual example is shown in figure 4.4.1, please note that while the textbook performs genetic operations based on fitness, we chose to opt for varying the probabilities of the genetic operations based on generation number. The goal being to start with a wide

search space early on (high mutation rate), then throttle the variety down (high replication and crossover rate) as generations increase to hone in on the local minima of cost. Performance of this decision will be discussed in the experiment results section.

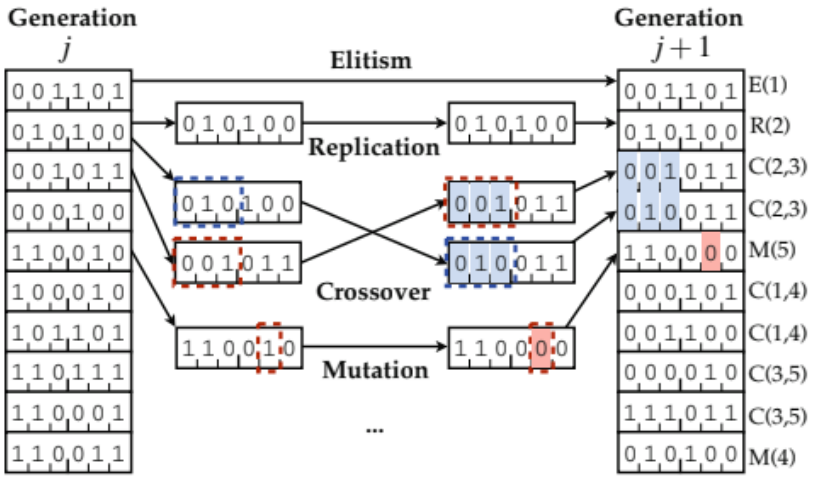


Figure 4.4.1: Visual example of how the evolution is performed between generations [3]

```
evolve(self, elitism amount, crossover probability, replication probability):
	Let prev_list be a copy of sorted(self.functions list)
	Let new_list be an empty list
	Move the first 'elitism amount' items from prev_list to new_list
	While prev_list is not empty:
		Let ratio be a random float from 0 to 1
		Let tree_A = prev_list.pop(0)
		If len(to_lisp(tree_A)) is greater than the maximum allowed length:
			self.add_to_generation(get_random_func(self.variables))
		Else if ratio < crossover_probability / 2 and len(prev_list) > 1:
			Let tree_B = prev_list.pop(0)
			Crossover tree_A and tree_B into new_tree_A and new_tree_B
			self.add_to_generation(new_list, new_tree_A)
			self.add_to_generation(new_list, new_tree_B)
		Else if ratio < (replication_probability +
crossover_probability/2):
			self.add_to_generation(new_list, tree_A)
		Else:
			self.add_to_generation(new_list, random_mutation(tree_A,
self.variables)
	self.functions list = copy of new_list
```

Using these functions, we can evolve our adjustable-length generation of candidate functions while being able to vary the ratios of the crossover, replication and mutation probabilities between generations, allowing us some control over the randomness of the evolution.

5. Test algorithm against a real dataset (datatesting.py)

We implemented testing against a real dataset at the end of the project to confirm the correctness or accuracy of the developed functions. This was done using the python file datatesting.py. The team used read functions to import a real dataset of infection rates of the COVID-19 virus from "Preliminary dataset on confirmed cases of COVID-19, Public Health Agency of Canada". This data included a large variety of information, but to keep the experiment simple and effective, the team focused solely on infections per week. Specifically, only infections per week in the first 52 weeks, since less than one percent of the supplied data was beyond this range and was therefore omitted. This was done using pseudo-code: It was important to allow the algorithm to randomly choose a mutation when one should occur, so the function random_mutation() helped with this.

```
extract_data(filename)
	count = 0
	weekly_infections =[]
	for each line f in filename:
		currentline = line.split(",") #split the entries by commas
		if count > 0: #omit the first entry because its all strings
		week_infected = currentline[2]
		if week_infected <= 52 #omit any entries for weeks above 52
			weekly_infections[week_infected] ++
		count ++
	weekly_infections.pop() #remove first index as it represents nothing
	return weekly_infections
```

After sorting the data into the number of people infected per week, a function had to be derived to explain the pattern in the data. This would be the master function to test our algorithm against. To accomplish this task, the team would score all the functions of a generation and choose the function with the lowest score to be the best candidate. This is another type of heuristic used in this project. Two helper functions were created to solve this problem. Firstly, the method to test a single function must be scored. The

score ‘total’ is calculated using regression, therefore a function with a lower score performs better.

```
Score_a_function(infections, root, variables)
	i = 0
	total = 0
	While i < 52:
		Variables[infections:weeklyinfections[i]]
		total += (infections[i] - evaluate(root,variables))^2) / 52
		i ++
	return total
```

