Assignment 4 Mohammad

Question 3

Part 1

> 💡 [25 pts] Code a simple PSO to solve the problem. To do this you need to encode
> the problem, initialize a population, select a velocity update equation, and select a
> stopping criterion. Run your code and report: final solution, plot the progress of the
> average fitness and the best particle fitness.

This part can be simulated by running the following command,

```
python run.py 0
```

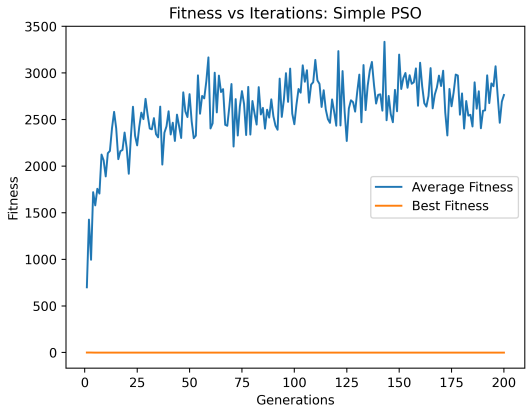
For simple PSO, the velocity update function is defined as,

$$v^{t+1} = v^t + c_1 r_1^t [P_{best}^t - x^t] + c_2 r_2^t [G_{best} - x^t]$$

where $c1 = c2 = 2.05$, and both r_1, r_2 are random values uniformly sampled from the range of $[0, 1]$.

The plot created from the above command is shown below and can be viewed under `./q1/plots/Fitness vs Iterations: Simple PSO.png`.

```
CONSOLE OUTPUT:
c1: 2.05
c2: 2.05
iterations: 200
particles: 100
best fitness: -1.0295688135918653
best x,y: -0.09005 0.69663
```



Simple PSO

The Simple PSO algorithm did not reach the optimal solution of `1.0316285` as can be seen by the console output. It can be seen from the average fitness that the algorithm did not converge and has lots of variance in fitness among particles.

Part 2

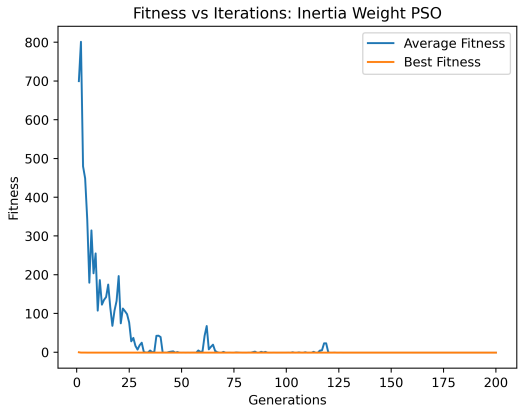
> 💡 [5 pts] Use the Inertia Weight version of velocity update equation with Global
> best. Run your code and report: final solution, and plot the progress of the average
> fitness of the population and the global best particle fitness

This part can be simulated by running the following command,

```
python run.py 1
```

and will create the following plot under `./q1/plots/Fitness vs Iterations: Inertia Weight PSO.png`

```
CONSOLE OUTPUT:
c1: 2.05
c2: 2.05
w: 0.5
```



Inertia Weight PSO

```
iterations: 200
particles: 100
best fitness: -1.0316284534898774
best x,y: -0.08984 0.71266
```

For Inertia Weight PSO, the velocity update is defined as,

$$v^{t+1} = w * v^t + c_1 r_1^t [P_{best}^t - x^t] + c_2 r_2^t [G_{best} - x^t]$$

where $c1 = c2 = 2.05$, $w = 0.5$, and both r_1, r_2 are random values uniformly sampled from the range of $[0, 1]$. w was chosen to be smaller than 1 to avoid large velocity updates over time (leading to divergance).

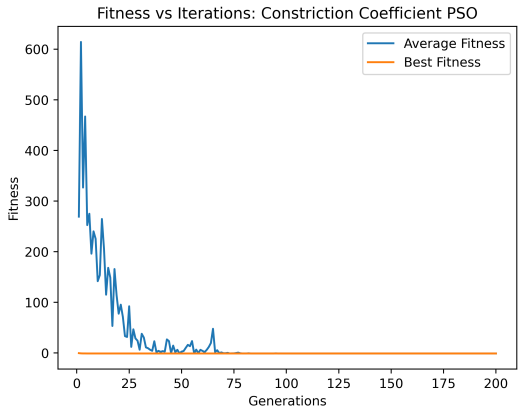
The Inertia Weight PSO algorithm did reach the optimal solution of `1.0316285` as can be seen by the console output. The average fitness has a clear trend, with minor variations before seeming to converge to an ideal population with low variance and good fitness.

