Real-Valued Genetic Algorithm

الگوریتم وراثتی حقیقی

مشخصات

درس : مبانی هوش محاسباتی

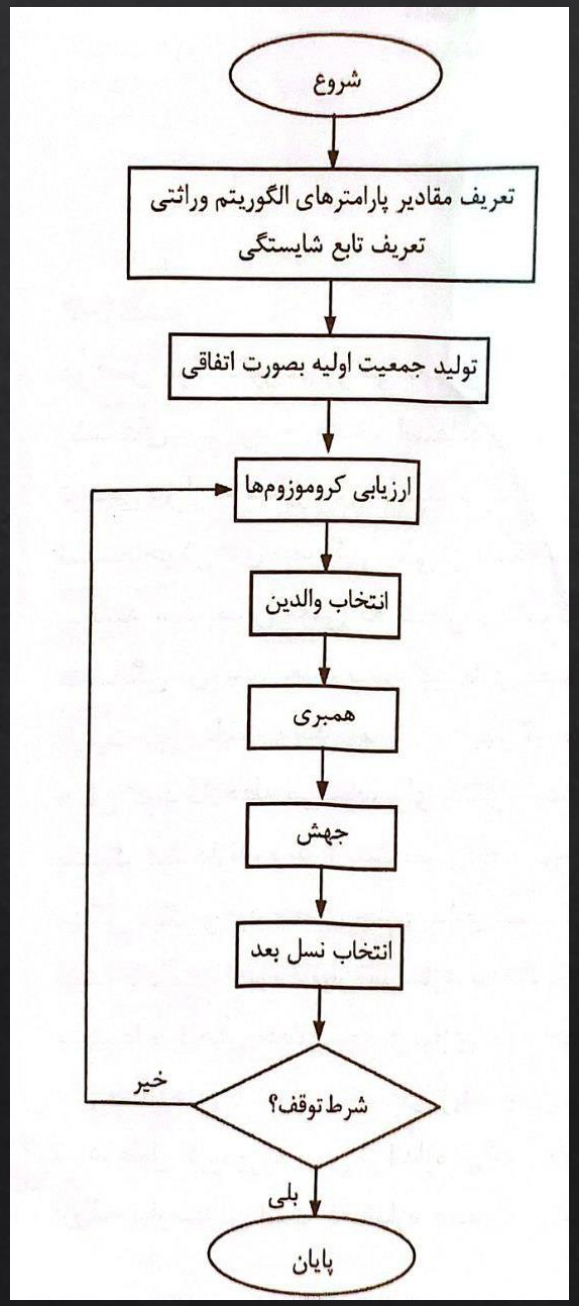
زمان : ترم اول 1402 - 1403

مدرس : دکتر عباس بحرالعلوم

اعضای گروه : امیرحسین ابوالحسنی, حمیدرضا بازیار, صدرا کوچک زاده

موضوع : پیاده سازی الگوریتم وراثتی حقیقی

پیاده سازی



بررسی اجمالی

الگوریتم وراثتی حقیقی , شامل 8 گام که در فلوچارت مقابل به آن اشاره شده است می باشد.

در این پروژه , با استفاده از شیوه شی گرایی(OOP) , این الگوریتم پیاده سازی شده است؛ یعنی ساختار کلی الگوریتم به عنوان یک کلاس در نظر گرفته شده و مراحل الگویتم به عنوان تابع هایی در درون کلاس پیاده سازی شده اند.

برای استفاده از این الگوریتم, کافی است که یک شی(object) از کلاس RGA ساخته و تابع ()Run را برای آن شی صدا بزنیم.

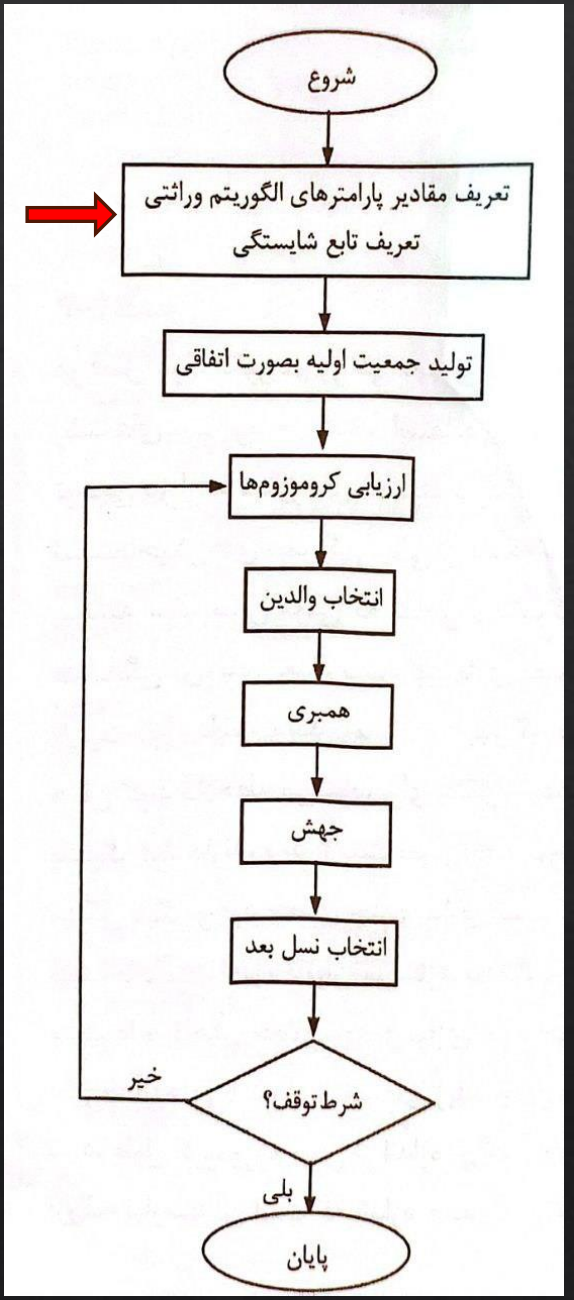
لازم به ذکر است در این پیاده سازی, الگوریتم چند بار اجرا می شود و در هر نسل, بین best so far ها و همچنین mean fitness میانگین گرفته می شود و درنهایت نمودار average mean fitness و average best so far نمایش داده می شود.