The following and final function used to score the functions is find_a_winner. This function will score all of the generated functions from a list of tree roots and return the one with the best and lowest score, resulting in the best candidate for the function most accurately matching the underlying function of the input data.

```
Find_a_winner(treeroots, infections,  variables)
	i = 0
	winner = treeroots[0]
	min_score = infinity
	While i < len(treeroots):
		new_score = score_a_function(infections, treeroots[i], variables)
		if new_score < min_score:
			Winner = treeroots[i]
			Min_score = new_score
		i++
Return winner
```

Questions, Problem Instances and Methodology

Preliminary experimental results proved promising, but once tested against our COVID19 dataset, there was very high variability in output performance, but more importantly, low convergence. The biggest question the group faced was that of the performance of the algorithm. How could the number of generations be reduced?

Another important question to ask was: How could the cost function incorporate function length? To develop a sophisticated algorithm to handle the task at hand was a feat of its own however it often led to long functions as the result of running the code. The algorithm would often present lengthy or complicated functions as its estimate of the dataset presented. The group quickly arrived at the consensus that this was not favourable should the resulting function be used for a meaningful purpose. It was clear that generating shorter functions that are still accurate was important. Changing the heuristic function to include length meant changing the line to read:

```
return abs(TARGET VAL - evaluate(temp_func, variables))**2
    + 0.1*len(to_list(temp_func))
```

Experiment Setup

Our experiments were run on a Microsoft Surface Pro 4 using the language python 3.10.0 in Pycharm 2021.3.3. The operating system used was windows 11 21H2 (10.0.22000.832) (July 21, 2022) with processor Intel(R) Core(TM) i5-1035G4 CPU @ 1.10GHz 1.50 GHz. The amount of memory available was 7.60 GB of usable RAM.

Initial experimental runs were performed against individual target values. That is, given constant variable values, how well could the algorithm find a function that matched this value.

The main experimental runs were using the 52-week COVID19 dataset described previously. The experiments were run with varying starting parameters of generation size, maximum generation number, and start and ending genetic operation probability ratios.

The probabilities of mutation versus crossover followed a sigmoid curve, from a starting crossover probability P_{CI} to an ending probability $P_{CF}=(1-P_{CI})$. This probability was varied as a function of the generation number, eventually reaching the close to the final

value as the generation number approached the maximum. The motivation was to encourage the system to perform a wide-search early on, honing in on the generated candidates later.

Since this is a local-search function, we needed exit criteria. We decided on exiting and returning the best function if either we reach the maximum generation value, or if the percent squared error between the generation function values and the real values was less than one percent.

Note that due to reaching maximum tree depth errors, or float overflow errors, many try-catch blocks were placed, if a function resulted in an error, it would be ignored with its cost set to the maximum possible float value. We also set a maximum function lisp string length to filter these as well.

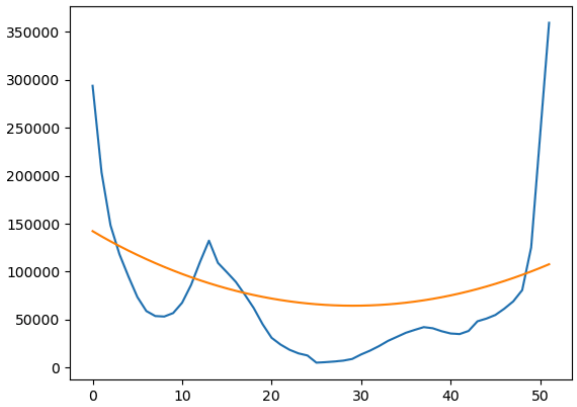
Experiment Results

Initial experiments proved promising. The system was able to fairly quickly converge to a solution function. close to the singular constant value target it was given. It was discovered that high variance (high mutation probability, low crossover and replication probabilities) resulted in the quickest solution convergence, this result held timewise regardless of generation size, being able to converge with fewer generations on larger generation sizes, but also being able to compute generations faster on smaller and thus achieving similar time performance. This stage of testing also showed high variance in generation number when controlling for generation size, and this was true throughout the experiments; A good solution could be found very quickly or it might take a long time to find an even halfway decent one, depending on the initial conditions and effects of random chance.

