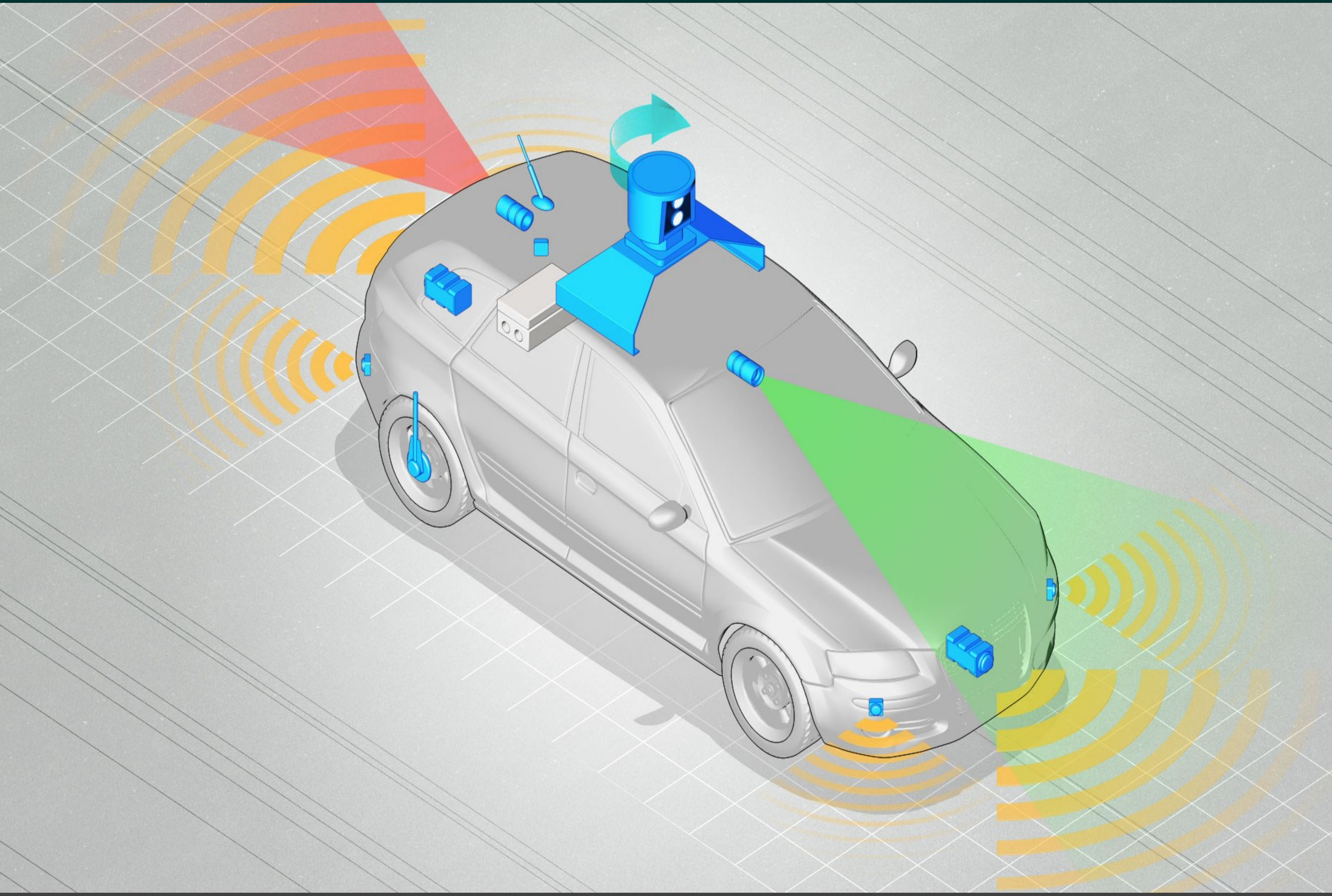
By: Sai Bharath Uppinakuduru Student ID: 160039454



Self-Driving Vehicle

IN3044 Artificial Intelligence (PRD1 A 2018/19)

Table of Contents

Experiment Details

Introduction

This is the report on the Artificial Intelligence application which is programmed for self-driving vehicles. In this piece, I will be discussing the path through and how I took to achieve my results as intended. In addition, I will be covering the environment I built to perform this experiment, the constrain of the generations, the logic/algorithm I used to determine fitness function, about variations of the average fitness when we vary mutation rate and final, I will be presenting the results of the experiment.

Experiment setup

Initial idea

Certain population of vehicles will be travelling on the highway and they will be created on the highway individually with five seconds interval amongst each other. Each vehicle will travel in different speed and each will haves its own radar with different radius and rules.

The vehicles will accelerate either till it hits its speed limits or till its radar come in contact other vehicles. The difference in speed creates collision between the vehicles because it does not deaccelerate to zero from its acceleration.

When a vehicle collides with others, the collided vehicle will be considered as dead and it deaccelerates till the speed hits zero. One which took the collision (victim vehicle) will have no impact and continue its lifecycle.

* The cover image is taken from the link https://techmalak.com/wp-content/uploads/2016/11/autonomoose-self-driving-car.jpg

Aim of the experiment/ AI Application

Ideally, at the end of the generation, there will be no collisions between vehicles. This will be achieved by maintaining an optimal distance between them. If the distance between two vehicles is greater than the optimal distance, the second vehicle accelerates till the distance between them will be optimal. If the distance between two vehicles is less than the optimal distance, the second vehicle will deaccelerate till the distance between them will be optimal.

Tools I used to build the application

Language: Python 3.7

IDE: PyCharm Professional 2018

GitHub Link: https://github.com/CodeXSai/Project-GA

File Size: 1.34GB

Experiment results with explanation

Environment

To achieve the aim of this project, I had to modify my initial ideas without amending the rules and effects of it. Having 50 to 200 population in one road decreases the efficiency exponentially as population creates. It is because, if I call a vehicle an object, every object must check whether every other object is in the collision course to it.