کتابخانه های های به کار رفته

```python
from random import random, uniform, gauss
import matplotlib.pyplot as plt
from tqdm import tqdm, trange
from tabulate import tabulate
import os
```

- از ماژول **random** برای تولید اعداد تصادفی در عملیات های همبری و جهش و همچنین تولید ماتریس اولیه مورد استفاده قرار می گیرد.
  - Uniform : تولید عدد رندم با توزیع یکنواخت
  - gauss: تولید عدد رندوم با توزیع نرمال
  - random: تولید عدد رندومی بین 0 و 1 با توزیع یکنواخت

- از ماژول **os** برای کار با فایل ها استفاده می کنیم.
- از ماژول **matplotlib** برای رسم نمودار استفاده می کنیم.
- از ماژول**tqdm** برای نمایش وضعیت پیشرفت الگوریتم استفاده می کنیم.
- برای نمایش اطلاعات مربوط به الگوریتم و چاپ پارامتر ها, از ساختار جدولی مربوط به ماژول **tabulate** استفاده می کنیم.



مقدار دهی پارامترها

مقدار دهی پارامترهای اولیه, اولین گام از الگوریتم وراثتی می باشد. همان طور که اشاره شد این الگوریتم به شیوه شی گرایی پیاده سازی شده است. برای مقدار دهی پارامتر های اولیه, پارامتر ها را از طریق تابع سازنده (__init__) مقدار دهی می کنیم.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

در تابع سازنده کلاس BGA , 9 مقدار را در هنگام ساخت شی از روی کلاس, دریافت می کنیم.

- **max_gen** : مقدار دریافتی **max_gen** یا همان **Maximum Generation** به معنای حداکثر تعداد نسل هایی که توسط این الگوریتم تولید شوند می باشد.
  مطمئنا این مقدار باید یک عدد صحیح(int) باشد. مقدار پیش فرض برای این پارامتر 50 نسل قرار داده شده است.
  مهم ترین کاربرد این پارامتر استفاده در شرط توقف می باشد. این مقدار را در خاصیتی با نام max_gen ذخیره می کنیم.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

- **target_function**: تابعی می باشد که قصد بهینه سازی آن را داریم. این تابع را در خاصیتی با نام **function** قرار می دهیم.
- **fitness_function**: یک تابع است که میزان کارآمدی کروموزوم های مسئله را نشان می دهد. این تابع را در خصوصیتی به اسم **fit_funct** ذخیره می کنیم.
- **population**: پارامتر **population** به تعداد کروموزوم ها یا به عبارتی اندازه ی جمعیت اشاره می کند. این پارامتر یک عدد صیح می باشد که مقدار آن در خصوصیتی با همین نام ذخیره می شود. این پارامتر حتما باید زوج باشد. (به علت روشی که برای انتخاب والدین به کار گرفته شده است.)

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

- **crossover_rate**: پارامتر بعدی که در ورودی دریافت می شود, نرخ همبری می باشد. این مقدار عددی مثبت و کمتر از یک است.
- **mutation_rate**: پارامتر دیگر, نرخ جهش می باشد. این مقدار عددی مثبت و کمتر از یک است.
- **function_config**: این پارامتر از نوع لیست می باشد. هر درایه از این لیست یک دیکشنری می باشد که دارای دو کلید high و low است. این دیکشنری دامنه ی یک ژن را نمایش می دهد و طبیعتا سایز لیست برابر با بعد ورودی تابع است.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

- **dim**: به تعداد پارامتر های تابع اشاره دارد, برابر است با طول لیستی که دامنه متغیر ها درآن قراردارد(**function_config**).
- **plot_dir**: آدرس پوشه ای را که نمودارهای تهیه شده از اجرا های این الگوریتم تولید شده اند را نشان می دهد. این مقدار را در خصوصیت plot_save_dir ذخیره می کنیم. اگر این مقدار را تعیین نکنیم, نمودار های ایجاد شده صرفا نمایش داده می شده و ذخیره نمی شوند.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

- **run_rga**: به تعداد دفعاتی که این الگوریتم اجرا شود اشاره می کند. این پارامتر بیانگر این می باشد که چند بار الگوریتم به طور کامل اجرا شود. مقدار این خصوصیت را که یک عدد صحیح می باشد در خصوصیت **runs** ذخیره می کنیم. اگر این مقدار را تعیین نکنیم, به طور پیش فرض 30 در نظر گرفته می شود.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

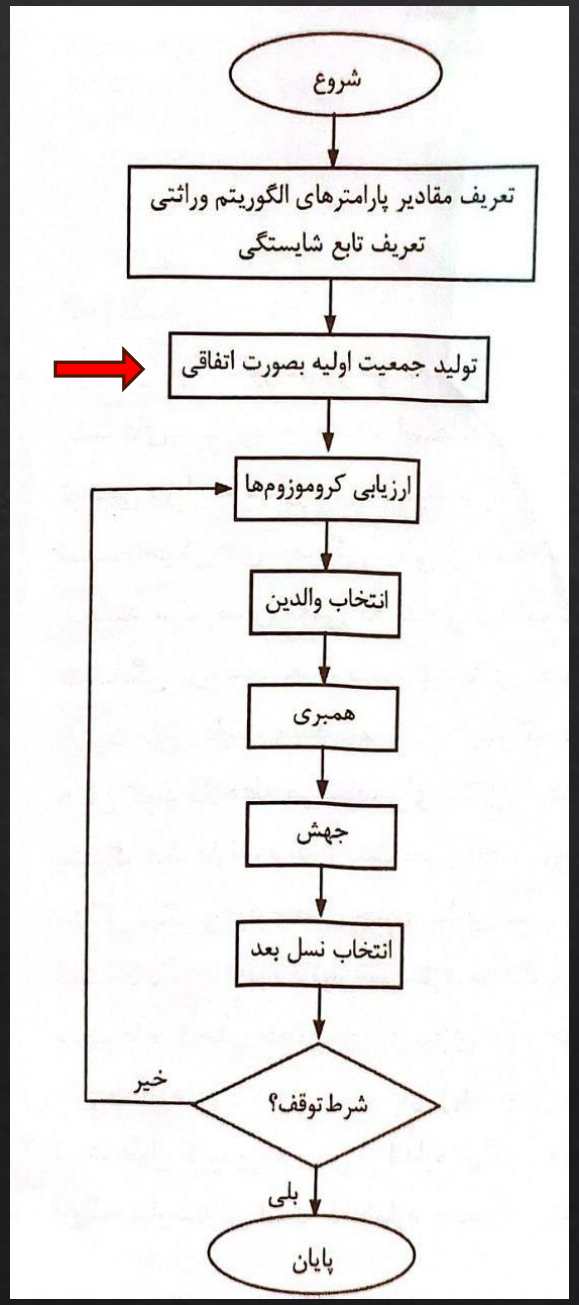
علاوه بر مقادیر دریافتی از ورودی تابع سازنده, این کلاس شامل خصوصیات دیگری نیز می باشد.

