



دانشکده مهندسی

گروه مهندسی کامپیوتر

گرایش هوش مصنوعی

الگوریتم ژنتیک ابتدایی

استاد:

دکتر مجتبی روحانی

دانشجو:

علی گلی

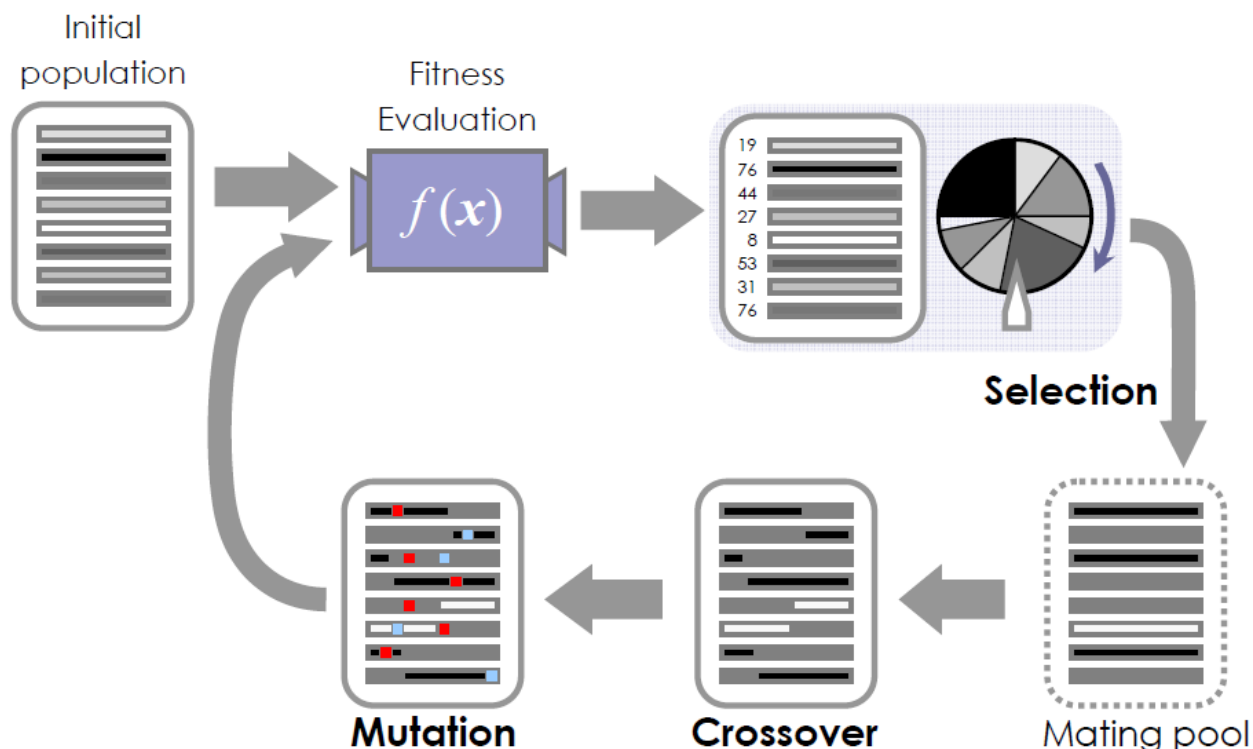
آبان ماه

سال ۱۴۰۲

۱ الگوریتم

۱.۱ تعریف الگوریتم

الگوریتم CGA یک الگوریتم ژنتیک ابتدایی می باشد که در آن یک جمعیت ابتدایی ساخته می شود سپس عملیات fitness روی این جمعیت صورت می گیرد که در آن به هر فرد یک امتیازی برحسب فنوتیپ آن ها اختصاص داده می شود. سپس در selection به هر فرد بابت امتیازی که دارد یک شانس برای انتخاب شدن داده می شود. یعنی آنان که فنوتیپ بهتری دارند به احتمال بیشتری انتخاب می شوند. از جمعیت انتخاب شده عملیات crossover انجام می شود. سپس عملیات mutation انجام می شود. این روند یک روند تکراری است. (شکل ۱)