Let's take a case:

In the population of 100, where every object (vehicle) is created in the environment five seconds after the previous creation where the first creation is taken place at t=0 sec,

After 10 seconds:

There will be three objects where the first object is created at 0th second and second object will be created at 5th second and the third object will be created at 10th second. Now, first object will check whether second and third objects are in collision course with it and it will check itself to determine whether it is in collision course with other objects.

Time complexity = 3 objects * 3 checks

After 60 seconds:

Time complexity = 31 objects * 31 checks

At the end of the creation of 100 objects:

Time complexity = 100 objects * 100 checks

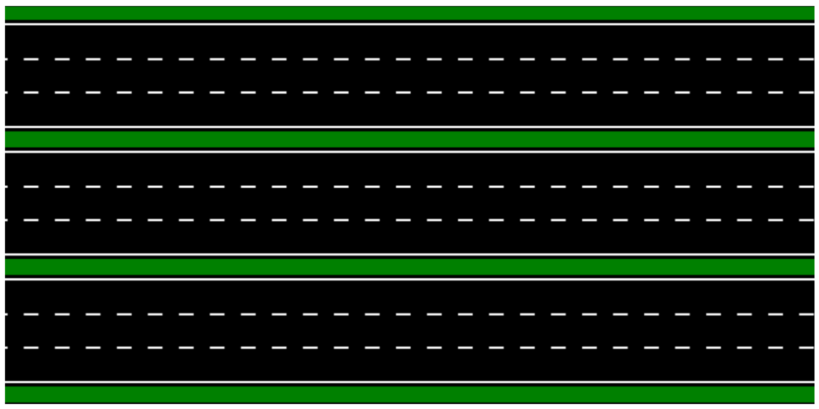
So, for 'n' population, there will be 'm' checks where n = m

Therefore,

Time complexity = n^2 which is exponential.

This resulting time complexity i.e. n^2 will create lag in the application

The main problem here is I created all the objects on single road where every object must check every other object and itself. To resolve this problem, I can have 3 roads parallel to each other and every road has three lanes. In total there are nine lanes and objects in the population will be created amongst the nine lanes randomly. Here, the load is divided by nine lanes and every object must check only the objects which are on same lane and itself. This gives somewhat realistic view of the traffics and reduces the time complexity by a large amount.



This image illustrates the solution for n^2 time complexity problem.

We can have 3 roads parallel to each other and every road has 3 lanes

DNA

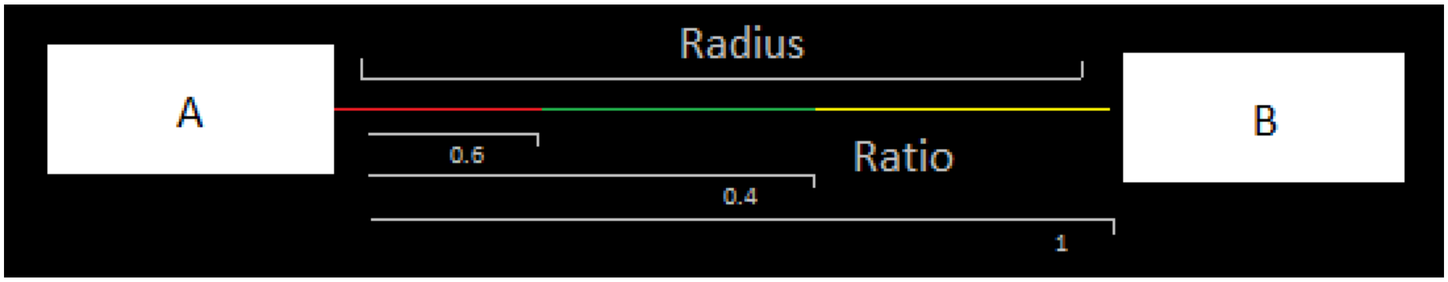
DNA is one of the major components of every object (vehicle). It contains the basic properties of the object. In this case, it has 2 properties.

1. Speed Limit: Highest speed an object can travel.
2. Radar Ratio: The radius of the radar.

The DNA will be passed to the next generation through the process of selection. This process will create new set of DNA list for the population which is used in next generation.

Radar

Radar is the component in the object (vehicle) which takes account of collision of another object with it. It has 2 components. Radius and Ratio of the parts of radar. Radius of the radar is embedded in object's DNA. Radar has 3 sub radars. Yellow, Green and Red.



This image illustrates the radar of object A (vehicle A) and its sub components

Red sub radar's radius will be 1/6th of the Radar's radius. Green sub radar's radius will be ¼th of the Radar's radius. Yellow sub radar's radius will be equal to the radius of the radar. The origin point of the radar and sub radars will be from the front of the object.

In the given image, the origin point of object A's radar and sub radar will be from the face of the object A.

If radius = r

Red sub radar radius= r * 1/6

Green sub radar radius= r * ¼

Yellow sub radar radius = r

Fuzzy logic

The main aim of this application is to maintain distance between two objects to avoid collision at any cost. Maintaining more distance may avoid collision but it increases the journey time. On the other hand, maintaining a close distance may leads to collision if the front vehicle hits brake. An optimal distancer is required to avoid both collision as well as over journey time. This means the resultant cannot be a constant or a well-defined value at all case. It is rather a range of values which varies throughout the time depending on the scenario.

The best approach to get the desired output is to implement fuzzy logic. This will accommodate the set of resultants. There are set of rules put in place to improve the behaviour of the object when it approaches another object. These rules implementation can be crosschecked using speed vs time graph which we will discuss in the 'Cross check' section

This image illustrates our case for using fuzzy logic

* This image is taken from the link https://i.ytimg.com/vi/rln_kZbYaWc/maxresdefault.jpg

Rules to implement fuzzy logic

Rules are one of the main parts to implement fuzzy logic. In our case, to get a crisp output, I must pass the distance from the object to the colliding object (vehicle). This will give the area colliding object situated in and what area it is travelling from.

