Assignment 4

Question 1

(a) Particle Swarm Optimization Experiments and Observations

Experiment	Population	Speed Limit	Inertia	Personal Best	Global Best	Convergence (Ticks)	Best Value
1	30	2	0.60	1.7	1.7	116, 74, 19	1
2	30	2	0.60	1.494	1.494	48, 118, 31	1
3	30	2	0.729	1.7	1.7	45, 920, 434	1
4	30	2	0.729	1.494	1.494	9, 56, 11	1
5	30	6	0.60	1.7	1.7	60, 11, 8	1
6	30	6	0.60	1.494	1.494	19, 12, 14	1
7	30	6	0.729	1.7	1.7	411, 14, 31	1
8	30	6	0.729	1.494	1.494	9, 731, 26	1
9	80	2	0.60	1.7	1.7	42, 16, 23	1
10	80	2	0.60	1.494	1.494	20, 7, 22	1
11	80	2	0.729	1.7	1.7	32, 8, 12	1
12	80	2	0.729	1.494	1.494	13, 11, 19	1
13	80	6	0.60	1.7	1.7	13, 9, 15	1
14	80	6	0.60	1.494	1.494	6, 9, 19	1
15	80	6	0.729	1.7	1.7	8, 10, 13	1
16	80	6	0.729	1.494	1.494	12, 15, 14	1

To examine the PSO algorithm's characteristics in greater detail, the first three speed of convergences (in units of ticks) were measured, as opposed to only one, to have a better idea of how the corresponding set of parameters perform. Moreover, many of the convergence values are overlapping, so comparing with multiple values gives a clearer picture of its performance. An important observation was that many of the experiments had to be run multiple times to converge to a value, some more than others, most likely due to getting stuck in local optima, combined with the large search space it has to cover. Also, it should be noted that all experiments converged to the best value of 1, so observations will be made solely for the convergence values. First, keeping all other parameters constant with changing personal/global best factors, it appears to have almost no effect on the speed of convergence. There are some outliers, such as between experiments 3/4 and 7/8, however, the rest of them have similar values. Next, keeping all other parameters the same but changing the particle's inertia, many of the results show that increasing inertia decreases the speed of convergence. This is particularly noticeable between experiments 1/3 and 5/7, where inertia is increased, promoting more exploration of the search space, and most likely causing the particles to overshoot during the search. Next, changing only the speed limit, there is a considerable effect, where increasing the speed limit resulted in increasing the speed of convergence. This is most prevalent between experiments 2/6 and 9/13, where all three of the measured convergence values improved for each comparison. In these cases, the speed limit of 2 proved to be lower than desired, as this could lead to particles getting stuck in local optima, so increasing it to 6 allowed them to escape and converge faster. Lastly, only changing the population size, and keeping everything else constant, had the most significant effect on convergence time. In almost every case, the speed of convergence improved, and this makes sense because more population results in exploring more of the search space, as well as having a higher chance at more optimal global best values, which other particles can use to steer towards the direction of the optimal solution.

(b) NetLogo Implementation Versus Classical Particle Swarm Optimization

The difference between the motion formulations is that NetLogo's implementation has a $(1 - inertia)$ term when calculating the velocity of each particle. It states that it was only added to allow the inertia slider to vary the motion of the particles on the full spectrum. This ranged from 0.0, where the particles were always moving towards the best spots and ignoring its previous velocity, to 1.0, where they were moving in a straight line. This term is not present in the classical PSO; however, it still has the same functionality, and would converge to the same results if accounted for. This difference is only to improve the user experience of the slider and make it more intuitive, as one would think that increasing the slider should increase the inertia.

Assignment 4

Question 2

(a) Update Mode of Particle Swarm Optimization Implementation

The update mode for the personal best and neighbours' best in the following implementation of the particle swarm optimization is asynchronous. This is due to how the neighbourhood best solution is being updated in the algorithm. In this case, the neighbourhood best solution is being updated within the particle's update loop, as opposed to outside of the loop, which would make it synchronous. This means that every time a new particle has been evaluated, the neighbourhood best is updated. The asynchronous version of the particle swarm optimization usually produces better results, as it causes the particles to use more up-to-date information. The trade-off to this is slightly more computations per iteration, however, this is outweighed by the benefit of having more current information, i.e., a more optimal neighbourhood best solution.