شکل ۱

۲ برنامه CGA

۱.۲ صورت کلی برنامه

برنامه شامل ۳ کلاس پایه ای می باشد. کلاس اول CGA می باشد که برای استفاده از آن باید ارث بری صورت بگیرد، و طرح کلی الگوریتم ژنتیک را در بر می گیرد. کلاس دوم `functionEvaluations` می باشد، که می توان در آن توابعی را تعریف کرد و از آنها استفاده کرد. کلاس سوم `main` می باشد، که کلاس CGA ارث بری شده است و متغیرهای مورد نیاز در آن تعریف شده اند. همچنین توابع بیشتری جهت اجرای بیش از ۱ بار استفاده از برنامه نیز به آن اضافه شده است که کمک می کند چندین بار الگوریتم CGA را اجرا کنید و بین نتایج مختلف میانگین گیری کنید.

توابع مورد نیاز این کلاس؛ به ترتیب از بالا به پایین نوشته می‌شود. این کلاس شامل تابع `init` هست که ابتدا تابع `fitness` را تنظیم می‌کند. سپس تابع `run` الگوریتم کلی را اجرا می‌کند. در تابع `run` ابتدا تابع `initialization` را اجرا می‌کند که ماتریس جمعیت، ماتریس والدین، تنظیم متغیرهای جمع آوری کننده بدترین، بهترین و متوسط فنوتیپ هر نسل مقدار دهی اولیه می‌کند. ماتریس جمعیت از بدترین حالت شروع به کار می‌کند. سپس در تابع `run` در یک حلقه نسل افزایش پیدا می‌کند و در هر نسل به ترتیب از چپ به راست توابع `selection, crossover, mutation` اجرا می‌گردد. حال داده‌ها را جمع آوری می‌کنیم. داده‌های جمع آوری شده شامل: بهترین فنوتیپ نسل، بدترین فنوتیپ نسل، متوسط فنوتیپ نسل می‌باشد. حال که یک نسل به اتمام رسیده بعضی داده‌ها نیازمند متغیردهی هستند. ابتدا ماتریس والدین به ۰ مقدار دهی می‌شود. سپس نسل افزایش می‌یابد.

تابع `selection` در ابتدا مقدار `fitness` تمامی افراد نسل را محاسبه و جمع می‌کند. سپس مقدار `fitness` هر فرد را تقسیم بر مقدار جمع شده می‌کند. حال احتمال انتخاب هر فرد به دست می‌آید. برای پیاده سازی الگوریتم `roulette wheel` با تابع `choices` از ماژول `random` با وزن احتمال انتخاب هر فرد اجازه می‌دهیم هر فرد با توجه به احتمال `fitness` انتخاب گردد. سپس به تعداد افراد جامعه با الگوریتم `roulette wheel` ما والدین را انتخاب می‌کنیم.

تابع `crossover` شروع می‌کند به انتخاب دو والد از ماتریس والدین و `crossover` با احتمال `Pc` اجازه `crossover` می‌دهد. در غیراینصورت همان دو نفر به عنوان دو فرزند خواهند بود.

تابع `mutation` برای همه افراد جامعه اجرا می‌شود. هر بیت از هر فرد جامعه با احتمال `Pm` می‌تواند جهش پیدا کند.

تابع `function_evaluation` می‌تواند ارث بری گردد و به نحوه دلخواه نوشته شود.

کد مربوط به این کلاس در شکل های ۲ و ۳ انتهای گزارش قابل مشاهده است.