For instance, if the radius of the radar of Object A is 20 meters.

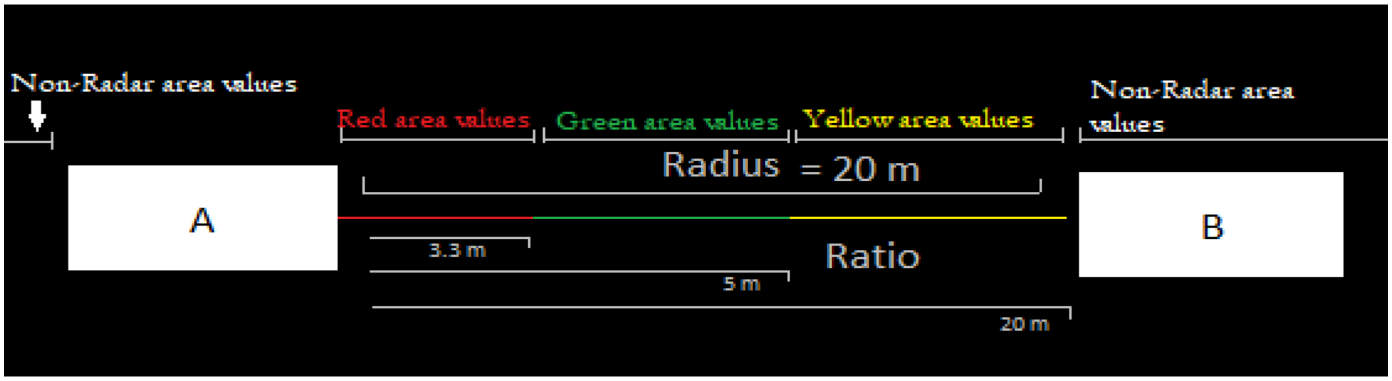
Then,

	Formulas (refer Radar section)
Radar radius = 20 meters	r
Yellow sub radar radius = 20 meters	r
Green sub radar radius = 5 meters	r * ¼
Red sub radar radius = 3.3 meters	r * 1/6

Let's call the colliding object as Object B.

Let's assume Object B is in front of Object A

Let's call the distance between object A and B as Distance



We fix this to different areas. i.e.

Object A is in front of Object B = Distance between backside of Object A > frontside of Object B

Non-Radar area values = (values > 20) or (Object A is in front of Object B)

Yellow area values = 5 < values < 20

Green area values = 3.3 < values < 5

Red area valued = 0 < values < 3.3

Overlap = Distance between backside of Object A <= frontside of Object B

Collision state = (Distance between frontside of Object A >= backside of Object B) and (Overlap)

Overlap checks whether Object A and Object B are overlapping. If it is overlapping, it returns 'true' else 'false'

Collision state checks whether Object A and Object B are collided. If it is collided, it returns 'true' else 'false'

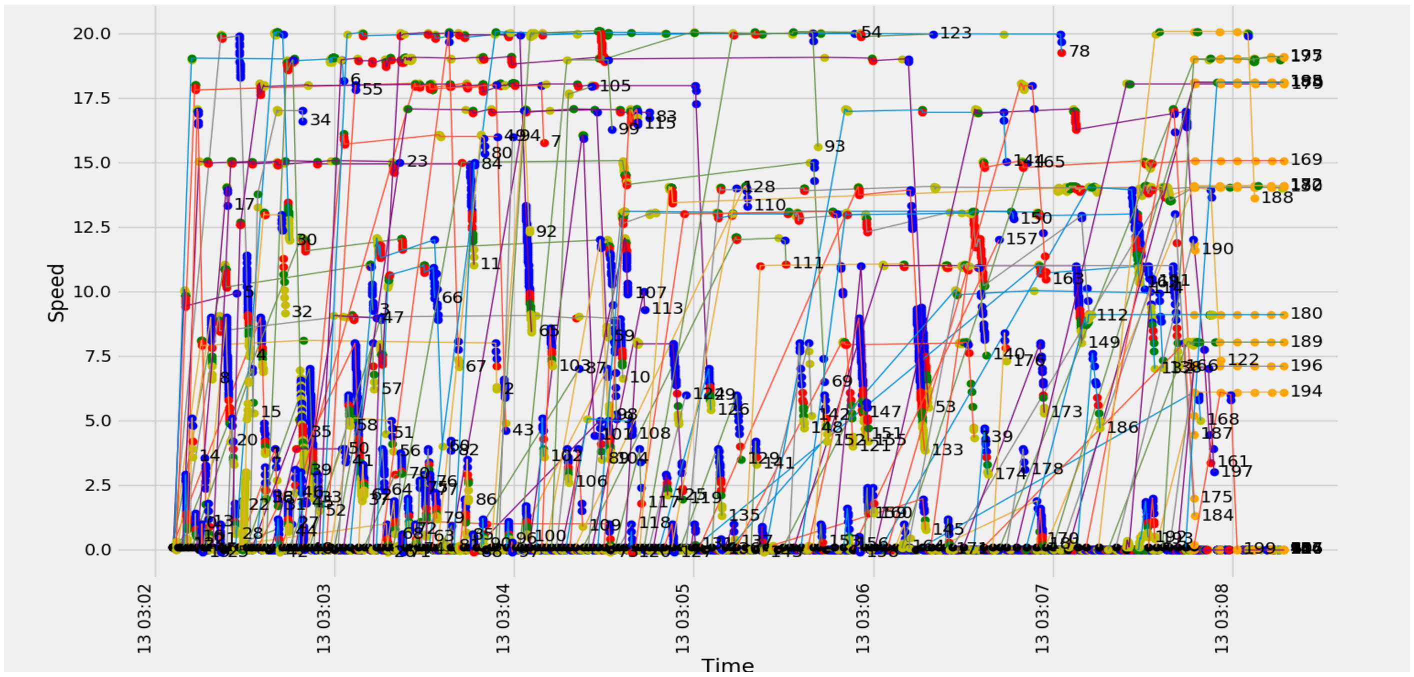
Rules Block:

```
If the Collision State is false:
    If the Distance matches with Yellow area values:
        If Object B is from Non-Radar area: (if object B is deaccelerating)
            Object A speed should slowly decrease
        Else if Object B is from Green area: (if object B is accelerating)
            Object A speed should slowly increase
    Else if the Distance matches with Green area values:
        Object A speed should not change
    Else if the Distance matches with Red area values:
        Object A speed should decrease as fast as possible
    Else: (if nothing is in collision course with object A)
        Object A speed should gradually increase to maximum
Else: (if object A is already collided)
    Object A speed should gradually decrease to zero
```