- **population_matrix** : این خصوصیت که در ابتدا مقدار **None** را دارد, یک ماتریس است که در ادامه کروموزوم های جمعیت در آن قرار می گیرند. برای دستیابی به کروموزوم ها از این خصوصیت استفاده می کنیم.
- **last_gen**: این خصوصیت نشان می دهد در حال حاضر چند نسل را پشت سر گذاشته ایم. این خصوصیت در ابتدا مقدار صفر را دارد و با هر بار تولید نسل جدید مقدار آن یک واحد افزایش می یابد.

```python
def __init__(self, target_function, fitness_function, population,
             crossover_rate, mutation_rate, function_config, plot_dir=None, max_gen=50, run_rga=30):

    self.func_config = function_config  # a list of dicts containing boundary of each dimension : [{"low": a, "high":b}, ...]
    self.population_matrix = None
    self.function = target_function  # target function
    self.fit_func = fitness_function  # the fitness function
    self.dim = len(function_config)  # dim of the function
    self.population = population if population % 2 == 0 else population + 1  # Note: Population must be an even integer.
    self.pc = crossover_rate
    self.pm = mutation_rate
    self.max_gen = max_gen  # maximum number of generations
    self.last_gen = 0
    self.history = {"mean_fitness": [], "best_so_far": [], "avg_mean_fitness": [], "avg_best_so_far": []}
    self.best_so_far = {'fitness': 0, "chromosome": list()}
    self.best_current = {'fitness': 0, "chromosome": list()}
    self.runs = run_rga
    self.best_answers = []  # best decoded value of the chromosomes found in each full run of the algorithm
    self.plot_save_dir = None if plot_dir is None else plot_dir
```

- **best_so_far, best_current**: به ترتیب برای نگهداری بهترین جواب دیده شده در هر نسل و بهترین جواب تا این نسل استفاده می شود.
- **best_answers** : این خصوصیت یک لیست می باشد که در هربار اجرای کامل, بهترین جواب پیدا شده را در خود نگه می دارد.

تولید جمعیت تصادفی اولیه



گام دوم از این الگوریتم, تولید جمعیت اولیه به صورت تصادفی است. هر کروموزوم به تعداد dim ژن دارد. هر ژن را در یک خانه نگه می داریم. پس طول یک کروموزوم برابر با dim می باشد.
برای مقدار دهی اولیه ماتریس جمعیت با اعداد رندوم کافی است که طبق رابطه ی زیر عمل کنیم:

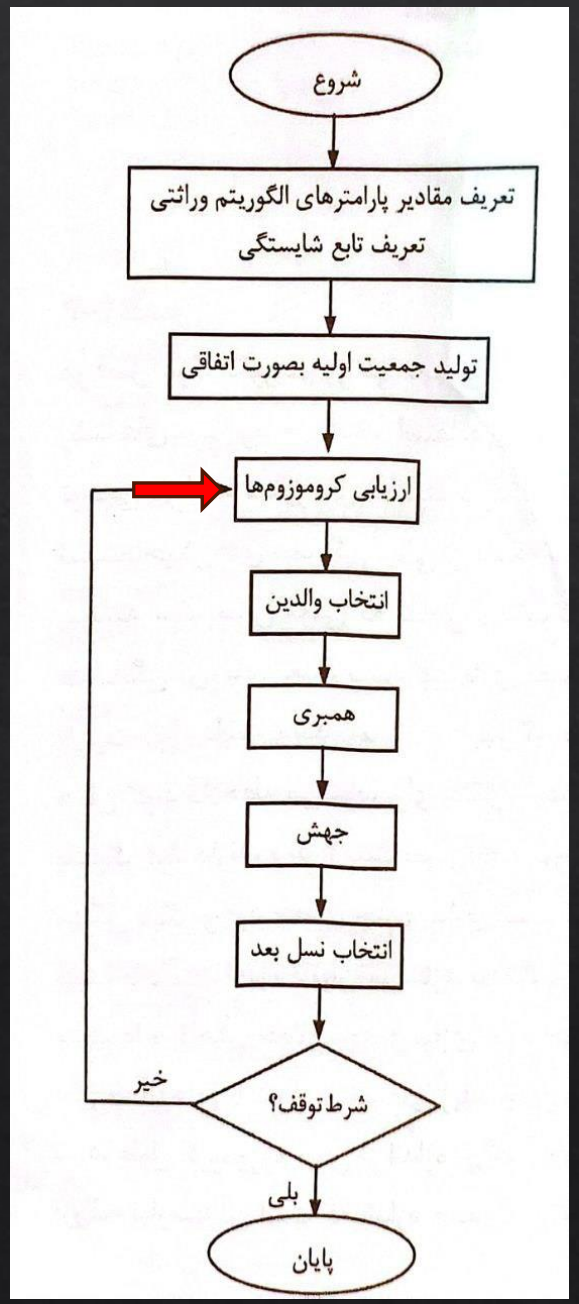
$$x_i^{real} = x_i^{low} + \left(x_i^{high} - x_i^{low}\right) \times \text{rand}(0,\ 1)$$

```python
def Random_population(self):
    """
    This method randomly init a population matrix with continuous uniform distribution
    """

    population_matrix = []  # N x d Matrix

    for i in range(self.population):
        chromosome = [0 for k in range(self.dim)]
        for j in range(self.dim):
            l_bound = self.func_config[j]['low']
            h_bound = self.func_config[j]['high']
            chromosome[j] = (uniform(0, 1) * (h_bound - l_bound)) + l_bound
        population_matrix.append(chromosome)

    self.population_matrix = population_matrix
```

تابعی که برای تولید جمعیت اولیه در نظر گرفته شده است, تابع Random_population است. این تابع هیچ پارامتر ورودی ندارد. در ابتدا یک ماتریس را برای ذخیره کروموزوم های تولید شده در نظر می گیرد. سپس با کمک دو حلقه که به ترتیب روی تعداد جمعیت و طول کروموزوم به کمک متغیر های i و j حرکت می کنند. در این تکرار ها متغیر i بیانگر تعداد سطر های ماتریس ما یا شماره ردیف کروموزوم است. پس باید در ابتدای حلقه یک فضای آرایه در نظر بگیریم که بیانگر یک کروموزوم باشد. در حلقه ی دوم, متغیر j بیانگرشماره ژن کروموزوم می باشد. ابتدا حد بالا و پایین هر ژن را از خصوصیت function_config به دست می آوریم و در متغیر های l_bound و h_bound ذخیره می کنیم. سپس تفاضل این حدود را در یک عدد رندوم بین صفر و یک که با تابع uniform ایجاد شده است ضرب می کنیم. سپس آن را با حد پایین دامنه جمع می کنیم.