(b) Change Algorithm to Work in Synchronous Mode

Since the implementation of the particle swarm optimization worked in asynchronous mode, the neighbourhood best update was within the particle's update loop. Hence, to change the algorithm to work in synchronous mode, move the neighbourhood best update outside of the particle's update loop. So, first for each particle, update their velocities, positions, and personal bests. Then, once all the particles have been evaluated, now update the neighbourhood best solution, before moving on to the next iteration.

(c) Effect of Parameters W, C_1, and C_2 in the Particle Swarm Optimization Model

When C_1 is set to zero, this reduces the velocity model to a social only model, completely removing the cognitive component, meaning all the particles are solely attracted to the neighbourhood best solution. It places all its trust in the swarm's experience and ignores its own.

When C_2 is set to zero, this reduces the velocity model to a cognition only model, completely removing the social component, meaning all the particles act as independent hill climbers. It places all its trust in its own experience and ignores the rest of the swarm.

The importance of the W parameter is that it acts as a balance between exploration and exploitation. Large values promote exploration of the search space, while small values promote exploitation, allowing more control to the cognitive and social components.

Assignment 4

Question 3

(a) Parameter Choices

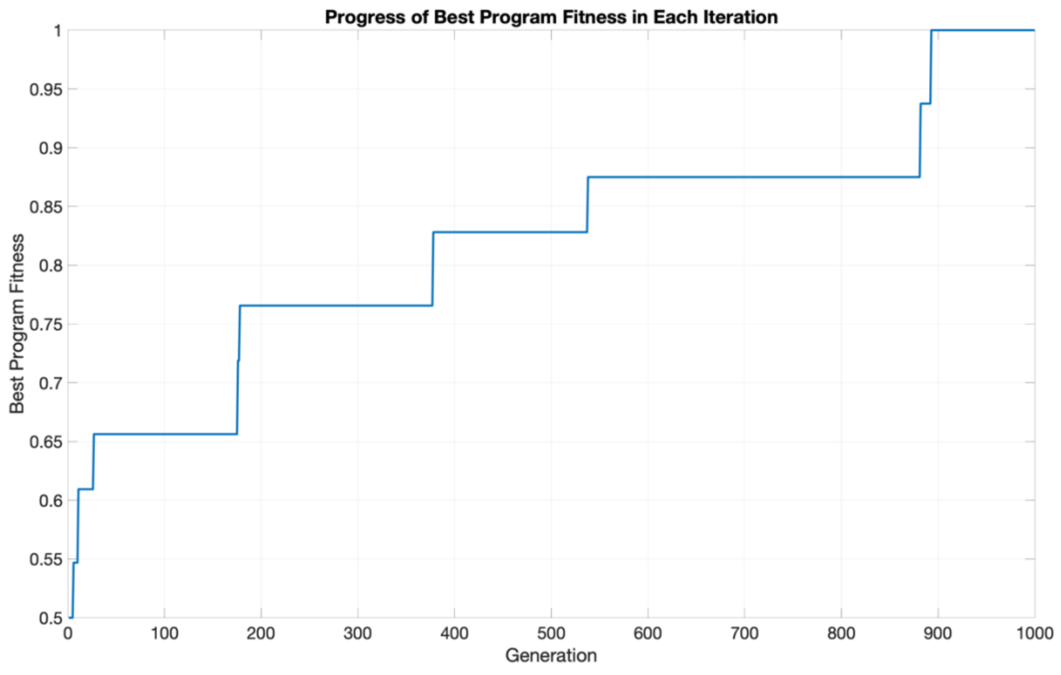
The following parameters were chosen for the genetic programming algorithm:

Population Size	100
Number of Generations	1000
Crossover Probability	0.9
Mutation Probability	0.6
Maximum Depth of Trees	8

The population size and number of generations were chosen as 100 and 1,000 respectively, to allow the algorithm to converge more often. I found that many times, the algorithm would result in almost the optimal solution, and with only higher populations and generations is it able to reach the most optimal solution. This is because a higher population allowed the algorithm to evaluate more of the search space, and a higher number of generations enabled it to evolve more and perform more crossover and mutation. The crossover probability was 90% and I found this as a reasonable number for the algorithm because it allowed the algorithm to explore more of the search space and escape local optima more often. The mutation probability was 60% and this performed well because it allowed the algorithm to hill climb and become more optimal locally, which appeared to help it converge. Lastly, the maximum depth of the trees was set to eight so that it would retain a relatively spread-out tree structure, and not imposing a maximum depth would transform it into almost a linear structure.