This *Rule Block* will loop whenever there is a movement of any objects(vehicles) in the entire simulation. As we covered in Environment section, this *Rule Block* will be checked between every object on the respective lane.

Rule Block crosscheck

This *rule block* determines the acceleration of every objects. We can check the efficacy of the *Rule Block* by plotting a speed vs time graph.



Speed vs Time Graph

Initial condition for above graph.

Population: 200

Mutation: 1%

Graph Shown: First Generation

Total Generation taken: 175

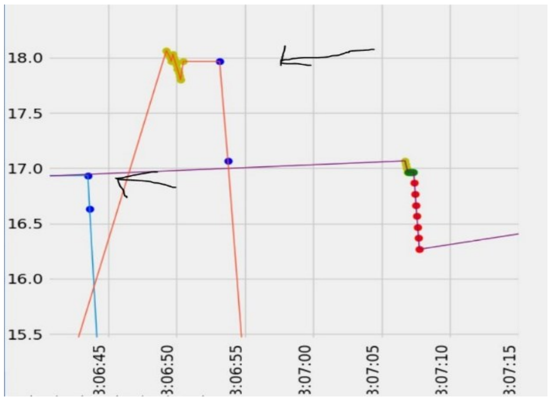
DNA

Speed Range: 5 to 20

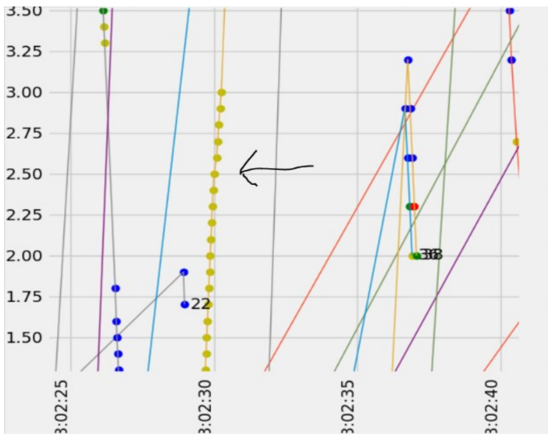
Radius Range: 100 to 200

NB: This graph gives information only about the different areas of the sub radars. This graph cannot be used to gain any other information.

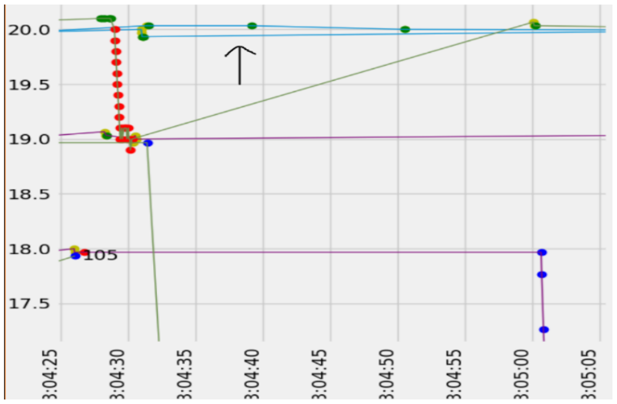
In the above graph, we see different colour plots. Blue, Yellow, Green, Red, Black and Orange.



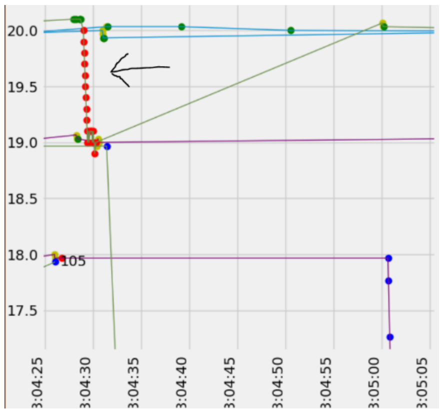
Blue Dot represents the collision. According to the Rule Block, when object A collides, it should slow down and stop moving. We can see that happening throughout the graph.



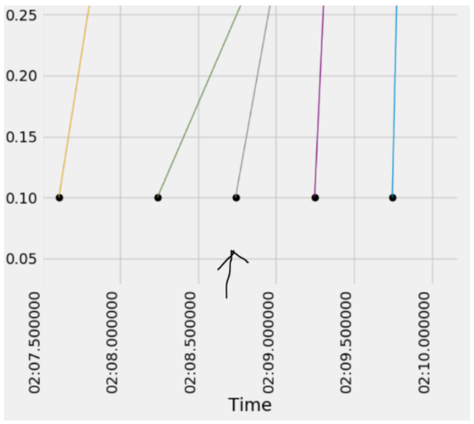
Yellow Dot represents object B in object A's yellow area. If object B is from Non-Radar area, object A's speed should decrease else if object B is from Green area, object A's speed should increase. We can see that happening throughout the graph.