۳.۲ کلاس `functionEvaluations`

این کلاس به عنوان یک کلاس کمکی است. در آن می‌توانید توابع مختلف `fitness` را برنامه ریزی کنید و هر تابع را به عنوان یک آرگومان استفاده کنید. همچنین یک تابع `map_to_range` وجود دارد که در آن می‌توانید مقادیر ورودی هر تابع را به یک بازه ای از اعداد اختصاص دهید.

۴.۲ کلاس `main`

کلاس `main` از کلاس `CGA` ارث بری کرده است. متغیرهای اولیه مقدار دهی شده‌اند. `L` اندازه هر ژن را نشان می‌دهد. `Meu` اندازه جمعیت را نشان می‌دهد. `Pc` احتمال کراس اوور را نشان می‌دهد. `Pm` احتمال جهش را نشان می‌دهد. همچنین `MaxGen` تعداد نسل‌ها را نشان می‌دهد. تابع اضافه شده `runs` اجازه می‌دهد چندین بار الگوریتم `CGA` اجرا شود.

۳ نتیجه

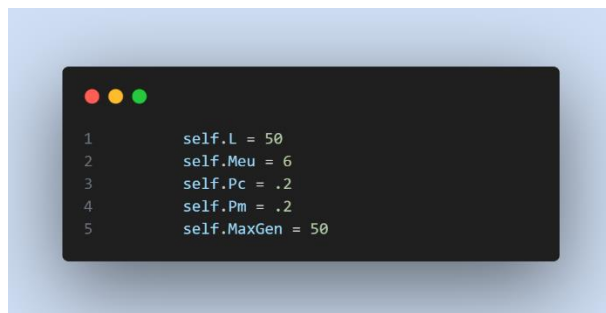
۳.۱ اجرای دو تابع fitness

دو تابع fitness با فرمول های زیر اجرا گردیده و ۱۰ بار اجرا شده است.

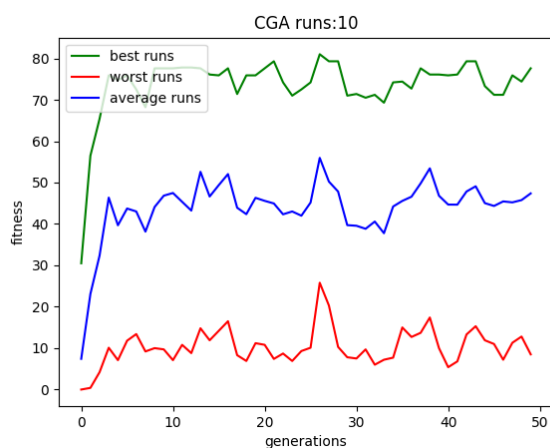
$$f(x) = x^2$$

$$f(x) = |\cos(x) e^{-\frac{|x|}{5}}|$$

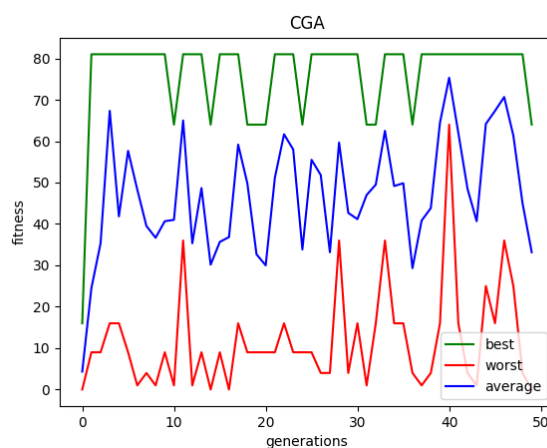
متغیر های اولیه به شرح ذیل مقدار دهی شده‌اند.