Use Constriction Factor version of velocity update equation with Global Best.
Run your code and report: final solution, and plot the progress of the average
fitness of the population and the global best particle fitness.

This part can be simulated by running the following command,

```
python run.py 2
```

and will create the following plot under `./q1/plots/Fitness vs Iterations: Constriction Coefficient PSO.png`



Constriction Coefficient PSO

```
CONSOLE OUTPUT
c1: 2.05
c2: 2.05
iterations: 200
particles: 100
best fitness: -1.0316284534898774
best x,y: 0.08984 -0.71266
```

For Constriction Coefficient PSO, the velocity update function is defined as,

$$v^{t+1} = k * [v^t + c_1 r_1^t [P_{best}^t - x^t] + c_2 r_2^t [G_{best} - x^t]]$$

$$k = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4 * \phi}|}, \phi = c_1 + c_2, \phi > 4$$

where $c1 = c2 = 2.05 > 4$, $w = 0.5$, and both r_1, r_2 are random values uniformly sampled from the range of $[0, 1]$.

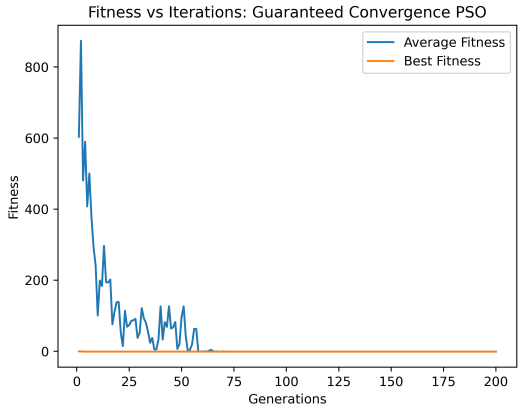
The Constriction Coefficiant PSO algorithm did reach the optimal solution of `1.0316285` as can be seen by the console output, similar to Inertia Weight PSO. The average fitness has a clear trend, with minor variations before seeming to converge to an ideal population with low variance and good fitness.

Use the Guaranteed Convergence PSO (GVPSO). Run your code and report: final
solution, and plot the progress of the average fitness of the population and the
global best particle fitness.

This part can be simulated by running the following command,

```
python run.py 3
```

and will the following plot under `./q1/plots/Fitness vs Iterations: Guaranteed Convergence PSO.png`



Guaranteed Convergance PSO

```
CONSOLE OUTPUT
c1: 2.05
c2: 2.05
w: 0.5
iterations: 200
particles: 100
best fitness: -1.0316284534898774
best x,y: 0.08984 -0.71266
```

For Guaranteed Convergance PSO, the velocity update function is defined as,

$$v^{t+1} = w * v^t + \rho^t * (1 - 2 * r_2^t),$$

for particles that satisfy $x^t = G_{best}$. For all other particles, the Inertia Weight PSO velocity described above is used.

Here, ρ^t is defined as:

- $\rho^0 = 1.0$
- $\rho^{t+1} = 2 * \rho^t$ if $\#success(t) > \epsilon_s$
- $\rho^{t+1} = 0.5 * \rho^t$ if $\#failures(t) > \epsilon_f$
- $\rho^{t+1} = \rho^t$ otherwise

A failure is defined as:

$$f(G^{t+1}) = f(G^t)$$

And the following updates to success and failure counters are applied:

- $\#success(t+1) > \#success(t) \rightarrow \#failures(t+1) = 0$
- $\#failures(t+1) > \#failures(t) \rightarrow \#sucess(t+1) = 0$

For this algorithm, the following parameter values where used:

- $w = 0.5$
- $c1 = c2 = 2.05$
- $r_2 = random[0, 1]$
- $\epsilon_s = 15$
- $\epsilon_f = 5$

The Guaranteed Convergence PSO algorithm did reach the optimal solution of `1.0316285` as can be seen by the console output, similar to Inertia Weight PSO and Constriction Coefficient PSO. However, there is less variation in the Guaranteed Convergence PSO, signifying that it converged/stabilized faster then the other two algorithms. The average fitness has a clear trend, with the least variations compared to the other algorithms before seeming to converge to an ideal population with low variance and good fitness.

