PES
UNIVERSITY

Dissertation on

"An Efficient Genetic Algorithm Framework in Python"

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology

in

Computer Science & Engineering

Submitted by:

Bharatraj S Telkar	**01FB15ECS066**
Daniel I	**01FB15ECS086**
Shreyas V Patil	**01FB15ECS286**

Under the guidance of

Internal Guide

Prof. Chitra G M,
Professor,
PES University

January – May 2018

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

'An Efficient Genetic Algorithm Framework in Python'

is a bonafide work carried out by

Bharatraj S Telkar	**01FB15ECS066**
Daniel I	**01FB15ECS086**
Shreyas V Patil	**01FB15ECS286**

In partial fulfilment for the completion of eighth semester project work in the Program of Study Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period Jan. 2018 – May. 2018. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 8th semester academic requirements in respect of project work.

Signature	Signature	Signature
Prof. Chitra G M	Dr.Shylaja S S	Dr.B.K.Keshavan
Professor	Chairperson	Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1.____________________	____________________
2.____________________	____________________

DECLARATION

We hereby declare that the project entitled **“An Efficient Genetic Algorithm Framework in Python”** has been carried out by us under the guidance **of Prof. Chitra GM, Professor, CSE Department** and submitted in partial fulfillment of the course requirements for the award of degree of Bachelor of Technology in Computer Science and Engineering of PES University, Bengaluru during the academic semester January – May 2019. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

01FB15ECS066	**Bharatraj S Telkar**
01FB15ECS086	**Daniel I**
01FB15ECS286	**Shreyas V Patil**

ACKNOWLEDGEMENT

We would like to thank our guide Prof. Chitra G M for her continued support and guidance without which the completion of our project would have not been possible.

We would like to thank Ms. Preet Kanwal and Ms. Sangeeta V I, Project coordinators for the batch of 2015-2019, Computer Science for taking the pains to organize our internships and coordinate with a batch of over 350 students.

We would like to thank the Chairperson of the Department of Computer Science,
Dr. Shylaja S Sharath, for giving students the opportunity to undertake such a technical project and for her guidance in learning CS Engineering Subjects.

We would like to thank the Dean of Faculty, Dr. B K Keshavan, for his determined work in ensuring students had the best possible faculty to prepare them for whatever activity they decide to pursue in the future.

We would like to thank the Vice Chancellor, Dr. K N B Murthy, for his continued work towards ensuring a vibrant learning environment within PES University existed for students to excel.

We would like to thank the CEO and Pro Chancellor, Prof. Jawahar, for his continued work towards PES University and ensuring good infrastructure for learning always existed.

We would like to thank the Chairperson and Pro Chancellor, Prof. Doreswamy, for ensuring our College’s progress and ensuring it becomes a top college in India.

Finally, we would like to thank our parents, without whose support, none of these things could have been achieved.

ABSTRACT

The goal of the project is to develop a highly efficient, usable and generic genetic algorithm framework in Python unlike most GA APIs currently available. Our python genetic algorithm API “pygenetic” intends to be user friendly, generic, having presence of both high level and low level API for users to use as per their need, having presence of famous GA optimizations like adaptive mutation and being time efficient by exploiting spark parallelization. It also has options for various features which many APIs ignore but tends to be useful for research purposes such as population control – to allow/disallow population in an iteration to go beyond limit, continue evolve – to continue from previous iterations, ability to add more than one crossover/mutation with different weights to improve diversity. Our API allows has a feature to find the best ANN topology to solve a particular classification problem in lesser time. Problems including nQueens and TSP have been tested using our python API. In addition, a website for online execution of genetic algorithms is developed as part of the project so that users can conveniently execute GAs all from the UI! As of present, there exists no such website for online GA execution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

Genetic Algorithms (GA) is the field of machine learning mainly meant to solve various optimization problems. It is based on the Darwin's theory of Evolution (the survival of the fittest) and hence known as "Genetic" Algorithm. Genetic Algorithms have been used for a long time to solve optimization problems in lesser time than a brute force approach. The fitness of the chromosome is calculated based on a fitness function decided as per the problem.

The main steps involved in a Genetic Algorithm are:

1. Generate initial population of chromosomes, where each chromosome is a possible solution to the problem at hand
2. Calculate the fitness using the fitness function of every chromosome in the population.
3. Selection is carried out where only a portion of the chromosomes are selected based on the selection method applied to reproduce and create children
4. Crossover is carried out on a percentage (crossover probability) to create two new children by the mating of two parents.
5. Mutation is carried out to mutate (change a bit) a small percentage (mutation probability) of the population members.
6. Step 2 to 5 are continued till best fitness value is reached or when max iterations are over.

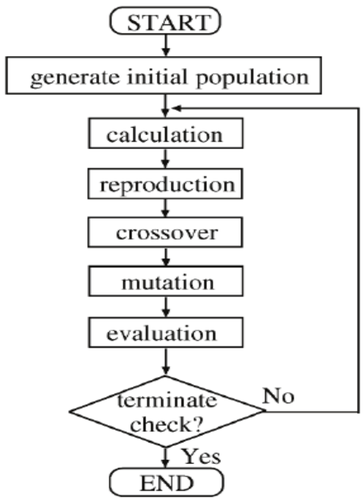


Fig 1.1. Flowchart of Genetic Algorithm

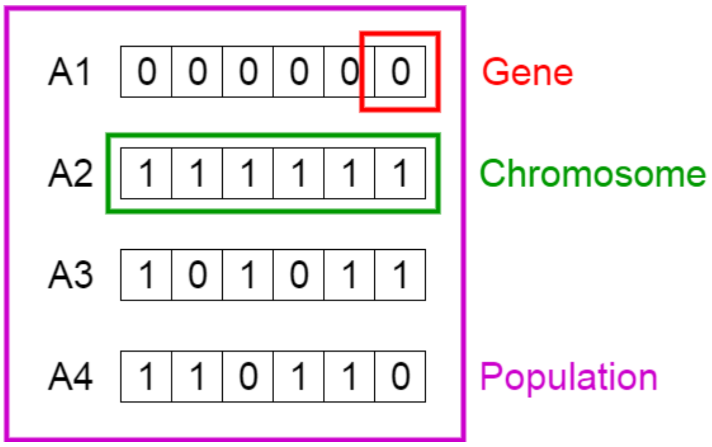


Fig 1.2 Representation of Chromosome

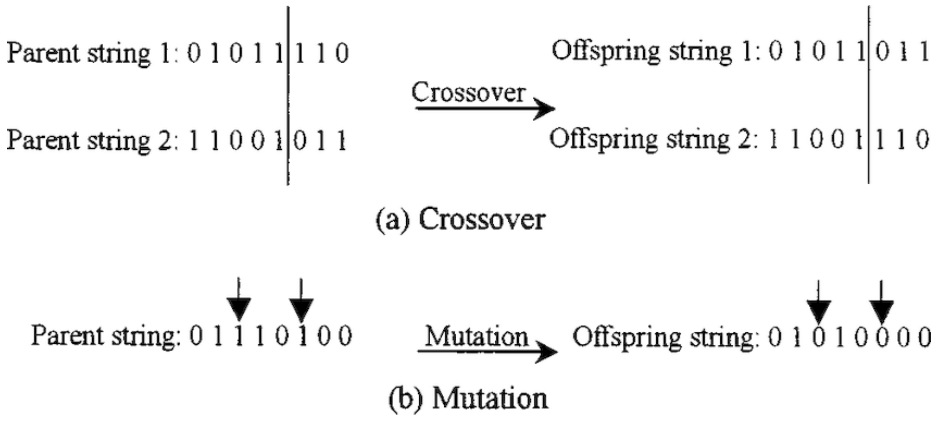

Parent string 1: 0 1 0 1 1 | 1 1 0
Parent string 2: 1 1 0 0 1 | 0 1 1
Crossover
Offspring string 1: 0 1 0 1 1 | 0 1 1
Offspring string 2: 1 1 0 0 1 | 1 1 0
(a) Crossover
Parent string: 0 1 1 1 0 1 0 0
Mutation
Offspring string: 0 1 0 1 0 0 0 0
(b) Mutation

Fig 1.3 Crossover and Mutation operation

The main steps for solving any problem using GAs include

1. Encode every possible solution/hypothesis to the problem in the form of a chromosome
2. Define the fitness function for that chromosome
3. Apply a good selection method depending on the problem to select a portion of the population.
4. Apply crossover followed by mutation to create offsprings.
5. Repeat steps 2 to 4 till termination condition is met
6. Decode the best chromosome. That is the solution to the problem.

In recent times, there has been intensive research to find good applications of GA algorithms most of which have been used to solve many NP complete problems in a short amount of time. There has also been lot of research to improve the efficiency of Genetic Algorithms to improve converging to a solution faster and research in improving efficiency by exploiting parallelization.

Apache Spark is a Big Data (BD) which can effectively handle parallelization of tasks involving large amounts of data. Apache Spark is a Big Data (BD) which can effectively handle parallelization of tasks involving large amounts of data. It has found usage in execution of many ML algorithms in order to improve efficiency.

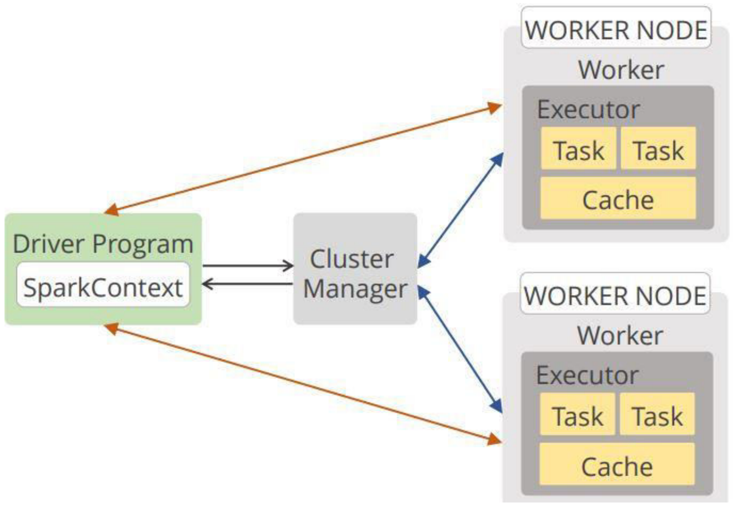


Fig 1.4 Spark System Architecture

The large datasets are first converted to RDDs (Resilient Distributed Datasets) which is effectively partitioned between different worker nodes.

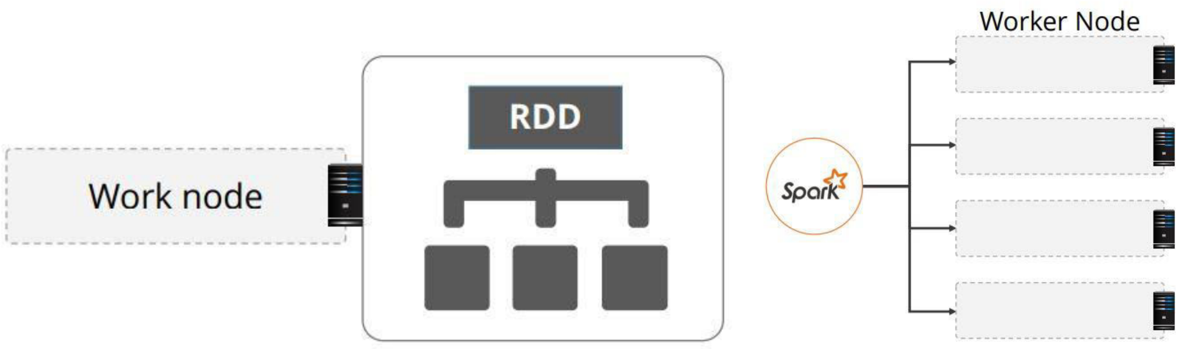


Fig 1.5 Resilient Distributed Dataset

All tasks defined in a RDD are first converted to a DAG and only when a final action is encountered is the entire DAG executed in parallel very efficiently.

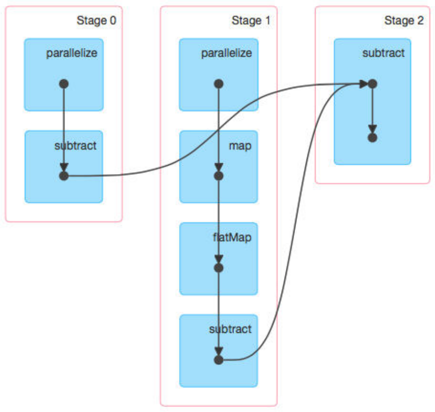


Fig 1.6 Representation of DAG in Spark

Spark has been put the test in many recent researches to check how efficiently it can run different ML algorithms. Usage of Spark in executing GAs is still a new area of exploration.

CHAPTER-2

PROBLEM DEFINITION

The aim is to develop a generic, user friendly API called "pygenetic" for execution of genetic algorithms. There already exist some APIs for genetic algorithm execution. As of present, there are mainly two worlds of genetic algorithm APIs in Python

1. DEAP which is a very generic, time efficient API which exploits parallelization
2. Other smaller APIs like gaft, genetics, pyevolve and pygalib which is very user friendly in nature.

There are however some disadvantages to both worlds of APIs – DEAP is poor in terms of user friendliness, doesn't implement any famous recent optimizations and lacks the presence of a high level API for executing simple genetic algorithms while other APIs like gaft, genetics, pyevolve and pygalib are time inefficient due to the lack of parallelization, lacks famous recent optimizations, less generic and doesn't give user much control of the GA he wishes to implement. Our python genetic algorithm API "pygenetic" intends to be user friendly, generic, having presence of both high level and low level API, having presence of famous GA optimizations and being time efficient by exploiting spark parallelization.

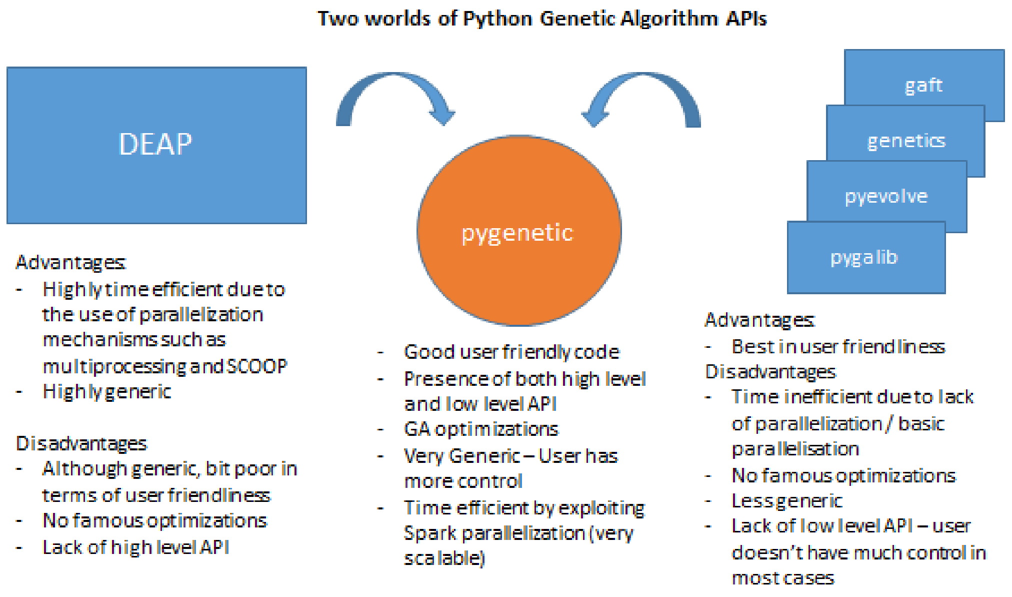

Two worlds of Python Genetic Algorithm APIs
DEAP
pygenetic
gaft
genetics
pyevolve
pygalib
Advantages
- Highly time efficient due to the use of parallelization mechanisms such as multiprocessing and SCOOP
- Highly generic
Disadvantages
- Although generic, bit poor in terms of user friendliness
- No famous optimizations
- Lack of high level API
- Good user friendly code
- Presence of both high level and low level API
- GA optimizations
- Very Generic – User has more control
- Time efficient by exploiting Spark parallelization (very scalable)
Advantages
- Best in user friendliness
Disadvantages
- Time inefficient due to lack of parallelization / basic parallelisation
- No famous optimizations
- Less generic
- Lack of low level API – user doesn't have much control in most cases

Fig 2.0 Comparison of DEAP and existing Pygenetic equivalents

CHAPTER-3

LITERATURE SURVEY

Most of the recent researches in Genetic Algorithms tend to be in figuring out how to further optimize GA algorithm execution, exploiting parallelization using big data technologies. However, there is no API in python which leverages all these research milestones and the research on parallelization is still basic with parallisation mainly being proposed for fitness calculation and selection. Our API in addition to having many GA optimizations, also has parallelization algorithm using Spark which parallelize most operations from fitness calculation to selection to crossovers and mutations.

3.1 "An adaptive genetic algorithm based on population diversity strategy" [1]

This work addresses problems of local minima stagnation that can occur when mutation rate is just a constant value. The paper defines the "diversity" of the GA population as a number which represents how diverse are the individuals in the population. Higher the diversity, there is a high variation in population members while lower the diversity means there is a less variation in population members. Mathematically, this diversity is the standard deviation of population fitness values. The paper suggests an adaptive mutation rate approach in which the mutation rate various with every generation – Higher the diversity, the mutation rate is made lesser to prevent too much randomness and lower the diversity, the mutation rate is made higher to add more diversity and prevent detecting local minimas. Thus this work prevents GAs from potentially stagnating at local minimas.

The adaptive mutation formula used is

$$p_m = M_a * (1 + \frac{f_{\max} - ASD_t}{f_{\max} + ASD_t})$$

Where p_m = adaptive mutation rate

M_a = fixed initial mutation rate

f_{min} and f_{max} = Minimum and Maximum fitness value in that generation

ASD_t = Standard Deviation of fitness values in that generation (Diversity)

Thus, with this approach we would have different mutation rates for different generations.

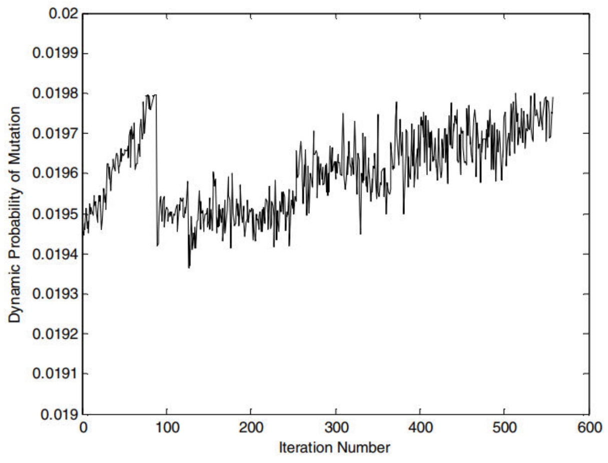
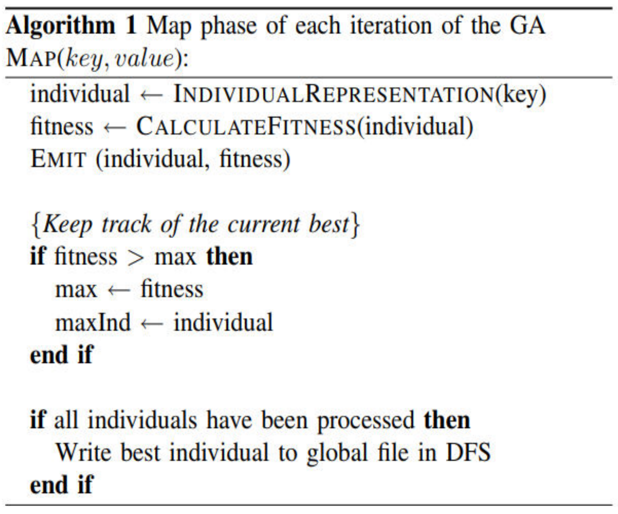


Fig 3.1 Dynamic Variation in mutation rate over iterations

3.2 "Scaling genetic algorithms using MapReduce" [2]

This work addresses problems of efficiency which GA which do not exploit parallelization suffer from. The paper mainly exploits MapReduce parallelization paradigm using which fitness values are calculated in parallel and selection is done accordingly. The paper also noted down improvements in execution time for generation of many members when executed MapReduce. This was one of the first research on exploiting Big Data tools for GA parallelization.

The Map phase mainly involved a phase where fitness values of each individual was calculated in parallel and emitted to the Reducer as depicted in the diagram.



Algorithm 1 Map phase of each iteration of the GA

```
MAP($key, value$):
  individual ← INDIVIDUALREPRESENTATION(key)
  fitness ← CALCULATEFITNESS(individual)
  EMIT (individual, fitness)

  {Keep track of the current best}
  if fitness > max then
    max ← fitness
    maxInd ← individual
  end if

  if all individuals have been processed then
    Write best individual to global file in DFS
  end if
```

Fig 3.2 Map phase for GA

The Reduce phase is made to do a selection based on all the values it receives from the Mapper. Thus this paper solves the problem of limitations of execution of Gas sequentially and suggests usage of Big Data technologies in solving GAs.

3.3 "Solving n Queens Problem using Genetic Algorithms" [3]

This work addresses problems in efficiency in solving n Queens's problem by using the backtracking algorithm. The work proposes a GA which can effectively solve the n Queens problem for different values of n.

3.3.1 Chromosome Representation

The work suggests usage of a chromosome to represent different chess board configurations For example,

For the given chess board, the chromosome is encoded as the column number in which each the queen is present in each row of the chess board.