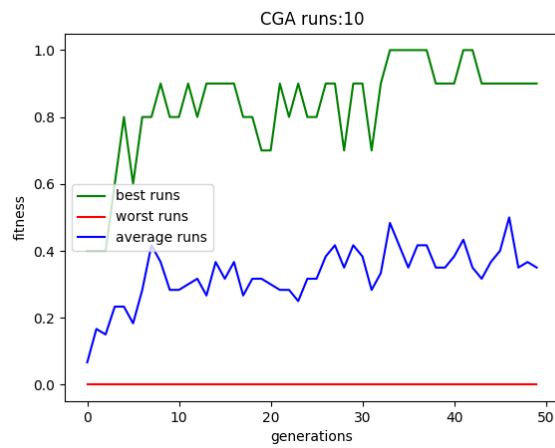
شکل ۲



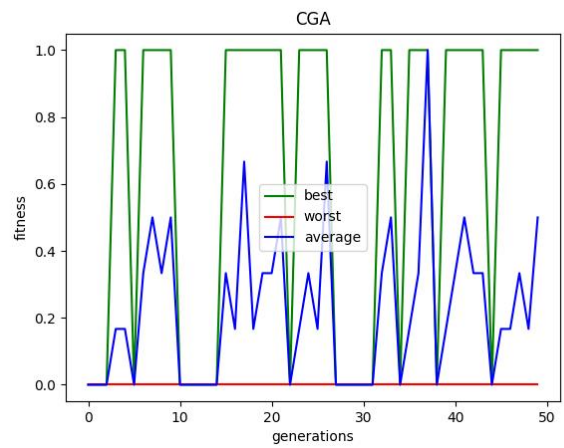
شکل ۴



شکل ۳

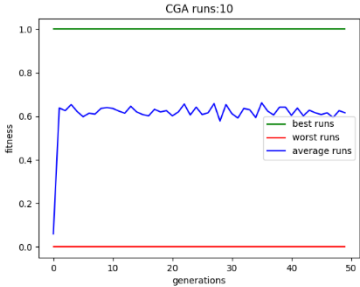
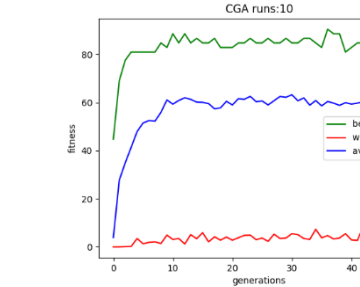
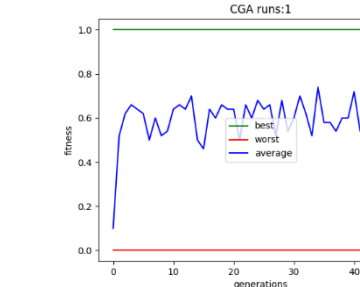
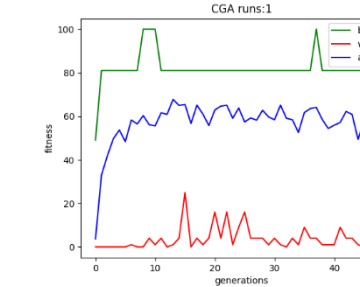
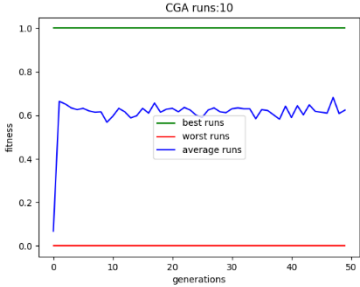
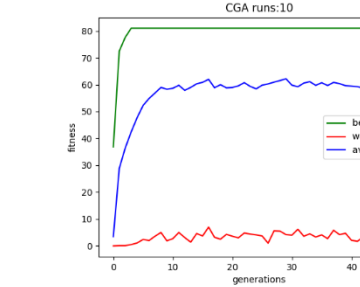
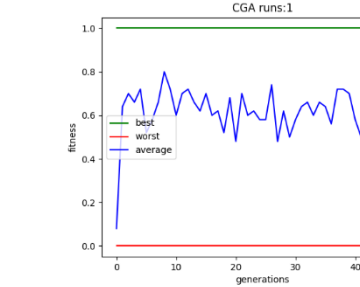
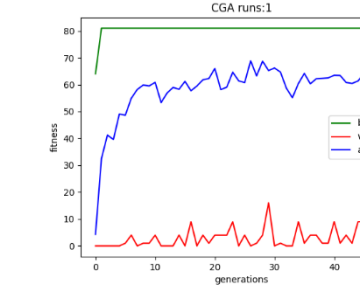
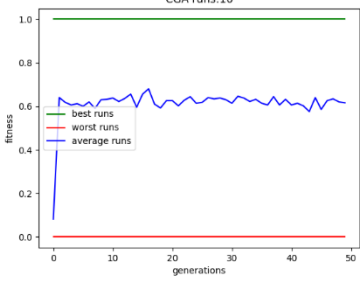
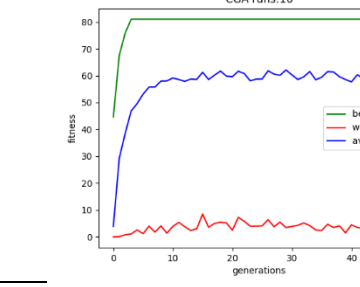
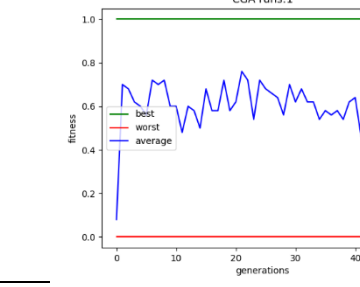