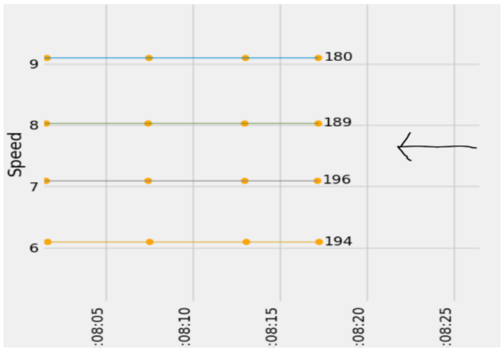
Green Dot represents object B in object A's green area. In this scenario, the speed of the object A should not change. It should remain constant. We can see that happening throughout the graph.



Red Dot represents object B in object A's red area. In this scenario, the speed of the object A should decrease as fast as possible. We can see that happening throughout the graph.



Black Dot represents the creation of an object. In this scenario, the speed of the object A should till it faces collision course. We can see that happening throughout the graph.

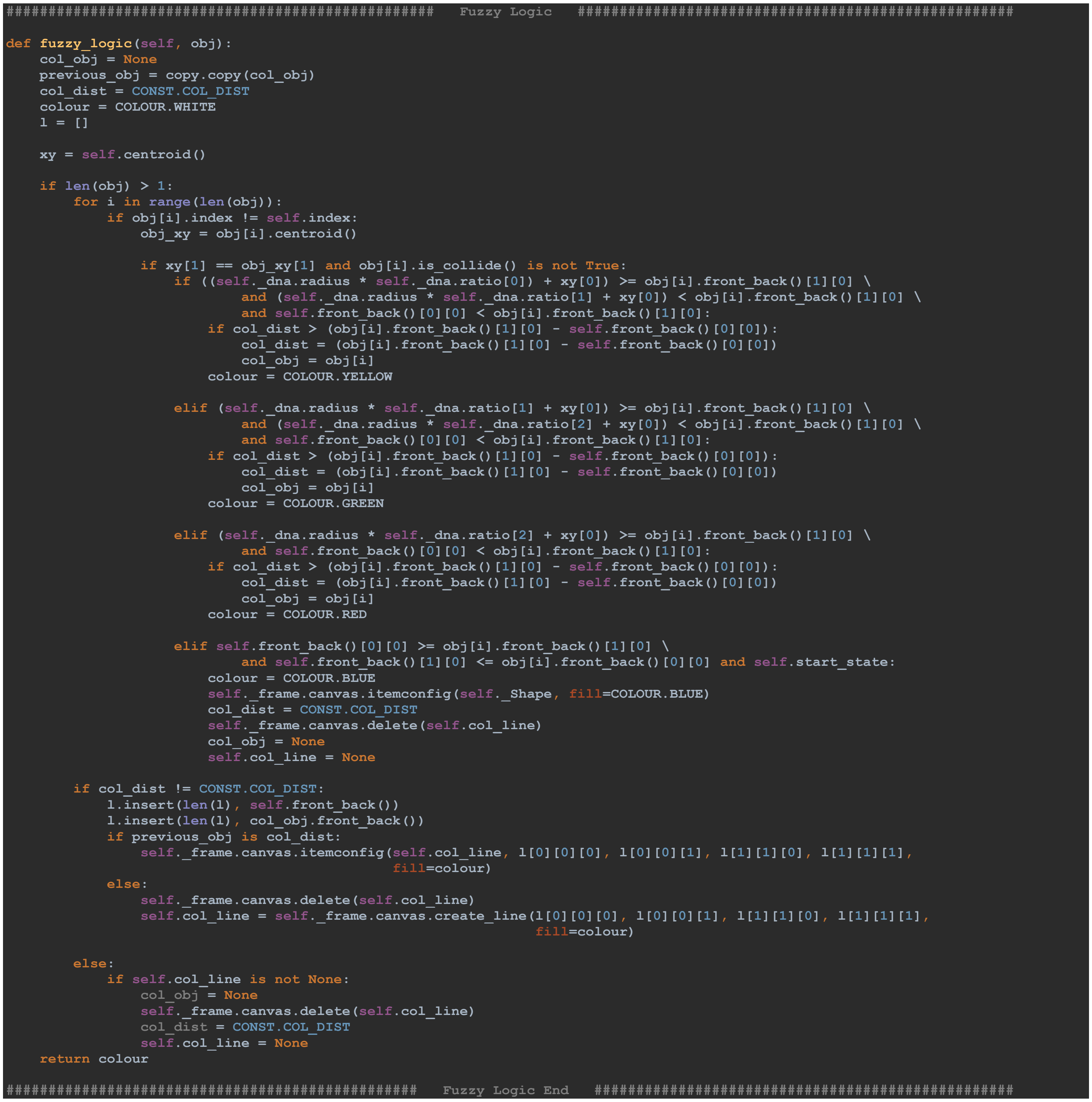


Orange Dot indicates that there is no further collision is possible from that object. When all the existing vehicle's status turns orange, we end that generation and initiate next generation. We can see that happening throughout the graph.

The above analysis shows that, our fuzzy logic is working. The Python code for fuzzy logic is given below.

The function `fuzzy_logic(self, obj)` takes one parameter called 'obj'. It is the list which contains all the objects on the current generation. It returns an object type called 'COLOUR' based on which the speed will be decided. This function will be called inside another function which is responsible for the movement of every object.