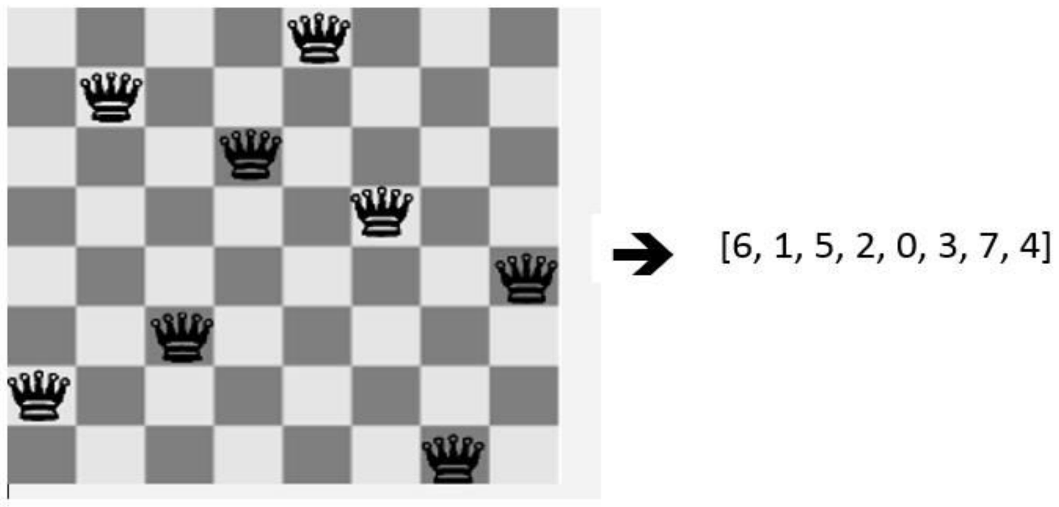


Fig 3.2 Chromosome Representation for N-Queens problem

3.3.2 Fitness Calculation

Fitness for a given chromosome is the number of non-intersecting queens for that given chess board configuration. Hence the highest fitness for a N X N chess board is N. Hence, we have a maximization GA problem with the aim of achieving fitness value N.

```
Fitness (chromosome) = Number of non-intersecting
                      queens
```

3.3.3 Crossover

The work suggests the use of a crossover technique as shown in the diagram which ensures that each element in the chromosome is a unique number. Hence traditional crossover methods such as 1-point, 2-point crossover cannot be used.

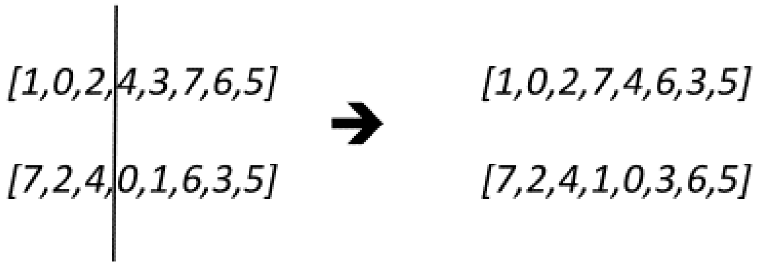


Fig 3.3 Single Point Crossover in N-Queens

In this crossover method, the first half of the chromosome is kept the same while the second half is obtained by sequentially traversing the second chromosome and adding elements only if that element is not already present.

3.3.4 Mutation

The work suggests the use of the mutation technique of swapping as shown in the diagram which also ensures that each element in the chromosome is a unique number and that there are no duplicates.

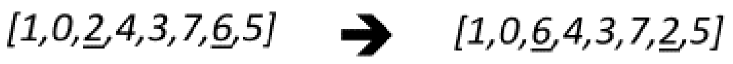


Fig 3.4 Swapping Mutation in N-Queens

3.3.5 GA Execution

The GA is executed in the standard genetic algorithm fashion. An initial population of different chess boards are created. The fitness of each of these boards are calculated and the best members are selected using any appropriate selection algorithm and then crossover and mutations are applied to them. This process repeats till the max fitness of N is reached for a N X N board. The best chromosome is the solution to the n-Queens problem.

3.4 "Travelling Salesman Problem using a Genetic Algorithm Approach" [5]

This paper addresses problems in efficiency in solving TSP problem by using the backtracking algorithm. The work proposes a GA which can effectively solve the TSP problem for large number of cities. The GA will provide an optimized solution and this may not be the best solution. Hence a tradeoff between time and accuracy is made which is indeed acceptable for this problem.

3.4.1 Chromosome Representation

The paper suggests usage of a chromosome to represent different city traversal orders.
For example,
For a 4 city problem, a chromosome can be represented as *[3,0,1,2]* which means city 3 is traversed first followed by city 0 followed by city 1 followed by city 2.

3.4.2 Fitness Calculation

Fitness for a given chromosome is the distance travelled by following the cities in the order present in the chromosome. So lesser the fitness, the better the chromosome. Hence our GA is a minimization problem.

```
Fitness (chromosome) = distance travelled by
following the cities in the order present in the
chromosome
```

3.4.3 Crossover

The work suggests the use of a crossover technique as shown in the diagram which ensures that each element in the chromosome is a unique number. Hence traditional crossover methods such as 1-point, 2-point crossover cannot be used.

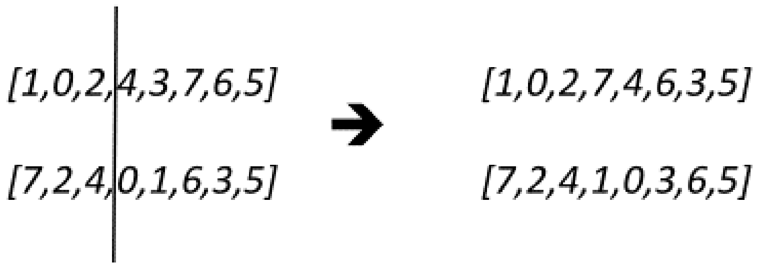


Fig 3.5 Single Point Crossover in TSP

Similar to n Queens problem

3.4.4 Mutation

The work suggests the use of the mutation technique of swapping as shown in the diagram which also ensures that each element in the chromosome is a unique number and that there are no duplicates. Similar to n Queens problem.

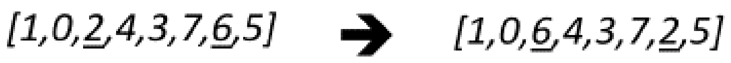


Fig 3.6 Swapping Mutation in TSP

3.4.5 GA Execution

The GA is executed in the standard genetic algorithm fashion. An initial population of different traversals are created. The fitness of each of these traversals are calculated and the best members are selected using any appropriate selection algorithm and then crossover and mutations are applied to them. This process repeats till max iterations are completed. The best chromosome is the optimized (good enough) solution to the TSP problem.

3.5 “Evolving a neural network with a genetic algorithm” [5]

This work addresses problems in neural network learning where choosing a bad topology can greatly affect how fast/slow the neural network learns. The work proposes a GA which can decide a good neural network topology which can train for a classification problem in less time. An end user can then use this topology to further train the network for longer time as per need.

3.5.1 Chromosome Representation

The work suggests usage of a chromosome to represent different network topology parameters such as number of neurons in each layer, activation functions used and optimization algorithm used to train the neural network.

For example,

For a 2 hidden layer neural network, a chromosome can be represented as *[2,"sigmoid",3,"sigmoid", "relu", "adam"]* which means first hidden layer has two neurons and sigmoid activation function, second layer has three neurons and sigmoid activation function while the output layer has "relu" activation. The neural network is trained using Adam optimization.

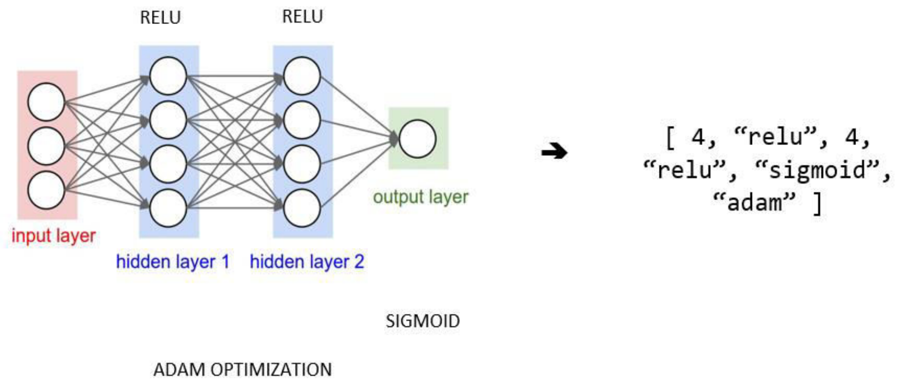


Fig 3.7 Chromosome Representation of Neural Network Topology

3.5.2 Fitness Calculation

Fitness for a given chromosome is the loss of the neural network of topology corresponding to that chromosome when trained for a given amount of time. So lesser the fitness, the better the chromosome. Hence our GA is a minimization problem.

```
Fitness (chromosome) = Neural network loss
corresponding to that topology when trained for
some amount of time
```

3.5.3 Crossover

The work suggests the use of any popular crossover methods such as 1-point, 2-point crossover, etc. Every child produced by such a crossover represents a new neural network topology which has the potential to possibly be better than others observed so far.

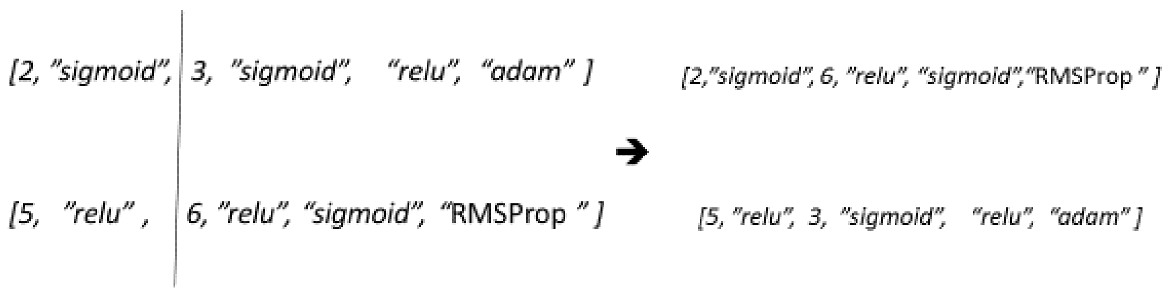
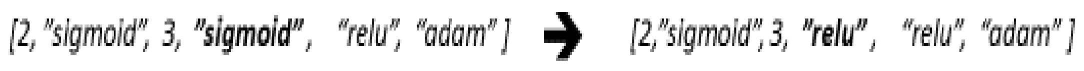

[2, "sigmoid", 3, "sigmoid", "relu", "adam"]
[5, "relu" , 6, "relu", "sigmoid", "RMSProp "]
[2,"sigmoid", 6, "relu", "sigmoid","RMSProp "]
[5, "relu", 3, "sigmoid", "relu", "adam"]

Fig 3.8 Single Point Crossover in Neural Network Topology

3.5.4 Mutation

The work suggests the use of mutations such as flipping an attribute to another of its possible values, swapping values between two same attributes of the chromosome. Each such child would represent a small change to a previous neural network which could potentially better than others observed so far.


[2, "sigmoid", 3, "sigmoid", "relu", "adam"]
[2,"sigmoid", 3, "relu", "relu", "adam"]

c

3.5.5 GA Execution

The GA is executed in the standard genetic algorithm fashion. An initial population of different neural network topologies are created. Each of these ANN are trained with the input data for a smaller amount of time than usual and the fitness values which is the neural network loss is calculated for each chromosome. The best members are selected using any appropriate selection algorithm and then crossover and mutations are applied to them. This process repeats till the max iteration limit is reached. The best chromosome is selected as the topology to be used for training the neural network.

CHAPTER-4

PROJECT REQUIREMENTS SPECIFICATION

4.1 Sequence Diagram

The following diagram describes the flow of the various modules and programs developed for this project:

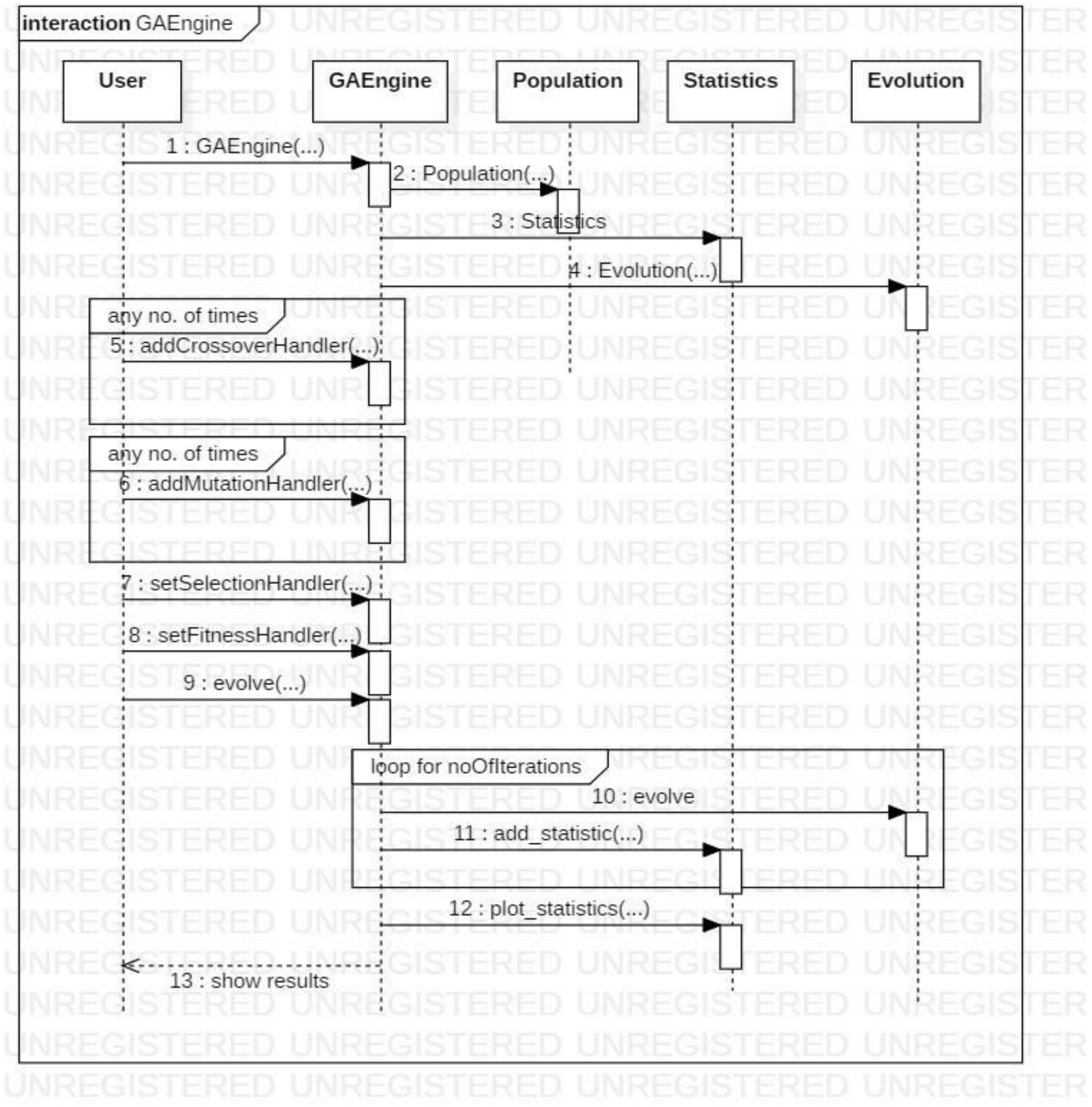


Fig 4.1 GAEngine sequence diagram

4.2 Modules

4.2.1 Module 1: Population

This module is mainly meant to maintain every generation of the genetic algorithm's population members. When a genetic algorithm is made to evolve for the first time, it is also responsible for creating all the population members based on an input Chromosome Factory which tells Population how the chromosomes are to be created. It maintains two lists – members and new_members which Evolution can use appropriately.

4.2.2 Module 2: ChromosomeFactory

This module is responsible for specifying how chromosomes for the Genetic Algorithm to be solved should be created. It contains an abstract base class which all other subclasses of ChromosomeFactory can inherit and specify the manner in which the chromosomes for the GA problem in hand are to be created. RangeChromosomeFactory can created chromosomes based on a numeric range while RangeChromosomeFactory can created chromosome based on a given regex. Users also have the ability to define their own subclasses and easily use that as part of the GA.

4.2.3 Module 3: Statistics

This module is responsible for keeping track of various statistics of every generation of the GA execution. It further provides options to visualize these statistics in the form of various graphs.

The Statistics it keeps track of in every generation include

- Maximum Fitness
- Minimum Fitness
- Average Fitness
- Population Diversity
- Mutation Rate (useful for adaptive mutation visualizations)

4.2.4 Module 4: Evolution

This module is responsible for carrying out evolution of one generation members to the next generation members. This includes carrying out fitness evaluation, selection, crossover and mutations. This module consists of a base class which many different Evolution types can subclass to define their own unique manner of carrying out Evolution. This gives the user the ability to easily define their own type of Evolutions easily. This module gives users two options – to evolve the GA sequentially or using pySpark parallelization.

4.2.5 Module 5: Utils (SelectionHandlers, CrossoverHandlers, MutationHandlers, Fitness)

This module contains all standard types of selection handlers, crossover handlers, mutation handlers and fitness functions which users can use in their GA without having to code them from scratch. These functions are given all details of the GA as one of it parameters to make them as generic as possible so users can define highly generic handlers based on any detail present in the GA.

The Standard Selection Handlers include:

- Random: select random chromosomes
- Best: select best chromosomes
- Worst: select worst chromosomes
- Tournament: ‘k’ individuals are selected in random and the best among them is chosen. This process is repeated for the required number of members needed.
- Roulette: selection of chromosomes is based on relative finesses.
- Rank: The selection of chromosomes depends on its rank value and directly on fitness values.

The Standard Crossover Handlers include:

- One Point: Crossover at one cut point
- Two Point: Crossover at two cut points
- Distinct: Ensures no duplicate elements in the chromosome

The Standard Mutation Handlers include:

- Swap: Swapping two genes in a chromosome
- Bit Flip: Flip a bit of a gene in a chromosome

4.2.6 Module 6: GAEngine

This module is responsible for executing the entire GA and is the main module used for genetic algorithm input from the end user. It includes options to carry out different optimizations such as adaptive mutation, hall of fame chromosome injection and efficient iterations halt. It provides options for population control to enable/disable maintaining same population size in every iteration. It also allows users to add more than one crossover/mutation handler with different weights so as to improve the diversity of newer generations. It also gives the option to continue from previous already executed evolutions. Hence, this module intends to be a very efficient/user friendly API access module for the end user.

4.2.7 Module 7: SimpleGA

This module is the high level GA API which can be used for executing simple genetic algorithms. It allows users to execute simple GAs with just a single line of code. However, this API is not generic and can only be used when the problem to be solved can be modelled to a very simple genetic algorithm.

4.2.8 Module 8: ANNEvolve

This module which is part of the GA framework enables users to find the best topology to train their neural networks with. It itself contains a GAEngine object which carries out the required Genetic Algorithm execution. The ANNEvolve is just a wrapper around this entire GA algorithm use case.

4.2.9 GUI Modules

Main Endpoints needed as part of the GA Online Execution Website

1. /ga_init: to send details of the genetic algorithm and get first generation details
2. /ga_evolve: to continue to get all the next generation details of the genetic algorithm
3. /plot_fitness: to return statistic graph which can be displayed on GUI
4. /download_code: to return equivalent code based on user form inputs

4.3 Constraints

4.3.1 Design Constraints

1. Only limited number of optimizations – adaptive mutation, hall of fame injection and efficient iteration halt are part of the API
2. Only one other ML application solved using GA (ANN Topology) is expected to be inbuilt into the API.
3. The more powerful Low Level API will take more lines to code than the less powerful High Level API.
4. Only 1D chromosomes and 1D crossover/mutations are part of this API. Users can however continue to use the API by defining his own 2 dimensional/custom chromosome factory, crossover, mutation. These commonly used chromosomes and operations can be made part of the API later in the future.

4.3.2 System Constraints

1. For executing Evolution using Spark, there is a need to install multiple packages – Apache Spark, Scala, JVMs and python package pySpark.
2. Linux based Operating Systems, Mac OSX and Windows are the operating systems LLVM is guaranteed to work on.
3. Python Package pygenetic works on Python 3.6
4. Website requires Flask in order to carry out online execution of Genetic Algorithms.

4.4 Product Perspective

pygenetic intends to be a Genetic Algorithm Framework which is very efficient, generic Framework where users can easily execute all variations of Genetic Algorithms very easily. It is a useful way to explore the problem solving ability of Genetic Algorithms. It supports usage of Spark to explore improvements in GA by using large populations. The GA website can be used by anyone to easily execute GAs from the GUI itself.

4.4.1 User Characteristics

There are two sections of users that will benefit from pygenetic and the GA website:

1. Researchers, programmers, company employees / entrepreneurs can all use our genetic algorithm framework while experimenting with different Machine Learning Algorithms and observing performance.
2. Teachers, students and even people new to ML and programming can get exposure and an understanding of genetic algorithms by playing around with our easy to use GA online execution website.

4.4.2 General Constraints, Assumptions and Dependencies

1. Speed of execution using Spark ultimately depends on the number of workers deployed, amount of memory in each worker, how large the population size actually is.
2. Dependence on pySpark for parallelization and matplotlib for statistic visualizations.
3. Assumption that fitness types can be MAX, MIN or EQUAL and that crossover/mutation operations are expensive and can be made efficient by parallelizing it.
4. Dependence on Flask as Website Backend Server and Travis CI for continuous integration.

4.4.3 Risks

1. Difficulty in parallelizing all operations for different kinds of problems to be solved using GA.
2. Execution of malicious code from the GUI. Need to have good input validation checks.
3. The API being difficult to use due to lack of documentation. Developing good documentation needs to be prioritized as well as part of the project.

CHAPTER-5

SYSTEM REQUIREMENTS SPECIFICATION

5.1 Functional Requirements

5.1.1 ChromosomeFactory interface for producing chromosomes

Pygenetic depends on a ChromosomeFactory instance for chromosome creation. The interface should be in such a way that users can easily create different chromosome factories which can be used for GA execution quite easily. RangeChromsomeFactory should produce chromosomes where each gene can take a value between a given min and max value while RegexChromosomeFactory should produce chromosomes based on an input regex.

5.1.2 Efficient Evolution Execution using Spark Parallelization

Usage of an efficient Spark DAG to parallelize the operations of fitness calculation, selection, crossover and mutation. This feature would allow processing populations of large size in lesser time than when done sequentially.

5.1.3 High Level and Low Level API

API provides users with both a high level API for executing simple GA in a single line of code and a low level API which gives users more control over what he wants to do and what he wants to change in his GA execution.

5.1.4 Supports Adaptive Mutation

API provides users with option to execute genetic algorithms with adaptive mutation rate – mutation rate which changes based on how diverse the individuals of the current population are. This ensures that the GA does not converge to a local minima.