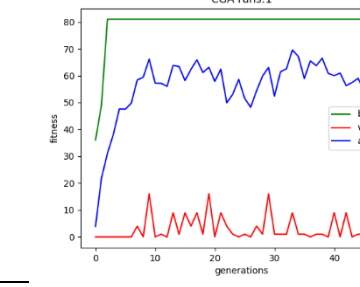
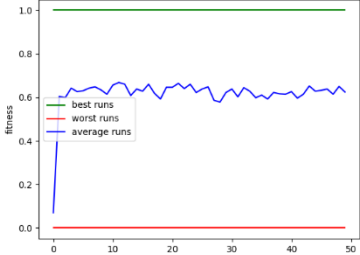
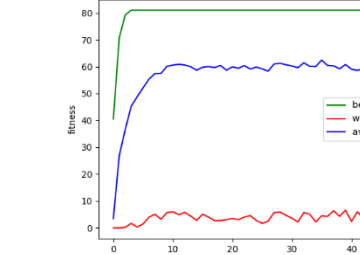
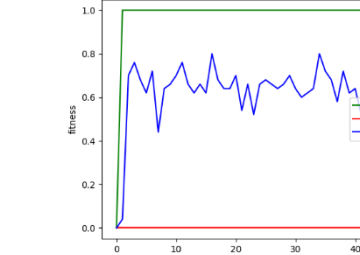
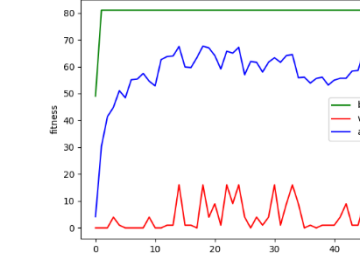


شکل ۶



شکل ۵

شکل ۳ نشان دهنده اجرای تابع اول برای ۱ بار، شکل ۴ نشان دهنده اجرای تابع دوم برای ۱۰ بار، شکل ۵ و ۶ نشان دهنده اجرای تابع دوم به ترتیب ۱ بار و ۱۰ بار است. جدول زیر نمودار مربوط به الگوریتم در ژن ها به طول ۱۰، ۲۰، ۳۰ و ۴۰ را نشان می دهد. ($\text{Meu}=50$, $\text{Pc}=0.1$, $\text{Pm}=0.1$, $\text{MaxGen}=50$)