پس از یک پایان اجرای حلقه ی داخلی, یک کروموزوم به طول dim به صورت رندوم تولید کرده ایم. حال کافی است که کروموزوم تولید شده را به ماتریسی که برای جمعیت در نظر گرفته بودیم اضافه کنیم.
پس از اتمام کار دو حلقه کافی است که ماتریس در نظر گرفته را به خصوصیت population_matrix انتساب دهیم.

ارزیابی کروموزوم ها

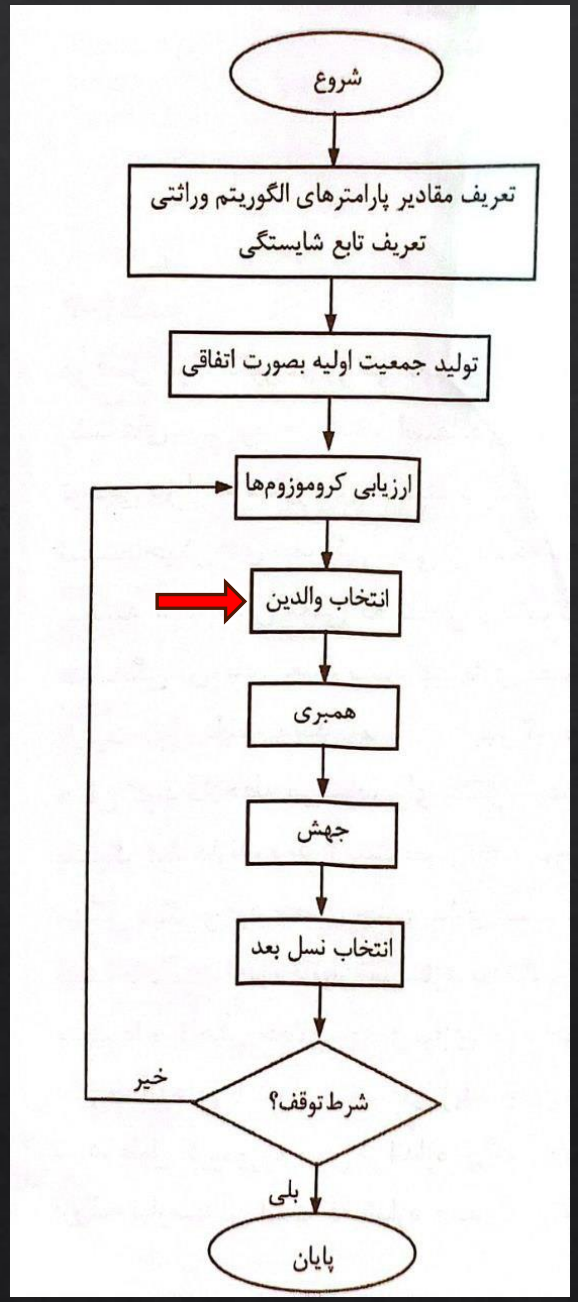


ارزیابی کروموزوم ها یعنی اینکه میزان کارایی هر کروموزوم را تعیین کنیم. این عمل به کمک تابع شایستگی انجام می شود.

```python
def get_Fitness(self, chromosomes):
    """returns a tuple of size N(population),
    that represents the fitness value for each chromosome"""

    fitness_values = [0 for i in range(self.population)]

    for i in range(len(chromosomes)):
        chromosome = chromosomes[i]
        fitness_values[i] = self.fit_func(chromosome)
    return tuple(fitness_values)
```

محاسبه ی شایستگی جمعیت توسط تابع get_Fitness انجام می شود.
برای این منظور باید در پارامتر های ورودی کروموزوم ها را برای آن مشخص کنیم. ابتدا یک لیست(آرایه) به اندازه ی جمعیت در نظر میگیریم؛ سپس به ازای هر عضو باید مقدار تابع شایستگی را محاسبه می کنیم و در فضای در نظر گرفته شده (آرایه ی fitness_values) قرار می دهیم. پس ابتدا هر کروموزوم را در یک متغیر با نام کروموزوم قرار می دهیم و سپس آن را به تابع شایستگی می دهیم. پس از اتمام حلقه مقدار شایستگی ها را به صورت یک tuple برمی گردانیم.

انتخاب والدین



انتخاب والدین به این منظور می باشد که مشخص کنیم کدام یک از کروموزوم ها در تولید نسل بعدی نقشی داشته باشد؛ پس باید تعدادی کروموزوم را به صورت رندوم انتخاب کنیم و به حوضچه ازدواج منتقل کنیم. شیوه مورد استفاده برای این کار در الگوریتم نوشته شده, استفاده از چرخ گردان مبتنی بر شایستگی است.

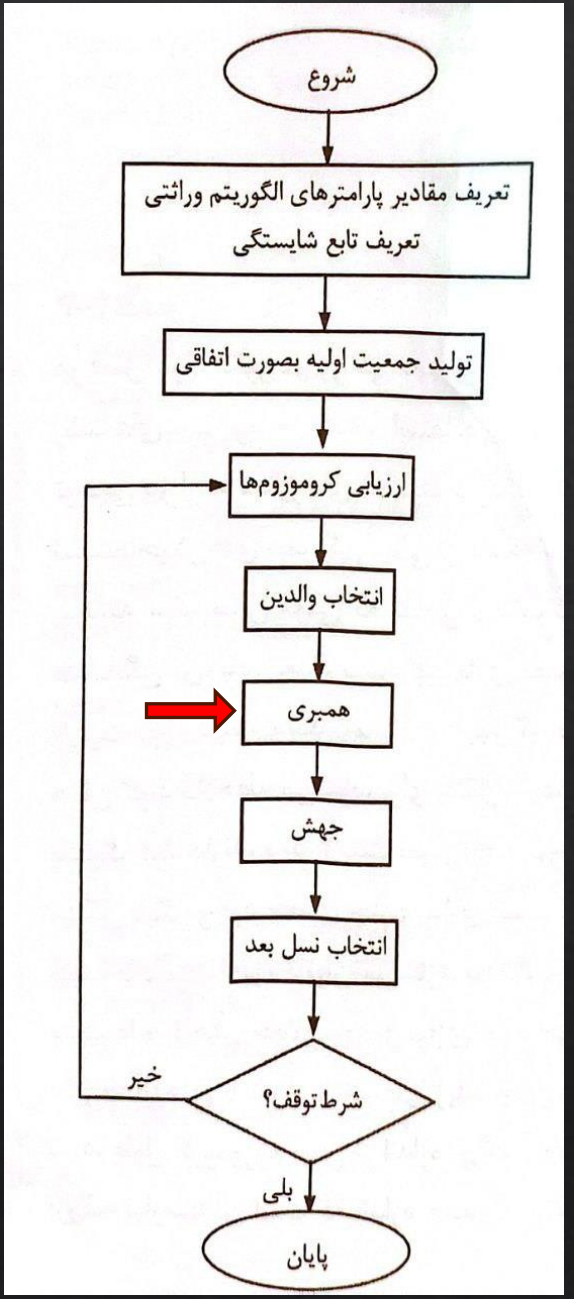
```python
def roulette_wheel(self,
                   fitness_values):
    """inputs fitness values of each chromosome
    return: Mating pool Matrix of size N x dim"""
    sum_fitness = sum(fitness_values)
    probs = [(fitness / sum_fitness) for fitness in fitness_values]

    prob = 0
    cumulative_probs = [0 for i in range(self.population)]

    for i in range(len(probs)):
        prob += probs[i]
        cumulative_probs[i] = prob

    # Selecting N chromosomes from population matrix
    mating_pool = []  # A matrix of size N x dim

    for i in range(self.population):
        r_num = random()
        idx = 0

        for j in range(len(cumulative_probs)):
            if cumulative_probs[j] > r_num:
                idx = j
                break

        mating_pool.append(self.population_matrix[idx])

    return mating_pool
```