5.1.5 Supports Hall of Fame Injection

API provides users with option to execute genetic algorithms with hall of fame injection optimization – the best encountered chromosome is injected back into the population after every 20 generations. This ensures that the GA does lose the good chromosomes while doing different selection methods.

5.1.6 Supports Efficient Iteration Halt

API provides users with option to halt genetic algorithms efficiently – when the same best fitness appears for 20 consecutive generations. This ensures that the GA does not continue executing for large number of generations unnecessarily and hence improves efficiency.

5.1.7 Supports Visualization of Statistics

API provides users with a way to visualize the statistic of different generation population. It provides graphs to visualize the best, worst, average finesses, diversities and mutation rates over different generations.

5.1.8 Supports Population Control

API provides users with a way to enable/disable population control option which decides whether the same population size is maintained or not in every generation of the GA execution. Since users may define sections handlers, custom evolutions, etc, the number of chromosomes in a population can go beyond/below the population size. Enabling this option ensures that same population size is maintained. Disabling it can be done when executing various GAs especially for research purposes.

5.1.9 Supports Multiple Crossovers and Mutations

API provides users with a way to add more than one crossover/mutation handler for one GA execution. Different weights can be provided to each handler. This ensures that there is good diversity in each GA generation.

5.1.10 Provides Standard Selection, Crossovers, Mutations and Fitness Functions

API provides users with a variety of Standard Selection, Crossovers, Mutations and Fitness Functions so that the end user need not have to code them from scratch every time.

5.1.11 Provides continue evolve feature

API provides users with a continue evolve feature to continue evolutions for some number of generations after a previous evolution has completed. Suppose, user executed for 100 generations, then realizes problem might have solved if he ran it for 120 generations, he doesn't need to start all over again. He can just continue evolve for 20 generations. This improves efficiency.

5.1.12 Website: Execute GA from UI and provide Statistics

The Genetic Algorithm Website allows users to execute GAs from the GUI itself and provides statistics at the end in the form of graphs. This allows users to experiment and solve problems with GAs from the UI itself. It is also useful for education purposes.

5.1.13 Website: Download Equivalent Code

The Genetic Algorithm Website allows users to download the equivalent Python GA code derived from all the user form inputs pertaining to the Genetic Algorithm.

5.2 Non-Functional Requirements

- Pygenetic is a user friendly and generic GA python API. Unlike most APIs which make tradeoffs between user friendliness and genericity, pygenetic seeks to strike a balance between the two.
- The API should be well documented
- The GA Website should be very intuitive and easy to use.
- The GA Website should be secure and reject invalid inputs.

5.2.1 Dependencies

- pySpark, Apache Spark, Scala, JVM
- matplotlib
- Flask
- Numpy

5.2.2 Assumptions

- The user using pygenetic must read the documentation and familiarize themselves with the API and its features.

- User has Spark currently configured either in his local setup or on cloud clusters when using Spark parallelization.

5.3 Hardware Requirements

The python API pygenetic can be run on any Linux based operating system (Ubuntu, etc), MAC OS and Windows.

5.4 Software Requirements

- Python 3.6 + : pygenetic requires Python version 3.6 and above
- Apache Spark, Scala, JVM: Needed for Spark parallelization when executing GAs.
- Matplotlib: For visualization of statistics
- Keras with TensorFlow backend: For ANN Topology using GA execution
- Numpy: for selection of chromosomes to undergo crossover, mutation, ANN input fata

5.5 User Interfaces

Users can input various details regarding the genetic algorithm they wish to execute in the GUI form and can execute the GA from the UI. The client will poll the server for every generation details and will display results and statistics graphs at the end. Users can also download equivalent python code of the various user inputs from the UI.

5.6 Performance Requirements

1. The GA website should be able to handle at least 100 consecutive users without any failures.
2. The GA website should return responses in at most 3 seconds from when request is sent under ideal conditions.
3. The GA API should perform efficiently without any unnecessary overheads and should outperform most other current python GA APIs.

5.7 External Interfaces

5.7.1 GA Website

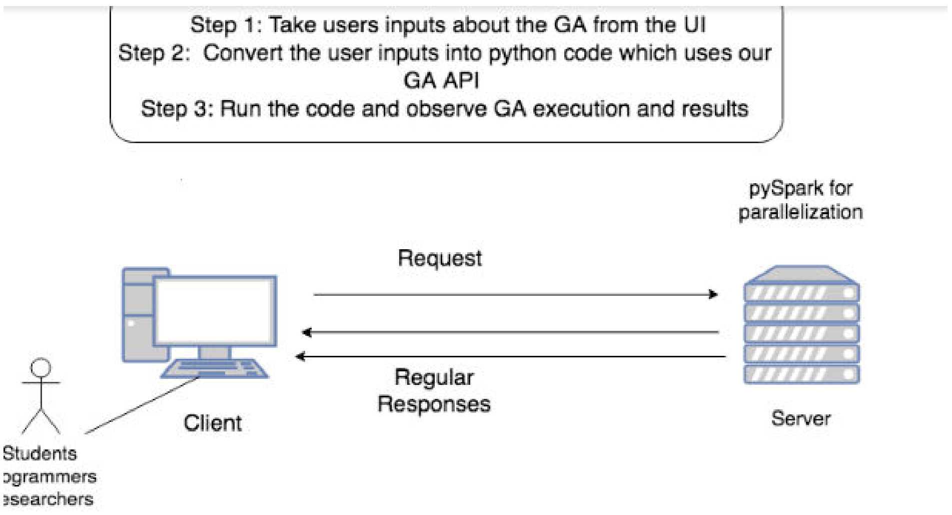

Step 1: Take users inputs about the GA from the UI
Step 2: Convert the user inputs into python code which uses our GA API
Step 3: Run the code and observe GA execution and results
pySpark for parallelization
Request
Regular Responses
Client
Server
Students
ɔgrammers
ɘsearchers

Fig 5.1 GA Client Server Architecture

5.7.1 GA API

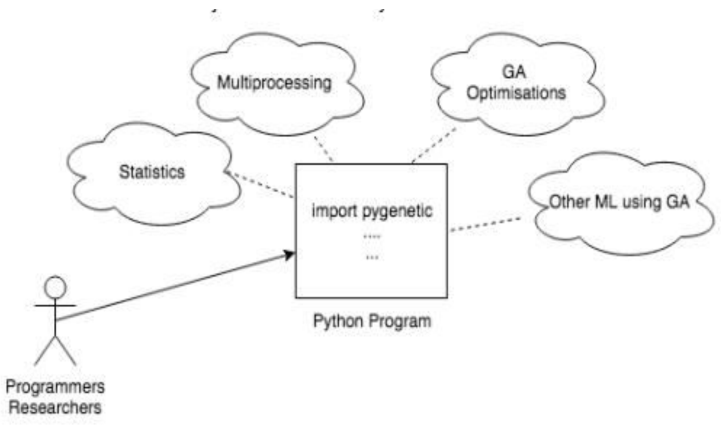


Fig 5.2 Modules of GA Framework

5.8 Help

Documentation for pygenetic is hosted online at readthedocs.org, which is one of the most famous websites for hosting documentation and tutorials. The entire API documentation is available at https://pygenetic.readthedocs.io/en/latest/. Also issues can be posted at our GitHub repository https://github.com/danny311296/pygenetic/ for any help.

CHAPTER-6

SYSTEM DESIGN

The user of pygenetic can interact with it in 2 major ways:

1. Through the website. The user visits the website hosted on our server. There, the user can input values in the places specified and either run the corresponding implementation or just download the code that is generated for the parameters he/she has entered. The user can also load a specific implementation he/she had saved earlier.
2. Through the console. Here, the user needs to have pygenetic installed. The user needs to write a python script using the modules included in pygenetic and run the script on his/her console to get the results.

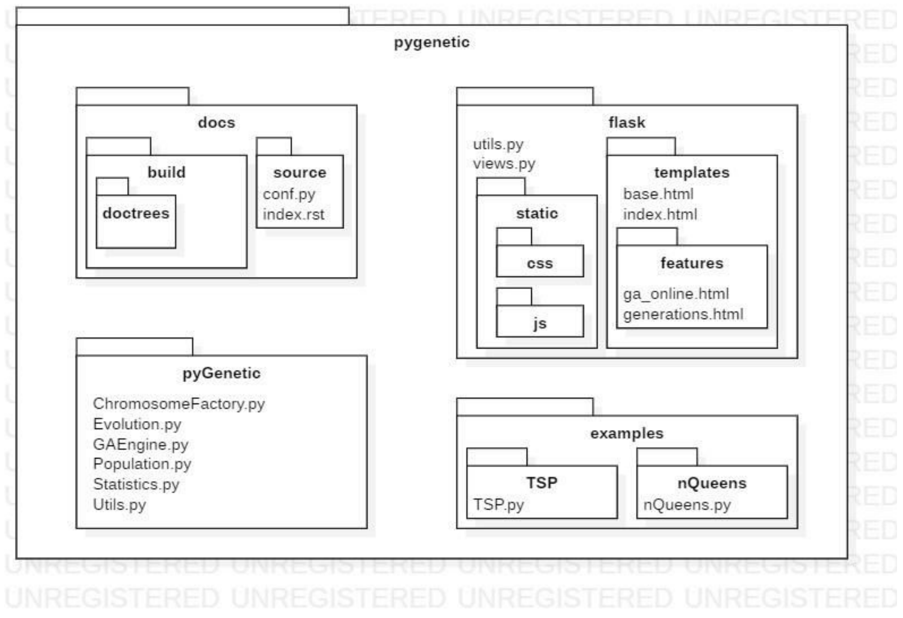


Fig 6.1 Packaging diagram

The above packaging diagram describes all the modules that are included in the pygenetic package. There is the core package “pyGenetic” containing all the working modules that perform the all the functions. Examples are also imcluded for common problems like TSP, n-Queens problem and others to aid the user in understanding the functioning of pygenetic better.

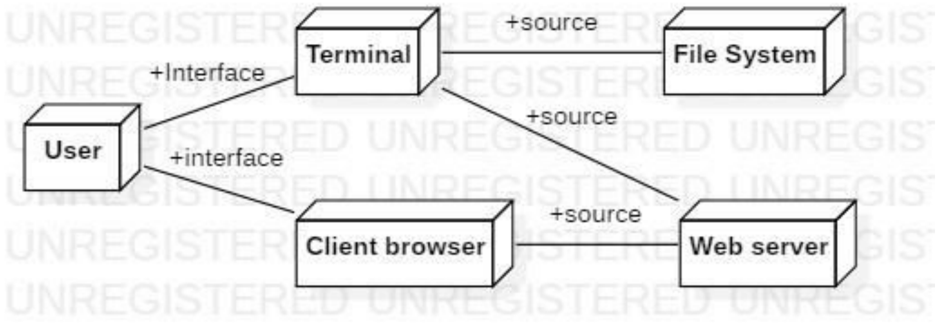


Fig 6.2 Deployment diagram

The above deployment diagram depicts the packaging of the whole system from the perspective of the user. It shows the 2 ways in which the user can interact with the system. For the user, he/she is interacting with either the terminal (in the case of using pre-installed pygenetic or referencing the modules with active internet connection) or the browser (in the case user is visiting the website). Internally, the codebase for both the local execution and website execution will be the same.

CHAPTER-7

DETAILED DESIGN

7.1 Master Class Diagram

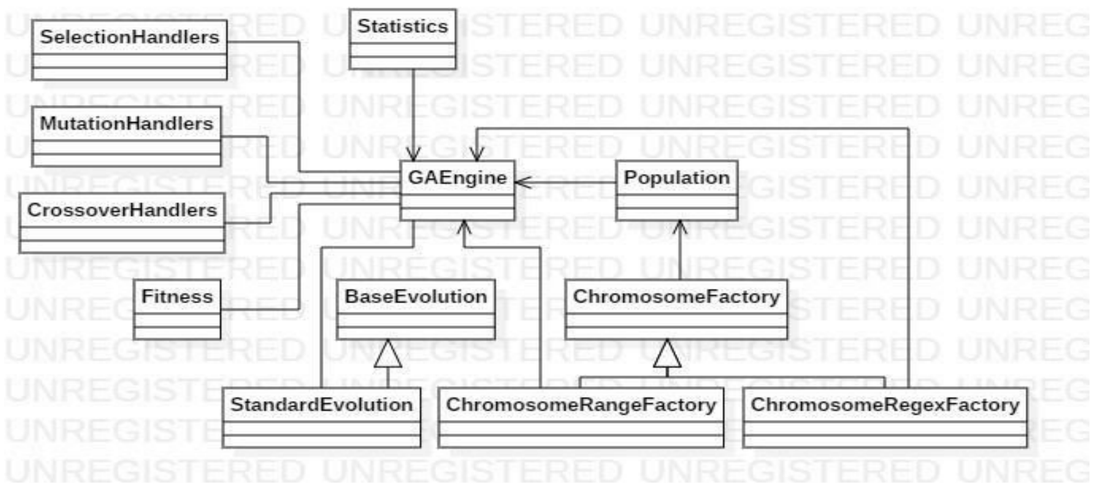


Fig 7.1 Master Class Diagram of GA Framework

Figure 7.1 depicts the entire structure of the modules that have been developed for the project. GAEngine requires a ChromosomeFactory using which it creates a Population and using user provided Selection, Mutation, Crossover and Fitness functions, it makes the population undergo evolution and this is how the GA in brief gets executed.

The below figure shows the class diagram in a little more detail – the list of some of the attributes each class contains, some of its methods, etc.

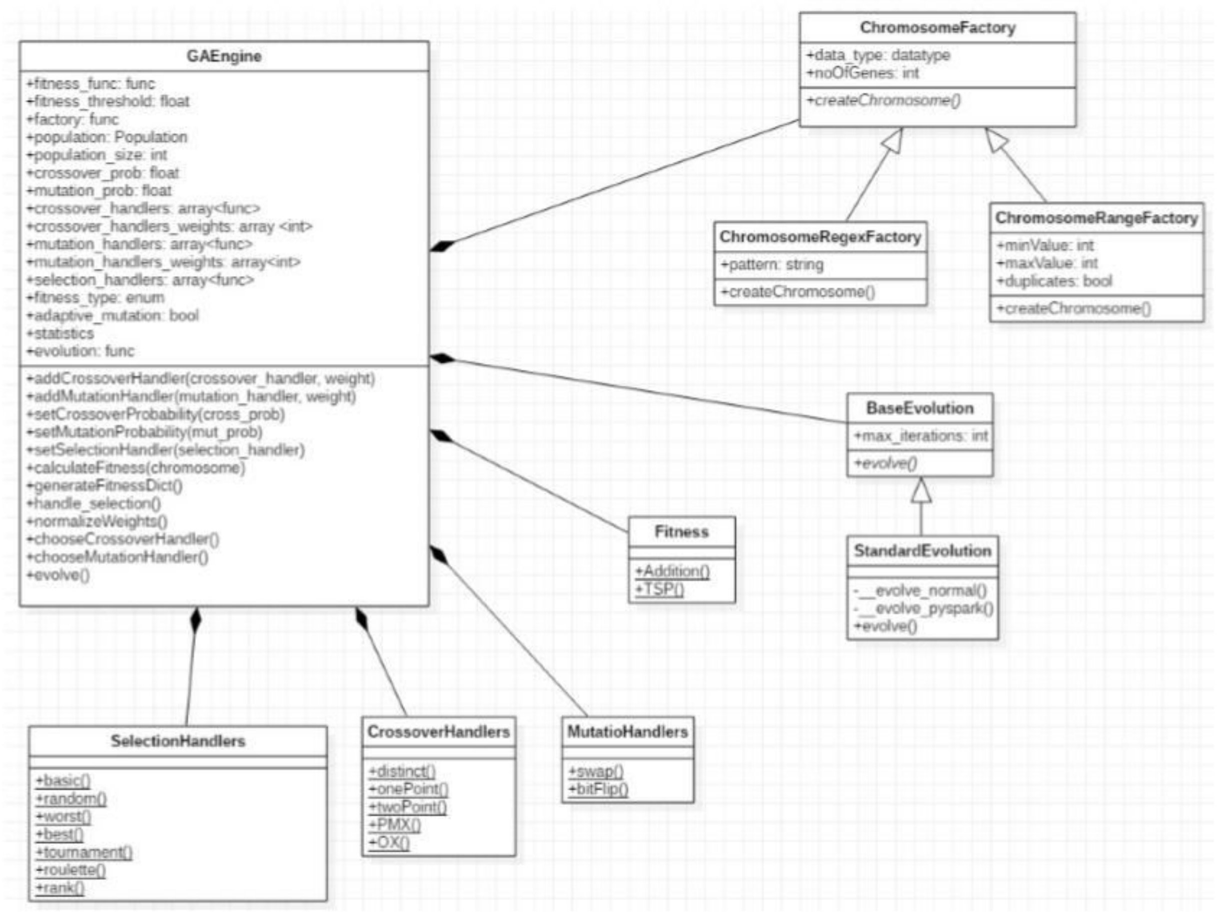


Fig 7.2 Detailed Class Diagram of GA Framework

7.1.1. Module 1: Population

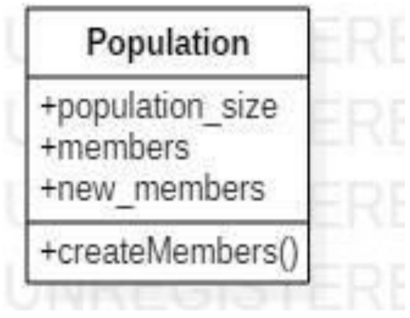


Fig 7.3 Population Class Data members and Methods

This module is mainly meant to maintain every generation of the genetic algorithm's population members. When a genetic algorithm is made to evolve for the first time, it is also responsible for creating all the population members based on an input Chromosome Factory which tells Population how the chromosomes are to be created. It maintains two lists – members and new_members which Evolution can use appropriately.

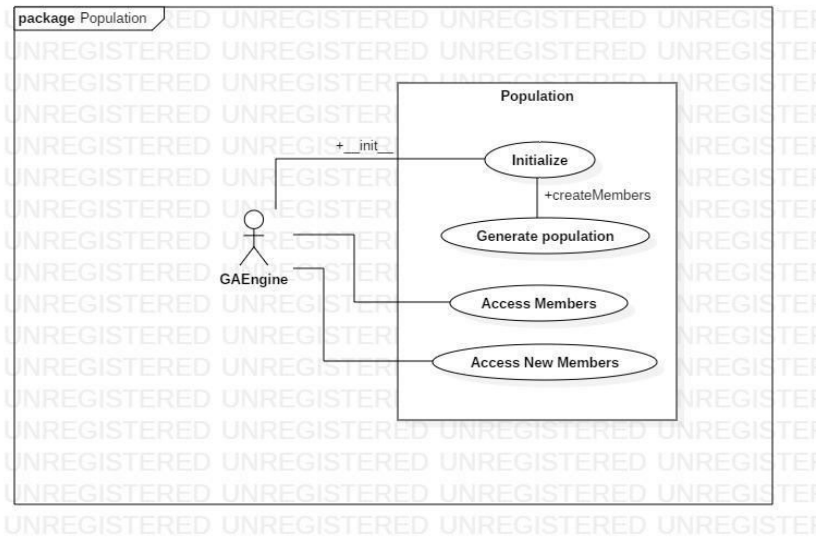


Fig 7.4 Use case Diagram of GAEngine

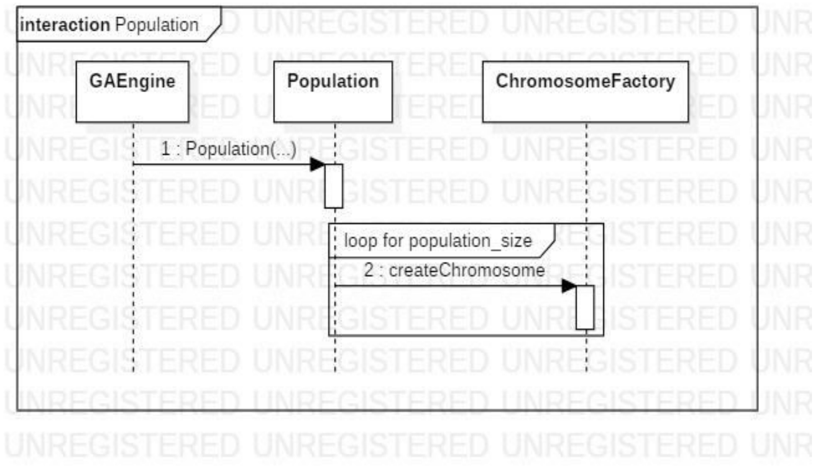


Fig 7.5 Interation of GAEngine with Population and ChromosomeFactory Class

7.1.2. Module 2: ChromosomeFactory

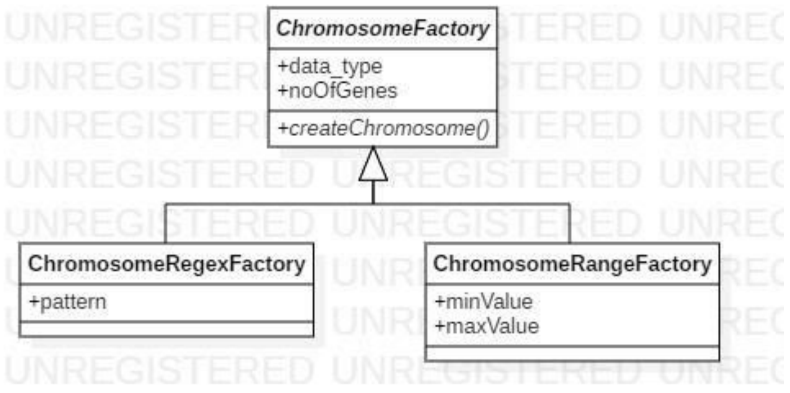


Fig 7.6 Inheritance of ChromosomeFactory

This module is responsible for specifying how chromosomes for the Genetic Algorithm to be solved should be created. It contains an abstract base class which all other subclasses of

ChromosomeFactory can inherit and specify the manner in which the chromosomes for the GA problem in hand are to be created. RangeChromosomeFactory can created chromosomes based on a numeric range while RangeChromosomeFactory can created chromosome based on a given regex. Users also have the ability to define their own subclasses and easily use that as part of the GA.