طول ژن	یک بار اجرا	۱۰ بار اجرا		
تابع اول	تابع دوم	تابع اول	تابع دوم	
				۱۰
				۲۰
				۳۰
				۴۰

```

import numpy as np
from numpy.typing import NDArray
from typing import Tuple, Any, List, Callable
from decimal import *
import random
import copy
import matplotlib.pyplot as plt

class CGA:
    population: NDArray
    parent: NDArray

    Meu: int
    L: int
    Pc: int
    Pm: int
    MaxGen: int
    generation: int

    fitness_func: Callable

    plot: plt

    best_cases_generation: List
    worst_cases_generation: List
    ave_cases_generation: List

    def __init__(self, fitness_func:Callable) -> None:
        self.fitness_func = fitness_func
        ...

    def run(self, plotting=True):
        self.initialization()
        self.generation = 0

        while self.generation < self.MaxGen:
            # Main Algorithm
            self.selection()
            self.crossover()
            self.mutation()
            # Gathering Data
            best, worst, ave = self.worst_best_average()
            self.best_cases_generation.append(best)
            self.worst_cases_generation.append(worst)
            self.ave_cases_generation.append(ave)
            # Reset Variables for Next Generation
            self.parent = np.zeros((self.Meu,self.L))
            self.generation += 1

        if plotting:
            #plotting results
            self.plotting()

    def initialization(self):
        # self.population = (np.random.rand(self.Meu, self.L) > .5) * 1
        self.population = (np.random.rand(self.Meu, self.L) > 1) * 1
        self.parent = np.zeros((self.Meu,self.L))

        self.best_cases_generation, self.worst_cases_generation, self.ave_cases_generation =
        [], [], []
        self.plot = plt

    def selection(self) -> None:
        # sigma(function evaluation)
        # this number is to large
        # we use Desimal Module to control this large data
        sum_eva = Decimal(1)
        for ind in self.population:sum_eva += self.function_evaluation(ind)
        # calculate probability for every population
        prob = np.array([[index,self.function_evaluation(self.population[index])/sum_eva] for index
in range(len(self.population))], dtype=np.float64)
        # roulette wheel
        for i in range(self.Meu):
            index = int(random.choices(prob[:,0], weights=prob[:,1])[0])
            self.parent[i] = self.population[index].copy()

```

```

1  def crossover(self) -> None:
2      for i in range(0, self.Meu, 2):
3          # copy two parents in "a" and "b" variables
4          a, b = (self.parent[i].copy(), self.parent[i+1].copy())
5          r = random.random()
6          if r<self.Pc:
7              # crossover happens
8              cutoff = random.randint(1, self.L-2)
9              temp = b[:cutoff].copy()
10             b[:cutoff], a[:cutoff] = a[:cutoff], temp
11             self.parent[i], self.parent[i+1] = a, b
12             self.population = self.parent.copy()
13
14
15  def mutation(self) -> None:
16      # Choose Individuals
17      for ind in self.population:
18          # search all genes
19          for index in range(self.L):
20              r = random.random()
21              if r<self.Pm:
22                  # flip the gene with r probability
23                  ind[index] = 1 - ind[index]
24
25  def worst_best_average(self, worst_case=None, best_case=None) -> Tuple:
26      # best evaluation is worst case from the start
27      best = self.function_evaluation(worst_case) if not worst_case == None \
28      else self.function_evaluation(np.zeros((self.L,)))
29      # worst evaluation is best case from the start
30      worst = self.function_evaluation(best_case) if not best_case == None \
31      else self.function_evaluation(np.ones((self.L,)))
32      sum_eva = Decimal(0)
33      for ind in self.population:
34          ind_eva = self.function_evaluation(ind)
35          if ind_eva > best:
36              best = copy.deepcopy(ind_eva)
37          if ind_eva < worst:
38              worst = copy.deepcopy(ind_eva)
39          sum_eva += ind_eva
40      return best, worst, sum_eva/self.Meu
41
42
43  def function_evaluation(self, args) -> int:
44      return self.bit_to_int(args)
45
46  def bit_to_int(self, bits) -> int:
47      bit_string = ''
48      for bit in bits: bit_string+=str(int(bit))
49      # Convert string of bits to integer
50      return int(bit_string, 2)
51
52
53  def plotting(self):
54      plt.figure()
55      self.plot.title("CGA")
56      self.plot.xlabel('generations')
57      self.plot.ylabel('fitness')
58
59      generations = [i for i in range(self.MaxGen)]
60
61      self.plot.plot(generations, self.best_cases_generation, label="best", c='g')
62      self.plot.plot(generations, self.worst_cases_generation, label="worst", c='r')
63      self.plot.plot(generations, self.ave_cases_generation, label="average", c='b')
64
65      plt.legend()
66      self.plot.show()