برای اجرای چرخ گردون مبتنی بر شایستگی نیازمند لیست شایستگی کروموزوم ها هستیم؛ پس این مقدار را در پارامتر های ورودی دریافت می کنیم. همچنین نیازمند محاسبه شانس کروموزوم ها هستیم؛ پس مجموع شایستگی ها را محاسبه می کنیم و در متغیر sum_fitness قرار می دهیم. اکنون شانس هر کروموزوم را با تقسیم شایستگی کروموز(fitness) به مجموع شایستگی ها(sum_fitness) محاسبه می کنیم و در لیستی به اسم probs قرار می دهیم.

اکنون نیازمند محاسبه احتمال تجمعی کروموزوم ها هستیم تا با تولید عدد رندوم و مقایسه آن با بازه ای که احتمال تجمعی هر کروموزوم در آن قرار می گیرد مشخص کنیم که چه کروموزوم هایی به حوضچه تزویج راه پیدا می کنند.

فضای در نظر گرفته شده برای ذخیره ی احتمالات تجمعی را یک لیست با نام cumulative_probs در نظر می گیریم. احتمال تجمعی هر مرحله را در متغیر prob ذخیره می کنیم و در هر مرحله شانس کروموزوم را به آن اضافه می کنیم و در نهایت مقدار احتمال تجمعی(prob) را به فضای در نظر گرفته شده اضافه میکنیم. اکنون در یک حلقه روی تعداد جمعیت جرکت می کنیم و در هر مرحله یک عدد رندوم ایجاد کرده و درمتغیر r_num قرار می دهیم. حال طی یک حلقه مشخص می کنیم که عدد رندوم در بازه مربوط به کدام کروموزوم قرار گرفته است.
شماره کروموزوم را در متغیر idx قرار داده و آن را به فضای در نظر گرفته شده(mating_pool) اضافه می کنیم. در نهایت فضا را بر می گردانیم.

همبری

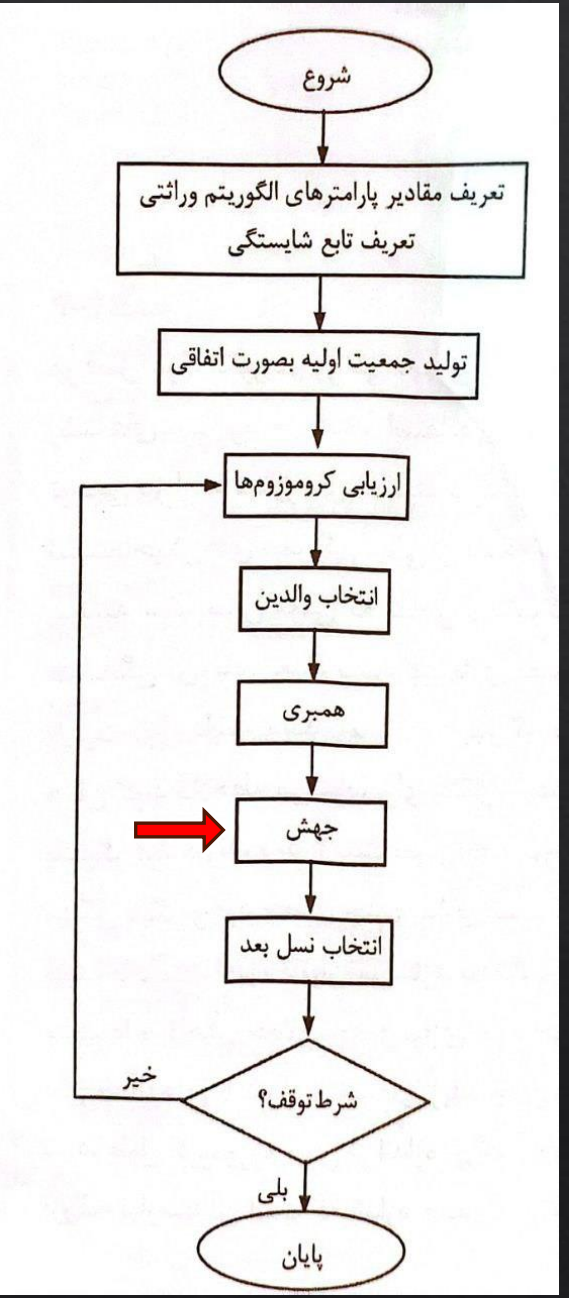


عملگر همبری, مهمترین عملگر الگوریتم وراثتی می باشد.
در اینجا از روش همبری حسابی محدب برای همبری استفاده شده است.

```
landa1 = uniform(0, 1)
landa2 = 1 - landa1

child_matrix = []
i = 0
while i < self.population - 1:
    if random() > self.pc:
        child_matrix.append(mating_pool_mat[i])
        child_matrix.append(mating_pool_mat[i + 1])
        i += 2
    else:
        parent1 = mating_pool_mat[i]
        parent2 = mating_pool_mat[i + 1]

        child1 = clip_vector(vadd(vmult(landa1, parent1), vmult(landa2, parent2)))
        child2 = clip_vector(vadd(vmult(landa2, parent1), vmult(landa1, parent2)))

        child_matrix.append(child1)
        child_matrix.append(child2)
        i += 2

return child_matrix
```

در تابع crossover, ابتدا λ_1 و λ_2 را مقدار دهی می کنیم.
سپس با حلقه While به اندازه جمعیت تکرار انجام می دهیم.
در هر بار تکرار عدد رندومی تولید می شود که اگر از نرخ همبری بیشتر بود, همبری انجام نشده و دو والد به ماتریکس فرزندان برده می شوند, در غیر این صورت عمل همبری حسابی محدب انجام می شود و سپس دو فرزند تولید شده به ماترکس فرزندان اضافه می شوند.