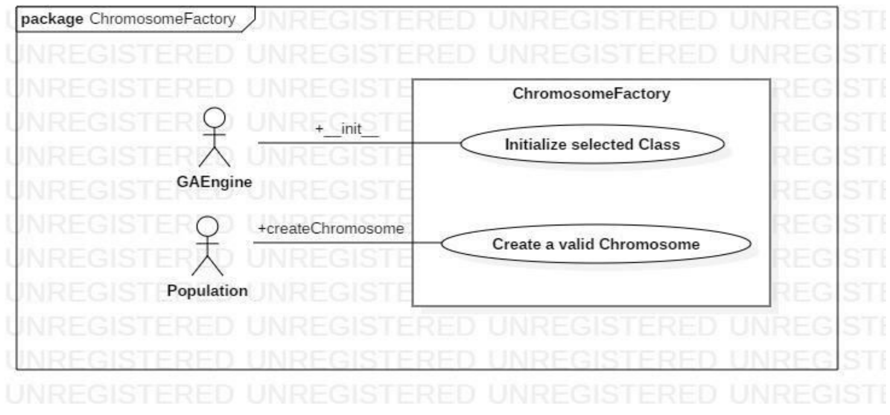

package ChromosomeFactory
ChromosomeFactory
GAEngine
+__init__
Initialize selected Class
Population
+createChromosome
Create a valid Chromosome
UNREGISTERED

Fig 7.7 Use Case diagram of ChromosomeFactory

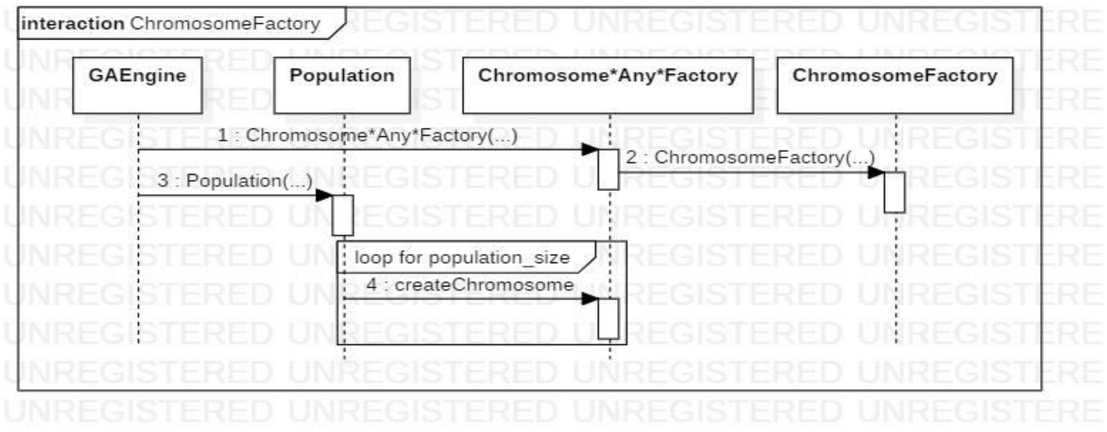


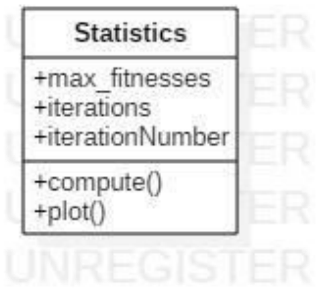
Fig 7.8 Sequence Diagram of ChromosomeFactory

7.1.3. Module 3: Statistics

This module is responsible for keeping track of various statistics of every generation of the GA execution. It further provides options to visualize these statistics in the form of various graphs.

The Statistics it keeps track of in every generation include

- Maximum Fitness
- Minimum Fitness
- Average Fitness
- Population Diversity
- Mutation Rate (useful for adaptive mutation visualizations)



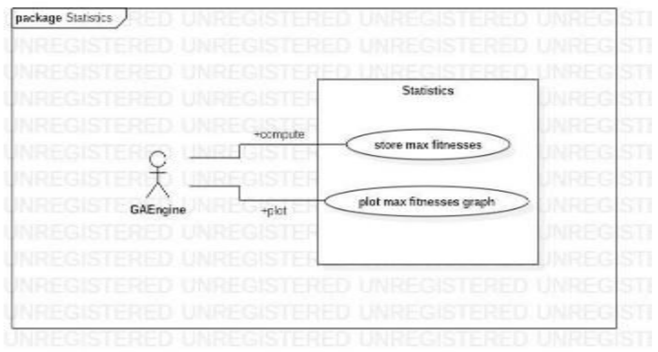


Fig 7.9 Use Case diagram of Statistics

7.1.3. Module 4: Evolution

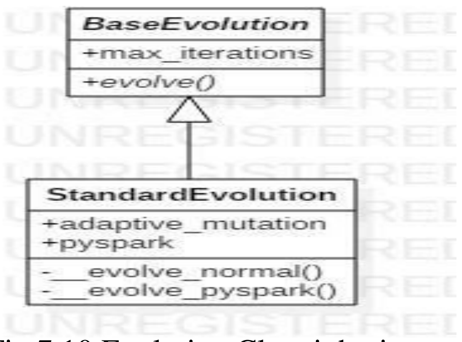


Fig 7.10 Evolution Class inheritance

This module is responsible for carrying out evolution of one generation members to the next generation members. This includes carrying out fitness evaluation, selection, crossover and mutations. This module consists of a base class which many different Evolution types can subclass to define their own unique manner of carrying out Evolution. This gives the user the ability to easily define their own type of Evolutions easily. This module gives users two options – to evolve the GA sequentially or using pySpark parallelization.

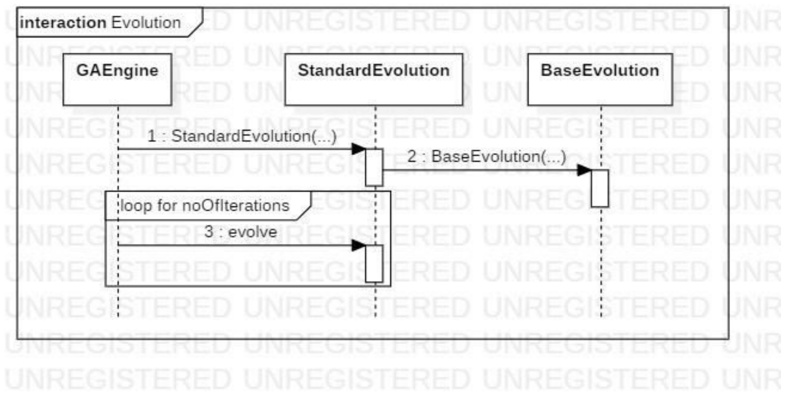


Fig 7.11 Interaction of GAEngine with Evolution Class

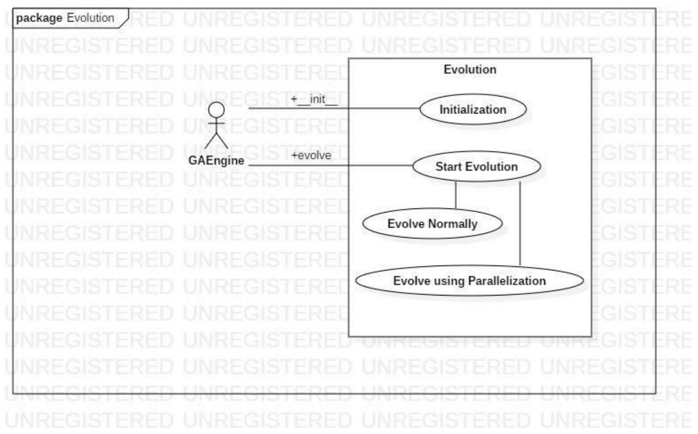


Fig 7.12 Use Case Diagram of Evolution

7.1.5. Module 5: Utils

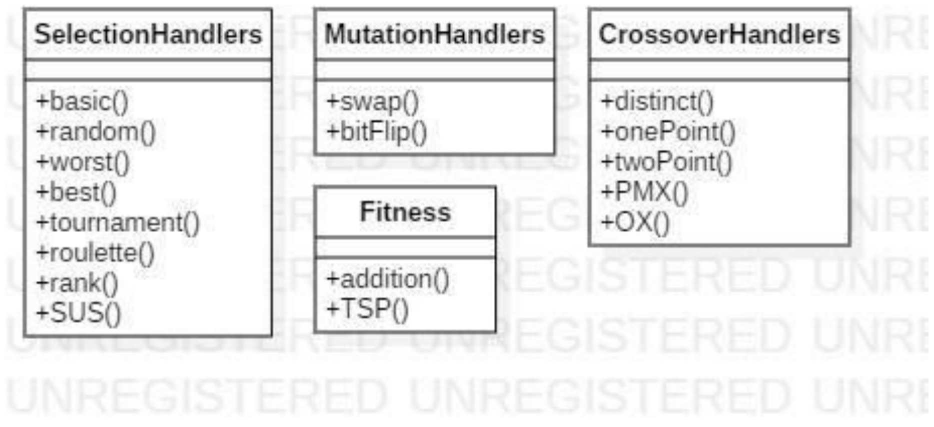


Fig 7.13 Class Details of Utils Module

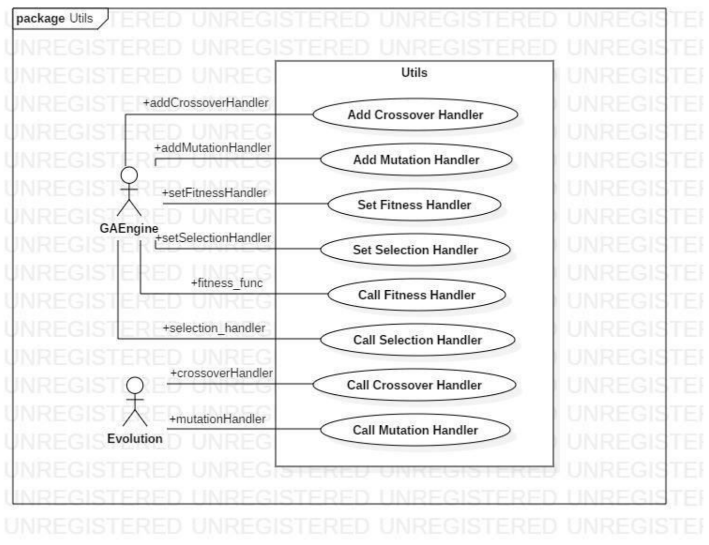


Fig 7.14 Use Case diagram of Utils

This module contains all standard types of selection handlers, crossover handlers, mutation handlers and fitness functions which users can use in their GA without having to code them from scratch. These functions are given all details of the GA as one of it parameters to make them as generic as possible so users can define highly generic handlers based on any detail present in the GA.

7.1.6. Module 6: GAEngine

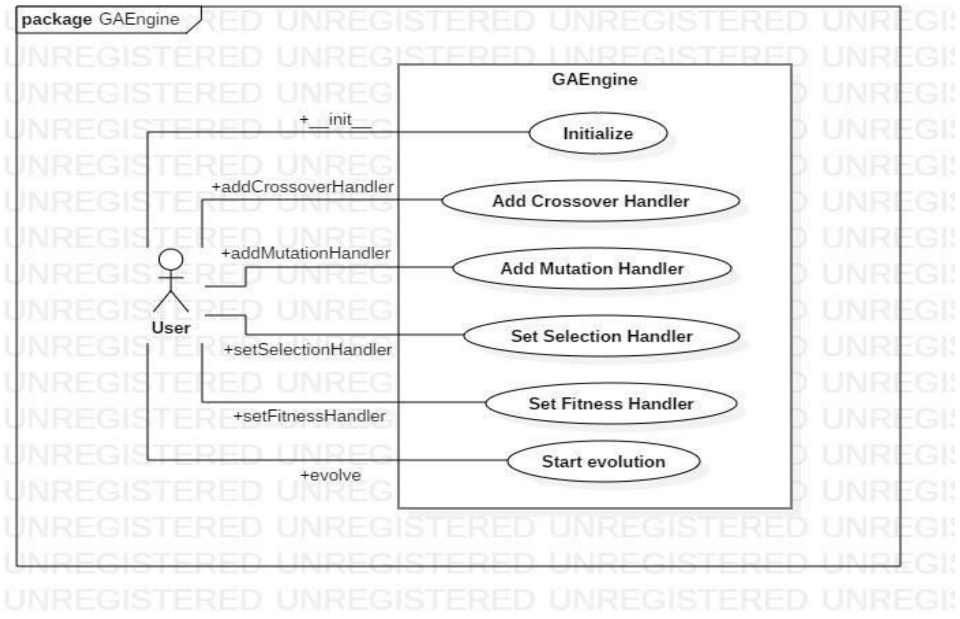


Fig 7.14 Use Case of GAEngine

This module is responsible for executing the entire GA and is the main module used for genetic algorithm input from the end user. It includes options to carry out different optimizations such as adaptive mutation, hall of fame chromosome injection and efficient iterations halt. It provides options for population control to enable/disable maintaining same population size in every iteration. It also allows users to add more than one crossover/mutation handler with different weights so as to improve the diversity of newer generations. It also gives the option to continue from previous already executed evolutions. Hence, this module intends to be a very efficient/user friendly API access module for the end user.

7.1.7. Spark DAG Used for parallelization

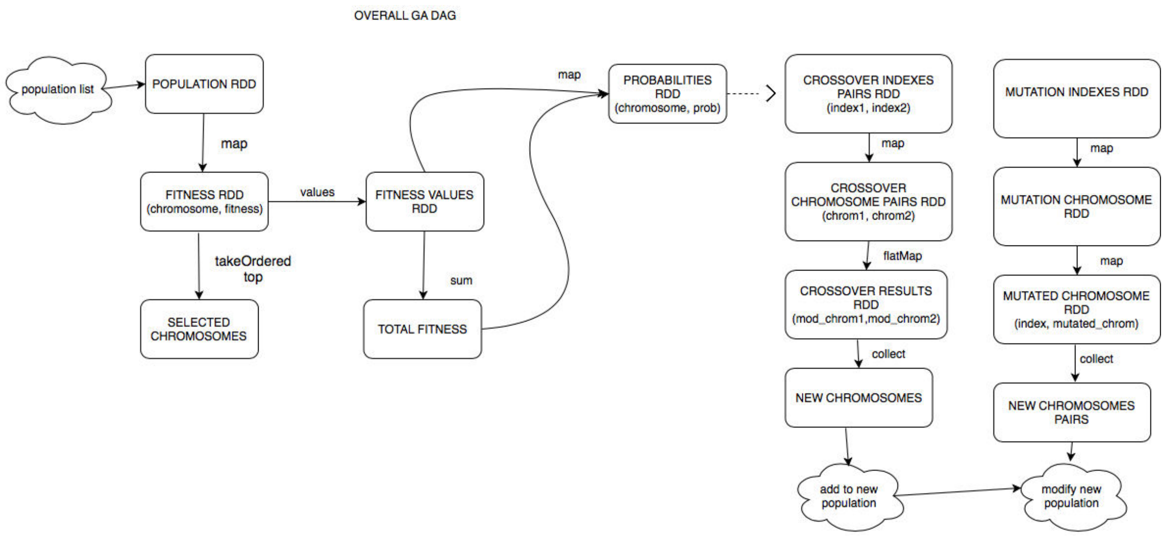


Fig 7.15 DAG used in Framework

- The population members is first made into a population RDD. This RDD is mapped to its corresponding fitness values and takeOrdered/top is applied on it to select the chromosomes accordingly. [Parallelize Selection]
- The Fitness RDD is summed up to get total_fitness value. This is used in the creation of the relative probabilities RDD – which contains the relative fitness of each chromosome. A crossover index pair RDD is created using these probabilities. This crossover index pair RDD is mapped to its corresponding crossover chromosome pairs and then mapped to the crossover results. [Parallelize Crossovers]
- A mutation index pair RDD is also created. This mutation index pair RDD is mapped to its corresponding mutation chromosome along with the index and then the corresponding chromosomes are modified accordingly. [Parallelize Mutations]

7.2 User Interfaces

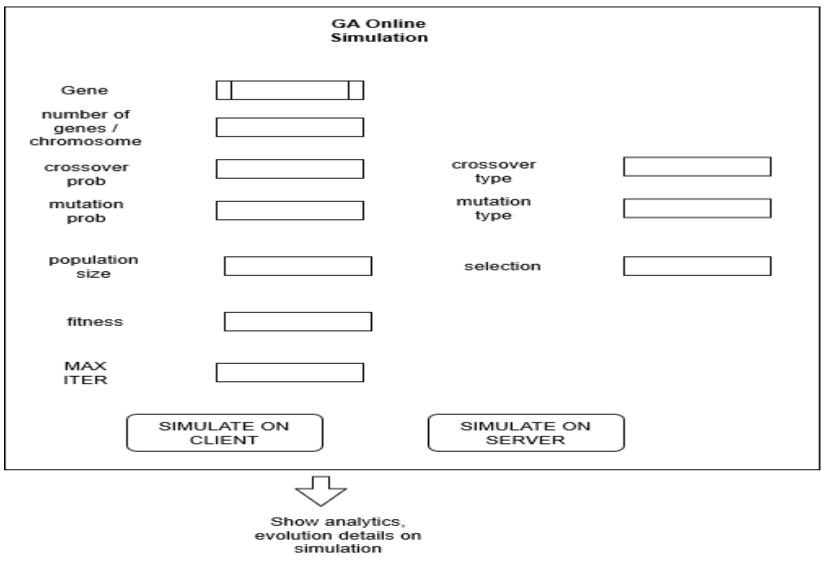


Fig 7.16 GUI for Framework

The User Interface should be a simple to use UI where users can

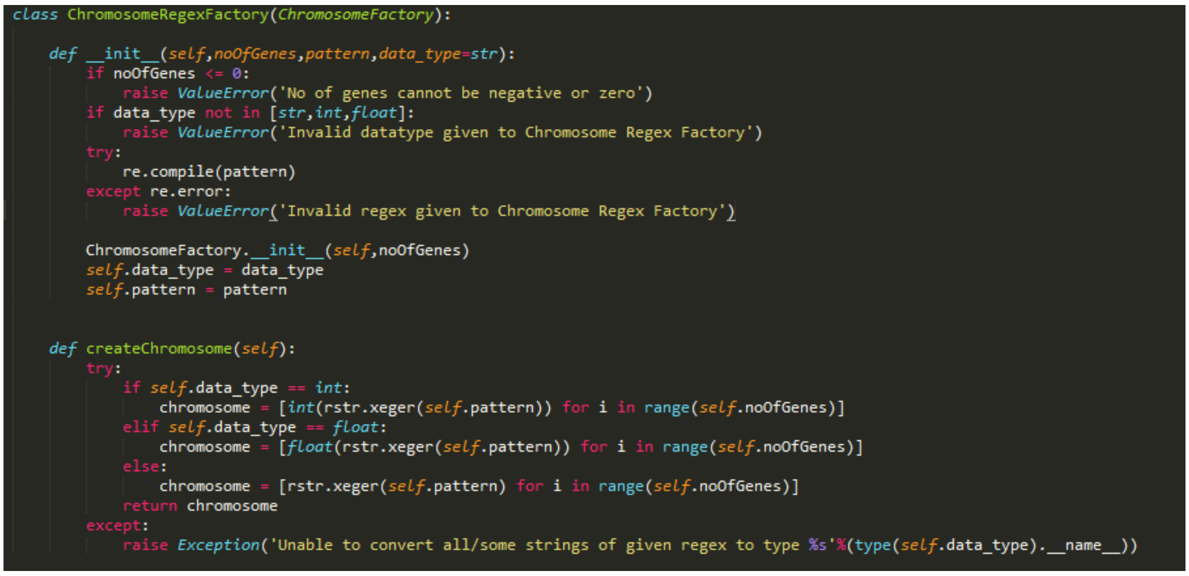
- Enter details about the Genetic Algorithm and run the code from the UI and observe various results
- Get visualizations of different generation statistics.
- Have the ability to enter custom code from the UI
- Enable Users to download the equivalent python code from the various user form inputs regarding the genetic algorithm.