```

```

1 class functionEvaluations:
2     def function_evaluation1(self, val, old_min, old_max):
3         val = self.map_to_range(val, 0, 10, old_min, old_max)
4         return val ** 2
5     def function_evaluation2(self, val, old_min, old_max):
6         val = self.map_to_range(val, -10, 10, old_min, old_max)
7         return int(np.abs(np.cos(val) * np.exp(-1*np.abs(val)/5)))
8
9     def map_to_range(self, val, new_min, new_max, old_min, old_max) -> int:
10        old_range = old_max - old_min
11        new_range = new_max - new_min
12        scaled = (val - old_min) / old_range
13        return int(new_min + (scaled * new_range))

```

```

1 class main(CGA):
2
3     def __init__(self, fitness_func: Callable) -> None:
4         self.L = 40
5         self.Meu = 50
6         self.Pc = .1
7         self.Pm = .1
8         self.MaxGen = 50
9         super().__init__(fitness_func)
10
11     def function_evaluation(self, args) -> int:
12         return self.fitness_func(self.bit_to_int(args), self.bit_to_int(np.zeros((self.L,))), self.bit_to_int(np.ones(self.L
13 ,)))
14
15     def runs(self):
16         self.runs_iterator = 10
17         # Matrix: Each Row shows run index and each coloumn show generation
18         self.best_runs = np.zeros((self.runs_iterator, self.MaxGen))
19         self.worst_runs = np.zeros((self.runs_iterator, self.MaxGen))
20         self.ave_runs = np.zeros((self.runs_iterator, self.MaxGen))
21         for i in range(self.runs_iterator):
22             self.run(plotting=False)
23             self.best_runs[i] = np.array(self.best_cases_generation)
24             self.worst_runs[i] = np.array(self.worst_cases_generation)
25             self.ave_runs[i] = np.array(self.ave_cases_generation)
26         self.plotting_runs()
27     def plotting_runs(self):
28         plot = plt
29         plot.title(f"CGA runs:{self.runs_iterator}")
30         self.plot.xlabel('generations')
31         self.plot.ylabel('fitness')
32
33         generations = [i for i in range(self.MaxGen)]
34
35         plot.plot(generations, np.mean(self.best_runs, axis=0), label="best runs", c='g')
36         plot.plot(generations, np.mean(self.worst_runs, axis=0), label="worst runs", c='r')
37         plot.plot(generations, np.mean(self.ave_runs, axis=0), label='average runs', c='b')
38
39         plt.legend()
40         plt.show()

```


۴.۴ نحوه اجرای کدها

```
1 cga = main(functionEvaluations().function_evaluation1)
2 cga.run()
3 cga.runs()
4 cga = main(functionEvaluations().function_evaluation2)
5 cga.run()
6 cga.runs()
```

۵.۴ کتابخانه ها

```
1 import numpy as np
2 from numpy.typing import NDArray
3 from typing import Tuple, Any, List, Callable
4 from decimal import *
5 import random
6 import copy
7 import matplotlib.pyplot as plt
```