NB: 'self' is not a parameter. It is equivalent of 'this' in C, C++ or Java. It is the syntax in Python to have 'self' at very first else we cannot use any other methods of the same class and it will become a static function.



```
##################################################   Fuzzy Logic   ###################################################

def fuzzy_logic(self, obj):
    col_obj = None
    previous_obj = copy.copy(col_obj)
    col_dist = CONST.COL_DIST
    colour = COLOUR.WHITE
    l = []

    xy = self.centroid()

    if len(obj) > 1:
        for i in range(len(obj)):
            if obj[i].index != self.index:
                obj_xy = obj[i].centroid()

                if xy[1] == obj_xy[1] and obj[i].is_collide() is not True:
                    if ((self._dna.radius * self._dna.ratio[0]) + xy[0]) >= obj[i].front_back()[1][0] \
                            and (self._dna.radius * self._dna.ratio[1] + xy[0]) < obj[i].front_back()[1][0] \
                            and self.front_back()[0][0] < obj[i].front_back()[1][0]:
                        if col_dist > (obj[i].front_back()[1][0] - self.front_back()[0][0]):
                            col_dist = (obj[i].front_back()[1][0] - self.front_back()[0][0])
                            col_obj = obj[i]
                        colour = COLOUR.YELLOW

                    elif (self._dna.radius * self._dna.ratio[1] + xy[0]) >= obj[i].front_back()[1][0] \
                            and (self._dna.radius * self._dna.ratio[2] + xy[0]) < obj[i].front_back()[1][0] \
                            and self.front_back()[0][0] < obj[i].front_back()[1][0]:
                        if col_dist > (obj[i].front_back()[1][0] - self.front_back()[0][0]):
                            col_dist = (obj[i].front_back()[1][0] - self.front_back()[0][0])
                            col_obj = obj[i]
                        colour = COLOUR.GREEN

                    elif (self._dna.radius * self._dna.ratio[2] + xy[0]) >= obj[i].front_back()[1][0] \
                            and self.front_back()[0][0] < obj[i].front_back()[1][0]:
                        if col_dist > (obj[i].front_back()[1][0] - self.front_back()[0][0]):
                            col_dist = (obj[i].front_back()[1][0] - self.front_back()[0][0])
                            col_obj = obj[i]
                        colour = COLOUR.RED

                    elif self.front_back()[0][0] >= obj[i].front_back()[1][0] \
                            and self.front_back()[1][0] <= obj[i].front_back()[0][0] and self.start_state:
                        colour = COLOUR.BLUE
                        self._frame.canvas.itemconfig(self._Shape, fill=COLOUR.BLUE)
                        col_dist = CONST.COL_DIST
                        self._frame.canvas.delete(self.col_line)
                        col_obj = None
                        self.col_line = None

        if col_dist != CONST.COL_DIST:
            l.insert(len(l), self.front_back())
            l.insert(len(l), col_obj.front_back())
            if previous_obj is col_dist:
                self._frame.canvas.itemconfig(self.col_line, l[0][0][0], l[0][0][1], l[1][1][0], l[1][1][1],
                                              fill=colour)
            else:
                self._frame.canvas.delete(self.col_line)
                self.col_line = self._frame.canvas.create_line(l[0][0][0], l[0][0][1], l[1][1][0], l[1][1][1],
                                                               fill=colour)

        else:
            if self.col_line is not None:
                col_obj = None
                self._frame.canvas.delete(self.col_line)
                col_dist = CONST.COL_DIST
                self.col_line = None
    return colour

################################################   Fuzzy Logic End   #################################################
```

This is the fuzzy logic function written in python 3.7

Population

Population is number of vehicles present in every generation. In the application, it randomly generates required number of DNAs. Population uses those DNAs to create vehicles for the first generation. After the end of every generation, it creates a new set of objects by picking the DNA of the objects of generation which is just ended. The picking will be based on the selection process where individual's fitness ranking will be considered. The new set of DNAs will undergo mutation based on the mutation rate and those DNA's will then be injected to the objects and start next generation.

Selection

Whenever a generation is ended, I have created a new set of DNAs using the old ones. We select two DNAs such a way that the DNA with high fitness score of will have more probity to get picked. I mate those two selected DNAs to get the child DNA. While mating, it takes half of the properties from the first selected DNA and another half from the second selected DNA.

In this case, it picks speed limit property will be taken from one DNA and Radar radius from the second DNA. The selection process is random.

The list of newly created DNAs will undergo mutation. The mutation will be based on mutation rate.

Mutation

Mutation is a process of modifying DNA with values which does not resembles the parent DNAs. The mutation will be set in our application through mutation rate. Mutation rate supposed to be very low.as it is something which happens rarely in reality. Having a high mutation rate is not favourable if you are looking for the desirable output. Having 100% mutation rate is nothing but Brute-forcing the DNA. In the case of 100% of mutation rate, it modifies all the newly created DNAs which most probably remove the properties of most fitness object of previous generation. Those resultants never give desired output. I will see the output of different mutation in 'Average fitness and variation mutation results' section.

Fitness function and Average Fitness

Fitness is the one determines the performance of every object (vehicle). In our case, our fitness will be lifespam of the object. Higher the value of fitness function, greater the chance of getting picked.

Fitness function of the object = Time at which the object is collided or killed - Time at which the object is created