CHAPTER-8

IMPLEMENTATION AND PSEUDO-CODE

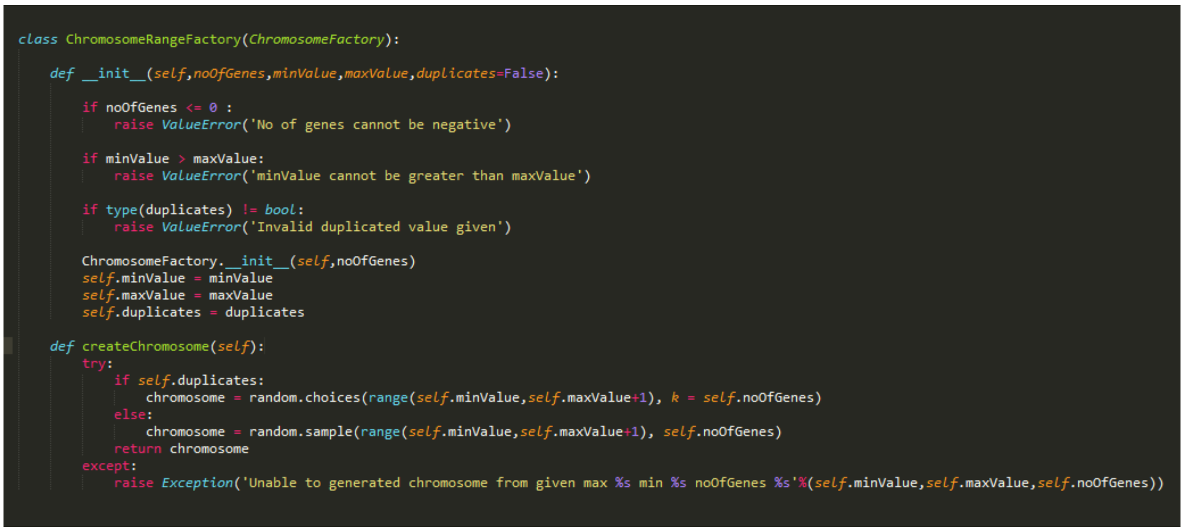
8.1 ChromosomeFactory

This module is responsible for generation of chromosomes of the genetic algorithm. Implemented two factories – RangeFactory for generating based on (min , max) values and RegexFactory for generating based on a regex. The following is the code for the regex and range factory.



```
class ChromosomeRegexFactory(ChromosomeFactory):

    def __init__(self,noOfGenes,pattern,data_type=str):
        if noOfGenes <= 0:
            raise ValueError('No of genes cannot be negative or zero')
        if data_type not in [str,int,float]:
            raise ValueError('Invalid datatype given to Chromosome Regex Factory')
        try:
            re.compile(pattern)
        except re.error:
            raise ValueError('Invalid regex given to Chromosome Regex Factory')

        ChromosomeFactory.__init__(self,noOfGenes)
        self.data_type = data_type
        self.pattern = pattern


    def createChromosome(self):
        try:
            if self.data_type == int:
                chromosome = [int(rstr.xeger(self.pattern)) for i in range(self.noOfGenes)]
            elif self.data_type == float:
                chromosome = [float(rstr.xeger(self.pattern)) for i in range(self.noOfGenes)]
            else:
                chromosome = [rstr.xeger(self.pattern) for i in range(self.noOfGenes)]
            return chromosome
        except:
            raise Exception('Unable to convert all/some strings of given regex to type %s'%(type(self.data_type).__name__))
```

Fig 8.1 ChromosomeRangeFactory code snippet

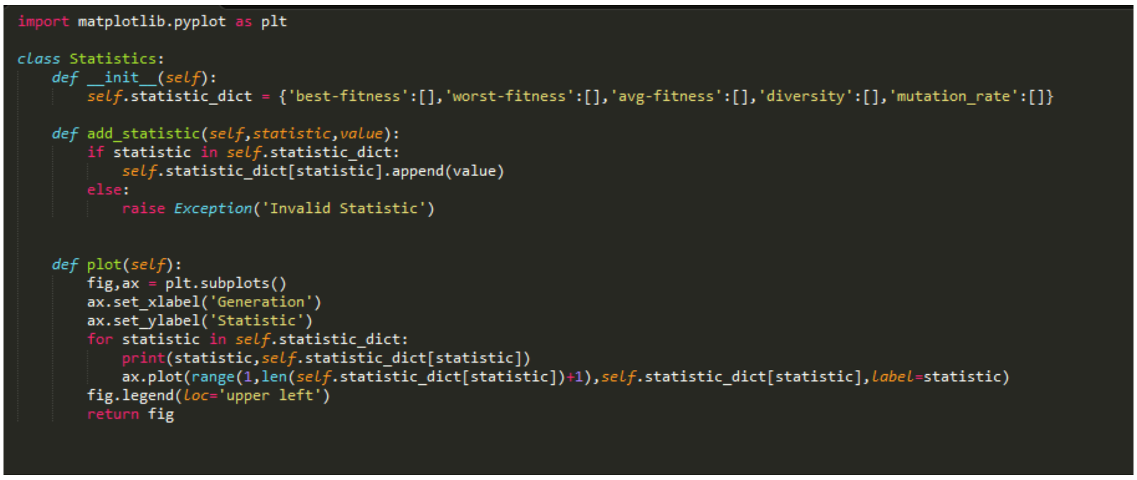


```
class ChromosomeRangeFactory(ChromosomeFactory):

    def __init__(self,noOfGenes,minValue,maxValue,duplicates=False):

        if noOfGenes <= 0 :
            raise ValueError('No of genes cannot be negative')

        if minValue > maxValue:
            raise ValueError('minValue cannot be greater than maxValue')

        if type(duplicates) != bool:
            raise ValueError('Invalid duplicated value given')

        ChromosomeFactory.__init__(self,noOfGenes)
        self.minValue = minValue
        self.maxValue = maxValue
        self.duplicates = duplicates

    def createChromosome(self):
        try:
            if self.duplicates:
                chromosome = random.choices(range(self.minValue,self.maxValue+1), k = self.noOfGenes)
            else:
                chromosome = random.sample(range(self.minValue,self.maxValue+1), self.noOfGenes)
            return chromosome
        except:
            raise Exception('Unable to generated chromosome from given max %s min %s noOfGenes %s'%(self.minValue,self.maxValue,self.noOfGenes))
```

Fig 8.2 ChromosomeRangeFactory code snippet

8.2 Statistics

Following is the basic implementation of the statistics module used to keep track of different generation statistics and provide ways to display it in a nice visual manner.

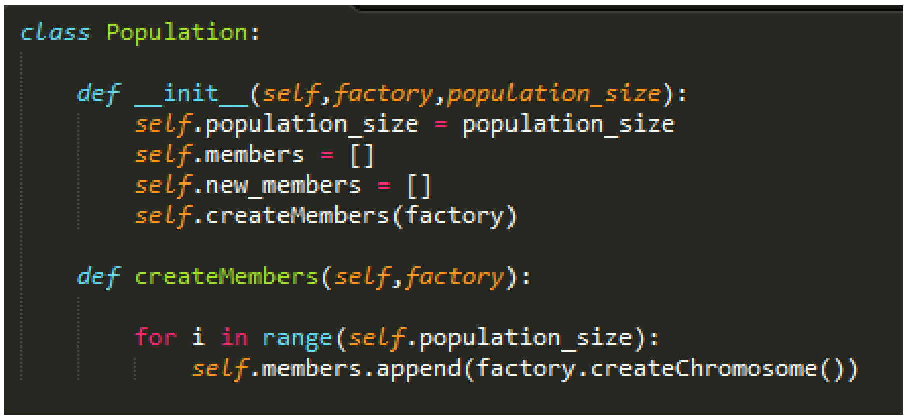


```python
import matplotlib.pyplot as plt

class Statistics:
    def __init__(self):
        self.statistic_dict = {'best-fitness':[],'worst-fitness':[],'avg-fitness':[],'diversity':[],'mutation_rate':[]}

    def add_statistic(self,statistic,value):
        if statistic in self.statistic_dict:
            self.statistic_dict[statistic].append(value)
        else:
            raise Exception('Invalid Statistic')


    def plot(self):
        fig,ax = plt.subplots()
        ax.set_xlabel('Generation')
        ax.set_ylabel('Statistic')
        for statistic in self.statistic_dict:
            print(statistic,self.statistic_dict[statistic])
            ax.plot(range(1,len(self.statistic_dict[statistic])+1),self.statistic_dict[statistic],label=statistic)
        fig.legend(loc='upper left')
        return fig
```

Fig 8.3 Statistics Class code snippet

8.3 Population

Following is the basic implementation of the population module used to keep track of current generation members and can create the initial population based on a given factory

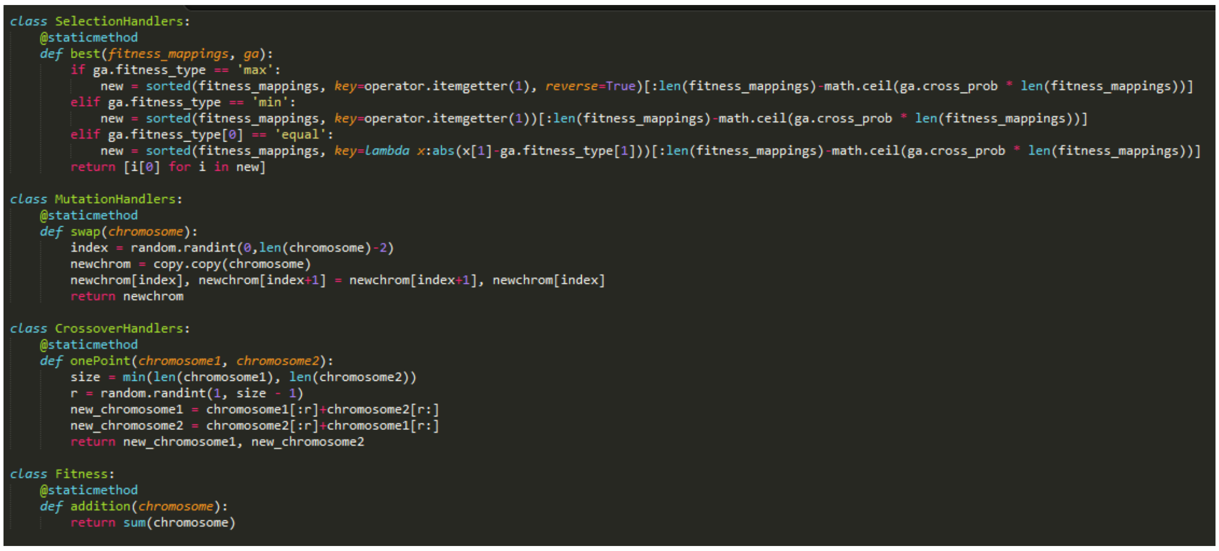


```python
class Population:

    def __init__(self,factory,population_size):
        self.population_size = population_size
        self.members = []
        self.new_members = []
        self.createMembers(factory)

    def createMembers(self,factory):

        for i in range(self.population_size):
            self.members.append(factory.createChromosome())
```

Fig 8.4 Population Class code snippet

8.4 Utils

Following is the basic implementation of the Utils module contains standard selection, crossover and mutation functions. The pseudo code below shows one function of each category.



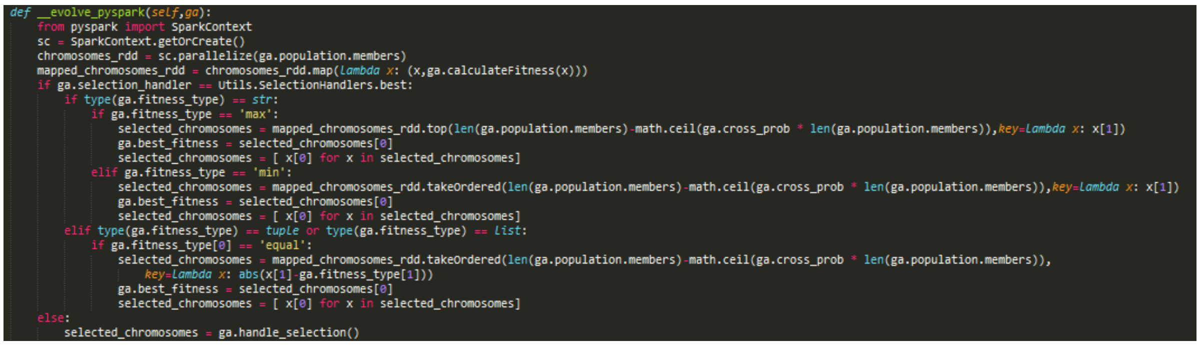
```
class SelectionHandlers:
    @staticmethod
    def best(fitness_mappings, ga):
        if ga.fitness_type == 'max':
            new = sorted(fitness_mappings, key=operator.itemgetter(1), reverse=True)[:len(fitness_mappings)-math.ceil(ga.cross_prob * len(fitness_mappings))]
        elif ga.fitness_type == 'min':
            new = sorted(fitness_mappings, key=operator.itemgetter(1))[:len(fitness_mappings)-math.ceil(ga.cross_prob * len(fitness_mappings))]
        elif ga.fitness_type[0] == 'equal':
            new = sorted(fitness_mappings, key=lambda x:abs(x[1]-ga.fitness_type[1]))[:len(fitness_mappings)-math.ceil(ga.cross_prob * len(fitness_mappings))]
        return [i[0] for i in new]

class MutationHandlers:
    @staticmethod
    def swap(chromosome):
        index = random.randint(0,len(chromosome)-2)
        newchrom = copy.copy(chromosome)
        newchrom[index], newchrom[index+1] = newchrom[index+1], newchrom[index]
        return newchrom

class CrossoverHandlers:
    @staticmethod
    def onePoint(chromosome1, chromosome2):
        size = min(len(chromosome1), len(chromosome2))
        r = random.randint(1, size - 1)
        new_chromosome1 = chromosome1[:r]+chromosome2[r:]
        new_chromosome2 = chromosome2[:r]+chromosome1[r:]
        return new_chromosome1, new_chromosome2

class Fitness:
    @staticmethod
    def addition(chromosome):
        return sum(chromosome)
```

Fig 8.5 Utils module code snippet

8.5 Evolution

This module is in charge of handling Evolutions. It creates a SparkContext and does parallelization of the various operations of fitness calculations, selection, crossover and mutation. The following pseudo code is the basic implementation of Evolution using pySpark.

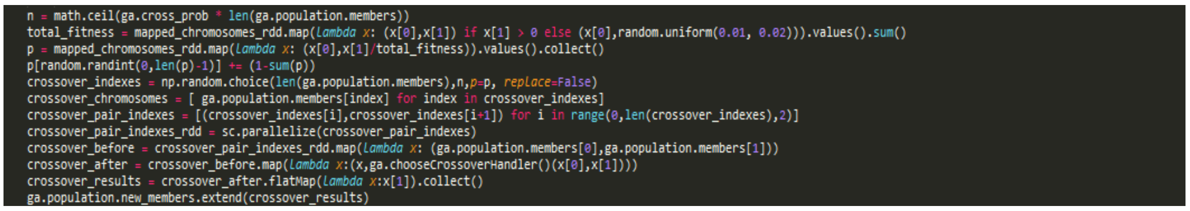
The below basic code achieves parallelization of fitness calculation and selection.



```
def __evolve_pyspark(self,ga):
    from pyspark import SparkContext
    sc = SparkContext.getOrCreate()
    chromosomes_rdd = sc.parallelize(ga.population.members)
    mapped_chromosomes_rdd = chromosomes_rdd.map(lambda x: (x,ga.calculateFitness(x)))
    if ga.selection_handler == Utils.SelectionHandlers.best:
        if type(ga.fitness_type) == str:
            if ga.fitness_type == 'max':
                selected_chromosomes = mapped_chromosomes_rdd.top(len(ga.population.members)-math.ceil(ga.cross_prob * len(ga.population.members)),key=lambda x: x[1])
                ga.best_fitness = selected_chromosomes[0]
                selected_chromosomes = [ x[0] for x in selected_chromosomes]
            elif ga.fitness_type == 'min':
                selected_chromosomes = mapped_chromosomes_rdd.takeOrdered(len(ga.population.members)-math.ceil(ga.cross_prob * len(ga.population.members)),key=lambda x: x[1])
                ga.best_fitness = selected_chromosomes[0]
                selected_chromosomes = [ x[0] for x in selected_chromosomes]
        elif type(ga.fitness_type) == tuple or type(ga.fitness_type) == list:
            if ga.fitness_type[0] == 'equal':
                selected_chromosomes = mapped_chromosomes_rdd.takeOrdered(len(ga.population.members)-math.ceil(ga.cross_prob * len(ga.population.members)),
                    key=lambda x: abs(x[1]-ga.fitness_type[1]))
                ga.best_fitness = selected_chromosomes[0]
                selected_chromosomes = [ x[0] for x in selected_chromosomes]
    else:
        selected_chromosomes = ga.handle_selection()
```

Fig 8.6.1 Evolution with Pyspark code snippet

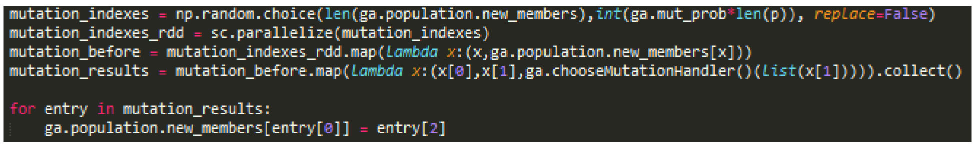
The below basic code achieves parallelization of crossover operations



```
n = math.ceil(ga.cross_prob * len(ga.population.members))
total_fitness = mapped_chromosomes_rdd.map(lambda x: (x[0],x[1]) if x[1] > 0 else (x[0],random.uniform(0.01, 0.02))).values().sum()
p = mapped_chromosomes_rdd.map(lambda x: (x[0],x[1]/total_fitness)).values().collect()
p[random.randint(0,len(p)-1)] += (1-sum(p))
crossover_indexes = np.random.choice(len(ga.population.members),n,p=p, replace=False)
crossover_chromosomes = [ ga.population.members[index] for index in crossover_indexes]
crossover_pair_indexes = [(crossover_indexes[i],crossover_indexes[i+1]) for i in range(0,len(crossover_indexes),2)]
crossover_pair_indexes_rdd = sc.parallelize(crossover_pair_indexes)
crossover_before = crossover_pair_indexes_rdd.map(lambda x: (ga.population.members[0],ga.population.members[1]))
crossover_after = crossover_before.map(lambda x:(x,ga.chooseCrossoverHandler()(x[0],x[1])))
crossover_results = crossover_after.flatMap(lambda x:x[1]).collect()
ga.population.new_members.extend(crossover_results)
```

Fig 8.6.2 Evolution with Pyspark code snippet

The below basic code achieves parallelization of mutation operations

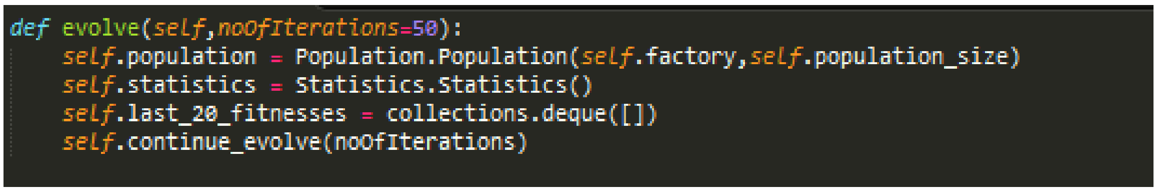


```python
mutation_indexes = np.random.choice(len(ga.population.new_members),int(ga.mut_prob*len(p)), replace=False)
mutation_indexes_rdd = sc.parallelize(mutation_indexes)
mutation_before = mutation_indexes_rdd.map(lambda x:(x,ga.population.new_members[x]))
mutation_results = mutation_before.map(lambda x:(x[0],x[1],ga.chooseMutationHandler()(list(x[1])))).collect()

for entry in mutation_results:
    ga.population.new_members[entry[0]] = entry[2]
```

Fig 8.6.3 Evolution with Pyspark code snippet

8.6 GAEngine

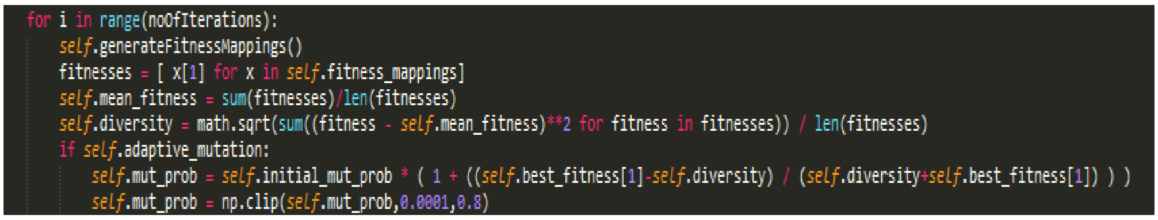
This GAEngine is the main module from which the end user can execute Genetic Algorithms. The following pseudo code is the basic implementation of evolve creates an initial population and further calls continue evolve.



```python
def evolve(self,noOfIterations=50):
    self.population = Population.Population(self.factory,self.population_size)
    self.statistics = Statistics.Statistics()
    self.last_20_fitnesses = collections.deque([])
    self.continue_evolve(noOfIterations)
```

Fig 8.7 GAEngine module code snippet

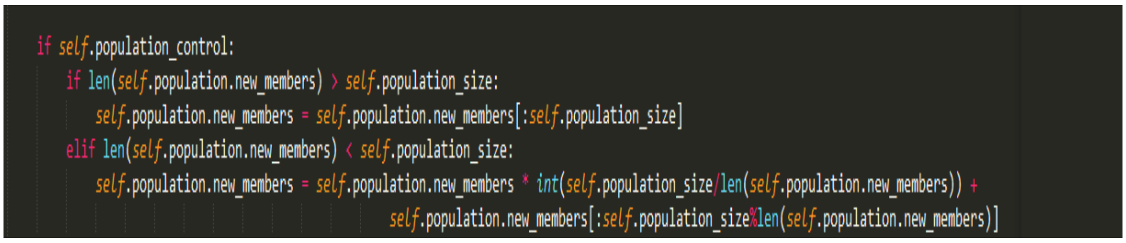
The following pseudo code is the basic implementation of adaptive mutation in continue evolve which calculated the population diversity and sets the mutation rate accordingly.



```python
for i in range(noOfIterations):
    self.generateFitnessMappings()
    fitnesses = [ x[1] for x in self.fitness_mappings]
    self.mean_fitness = sum(fitnesses)/len(fitnesses)
    self.diversity = math.sqrt(sum((fitness - self.mean_fitness)**2 for fitness in fitnesses)) / len(fitnesses)
    if self.adaptive_mutation:
        self.mut_prob = self.initial_mut_prob * ( 1 + ((self.best_fitness[1]-self.diversity) / (self.diversity+self.best_fitness[1])) )
        self.mut_prob = np.clip(self.mut_prob,0.0001,0.8)
```

Fig 8.8 Iteration code snippet

The following pseudo code is the basic implementation of population control which can be enabled/disabled to ensure same population size in each iteration.