Part 3

Write a report [max 2 pages, 700 words, 12pt font size] in which you describe: (1) the
choices you made to set the parameters of the populations, (2) Your insights and
observations on the perforamce of each implementation, (3) Your observations on the
comparative performance of the different implementations.

The code for all implementation can be found under the following files:

- `q1/particle_swarm_optimization.py`
- `q1/particle.py`
- `q1/velocity_position_update.py`
- `q1/fitness.py`

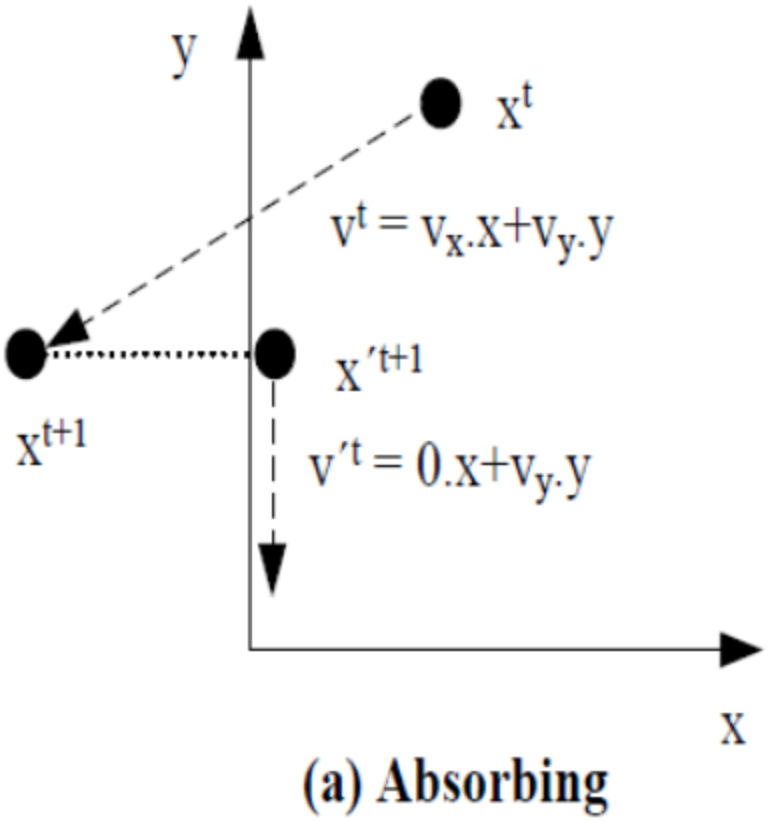
The file running the simulation can be found under,

- `q1/run.py`

and it can be run with or without command line arguments. Command line arguments accepts a single integer number from [0, 3], each corresponding to a different question for question 3.

A swarm size of 100 with 200 iterations (stopping criterion) were chosen for all test cases. All random modules are seeded with the same random number to give consistent comparisons.

When particles go out of bound, they are clipped via the `Absorbing` method.



(a) Absorbing

From the analysis in each section, it seems that Guaranteed Convergence PSO provides the best convergence and performance of the implemented algorithms. Constricted Coefficient PSO and Inertia Weight PSO perform similarly for the parameters chosen. Simple PSO performed the worse, and is the only algorithm that did not reach the ideal solution.