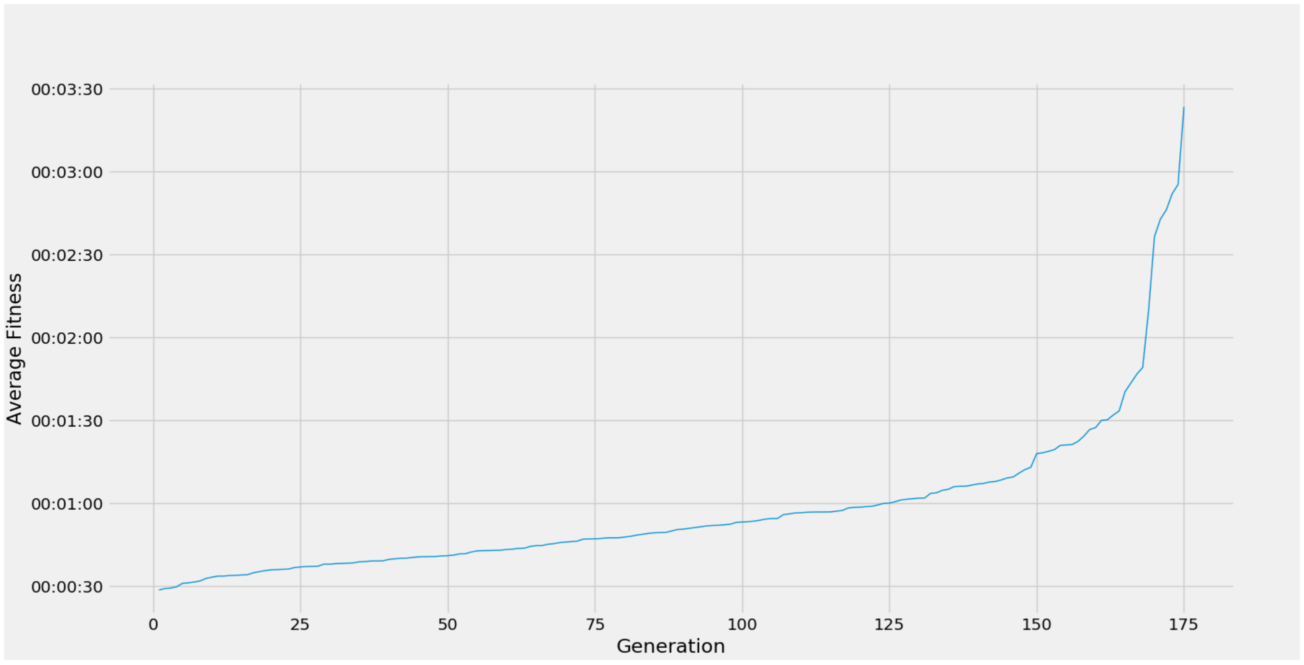
At the end of every generation, average fitness will be calculated for that generation. The resultant should be greater than average fitness of the previous generation. If the difference between average fitness of any given generation to the previous generation is negitive, it means there is a hinderance and if this continues, gradually, all the vehicles will have nearly zero lifespan.

Average fitness function of the generation = Sum of the fitness function of all objects / Population length

We can plot average fitness vs generation graph to see the effiency of our application. We will see this in next section.

On graph, we should see an increasing trend which eventually leads to nearly flat. That's the indication that it cannot be improved anymore so we stop the simulation.

Average fitness and varying mutation results



Average fitness vs Generation Graph

Initial condition for above graph.

Population: 200

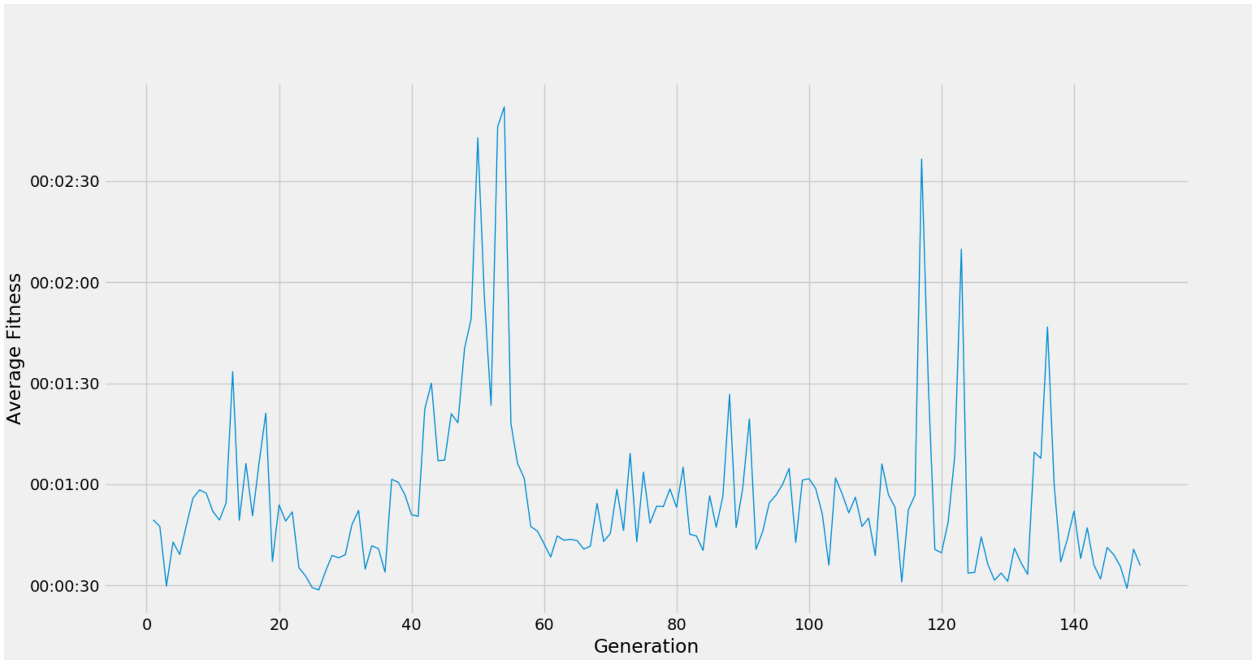
Mutation: 1%

Total Generation taken: 175

DNA

Speed Range: 5 to 20

Radius Range: 100 to 200



Average fitness vs Generation Graph

Initial condition for above graph.