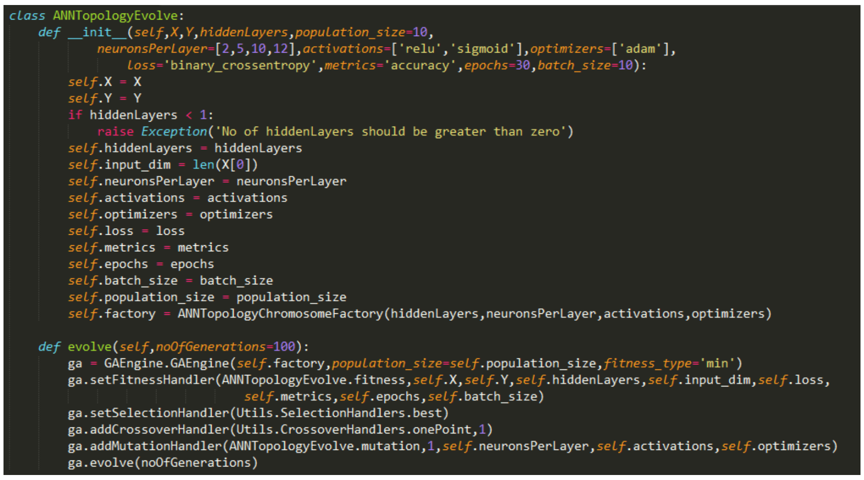


```python
if self.population_control:
    if len(self.population.new_members) > self.population_size:
        self.population.new_members = self.population.new_members[:self.population_size]
    elif len(self.population.new_members) < self.population_size:
        self.population.new_members = self.population.new_members * int(self.population_size/len(self.population.new_members)) +
                                      self.population.new_members[:self.population_size%len(self.population.new_members)]
```

Fig 8.9 Population control code snippet

8.7 ANNEvolve

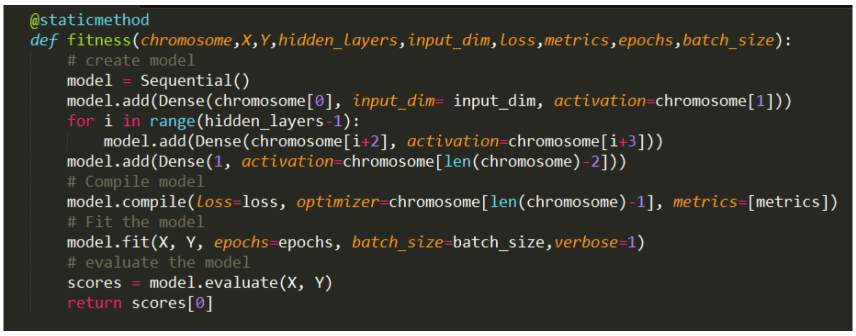
This module is responsible for finding the best ANN topology which can learn fast by using a GA. The following is its basic implementation.



```
class ANNTopologyEvolve:
    def __init__(self,X,Y,hiddenLayers,population_size=10,
            neuronsPerLayer=[2,5,10,12],activations=['relu','sigmoid'],optimizers=['adam'],
                loss='binary_crossentropy',metrics='accuracy',epochs=30,batch_size=10):
        self.X = X
        self.Y = Y
        if hiddenLayers < 1:
            raise Exception('No of hiddenLayers should be greater than zero')
        self.hiddenLayers = hiddenLayers
        self.input_dim = len(X[0])
        self.neuronsPerLayer = neuronsPerLayer
        self.activations = activations
        self.optimizers = optimizers
        self.loss = loss
        self.metrics = metrics
        self.epochs = epochs
        self.batch_size = batch_size
        self.population_size = population_size
        self.factory = ANNTopologyChromosomeFactory(hiddenLayers,neuronsPerLayer,activations,optimizers)

    def evolve(self,noOfGenerations=100):
        ga = GAEngine.GAEngine(self.factory,population_size=self.population_size,fitness_type='min')
        ga.setFitnessHandler(ANNTopologyEvolve.fitness,self.X,self.Y,self.hiddenLayers,self.input_dim,self.loss,
                                self.metrics,self.epochs,self.batch_size)
        ga.setSelectionHandler(Utils.SelectionHandlers.best)
        ga.addCrossoverHandler(Utils.CrossoverHandlers.onePoint,1)
        ga.addMutationHandler(ANNTopologyEvolve.mutation,1,self.neuronsPerLayer,self.activations,self.optimizers)
        ga.evolve(noOfGenerations)
```

Fig 8.10 ANN Topology using GA code snippet

The fitness can be computed by training the ANN as per the chromosome and calculating its loss.

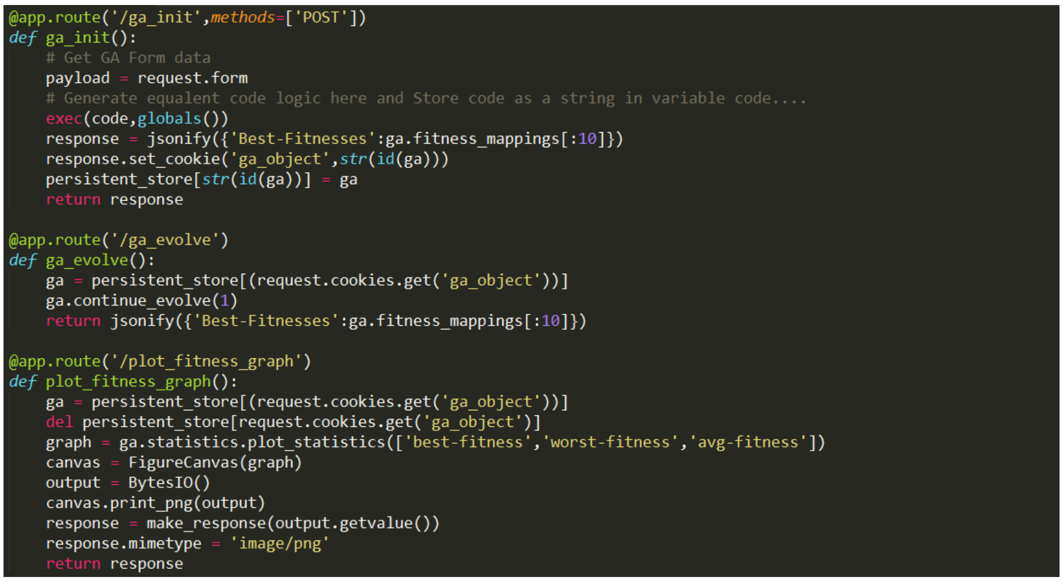


```
@staticmethod
def fitness(chromosome,X,Y,hidden_layers,input_dim,loss,metrics,epochs,batch_size):
    # create model
    model = Sequential()
    model.add(Dense(chromosome[0], input_dim= input_dim, activation=chromosome[1]))
    for i in range(hidden_layers-1):
        model.add(Dense(chromosome[i+2], activation=chromosome[i+3]))
    model.add(Dense(1, activation=chromosome[len(chromosome)-2]))
    # Compile model
    model.compile(loss=loss, optimizer=chromosome[len(chromosome)-1], metrics=[metrics])
    # Fit the model
    model.fit(X, Y, epochs=epochs, batch_size=batch_size,verbose=1)
    # evaluate the model
    scores = model.evaluate(X, Y)
    return scores[0]
```

Fig 8.11 Fitness function for ANN topology code snippet

8.8 GA Website Endpoints

The following is the basic implementation of the GA Website Backend using Flask. It consists of ga_init – which receives form data, generates equivalent code and runs one iteration and returns best chromosomes, ga_evolve – which continues further evolutions and returns best chromosomes and plot_fitness_graph which generates a graph depicting the various statistics.



```
@app.route('/ga_init',methods=['POST'])
def ga_init():
    # Get GA Form data
    payload = request.form
    # Generate equalent code logic here and Store code as a string in variable code....
    exec(code,globals())
    response = jsonify({'Best-Fitnesses':ga.fitness_mappings[:10]})
    response.set_cookie('ga_object',str(id(ga)))
    persistent_store[str(id(ga))] = ga
    return response

@app.route('/ga_evolve')
def ga_evolve():
    ga = persistent_store[(request.cookies.get('ga_object'))]
    ga.continue_evolve(1)
    return jsonify({'Best-Fitnesses':ga.fitness_mappings[:10]})

@app.route('/plot_fitness_graph')
def plot_fitness_graph():
    ga = persistent_store[(request.cookies.get('ga_object'))]
    del persistent_store[request.cookies.get('ga_object')]
    graph = ga.statistics.plot_statistics(['best-fitness','worst-fitness','avg-fitness'])
    canvas = FigureCanvas(graph)
    output = BytesIO()
    canvas.print_png(output)
    response = make_response(output.getvalue())
    response.mimetype = 'image/png'
    return response
```

Fig 8.12 GA website endpoints code snippet

This concludes the basic implementation and pseudo code of the various modules of pygenetic API and the online GA execution website.

CHAPTER-9

TESTING

Testing for this project has been carried out in all the various stages of the development cycle. Making a good test strategy is vital to the success of an API and whether its use gets widely accepted or not.

9.1 Scope

Identification of features whose failures are catastrophic to the project, features which are new and critical for release and those features which are complex to test get more priority in the list of various features to be tested.

On these grounds, the features to be included in our testing include

1. Testing the **functionalities of classes** of the GA.
2. Testing if the **performance of the Genetic Algorithm** is acceptable or not. (avg time taken, memory used, number of failures). Need for some acceptance criteria.
3. Testing the **security of the application**. No invalid inputs are fed into the API.
4. Testing the **usability of our website**.
5. Testing the **documentation of our API** to ensure that every end user of our API can effectively know how to use our API.
6. Testing the **conformance of the object oriented code to SOLID principles** to ensure high maintainability.
7. Testing the GA website **performance under heavy loads**.

On these grounds, the features which are cautiously not being included for testing include

1. Cross browser testing of our website
2. UI tests which involve checking whether **certain css styles are applied when an event occurs** (eg: red color for bad input) or **DOM elements are added when an event occurs.** These can be checked manually and we need not waste time coding test scripts.

9.2 Test Strategies

The test strategies which have been planned to be used in the project include:

Types of Testing to be used for testing the functionality:

2. **White Box Testing:** to ensure each basic class of the Genetic Algorithm Framework work as intended. All the main types of white box testing are done here –

    a. Static Testing:

    i. Done by humans to find any errors in the GA code, check if code conforms to planned GA design, check error handling capabilities of the framework using test cases with all different input data types. This is done through desk checking, code walkthroughs and formal inspection once every piece of code of the GA framework is developed.

    ii. By Static Analysis Tools to detect coding errors like not freeing memory after use, unused variables and unreachable code among other errors.

    It needs to be done as soon as a module of code is developed. **Manually examine** each class of our GA API to find errors in GA code, further possible optimizations and check error handling capabilities of our GA API. Use of **static analysis tools pylint** to detect static coding errors in API

    b. Structural Testing:

    i. Code Functional Testing: Test scripts to test the functionality of every function of every written module of the GA Library are created and added to the test suite to detect any errors in the code functionality considering the internal data structures.

    ii. Code Coverage Testing: Is to be done after most of the code base has been developed to ensure that a higher percentage of the GA framework code is covered by the test cases.

3. **Black Box Testing:** is to be done to ensure the requirements of the various classes of the GA framework work as intended. We are to incorporate this in our project by using

    a. Requirements based testing: to be done to test the various requirements specified in the SRS of the GA framework.

    Eg: Test if Regex Chromosome Factory mets its functional requirements.

a. Positive/negative testing: to be done to test that the code works under positive and negative conditions. Eg: Normal code sequence: ga.evolve(10)

ga.continue_evolve(5)

Abnormal Code sequence: ga = ….

ga.continue_evolve(5)

Handle these errors

b. Boundary value analysis: to analyze if code works as intended given boundary values of input variables.

   Eg: GAEngine even works with 1 gene in chromosomes in the population provided there are no crossover handlers, population = 1

c. User Documentation Testing: to test the documentation created for end users so as to allow users to easily. Need to review code hosted on readthedocs regularly.

4. **Error Handling Tests**

   To ensure that all possible error conditions are handled by rasing valid exceptions

   Eg: 1. Invalid Regex given to regex chromosome factory

   2. Negative population size

   3. max value less than minvalue in range factory

5. **GUI Tests**

   Usage of selenium

   - To verify form validation tests – form does not support on invalid input conditions
   - To verify Evolution details come to the UI on selecting "RUN GA" button
   - Verify the time of receiving a response from the Server is acceptable or not

6. **System and Acceptance Testing:** is done on the complete integrated GA framework to evaluate the systems compliance with specified requirements. The NFR testing to be carried out include

   a. Performance/Load testing: is to be done to evaluate the average time taken by the Genetic Algorithm to solve various types of problems either using Spark or through normal execution and to verify if it is acceptable or not. And Stress Testing: to ensure that the GA website will not break when resource limits like disk space, memory, processor utilization have been exceeded.

b. Concurrent Testing: detect the defects in the application when multiple users are logged in

c. User Friendliness Testing and Alpha/Beta Testing: functionality verification, etc. from end users

9.3 Test Tools Used

This section shall describe the test tools required / used for testing the product.

1. pyTest: Can be used for various unit tests, integration testing.
2. Selenium: For testing various functional aspects of our web based application and wide range of browsers.
3. Coverage: For carrying out code coverage tests.
4. Pylint: For static python code testing
5. TravisCI: For Continuous Integration
6. Locust: For various performance testing
7. Unittest.mock: Used for various mock tests. Since most of our functions involve random values, mock.patch helps us set random() to return a determined value, prevent calling unneeded code, etc.

9.4 Test Case List

The following are the list of test cases use to test the API and the GA website.

Test Case Number	Test Case	Required Output
1	Verify that the various chromosome factories produce the correct chromosomes	Correct chromosome is generated
2	Verify that the various selection handler utilities produce the right output for a given input	Correct selection of chromosomes is done
3	Verify that the various crossover handler utilities produce the right output for a given chromosome set	Correct crossover of the chromosome set is done
4	Verify that the various mutation handlers utilities produce the right output for a	Correct mutation of the chromosome is done

	given chromosome	
5	Verify that the standard fitness utilities is calculated correctly for a given chromosome	Correct fitness of the chromosome is calculated
6	Verify that the Statistics class works as intended to plot various graphs for fitness over different generation	Correct max, min, average fitness is calculated and Correct graph is plotted
7	Verify that the Population class produces correct number and type of chromosomes given the Chromosome Factory	Correct Population members are generated
9	Verify that the StandardEvolution steps take place as expected	Correct Procedure of Standard Evolution is executed
10	Verify that the Evolution using pySpark parallelization work as expected	Correct RDD are created and map, collect, takeordered executed
11	Verify handling of MAX,MIN and EQ fitness types	Correct execution of evolution based on fitness types are executed
12	Verify population diversity is calculated as expected	Correct calculation of population diversity
13	Verify adaptive mutation rate is calculated over different generations correctly	Correct adaptive mutation rate is calculated
14	Verify that correct Evolution is used, optimization setting is used when set in the GAEngine class	Correct evolution/optimization takes place
15	Verify that Genetic Algorithms can be executed from the UI	Correct execution using our API takes place
16	Verify that code generated from the UI inputs is correct	Correct code is generated
17	Verify that regular responses come from the server side in not more than 4 seconds	Responses in less than 4 seconds
18	Verify that at least 100 users can use the website concurrently with response not taking more than 20 seconds	Responsive Website with responses in less than 20 seconds
19	Verify that the form does not submit when invalid inputs are submitted	Verification that form does not get submitted and error is thrown

20	Verify that right code file is downloaded when inputs are given to the GA form	Verification of code generated from UI form
21	Verify population control mechanism	Verification of population control
22	Verify hall of fame injection	Verification of hall of fame injection
23	Verify iteration halt optimization	Verification of iteration halt optimization
24	Verify working of SimpleGA Evolutions	Verification of SimpleGA evolutions
25	Verify functionality of ANN Evolution using GA	Verification of ANN Evolution using GA

Table 9.1 Test Case List

This concludes the manner in which testing is carried for our python GA API which ensures that the API is indeed working as expected and isn't buggy.

CHAPTER-10

RESULTS AND DISCUSSIONS

The results of making this pygenetic framework are:

1. It succeeded in making an API for GA execution which is both good in user friendliness and genericity unlike most current day APIs which make tradeoffs between the two. DEAP makes it highly generic while it has poor user friendliness while genetics, pygalib make it user friendly with lesser genericity. Our API succeeds in making an API which lies in between – good user friendliness and good genericity.

2. Our API pygenetic in normal execution mode is equally good in terms of execution times as every other python API (genetics, pygalib) which runs sequentially/does basic parallelization.

3. Our API pygenetic in pyspark parallel execution mode is very scalable and comparable to APIs which use large scale parallelization like DEAP. Its performance has been observed when deployed with 2 workers and a remarkable improvement in execution time has been observed when compared with normal sequential execution. This performance is scalable and can theoretical match performances of DEAP when more workers are deployed.

4. Our API has been proven to solve real genetic algorithm applications including nQueens, TSP and ANN Best Topology Finder using GA using normal execution and pyspark parallelization in effective times.

5. Our API has been tested to excel in features including adaptive mutations, hall of fame injection, early iteration halt, population control, continue_evolve which can be exploited in various research applications.

6. Our Website has been proven to support execution of highly generic Genetic Algorithms from the UI itself. Statistics of the GA were successfully generated at the end of the GA execution.
7. Our Website has also undergone various performance tests and has been proven to excel in supporting as many as 100 concurrent users without breaking.

8. Our Website has also performed well with regard to polling the server for responses for each generation data. Under less load conditions of less than 100 concurrent users and population size of less than 1000, responses come back in less than 3 seconds and under heavy load conditions of more than 500 simultaneous users and population size of less than 1000, response come back in less than 10 seconds. Performance can further be improved by using a system with more RAM and processing speed as the server.

10.1 Performance comparison of pygenetic

The time comparisons of pygenetic vs DEAP vs other python GA APIs were monitored for execution of a TSP GA of different population sizes and their execution times were examined. It was run for 20 evolutions.

Population size	DEAP	pygenetic	pygenetic(2workers)	genetics	pygalib
100	1.923	1.093	2.620	1.289	1.278
500	2.879	1.437	3.027	1.983	1.762
1000	3.259	1.922	3.996	2.345	2.093
5000	5.129	4.201	5.798	4.892	4.231
10000	6.142	6.987	6.823	7.234	7.102
50000	20.123	32.738	24.982	39.902	34.234
100000	39.332	70.487	44.729	82.234	74.956

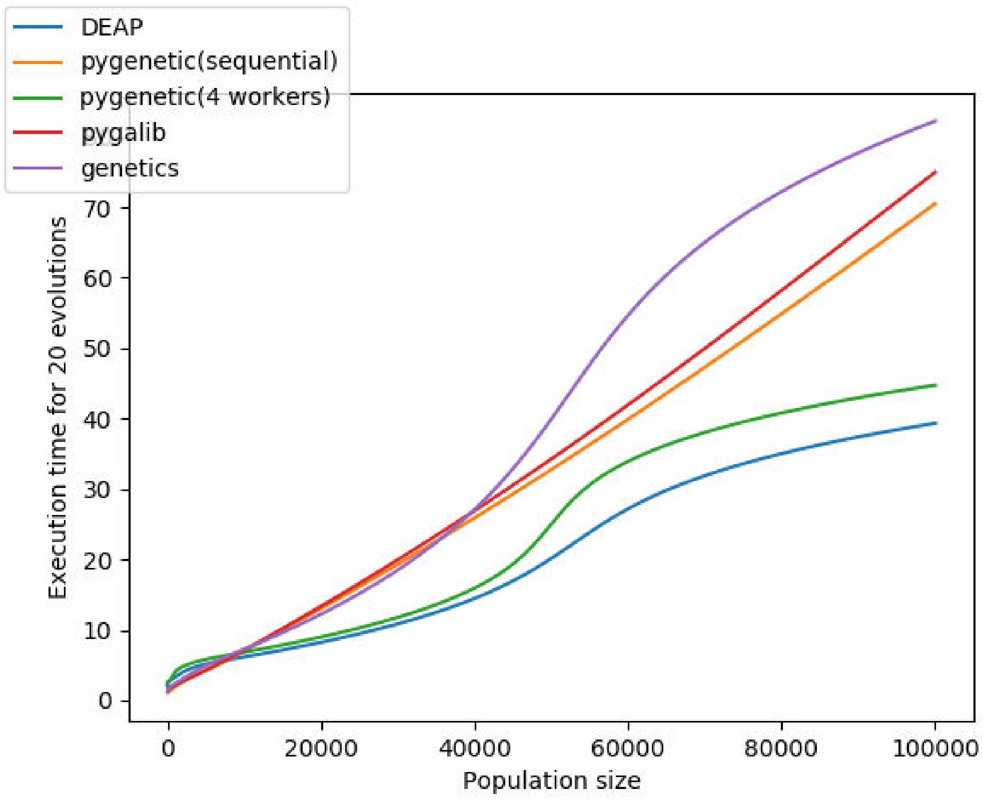


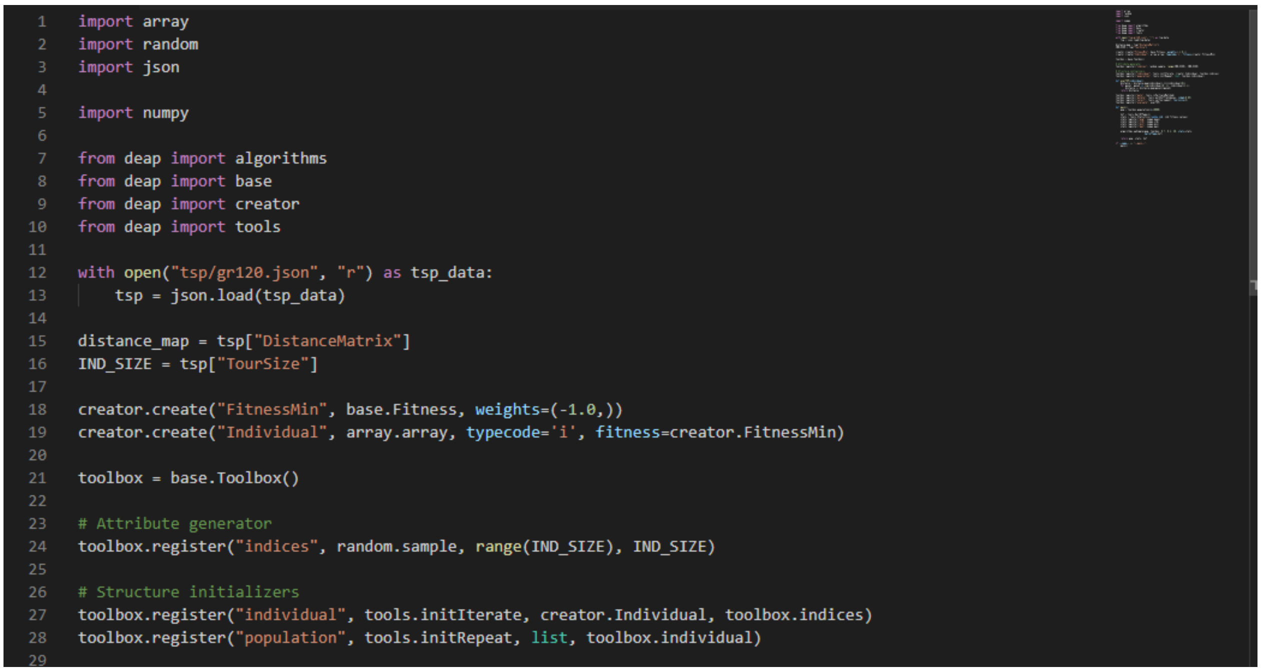
Fig 10.1 Execution time based on population size

As observed, pygenetic performs better than all the other GA APIs genetics and pygalib, while it can be scalable and when deployed using more workers, efficiency approaches that of DEAP and can be even better when deployed using many more workers.