During our main experimental block, it was found that over time, the algorithm does indeed find better and better solutions to the problem and we were able to identify some evolutionary strategies that the algorithm would attempt in order to gain cost performance. These best candidates would often take the form of either *edge-peak-focus* or *mean-curve-matching* focus with the peak focus prioritising getting its score points from matching either start or end peak and mostly ignore the rest, while the mean

matching, would focus on getting a good median performance, disincentivizing heavy focus on peaks.

However, both the long-term convergence and the variance in function structure left a lot to be desired in terms of consistency. Namely, we have a guaranteed non-negative convergence due to an elitism count of one. This meant that we always passed the best candidate into the next generation without changes so we could at least guarantee that the fitness of the best candidate will never decrease. It was found that the output function's performance was much more heavily dependent on random chance than generation number, and for the overwhelming majority of generations there was 0 improvement in best score. What running the algorithm for longer would allow for is "leaps" in performance in which a new best candidate is selected while the others continue their evolution. In order to improve this algorithm's performance, it was deemed most important to increase the consistency of this convergence. The following functions were generated with a generation size of 2500, and a generation-number varying crossover probability. A lower score means better performance.




```
(+ (- (* (/ x (u_ (exp x))) (+ (u_ (u_ x)) 28.61)) (- (cos (* -69.78 x)) (*
    (* 42.11 67.49) 50.04))) (- 2.57 (* (- (- 63.55 x) (+ (log x) (/ x x))) (+ x (- (* 88.72 x) x)))))
```

Figure 5.1: The best performing candidate we were able to generate.