Population: 200

Mutation: 50%

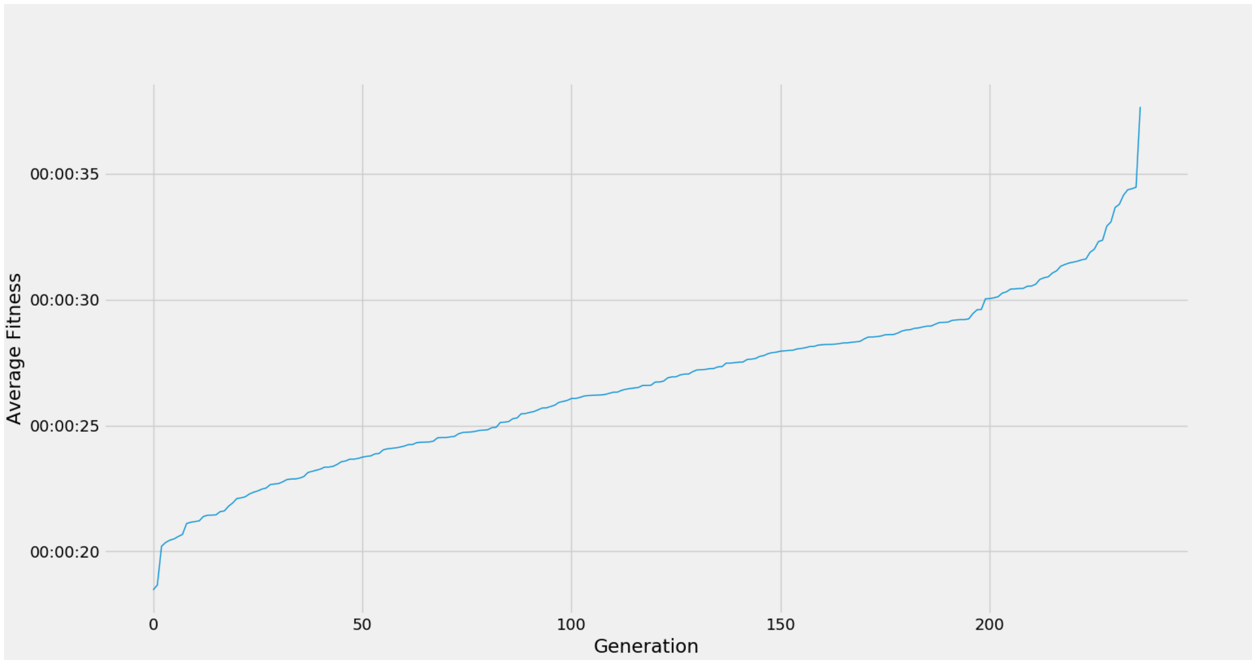
Total Generation taken: 160

DNA

Speed Range: 5 to 20

Radius Range: 100 to 200

The only variation in initial condition from this graph to the previous graph is the mutation rate. If we have very high mutation rate, we won't get desired output, so the graph continues and there is no point in continuing the simulation.



Average fitness vs Generation Graph

Initial condition for above graph.

Population: 50

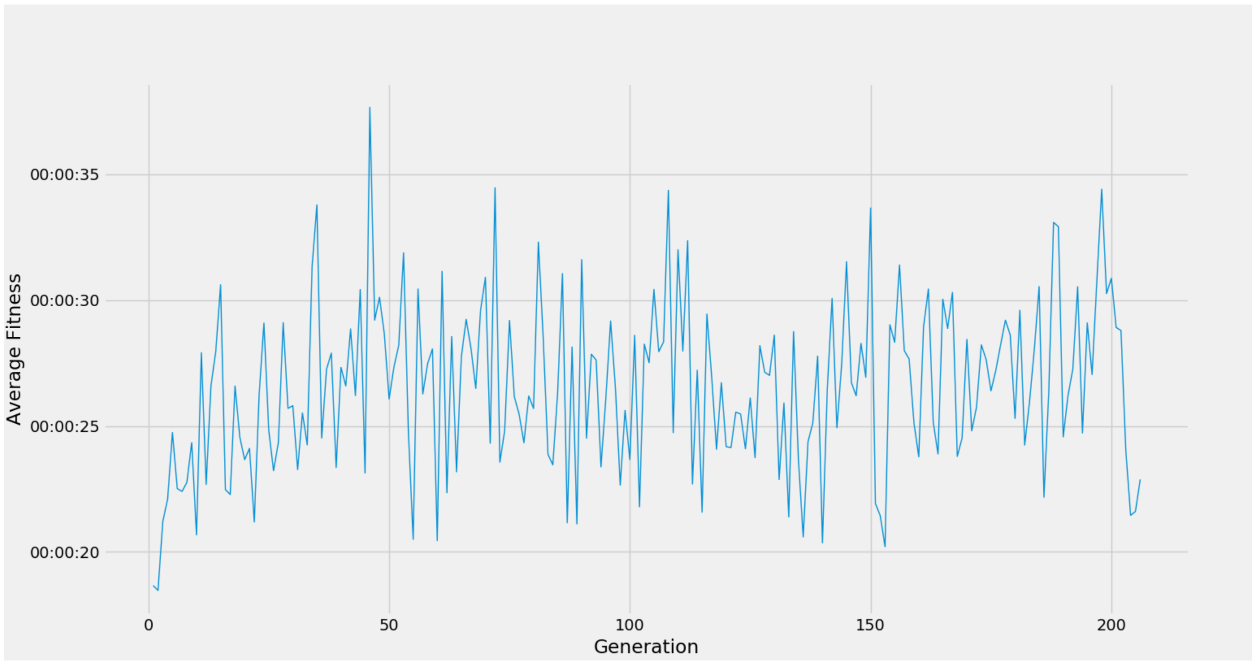
Mutation: 1%

Total Generation taken: 236

DNA

Speed Range: 10 to 20

Radius Range: 100 to 200



Average fitness vs Generation Graph

Initial condition for above graph.

Population: 50

Mutation: 75%

Total Generation taken: 236

DNA

Speed Range: 10 to 20

Radius Range: 100 to 200

The only variation in initial condition from this graph to the previous graph is the mutation rate. If we have very high mutation rate, we won't get desired output, so the graph continues and there is no point in continuing the simulation.

Recommendation and possible extension

The average death due to the traffic is around 1.24 million people in the world per year*. Because of its promising results, I would recommend this project (or improved version) to implement in all the vehicles. This will reduce the traffic related deaths occurring throughout the world. This project can be extended to accommodate the reality by adding more variables to tackle accident causing factors.

* This value is taken from the website 'Progressive Economy' http://www.progressive-economy.org/trade_facts/traffic-accidents-kill-1-24-million-people-a-year-worldwide-wars-and-murders-0-44-million/