10.2 Usage comparison between DEAP and pygenetic

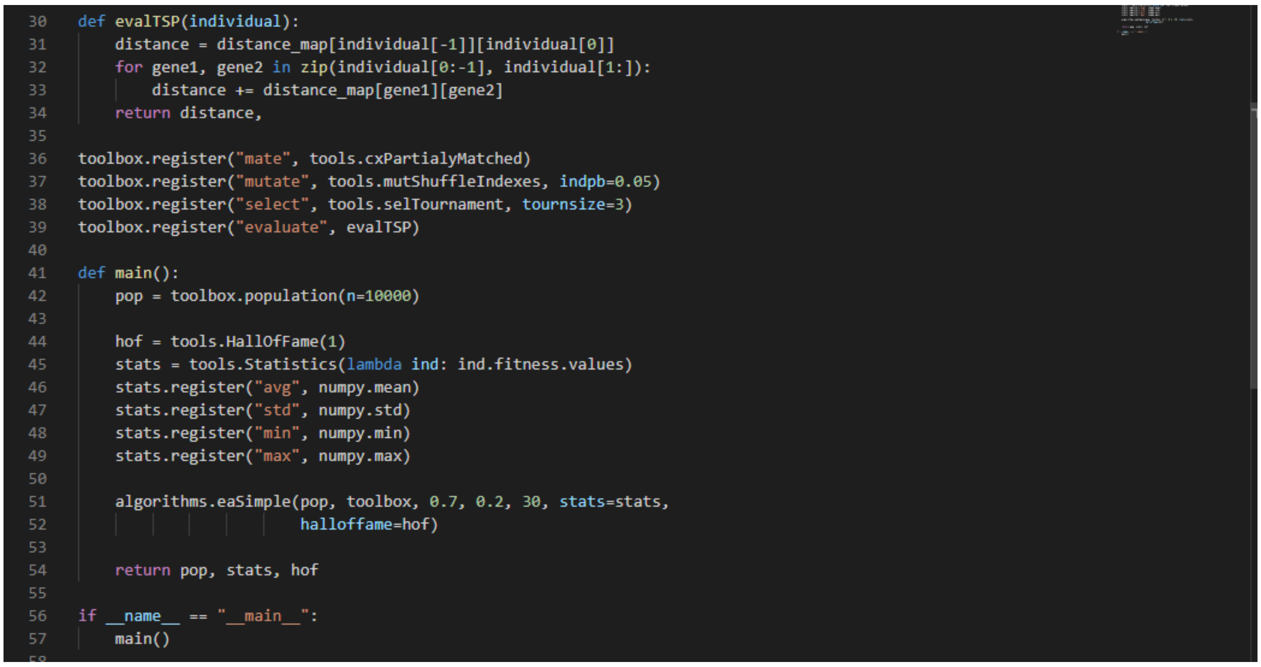
Let’s compare pygenetic with DEAP over the critical aspect of user friendliness.

Here is an example of DEAP that solves a Travelling Salesman Problem (data in json):



```python
import array
import random
import json

import numpy

from deap import algorithms
from deap import base
from deap import creator
from deap import tools

with open("tsp/gr120.json", "r") as tsp_data:
    tsp = json.load(tsp_data)

distance_map = tsp["DistanceMatrix"]
IND_SIZE = tsp["TourSize"]

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", array.array, typecode='i', fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Attribute generator
toolbox.register("indices", random.sample, range(IND_SIZE), IND_SIZE)

# Structure initializers
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

```

Fig 10.2 DEAP code 1/2

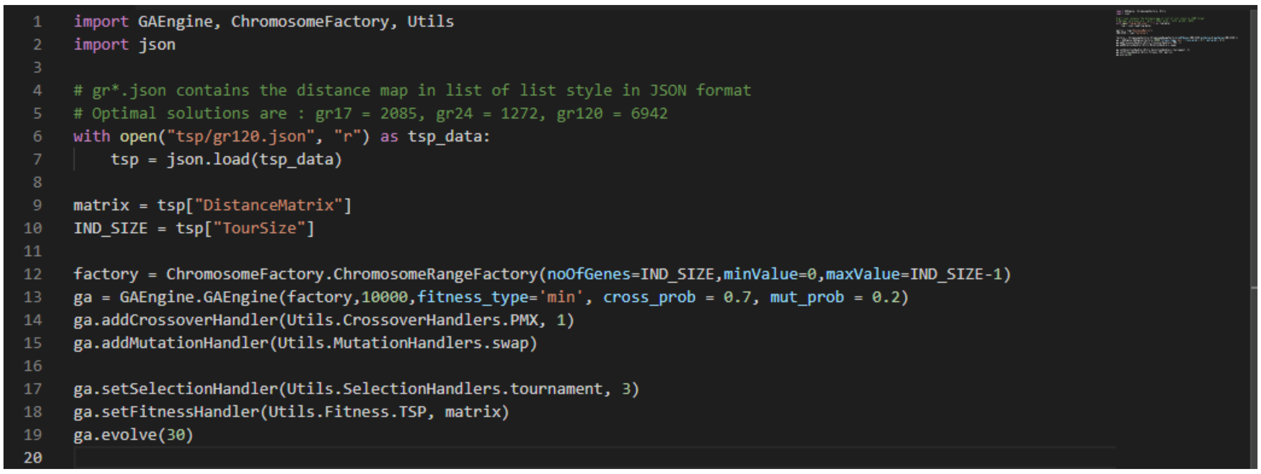


```python
def evalTSP(individual):
    distance = distance_map[individual[-1]][individual[0]]
    for gene1, gene2 in zip(individual[0:-1], individual[1:]):
        distance += distance_map[gene1][gene2]
    return distance,

toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evalTSP)

def main():
    pop = toolbox.population(n=10000)

    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean)
    stats.register("std", numpy.std)
    stats.register("min", numpy.min)
    stats.register("max", numpy.max)

    algorithms.eaSimple(pop, toolbox, 0.7, 0.2, 30, stats=stats,
                        halloffame=hof)

    return pop, stats, hof

if __name__ == "__main__":
    main()
```

Fig 10.3 DEAP code 2/2

You can see the problem here. You will need to import a lot of modules to even start writing the code. You also need to instantiate statistics yourself to get more meaningful information.

Also, you can observe that there are a lot of instantiations going around, making any person who first looks at the code to feel that it is tough to implement problem solving using genetic algorithms. This is where pygenetic reigns supreme. Take a look at the same problem being solved, with the same parameters as in the previous piece of code, no difference at all!



```python
import GAEngine, ChromosomeFactory, Utils
import json

# gr*.json contains the distance map in list of list style in JSON format
# Optimal solutions are : gr17 = 2085, gr24 = 1272, gr120 = 6942
with open("tsp/gr120.json", "r") as tsp_data:
    tsp = json.load(tsp_data)

matrix = tsp["DistanceMatrix"]
IND_SIZE = tsp["TourSize"]

factory = ChromosomeFactory.ChromosomeRangeFactory(noOfGenes=IND_SIZE,minValue=0,maxValue=IND_SIZE-1)
ga = GAEngine.GAEngine(factory,10000,fitness_type='min', cross_prob = 0.7, mut_prob = 0.2)
ga.addCrossoverHandler(Utils.CrossoverHandlers.PMX, 1)
ga.addMutationHandler(Utils.MutationHandlers.swap)

ga.setSelectionHandler(Utils.SelectionHandlers.tournament, 3)
ga.setFitnessHandler(Utils.Fitness.TSP, matrix)
ga.evolve(30)

```

Fig 10.4 pygenetic code

You can see the difference. Also make a note that pygenetic is as customizable, if not, more than DEAP (we provide multiple crossover and mutation handler support). All you need to do is import 3 modules (2 if you want to define your own handlers), write 7 lines of pure function-calling code (you need to know the parameters though) and you are done. You have solved a problem using genetic algorithms. Seeing that all you need is this short code to be able to implement problem solving in genetic algorithms can inspire any person interested in trying out genetic algorithms to actually get familiar with them, play around and gradually understand how it functions practically. That makes a lot of difference.

CHAPTER-11

SNAPSHOTS

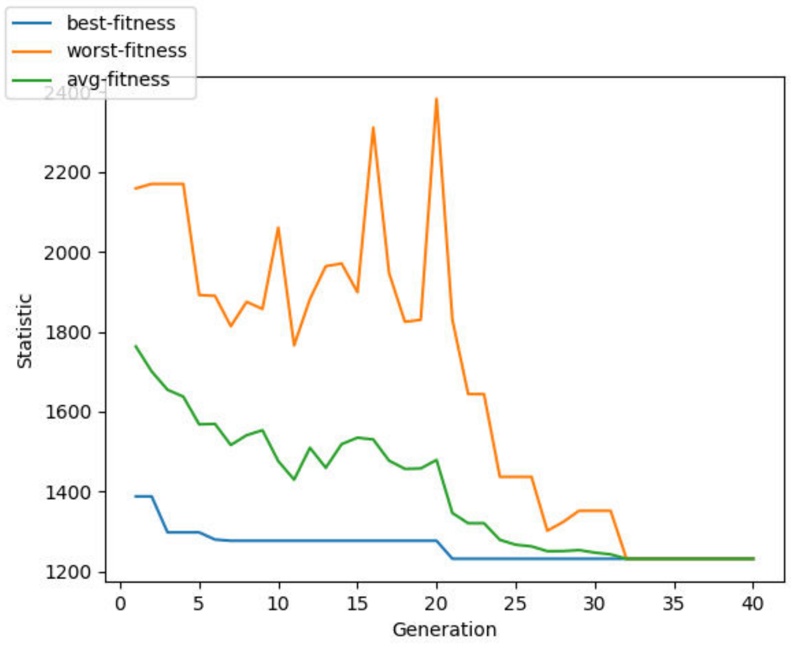


Fig 11.1 Fitnesses achieved while solving TSP

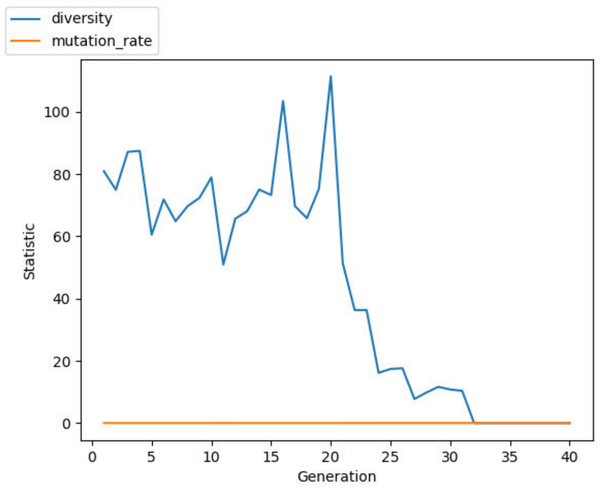


Fig 11.2 Diversities achieved while solving TSP

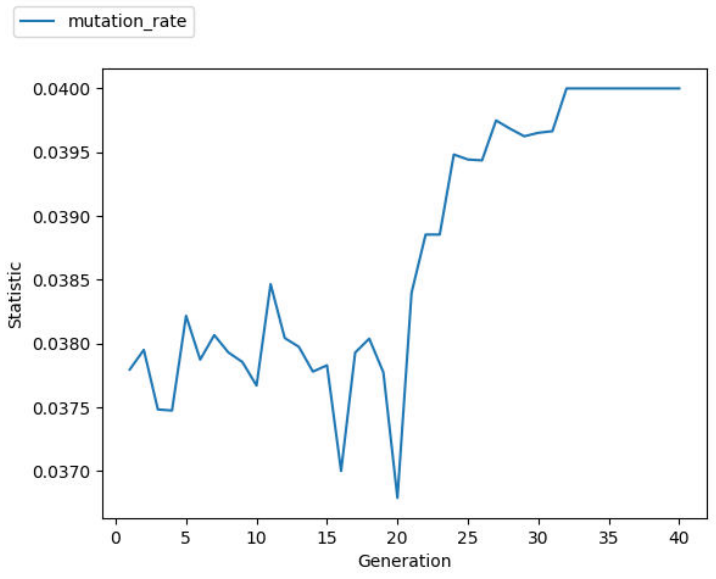


Fig 11.3 Adaptive Mutation in Action

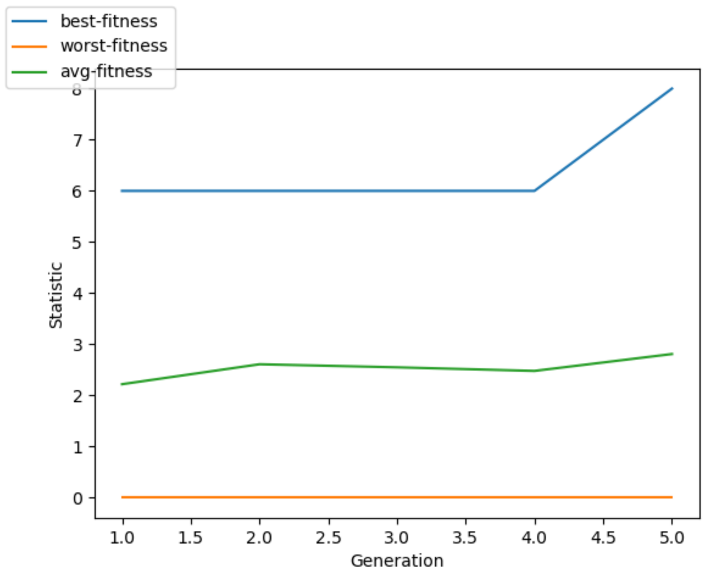
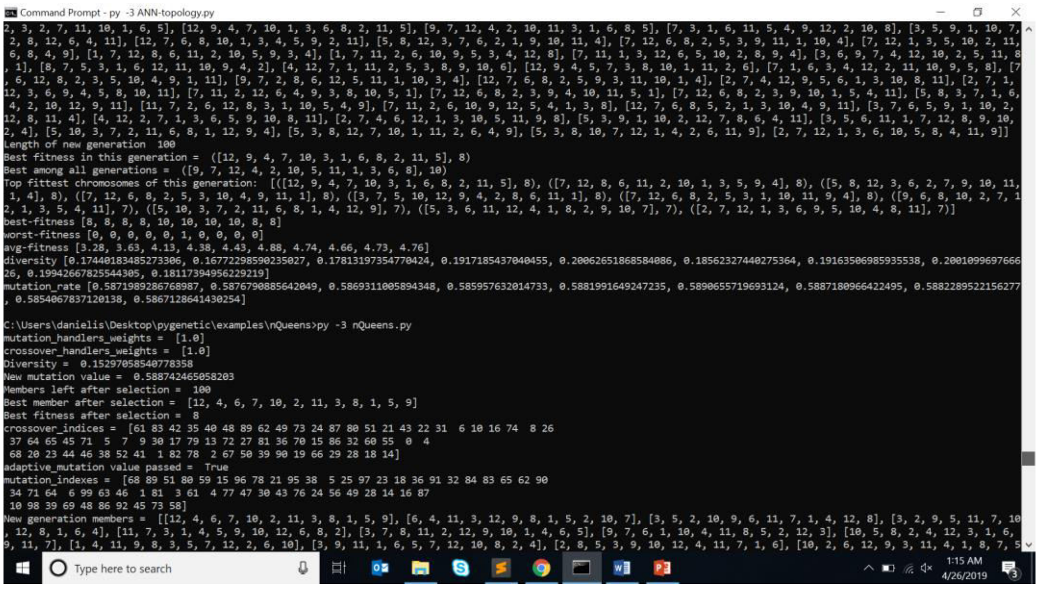
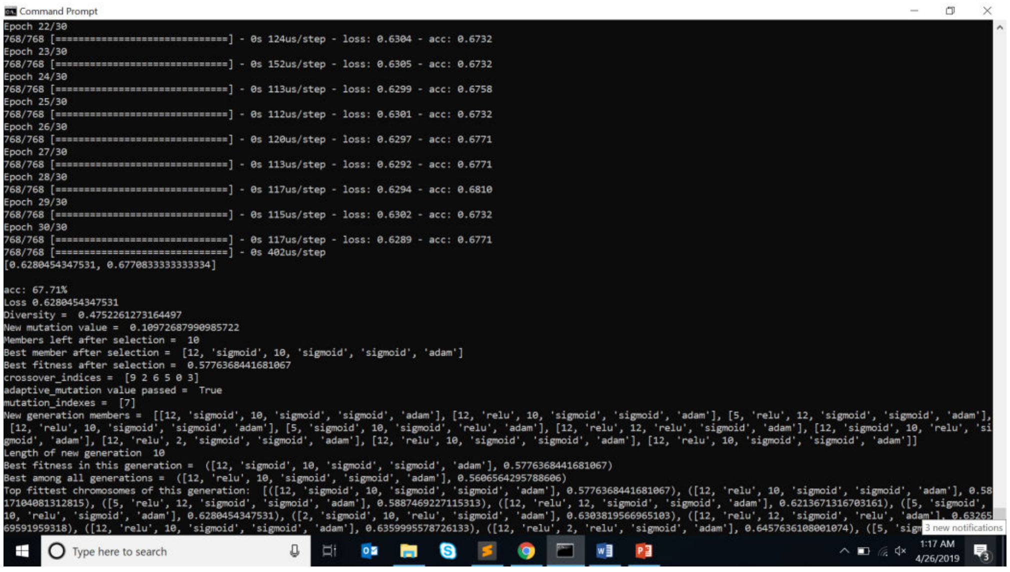


Fig 11.4 8-Queens Problem { [6, 3, 1, 7, 5, 8, 2, 4] }



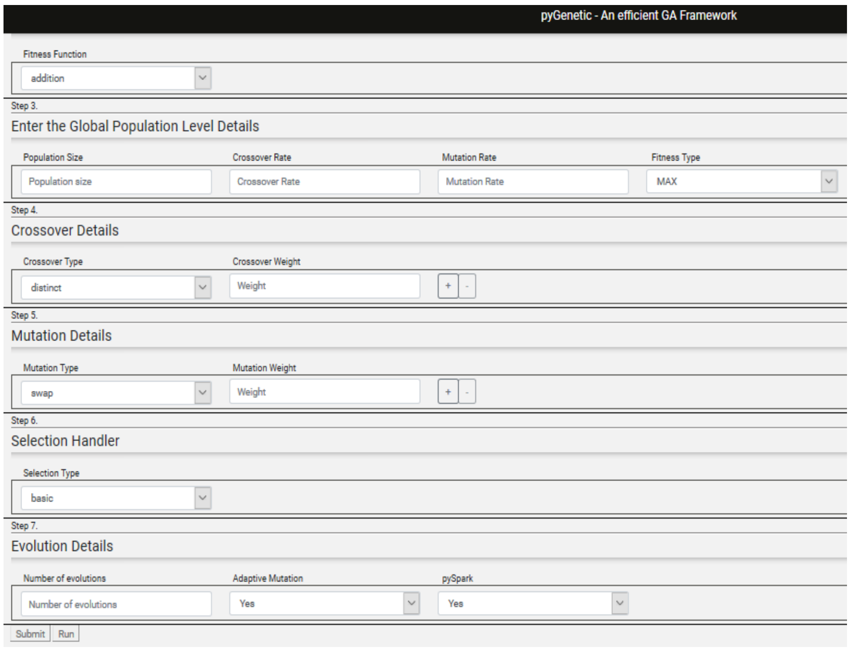
```
Command Prompt - py  -3 ANN-topology.py
2, 3, 2, 7, 11, 10, 1, 6, 5], [12, 9, 4, 7, 10, 1, 3, 6, 8, 2, 11, 5], [9, 7, 12, 4, 2, 10, 11, 3, 1, 6, 8, 5], [7, 3, 1, 6, 11, 5, 4, 9, 12, 2, 10, 8], [3, 5, 9, 1, 10, 7,
 2, 8, 12, 6, 4, 11], [12, 7, 6, 8, 10, 1, 3, 4, 5, 9, 2, 11], [5, 8, 12, 3, 7, 6, 2, 1, 9, 10, 11, 4], [7, 12, 6, 8, 2, 5, 3, 9, 11, 1, 10, 4], [7, 12, 1, 3, 5, 10, 2, 11,
 6, 8, 4, 9], [1, 7, 12, 8, 6, 11, 2, 10, 5, 9, 3, 4], [1, 7, 11, 2, 6, 10, 9, 5, 3, 4, 12, 8], [7, 11, 1, 3, 12, 6, 5, 10, 2, 8, 9, 4], [3, 6, 9, 7, 4, 12, 10, 2, 5, 11, 8
, 1], [8, 7, 5, 3, 1, 6, 12, 11, 10, 9, 4, 2], [4, 12, 7, 1, 11, 2, 5, 3, 8, 9, 10, 6], [12, 9, 4, 5, 7, 3, 8, 10, 1, 11, 2, 6], [7, 1, 6, 3, 4, 12, 2, 11, 10, 9, 5, 8], [7
, 6, 12, 8, 2, 3, 5, 10, 4, 9, 1, 11], [9, 7, 2, 8, 6, 12, 5, 11, 1, 10, 3, 4], [12, 7, 6, 8, 2, 5, 9, 3, 11, 10, 1, 4], [2, 7, 4, 12, 9, 5, 6, 1, 3, 10, 8, 11], [2, 7, 1,
12, 3, 6, 9, 4, 5, 8, 10, 11], [7, 11, 2, 12, 6, 4, 9, 3, 8, 10, 5, 1], [7, 12, 6, 8, 2, 3, 9, 4, 10, 11, 5, 1], [7, 12, 6, 8, 2, 3, 9, 10, 1, 5, 4, 11], [5, 8, 3, 7, 1, 6,
 4, 2, 10, 12, 9, 11], [11, 7, 2, 6, 12, 8, 3, 1, 10, 5, 4, 9], [7, 11, 2, 6, 10, 9, 12, 5, 4, 1, 3, 8], [12, 7, 6, 8, 5, 2, 1, 3, 10, 4, 9, 11], [3, 7, 6, 5, 9, 1, 10, 2,
12, 8, 11, 4], [4, 12, 2, 7, 1, 3, 6, 5, 9, 10, 8, 11], [2, 7, 4, 6, 12, 1, 3, 10, 5, 11, 9, 8], [5, 3, 9, 1, 10, 2, 12, 7, 8, 6, 4, 11], [3, 5, 6, 11, 1, 7, 12, 8, 9, 10,
2, 4], [5, 10, 3, 7, 2, 11, 6, 8, 1, 12, 9, 4], [5, 3, 8, 12, 7, 10, 1, 11, 2, 6, 4, 9], [5, 3, 8, 10, 7, 12, 1, 4, 2, 6, 11, 9], [2, 7, 12, 1, 3, 6, 10, 5, 8, 4, 11, 9]]
Length of new generation  100
Best fitness in this generation =  ([12, 9, 4, 7, 10, 3, 1, 6, 8, 2, 11, 5], 8)
Best among all generations =  ([9, 7, 12, 4, 2, 10, 5, 11, 1, 3, 6, 8], 10)
Top fittest chromosomes of this generation:  [([12, 9, 4, 7, 10, 3, 1, 6, 8, 2, 11, 5], 8), ([7, 12, 8, 6, 11, 2, 10, 1, 3, 5, 9, 4], 8), ([5, 8, 12, 3, 6, 2, 7, 9, 10, 11,
 1, 4], 8), ([7, 12, 6, 8, 2, 5, 3, 10, 4, 9, 11, 1], 8), ([3, 7, 5, 10, 12, 9, 4, 2, 8, 6, 11, 1], 8), ([7, 12, 6, 8, 2, 5, 3, 1, 10, 11, 9, 4], 8), ([9, 6, 8, 10, 2, 7, 1
2, 1, 3, 5, 4, 11], 7), ([5, 10, 3, 7, 2, 11, 6, 8, 1, 4, 12, 9], 7), ([5, 3, 6, 11, 12, 4, 1, 8, 2, 9, 10, 7], 7), ([2, 7, 12, 1, 3, 6, 9, 5, 10, 4, 8, 11], 7)]
best-fitness [8, 8, 8, 8, 10, 10, 10, 10, 8, 8]
worst-fitness [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
avg-fitness [3.28, 3.63, 4.13, 4.38, 4.43, 4.88, 4.74, 4.66, 4.73, 4.76]
diversity [0.17440183485273306, 0.16772298590235027, 0.17813197354770424, 0.1917185437040455, 0.20062651868584086, 0.18562327440275364, 0.19163506985935538, 0.2001099697666
26, 0.19942667825544305, 0.18117394956229219]
mutation_rate [0.5871989286768987, 0.5876790885642049, 0.5869311005894348, 0.585957632014733, 0.5881991649247235, 0.5890655719693124, 0.5887180966422495, 0.5882289522156277
, 0.5854067837120138, 0.5867128641430254]

C:\Users\danielis\Desktop\pygenetic\examples\nQueens>py -3 nQueens.py
mutation_handlers_weights =  [1.0]
crossover_handlers_weights =  [1.0]
Diversity =  0.15297058540778358
New mutation value =  0.588742465058203
Members left after selection =  100
Best member after selection =  [12, 4, 6, 7, 10, 2, 11, 3, 8, 1, 5, 9]
Best fitness after selection =  8
crossover_indices =  [61 83 42 35 40 48 89 62 49 73 24 87 80 51 21 43 22 31  6 10 16 74  8 26
 37 64 65 45 71  5  7  9 30 17 79 13 72 27 81 36 70 15 86 32 60 55  0  4
 68 20 23 44 46 38 52 41  1 82 78  2 67 50 39 90 19 66 29 28 18 14]
adaptive_mutation value passed =  True
mutation_indexes =  [68 89 51 80 59 15 96 78 21 95 38  5 25 97 23 18 36 91 32 84 83 65 62 90
 34 71 64  6 99 63 46  1 81  3 61  4 77 47 30 43 76 24 56 49 28 14 16 87
 10 98 39 69 48 86 92 45 73 58]
New generation members =  [[12, 4, 6, 7, 10, 2, 11, 3, 8, 1, 5, 9], [6, 4, 11, 3, 12, 9, 8, 1, 5, 2, 10, 7], [3, 5, 2, 10, 9, 6, 11, 7, 1, 4, 12, 8], [3, 2, 9, 5, 11, 7, 10
, 12, 8, 1, 6, 4], [11, 7, 3, 1, 4, 5, 9, 10, 12, 6, 8, 2], [3, 7, 8, 11, 2, 12, 9, 10, 1, 4, 6, 5], [9, 7, 6, 1, 10, 4, 11, 8, 5, 2, 12, 3], [10, 5, 8, 2, 4, 12, 3, 1, 6,
9, 11, 7], [1, 4, 11, 9, 8, 3, 5, 7, 12, 2, 6, 10], [3, 9, 11, 1, 6, 5, 7, 12, 10, 8, 2, 4], [2, 8, 5, 3, 9, 10, 12, 4, 11, 7, 1, 6], [10, 2, 6, 12, 9, 3, 11, 4, 1, 8, 7, 5
```