150,000 Generations

Score: 0.3529

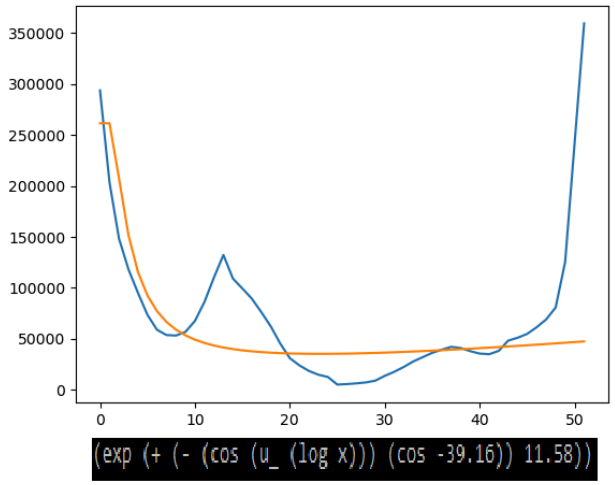


Figure 5.2: A left-biased function edge-peak-focus candidate

4,000 Generations

Score: 0.3657

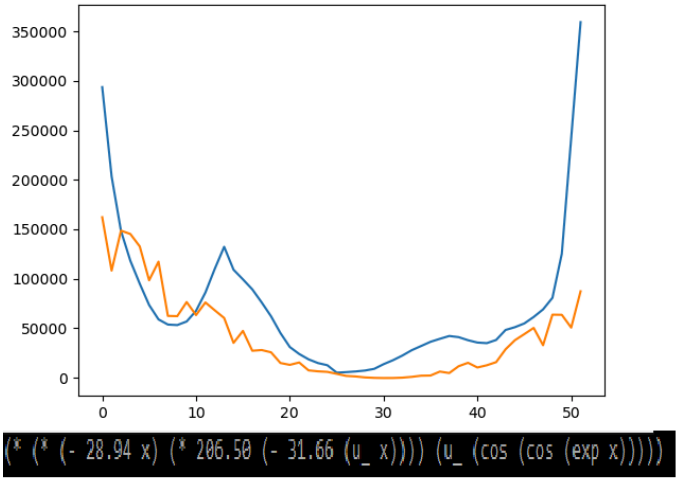


Figure 5.3: A heavily saw-toothed mean-curve-matching candidate

8,000 Generations

Score: 0.3548

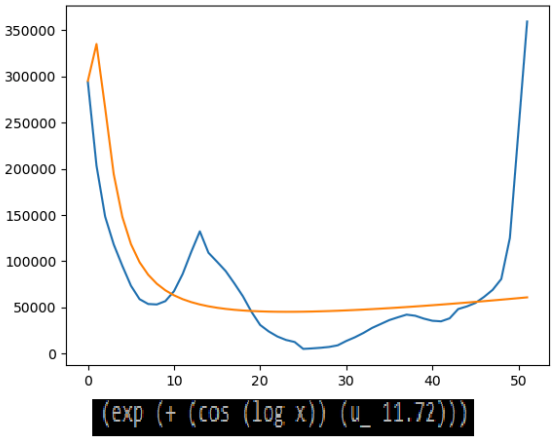


Figure 5.4: Another left-biased function edge-peak-focus candidate

10,000 Generations

Score: 0.3726

Note that just because a candidate evolved from more generations, doesn't necessarily mean that it performed better. Lots of optimization strategies were tried in attempts to increase the convergence. Variance in crossover ratio proved somewhat performant, so too did increasing generation size and running the algorithm for longer. Many combinations of input parameters were attempted but it had little boost in the performance of the system.

Conclusion

Given the results obtained, the group concluded that this experiment was a success and that the algorithm designed was able to generate some decent candidate functions to match our input COVID19 dataset.

To answer the first question posed by the group: How could the number of generations be reduced? Through our experiments, varying the crossover probability over the course of the evolution proved an improvement, albeit a minor one. By far the most consistent way we discovered to reduce the number of generations was to increase the generation size, computing more candidates in the same number of generations.

Also, the group was able to create a line of code to answer one the other of our initial questions: How could the cost function incorporate function length? The result in adding a function length parameter to the function cost resulted in a preference for shorter functions when possible, but would be, as intended, overshadowed in the case of finding a much better performing function.

More work should be done in order to flesh out the convergence of the system so as to approach a much more performant candidate function with a higher degree of probability in a shorter search time.

What we learned

This project provided each member of the team with valuable experience both in terms of technical knowledge and working in a team environment. The first major takeaway from this project is a deeper understanding of search algorithms. Our group found that actually taking the time to think about implementation provided a deeper insight into how these algorithms work. Secondly, this undertaking gave the group a comprehension of the process of natural selection in a more technical sense which became very eye-opening to the world we live in today and how it came to be. Lastly, the general experience of working in a team setting remotely was highly valuable as most modern computing science jobs require a strength in this ability. Overall, the group's efforts in implementations, time management and

Notable Problem Instances

As mentioned before, the system performed extremely well on instances of single constant target values, generating close-value candidate functions even before reparameterization was introduced, in which the best candidate function would often become a 1-size tree of just the target value. This was a good result since it meant that our system worked in finding a high performing, short length candidate function.

The system had poor convergence consistency when trying to compare against the input array of values provided by the COVID19 dataset. We learned that a different genetic operator selection strategy would have to be developed in order to encourage the system to better propagate well-performing traits so as to have a more likely convergent search strategy.

What Would Be Done With More Time?

The primary area of focus would be in further developing the genetic operator selection strategy. The immediate possible implementation would be that of the text [3], having well-performing candidates have a higher chance of crossover, and with a higher chance of the crossover partner being a well performing function, while worse performing functions would more likely be mutated or regenerated. Another solution that was discussed was to have each genetic operation be a two step process, in which either replication or crossover would be performed, with a mutation on the end result in order to guarantee wide-area search while also having a form of fitness propagation. Lastly, other biomimetic solutions would be investigated, such as a no-crossover "asexual reproduction" situation, in which the top N candidates of each generation would be selected, and the whole next generation would be created just from mutations of these N best candidates.

The group could have also spearheaded a more detailed analysis of real world infection rates using a greater number of parameters. For example, the group could find a way to incorporate age, gender and occupation into the data instead of just using the number of infections per week. This was something that was discussed briefly, but not implemented due to time restrictions. It is worth noting that with more time to spend on this project, it could be developed as a go-to tool for the field of epidemiology, in the study of the driving factors of infection rates and could be highly useful if presented with another pandemic scenario like COVID19.

Bibliography

[1] Ma, H. CMPT 417 Intelligent Systems, lecture material. (2022)
[2] Preliminary dataset on confirmed cases of COVID-19, Public Health Agency of Canada (2022-07-08) Data retrieved from: https://www150.statcan.gc.ca/n1/pub/13-26-0003/132600032020001-eng.htm
[3] Thomas D., Steven B., Bernd N., *Machine Learning Control – Taming Nonlinear Dynamics and Turbulence* (2018), Springer International Publishing.

The idea for the algorithms used stemmed from the class slides and research done by Thomas D., Steven B., Bernd N. in their guide *Machine Learning Control – Taming Nonlinear Dynamics and Turbulence.*