جهش



عملگر جهش, به این صورت است که یک تغیر جزئی با احتمال بسیار کم را روی یک کروموزوم اعمال می کند.

```python
def Mutation(self, child_matrix):
    """
    This method applies Mutation phase on Child Matrix
    :return: Next generation, a matrix of size N x dim
    """
    for i in range(self.population):
        if random() < self.pm:
            # Perform mutation on each gene of the chromosome
            for j in range(self.dim):
                l_bound = self.func_config[j]['low']
                h_bound = self.func_config[j]['high']
                mutation_mean = (h_bound + l_bound) / 2
                mutation_std = (h_bound - l_bound) / 4  # Mutation Standard # todo
                mutation_value = gauss(mu=mutation_mean, sigma=mutation_std)
                mutation_value = max(min(mutation_value, h_bound),
                                     l_bound)  # Clip the value to be within the bounds
                child_matrix[i][j] = mutation_value

    return child_matrix
```

عملگر جهش روی ماتریس فرزندان اعمال می شود. بنابراین ماتریس فرزندان را از طریق پارامتر **child_matrix** دریافت کرده ایم.
در یک حلقه ی خارجی, به کمک متغیر i روی اعضای جمعیت حرکت می کنیم. در ابتدای حلقه یک عدد رندوم ایجاد کرده و اگر این مقدار از نرخ جهش کمتر بود, طی یک حلقه ی دیگر به کمک متغیر j روی ژن های کروموزوم انتخابی حرکت می کنیم. به ازای هر ژن مقادیر حد پایین دامنه و حد بالای آن را محاسبه می کنیم. حد متوسط جهش برابر با میانگین دامنه ی ژن است. این مقدار را محاسبه کرده و در متغیر **mutation_mean** قرار می دهیم. حد استاندارد جهش برابر با یک چهارم طول دامنه است. این مقدار را در متغیر **mutation_std** قرار می دهیم.

سپس یک مقدار نرمال بین حد متوسط و حد استاندار جهش به کمک تابع **guess** ایجاد می کنیم و در متغیر **mutation_value** قرار می دهیم. بین این مقدار و حد بالای ژن مینیمم را پیدا می کنیم. بین مقدار انتخاب شده و حد پایین تابع ماکسیمم را انتخاب کرده و به عنوان مقدار جهش در نظر می گیریم. در آخر مقدار جهش پیدا کرده را در درایه ی مربوطه در ماتریس فرزندان قرار می دهیم. پس از اتمام حلقه ها, ماتریس فرزندان را باز می گردانیم.

سایر توابع

توابعی وجود دارند که به طور مستقیم جزوی از ساختار الگوریتم وراثتی نمی باشند اما برای مسائلی مانند نمایش داده ها یا محاسبه ی برخی از پارامتر های خاص به آن ها نیاز داریم.

تابع clip_vector

```python
def clip_vector(vector):
    """
    This method clips the values of the vector with respect to high bound and low bound of the vector

    :return: Clipped vector(list)
    """
    return [max(min(vector[i], self.func_config[i]['high']), self.func_config[i]['low']) for i in
            range(self.dim)]
```

این تابع عمل **clipping** را روی یک لیست با توجه به کران بالا و پایین هر متغیر انجام می دهد.

تابع vmult و vadd

```python
def vmult(value, vector):
    """
    This method multiplies value in each element of the list and returns it
    :param value: value you want to multiply in a vector
    :param vector: you know what it is :)
    :return: Value * vector
    """
    return [value * element for element in vector]

def vadd(v1, v2):
    """
    This method adds v1 and v2 elementwise and returns a list v3
    :param v1: vector 1
    :param v2: vector 2
    :return: v1 + v2 (list of len(v1))
    """
    assert len(v1) == len(v2)

    return [e1 + e2 for e1, e2 in zip(v1, v2)]
```

تابع **vmult** یک مقدار را در یک لیست ضرب می کند و لیست حاصل را خروجی می دهد.

تابع **vadd** دو لیست را متناظرا با یکدیگر جمع کرده و در لیست جدید قرار می دهیم, سپس آن را خروجی می دهد.

تابع print_parameters

```python
def print_parameters(self):
    # This is a method that prints parameters in tabular structure
    parameters = [
        ["Algorithm Runs", self.runs],
        ["Maximum Generation", self.max_gen],
        ["Population Size", self.population],
        ["CrossOver Rate", self.pc],
        ["Mutation Rate", self.pm],
        ["Function Input Shape", self.dim]
    ]
    print(tabulate(parameters, headers=["Parameter", "Value"], tablefmt="fancy_grid"))
```

تابع **print_parameters** , پارامترهایی از جمله نرخ همبری, اندازه جمعیت, نرخ جهش, حداکثر تعداد نسل ها(تعداد تکرار الگوریتم) و تعداد دفعات اجرا را به کمک ماژول **tabulate** به صورت یک جدول نمایش می دهد.

تابع one_gen

```python
def one_gen(self):
    """This method produces one generation of the algorithm"""

    fitness_values = self.get_Fitness(chromosomes=self.population_matrix)
    try:
        assert min(fitness_values) >= 0  # to make sure there is no zero fittness value
    except:
        print(min(fitness_values))

    self.log_gen(fitness_values)  # log the info of the last generation
    mating_pool = self.roulette_wheel(fitness_values)
    next_gen_v1 = self.Crossover(mating_pool)
    nex_gen = self.Mutation(next_gen_v1)
    self.population_matrix = nex_gen
    self.last_gen += 1
```

تابع **one_gen** , توابع دیگر را در خود فراخوانی می کند تا بتوانند یک نسل جمعیت ایجاد کند. در این تابع در ابتدا تابع مربوط به محاسبه شایستگی ها را فراخوانی می کنیم و مقدار شایستگی ها را در متغیر **fitness_values** قرار می دهیم. در یک ساختار **try/except** برسی می کنیم که حتما شایستگی هت مثبت باشند. در این مرحله تابع log را فرا می خوانیم(جلوتر توضیح داده خواهد شد.). سپس به کمک تابع **roulette_weel** والد های منتخب را در متغیر **mating_pool** قرار می دهیم. در نهایت تابع **crossover** را فراخوانی کرده و حاصل را در **next_gen_v1** قرار می دهیم. این متغیر را به تابع **mutation** جهت همبری ارسال می کنیم. در نهایت حاصل همبری یا همان متغیر **nex_gen** را به عنوان نسل جدید تثبیت میکنیم ودر مرحله آخر, شماره نسل را بروزرسانی می کنیم.