Assignment 4

(b) Plot of Progress of Best Program Fitness in Each Iteration



First, note that C++ was used for the main genetic programming algorithm, and then the resulting values were plotted in MATLAB. Analyzing the plot of the progress of the best program fitness in each iteration, this particular run started with a fitness of 0.5, meaning initially half of the cases were correct. It stayed here until generation six, when it increased to around 0.547 and stayed there until generation 10. At generation 11 it improved again to 0.609 and stayed there until generation 26. From generation 27 to generation 175 it was 0.656, then 0.719 from generation 176 to generation 177, 0.766 from generation 178 to generation 377, 0.828 from generation 378 to generation 537, 0.875 from generation 538 to generation 881, 0.938 from generation 882 to generation 892, and finally 1 from generation 893 to generation 1000. The following table summarizes its progress:

Generation	**Fitness**
$1-5$	0.5
$6-10$	0.546875
$11-26$	0.609375

Paolo Torres

$27 - 175$	0.65625
$176 - 177$	0.71875
$178 - 377$	0.765625
$378 - 537$	0.828125
$538 - 881$	0.875
$882 - 892$	0.9375
$893 - 1000$	1

(c) Fitness of the Finalist Program

The fitness of the finalist program was one, meaning it got all of the cases correct. The algorithm starts by generating random trees given the population size of 100. Given a maximum depth of 8, it builds each tree using a depth-first search recursive approach, randomly selecting a direction (either left or right), and randomly selecting a function (AND, OR, NOT, IF) or terminal ($a0, a1, d0, d1, d2, d3$) to use as the value of each node. If a function is used it can still have a chance to go to that node to place more children, however if a terminal is used, then that spot is made restricted, causing all the leaf nodes to be terminals. To simplify crossover and mutation later in the algorithm, the trees were converted to corresponding vectors containing the functions and terminals, where all the parents were linked to their associated children. These vectors were then combined together, and the resulting 2D vector acted as the initial set of solutions for the genetic programming algorithm.

With its initial set of solutions, the evolution process begins for $1{,}000$ generations. For each generation, it evaluates each solution, updates the best program and fitness found so far if necessary, and randomly performs either crossover or mutation on each solution probabilistically, given crossover and mutation probabilities of 90% and 60%, respectively. The evaluation process works by parsing each solution as a fixed length character string and comparing its output to the correct program. The correct program is as follows:

$$IF\ a0\ AND\ a1\ return\ d3$$

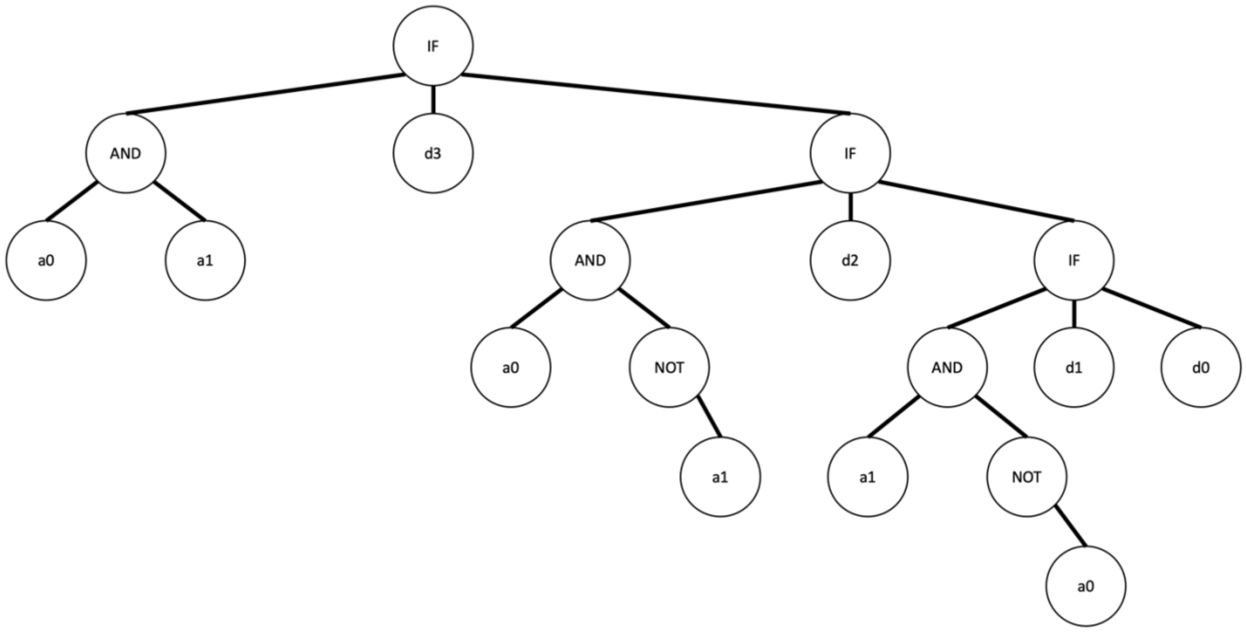
$$IF\ a0\ AND\ NOT\ a1\ return\ d2$$

$$IF\ NOT\ a0\ AND\ a1\ return\ d1$$

$$return\ d0$$

Moreover, for each solution, either crossover or mutation (not both) occurs, given their probabilities. For crossover, the algorithm randomly selects two individuals in the solution set, and then randomly selects two portions of each individual and swaps them. Additional logic was added to check if this swap improved the overall fitness of these new solutions, and if not, then these portions were swapped back. This added logic is a slight hill climbing technique that resulted in significantly improving its performance and chance of convergence. For mutation, a random portion of each solution was selected, and randomly changed using the set of functions and terminals of the problem. Similar to crossover, additional logic was added to check if this mutation improved the overall fitness, and if not, then it was changed back.

(d) Tree of the Finalist Program



This tree corresponds to the results of the finalist program of the best overall program and fitness after 1,000 generations. As mentioned previously, the trees were converted to corresponding vectors to simplify the processing and evaluation. Thus, the associated solution vector of this tree is:

$$program = \{IF, a0, AND, a1, d3, IF, a0, AND, NOT, a1, d2, IF, a1, AND, NOT, a0, d1, d0\}$$

Since the functions and terminals were split up, while keeping the components still linked together, the modified solution vector is as follows:

$$modified = \{IF, AND, IF, AND, NOT, IF, AND, NOT, a0, a1, d3, a0, a1, d2, a0, a1, d1, d0\}$$

In this particular run, this is the result returned by the best program after 893 generations. This modification is read as: "if a0 is true and a1 is true then return d3, else if a0 is true and a1 is false then return d2, else if a0 is false and a1 is true then return d1, else return d0." This exactly matches the correct program mentioned in part C. This can also be seen in a logic table:

Inputs		**Output**
$\boldsymbol{a0}$	$\boldsymbol{a1}$	$\boldsymbol{d0/d1/d2/d3}$
0	0	$d0$
0	1	$d1$
1	0	$d2$
1	1	$d3$

Evaluating each of these four cases using both the tree, as well as the solution vector, results in the expected corresponding outputs. Furthermore, evaluating each of the 64 cases, from 000000 to 111111, where the setup is $\{a0, a1, d0, d1, d2, d3\}$, returns the correct result every time. Thus, the finalist program is depicted correctly by the tree above, and represented by the corresponding solution vector:

$$program = \{IF, a0, AND, a1, d3, IF, a0, AND, NOT, a1, d2, IF, a1, AND, NOT, a0, d1, d0\}$$

In the algorithm, this optimal solution vector is modified as mentioned previously:

$$modified = \{IF, AND, IF, AND, NOT, IF, AND, NOT, a0, a1, d3, a0, a1, d2, a0, a1, d1, d0\}$$

Therefore, for this particular run, the final fitness of this program at 893 generations achieved the optimal fitness of one. These results change per run depending on how the initial solution trees are randomly generated, and how often crossover and mutation are performed.