Fig 11.5 pygenetic terminal output



```
Command Prompt
Epoch 22/30
768/768 [==============================] - 0s 124us/step - loss: 0.6304 - acc: 0.6732
Epoch 23/30
768/768 [==============================] - 0s 152us/step - loss: 0.6305 - acc: 0.6732
Epoch 24/30
768/768 [==============================] - 0s 113us/step - loss: 0.6299 - acc: 0.6758
Epoch 25/30
768/768 [==============================] - 0s 112us/step - loss: 0.6301 - acc: 0.6732
Epoch 26/30
768/768 [==============================] - 0s 120us/step - loss: 0.6297 - acc: 0.6771
Epoch 27/30
768/768 [==============================] - 0s 113us/step - loss: 0.6292 - acc: 0.6771
Epoch 28/30
768/768 [==============================] - 0s 117us/step - loss: 0.6294 - acc: 0.6810
Epoch 29/30
768/768 [==============================] - 0s 115us/step - loss: 0.6302 - acc: 0.6732
Epoch 30/30
768/768 [==============================] - 0s 117us/step - loss: 0.6289 - acc: 0.6771
768/768 [==============================] - 0s 402us/step
[0.6280454347531, 0.6770833333333334]

acc: 67.71%
Loss 0.6280454347531
Diversity =  0.4752261273164497
New mutation value =  0.10972687990985722
Members left after selection =  10
Best member after selection =  [12, 'sigmoid', 10, 'sigmoid', 'sigmoid', 'adam']
Best fitness after selection =  0.5776368441681067
crossover_indices =  [9 2 6 5 0 3]
adaptive_mutation value passed =  True
mutation_indexes =  [7]
New generation members =  [[12, 'sigmoid', 10, 'sigmoid', 'sigmoid', 'adam'], [12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], [5, 'relu', 12, 'sigmoid', 'sigmoid', 'adam'],
 [12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], [5, 'sigmoid', 10, 'sigmoid', 'relu', 'adam'], [12, 'relu', 12, 'relu', 'sigmoid', 'adam'], [12, 'sigmoid', 10, 'relu', 'si
gmoid', 'adam'], [12, 'relu', 2, 'sigmoid', 'sigmoid', 'adam'], [12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], [12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam']]
Length of new generation  10
Best fitness in this generation =  ([12, 'sigmoid', 10, 'sigmoid', 'sigmoid', 'adam'], 0.5776368441681067)
Best among all generations =  ([12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], 0.5606564295788606)
Top fittest chromosomes of this generation:  [([12, 'sigmoid', 10, 'sigmoid', 'sigmoid', 'adam'], 0.5776368441681067), ([12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], 0.58
17104081312815), ([5, 'relu', 12, 'sigmoid', 'sigmoid', 'adam'], 0.5887469227115313), ([12, 'relu', 12, 'sigmoid', 'sigmoid', 'adam'], 0.6213671316703161), ([5, 'sigmoid',
10, 'relu', 'sigmoid', 'adam'], 0.6280454347531), ([2, 'sigmoid', 10, 'relu', 'sigmoid', 'adam'], 0.6303819566965103), ([12, 'relu', 12, 'sigmoid', 'relu', 'adam'], 0.63265
69591959318), ([12, 'relu', 10, 'sigmoid', 'sigmoid', 'adam'], 0.6359995578726133), ([12, 'relu', 2, 'relu', 'sigmoid', 'adam'], 0.6457636108001074), ([5, 'sigm
```

Fig 11.6 ANN Topology Evolution {[12,’sigmoid’,10,’sigmoid’,’sigmoid’,’adam’]}



pyGenetic - An efficient GA Framework

Fitness Function: addition

Step 3.
Enter the Global Population Level Details

Population Size	Crossover Rate	Mutation Rate	Fitness Type
Population size	Crossover Rate	Mutation Rate	MAX

Step 4.
Crossover Details

Crossover Type	Crossover Weight
distinct	Weight

Step 5.
Mutation Details

Mutation Type	Mutation Weight
swap	Weight

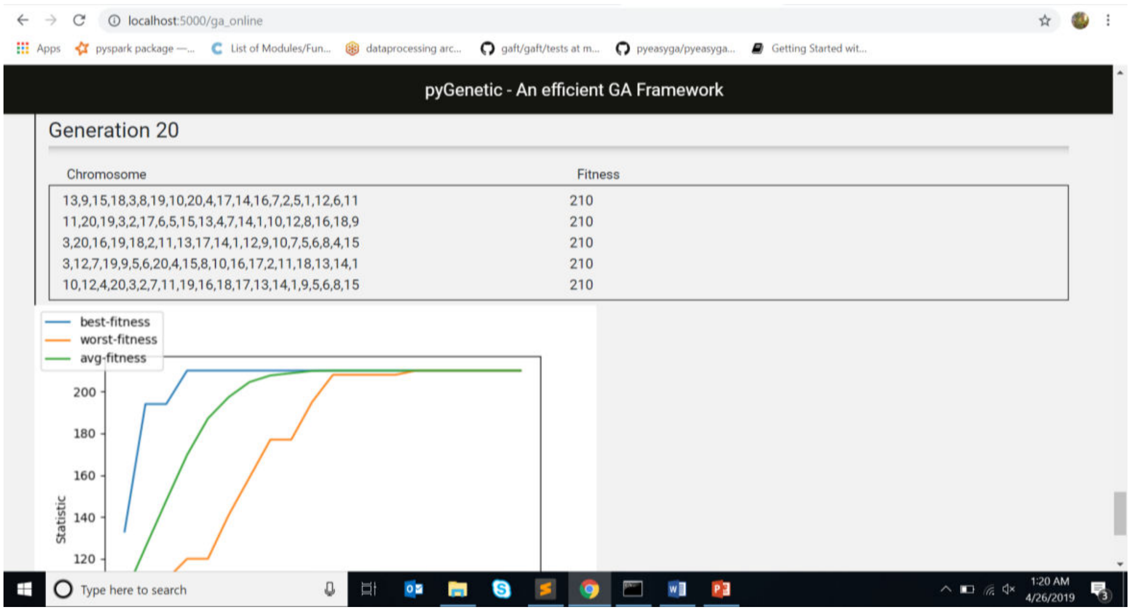
Step 6.
Selection Handler

Selection Type: basic

Step 7.
Evolution Details

Number of evolutions	Adaptive Mutation	pySpark
Number of evolutions	Yes	Yes

Fig 11.7 GA Input UI



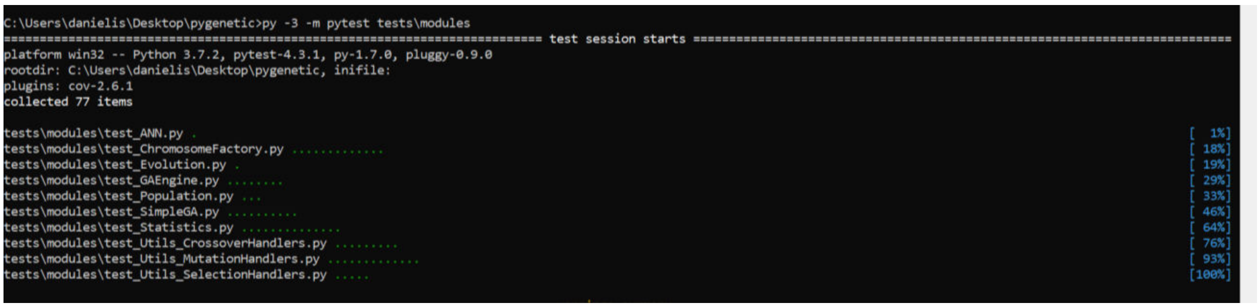
localhost:5000/ga_online

pyGenetic - An efficient GA Framework

Generation 20

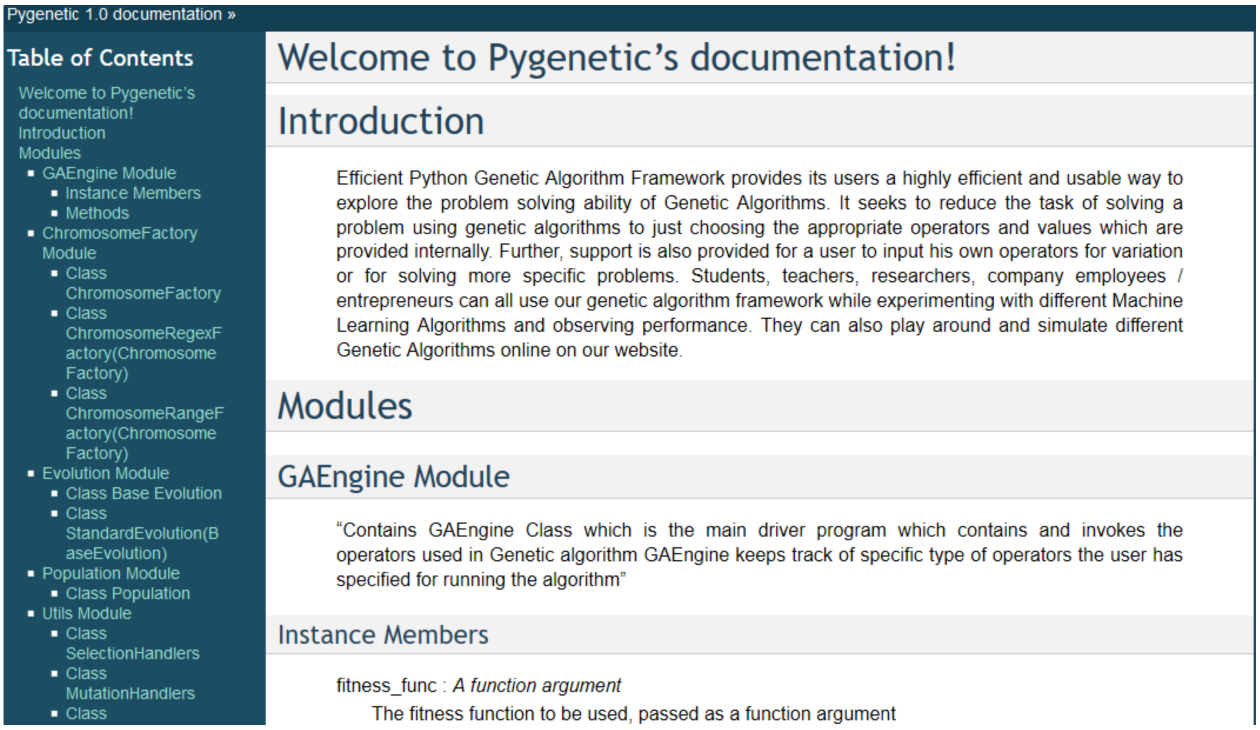
Chromosome	Fitness
13,9,15,18,3,8,19,10,20,4,17,14,16,7,2,5,1,12,6,11	210
11,20,19,3,2,17,6,5,15,13,4,7,14,1,10,12,8,16,18,9	210
3,20,16,19,18,2,11,13,17,14,1,12,9,10,7,5,6,8,4,15	210
3,12,7,19,9,5,6,20,4,15,8,10,16,17,2,11,18,13,14,1	210
10,12,4,20,3,2,7,11,19,16,18,17,13,14,1,9,5,6,8,15	210

Fig 11.8 GA Execution from UI



```
C:\Users\danielis\Desktop\pygenetic>py -3 -m pytest tests\modules
=========================================================================== test session starts ===========================================================================
platform win32 -- Python 3.7.2, pytest-4.3.1, py-1.7.0, pluggy-0.9.0
rootdir: C:\Users\danielis\Desktop\pygenetic, inifile:
plugins: cov-2.6.1
collected 77 items

tests\modules\test_ANN.py .                                                                                                                                          [  1%]
tests\modules\test_ChromosomeFactory.py .............                                                                                                                [ 18%]
tests\modules\test_Evolution.py .                                                                                                                                    [ 19%]
tests\modules\test_GAEngine.py ........                                                                                                                              [ 29%]
tests\modules\test_Population.py ...                                                                                                                                 [ 33%]
tests\modules\test_SimpleGA.py ..........                                                                                                                            [ 46%]
tests\modules\test_Statistics.py ..............                                                                                                                      [ 64%]
tests\modules\test_Utils_CrossoverHandlers.py .........                                                                                                              [ 76%]
tests\modules\test_Utils_MutationHandlers.py .............                                                                                                           [ 93%]
tests\modules\test_Utils_SelectionHandlers.py .....                                                                                                                  [100%]
```

Fig 11.9 Testing the Framework



Pygenetic 1.0 documentation »

Table of Contents

Welcome to Pygenetic's documentation!
Introduction
Modules
- GAEngine Module
  - Instance Members
  - Methods
- ChromosomeFactory Module
  - Class ChromosomeFactory
  - Class ChromosomeRegexFactory(ChromosomeFactory)
  - Class ChromosomeRangeFactory(ChromosomeFactory)
- Evolution Module
  - Class Base Evolution
  - Class StandardEvolution(BaseEvolution)
- Population Module
  - Class Population
- Utils Module
  - Class SelectionHandlers
  - Class MutationHandlers
  - Class

Welcome to Pygenetic's documentation!

Introduction

Efficient Python Genetic Algorithm Framework provides its users a highly efficient and usable way to explore the problem solving ability of Genetic Algorithms. It seeks to reduce the task of solving a problem using genetic algorithms to just choosing the appropriate operators and values which are provided internally. Further, support is also provided for a user to input his own operators for variation or for solving more specific problems. Students, teachers, researchers, company employees / entrepreneurs can all use our genetic algorithm framework while experimenting with different Machine Learning Algorithms and observing performance. They can also play around and simulate different Genetic Algorithms online on our website.

Modules

GAEngine Module

"Contains GAEngine Class which is the main driver program which contains and invokes the operators used in Genetic algorithm GAEngine keeps track of specific type of operators the user has specified for running the algorithm"

Instance Members

fitness_func : *A function argument*
The fitness function to be used, passed as a function argument

Fig 11.10 Our documentation hosted on readthedocs.org

CHAPTER-12

CONCLUSIONS

The outcome of this project, **pygenetic**, was to make a user friendly and generic genetic algorithm API. As observed from the results, our API succeeds in being good in terms of user friendliness as well as efficiency (execution time) when compared with most other currently available Genetic Algorithm API. Also, our API succeeds in implementing optimizations like adaptive mutation, hall of fame injection, early halts which most APIs ignore. It also successfully provides users the option for population control which is very useful for research purposes. The usage of Spark makes execution of GAs very scalable with users having the ability to run very large Genetic Algorithms in less time by deploying more Spark Workers. Finding the best API topology is also very fast with our API leveraging Spark parallelizations. The Website for Online Execution of Genetic Algorithms was also successfully developed with users having the capability to execute Genetic Algorithms from the User Interface itself, thereby being very useful for students and teachers. The API was also well documented on readthedocs for new users to be easily able to pick up our API. Hence, the development and publishing of the python package “pygenetic” was found to be an overall success.

CHAPTER-13

FURTHER ENHANCEMENTS

The python API pygenetic can have many future enhancements. It currently only supports 1 dimensional chromosomes and 1 dimensional chromosome based operations. It can be made to support it in future versions. Also more functions can be added to Utils in future versions.

More efficient parallelization can be implemented in many other types of selection handlers which can be made part of pygenetic future versions. Also more optimizations like long and short term memory can be implemented in newer versions.

The GA Online Execution Website must be made more robust to heavy load conditions by various load based optimizations such as batching outputs to some limit and then sending responses. Also the documentation of out API and website can be further improved with the inclusion of various tutorial videos.

REFERENCES / BIBLIOGRAPHY

[1] Lin, Chen. "An adaptive genetic algorithm based on population diversity strategy." In *2009 Third International Conference on Genetic and Evolutionary Computing*, pp. 93-96. IEEE, 2009.

[2] Verma, Abhishek, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. "Scaling genetic algorithms using mapreduce." In *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pp. 13-18. IEEE, 2009.

[3] Al-Khateeb, Belal. " Solving 8-Queens Problem by Using Genetic Algorithms, Simulated Annealing, and Randomization Method" In *International Conference on Developments in eSystems Engineering*

[4] Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, and Showkat Gani, "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator," *Computational Intelligence and Neuroscience, vol. 2017, Article ID 7430125*

[5] Rivero, D., Dorado, J., Rabuñal, J.R., Pazos, A.: Modifying genetic programming for artificial neural network development for data mining. *Soft Computing - A Fusion of Foundations, Methodologies and Applications 13(3), 291–305 (2008)*

REFERENCES / BIBLIOGRAPHY

- https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164
- https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm
- https://stackoverflow.com/questions/20290831/how-to-perform-rank-based-selection-in-a-genetic-algorithm
- https://www.tutorialspoint.com/pyspark/index.htm
- https://deap.readthedocs.io/en/master/
- http://flask.pocoo.org/docs/1.0/tutorial/
- https://cloud.google.com/dataproc/

APPENDIX A

DEFINITIONS, ACRONYMS AND ABBREVIATIONS

The key words given below are used in the document unambiguously. The meaning of these keywords:

- GA – Stands for Genetic Algorithm and is used in different places to refer to components pertaining to the framework.
- GA = Genetic Algorithm. Genetic algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. Genetic algorithms are inspired by Darwin's theory about evolution where some chromosomes in every generation are evolved over many generations. It is the survival of the fittest.
- A chromosome consist of genes, blocks of DNA.
- Possible settings for a trait (e.g. blue, brown) are called alleles.
- Complete set of genetic material (all chromosomes) is called genome.
- Particular set of genes in genome is called genotype. The genotype is with later development after birth base for the organism's phenotype, its physical and mental characteristics, such as eye color, intelligence etc.
- During reproduction, first occurs recombination (or crossover). Genes from parents form in some way the whole new chromosome.
- Mutation means, that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.
- The fitness of an organism is measured by success of the organism in its life. In GA, it is modeled as a mathematical function.
- MR = Map Reduce. It is a popular parallelization paradigm.
- ANN = artificial neural network. A ML algorithm inspired by our own biological brain neural networks.