تابع plot_info

```python
def plot_info(self):

    plt.style.use('Solarize_Light2')

    # Plot of avg fitness
    y2 = self.history['avg_best_so_far']
    y1 = self.history['avg_mean_fitness']
    x = list(range(self.max_gen))

    plt.plot(x, y1, label="Average Mean Fitness", color="blue")
    plt.plot(x, y2, label="Average Best So Far", color="red", ls="--")

    plt.legend()
    if self.plot_save_dir is not None:
        plt.savefig(os.path.join(self.plot_save_dir, "RGA_plot.png"))
        print("_" * 5, "Plot Saving", "_" * 5)
        print(f"Plot Saved in : {self.plot_save_dir}")

    plt.show()
```

تابع نام برده, وظیفه ی رسم نمودار ها را به عهده دارد.
در این تابع به کمک ابزاری(**plt.plot()**) که کتابخانه **matplotlib** در اختیار ما قرار می دهد, نمودار اطلاعات مربوط به تابع را رسم می کنیم. مهمترین اطلاعاتی که به آن نیاز داریم, معیار های **average best so far** و **average mean fitness** هستند. این معیار ها را در خصوصیت **history** از کلاس نگهداری, و در هر نسل آن را بروزرسانی کرده ایم. اگر در هنگام ایجاد کلاس مسیری را برای ذخیره اطلاعات مربوطه مشخص کرده باشیم, نمودار ها ذخیره می شوند, در غیر این صورت تنها نمایش داده می شوند.

تابع one_run

```python
def one_run(self):  # This method fully runs the algorithm once
    self.Random_population()  # Making random population

    # running the algorithm
    for generation in trange(self.max_gen):
        self.one_gen()

    # Saving fitness values of the last produced population
    fitness_values = self.get_Fitness(chromosomes=self.population_matrix)
    self.log_gen(fitness_values)
```

تابع نام برده, یکبار الگوریتم را اجرا می کند. برای اینکار در ابتدا جمعیت رندوم را تولید کرده, سپس طی یک حلقه نسل های متوالی(به تعداد دفعات مشخص شده در خصوصیت **max_gen**) تولید می کند. این عمل به معنای فراخوانی متعدد تابع **one_gen** می باشد. پس از اتمام اجرا مقادیر مورد نیاز مانند متوسط شایستگی را در خصوصیات مربوطه ذخیره کرده و باقی لاگ گذاری ها را توسط تابع **log_gen** انجام می دهیم.

تابع log_gen

```python
def log_gen(self, fitness_list):
    """saves the best so far and avg fitness of a generation in one run of RGA"""

    # updating best so far property
    if self.best_so_far['fitness'] < max(fitness_list):
        self.best_so_far['fitness'] = max(fitness_list)
        self.best_so_far['chromosome'] = self.population_matrix[fitness_list.index(self.best_so_far['fitness'])]

    # updating the best current property
    self.best_current['fitness'] = max(fitness_list)
    self.best_current['chromosome'] = self.population_matrix[fitness_list.index(self.best_current['fitness'])]

    # Saving history
    self.history["best_so_far"].append(self.best_so_far['fitness'])
    self.history["mean_fitness"].append(sum(fitness_list) / len(fitness_list))
```

تابع فوق, وظیفه دارد که مقادیر بهترین جواب دیده شده و متوسط شایستگی کروموزوم ها را در یک بار اجرا ذخیره کند. برای این منظور ابتدا طی یک شرط مقدار بهترین جواب دیده شده را چک می کنیم و در صورت برقراری شرط(بهتر بودن جواب) آن را ذخیره می کنیم. در گام بعد بهترین جواب نسل فعلی را به ضای در نظر گرفته شده منتقل می کنیم. در گام آخر تاریخچه را بروزرسانی می کنیم.

تابع log_algo

```python
def log_algo(self):
    self.history["avg_mean_fitness"].append(self.history['mean_fitness'])
    self.history["avg_best_so_far"].append(self.history['best_so_far'])

    # Saving the best decoded value of the chromosome of the last generated population
    self.best_answers.append(self.best_current['chromosome'])
```

تابع نام برده, وظیفه ذخیره و لاگ گذاری نتایج کل الگوریتم را به عهده دارد. در هربار اجرای این الگوریتم, این تابع فراخوانی شده و مقادیر تاریخچه و بهترین جواب را بروزرسانی می کند.

تابع reset

```
def reset(self):
    """This method empties the lists and values
    used for tracking the metrics for one fully run of the algorithm"""

    self.history['mean_fitness'] = []
    self.history['best_so_far'] = []

    self.best_so_far['fitness'] = 0
    self.best_so_far['chromosome'] = []

    self.best_current['fitness'] = 0
    self.best_current['chromosome'] = []

    self.last_gen = 0
```

تابع ریست, وظیفه دارد که فضایی را که برای یکبار اجرای الگوریتم در نظر گرفته, برای اجرای بعدی خالی کند و عملا خصوصیات کلاس را که مربوط به هر بار اجرا کردن کامل الگوریتم هستند مقدار دهی اولیه کند.

تابع Run

```python
def Run(self):
    # Starting the algo
    print("\033[1;36;40m" + "=" * 50)
    print("\033[1;36;40m" + "   Running Real-Valued Genetic Algorithm   ")
    print("\033[1;36;40m" + "=" * 50 + "\033[0m")

    # printing the params in tabular structure
    self.print_parameters()
    print("_____________________________________")

    for run in range(self.runs):
        # Reset the properties that were given values for one fully run of the algorithm
        self.reset()

        # Fully running the algorithm
        print(f"Run : {run + 1}")
        self.one_run()

        # Logging lists of best_so_far and mean fitness produced by one_run().
        self.log_algo()

    solution, opt = self.post_process()
    # Showing results
    print("_____________________ Results _____________________")
    print(f"Average Best Solution     :       {solution}")
    print(f"Optimum Value             :       {opt}")
    print(f"Last Average Best So Far  :       {self.history['avg_best_so_far'][self.last_gen - 1]}")
    print(f"Last Average Mean Fitness :       {self.history['avg_mean_fitness'][self.last_gen - 1]}")

    # Plotting absf, amf
    self.plot_info()
```

تابع **Run** مهمترین تابع و تابع اجرا کننده الگوریتم در این کلاس می باشد. ما در این تابع, توابع دیگر را که مراحل الگوریتم وراثتی را اجرا می کردند صدا می زنیم.