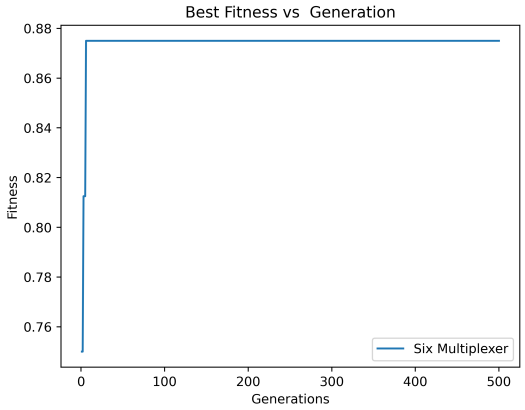
Question 2

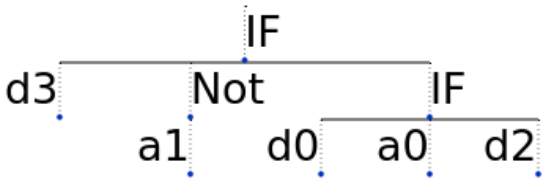
Part 1

💡 [30/50 pts]: Use genetic programming to discover the solution to the 6-multiplexer
problem.

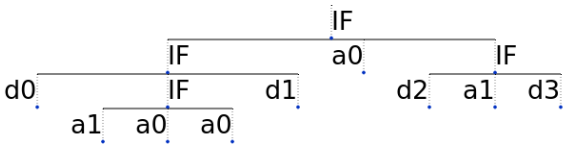
An approximate solution to the six `-multiplexer` is shown in the image below. A fitness of `0.875` was reached with this solution.

However it should be noted that a fitness of `1.0` was reached, however the graph data was lost for this. The diagram for this can be seen below.





6-Mux, Fitness 1.0



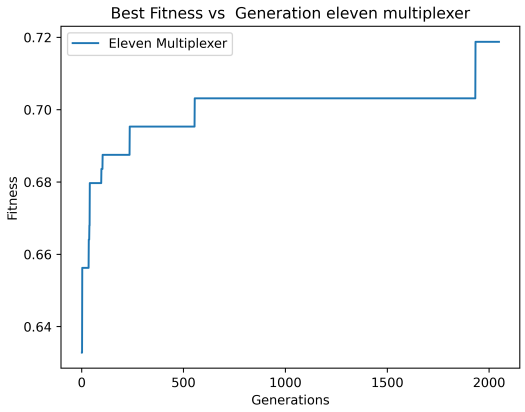
6-Mux, Fitness 1.0

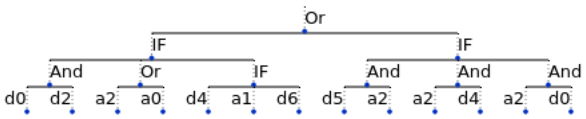
As can be seen from the graph of best fitness per generation, the progress of the algorithm stagnated very quickly, in the first 100 generations.

Part 2

[15/50 pts]: Use genetic programming to discover a solution to the 11-multiplexer (3
address, 8 data lines) problem. You may find it computationally expensive to use the full
test set on this problem (2048 cases), thus you may find it necessary to develop a technique
which only use a fraction (changing) of the test cases each generation.

An approximate solution to the `eleven-multiplexer` is shown in the image below. A fitness of `0.71875` was reached with this solution.





As the problem space for this is a lot larger than the six-multiplexer problem, it was run for more generations. Progress can be seen as the program progresses, and it may have reached the ideal solution with enough time (however time is limited!). A variety of methods where used to combat this issue such as limiting recombination to maintain the same max depth, implementing tournament parent selection, and changing mutation probabilities, and survivor selection count. However none of these assisted in convergence to an ideal solution.

Part 3

> 💡 [5/50 pts]: Use genetic programming to discover a solution to the 16-middle-3
> problem. The correct logic for 16-middle-3 given input x ∈ {0, 1}16 then for each part: report your parameter choices, and a plot of the progress of the best program
> fitness in each iteration, and the fitness of the finalist program. Report the tree of the finalist program in each part.

16-middle-3 has not been implemented.

The following parameters where used for parts 1 and 2:

- Population size of 1024
- Max depth of 3 (six-mux) and 4 (eleven-mux)

- Generation count of 500 (six-mux) and 2048 (eleven-mux)
- survivor count of 50 individuals
- probability of mutation of `0.05`
- Tournament size of 7

A variety of methods where used to try and reach convergence to an ideal solution such as limiting recombination to maintain the same max depth, implementing tournament parent selection, changing mutation probabilities, and survivor selection count. However none of these assisted in convergence to an ideal solution.

Please view sections above for graphs and the tree of the best solution for each of the problems.

All the code for this question is under the folder `q2`.