در ابتدا تابع **print_pararmeters** را صدا زده و پارامترهایمان را به کاربر نمایش می دهیم. سپس یک حلقه را ایجاد کرده که بتوانیم چندین بار الگوریتم را اجرا کنیم. در مرحله ی بعد به کمک تابع **reset** مقادیر را به حالت اولیه باز می گردانیم. سپس به کمک تابع **one_run** یک بار الگوریتم را اجرا می کنیم. چون توابع نامبرده را در یک حلقه که به تعداد دفعاتی که در خصوصیت **runs** مشخص کرده ایم اجرا می شود, فراخوانی کرده ایم, چندین بار اجرا می شود. پس از اتمام کار حلقه, نتایج به دست آمده را به کاربر نمایش می دهیم. دقت کنید که نتایج هر بار اجرا کامل الگوریتم را جداگانه به کمک تابع **log_algo** ذخیره کرده ایم. در انتها نتایج را چاپ کرده و نمودار مربوطه را به کاربر نمایش می دهیم.

تابع post_process

```python
def post_process(self):
    """This method processes self.history['avg_mean_fitness'] and self.history['avg_best_so_far'] and
    self.best_answers}
    :returns: 1. Mean Best Solution (mean of the best answers found in each run) 2. Optimum
    value found for the target function
    """

    # Evaluating the best solution
    mean_best_solutions = [0 for i in range(self.dim)]

    for i in range(self.dim):
        _sum = 0
        for j in range(self.runs):
            _sum += self.best_answers[j][i]

        mean_best_solutions[i] = _sum / len(self.best_answers)

    # avg best so far, and avg mean fitness
    avg_mean_fitness = [0 for i in range(self.max_gen)]
    avg_best_so_far = [0 for i in range(self.max_gen)]

    for i in range(self.max_gen):
        for j in range(self.runs):
            avg_mean_fitness[i] += self.history['avg_mean_fitness'][j][i]
            avg_best_so_far[i] += self.history['avg_best_so_far'][j][i]
        avg_mean_fitness[i] = avg_mean_fitness[i] / self.runs
        avg_best_so_far[i] = avg_best_so_far[i] / self.runs

    self.history['avg_mean_fitness'] = avg_mean_fitness
    self.history['avg_best_so_far'] = avg_best_so_far

    # Optimum found
    opt = self.function(mean_best_solutions)

    return tuple(mean_best_solutions), opt
```

تابع نام برده, طی دو حلقه average mean و fitness و average best so far را محاسبه میکند و در history قرار می دهد.
در اتمام کار این تابع میانگین بهترین جواب ها در یک اجرا و مقدار جواب بهینه محاسبه شده و آماده برای متد مربوطه برای کشیدن نمودار می شود.

فایل theorem.py

علاوه بر فایل اصلی, که کد های مربوط به پیاده سازی الگوریتم و کلاس RGA و اجرای آن در این فایل ها قرار دارند, فایل theorem.py حاوی توابع محک یا همان توابع تست متفاوتی است که الگوریتم با این توابع تست شده و به جواب مطلوبی رسیده است.

نتایج و گزارشات

Rastrigin Function

Formula : $f_{(x)} = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)]$

Global Minimum : $f(x^*) = 0$ Where $x^* = (0, \ldots, 0)$

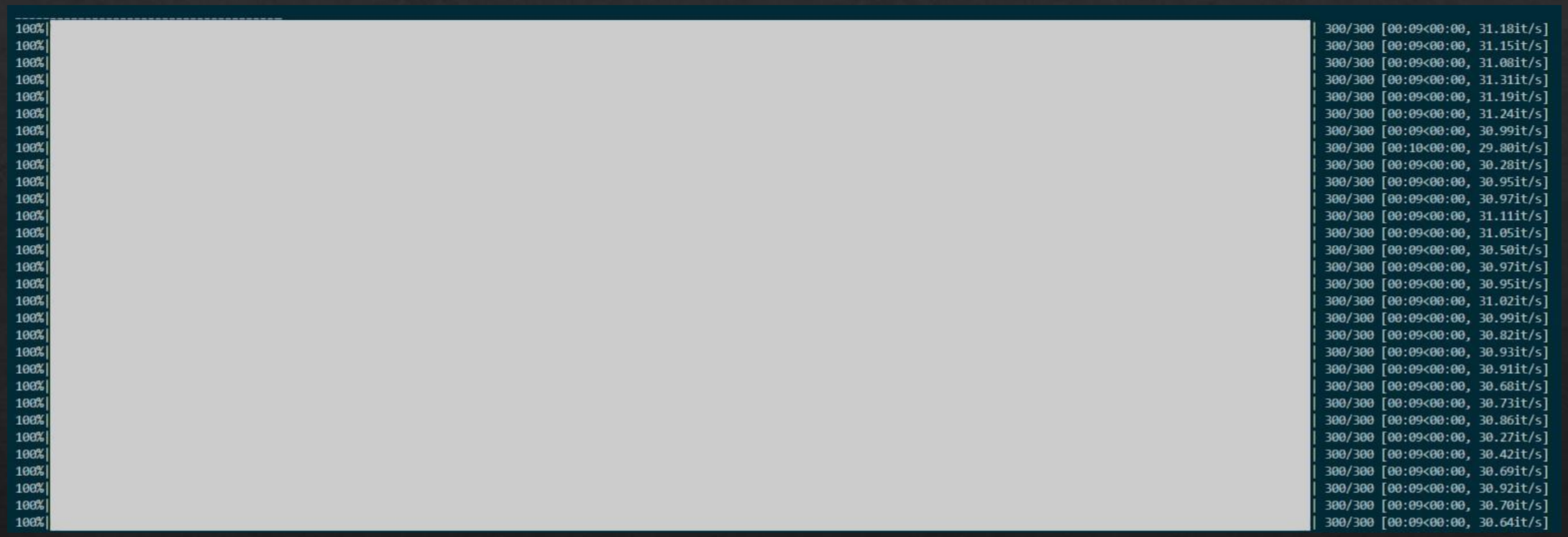
Domain : $x_i \in [-5.12, 5.12]$

پارامتر ها

```
rga = RGA(target_function=theorem.rastrigin, fitness_function=lambda x: 165 - theorem.rastrigin(x),
          function_config=[{'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12},
                           {'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12}], crossover_rate=0.5,
          mutation_rate=0.01, max_gen=300, population=700, run_bga=5)
rga.Run()
```

Parameter	Value
Algorithm Runs	30
Maximum Generation	300
Population Size	700
CrossOver Rate	0.5
Mutation Rate	0.01
Function Input Shape	4

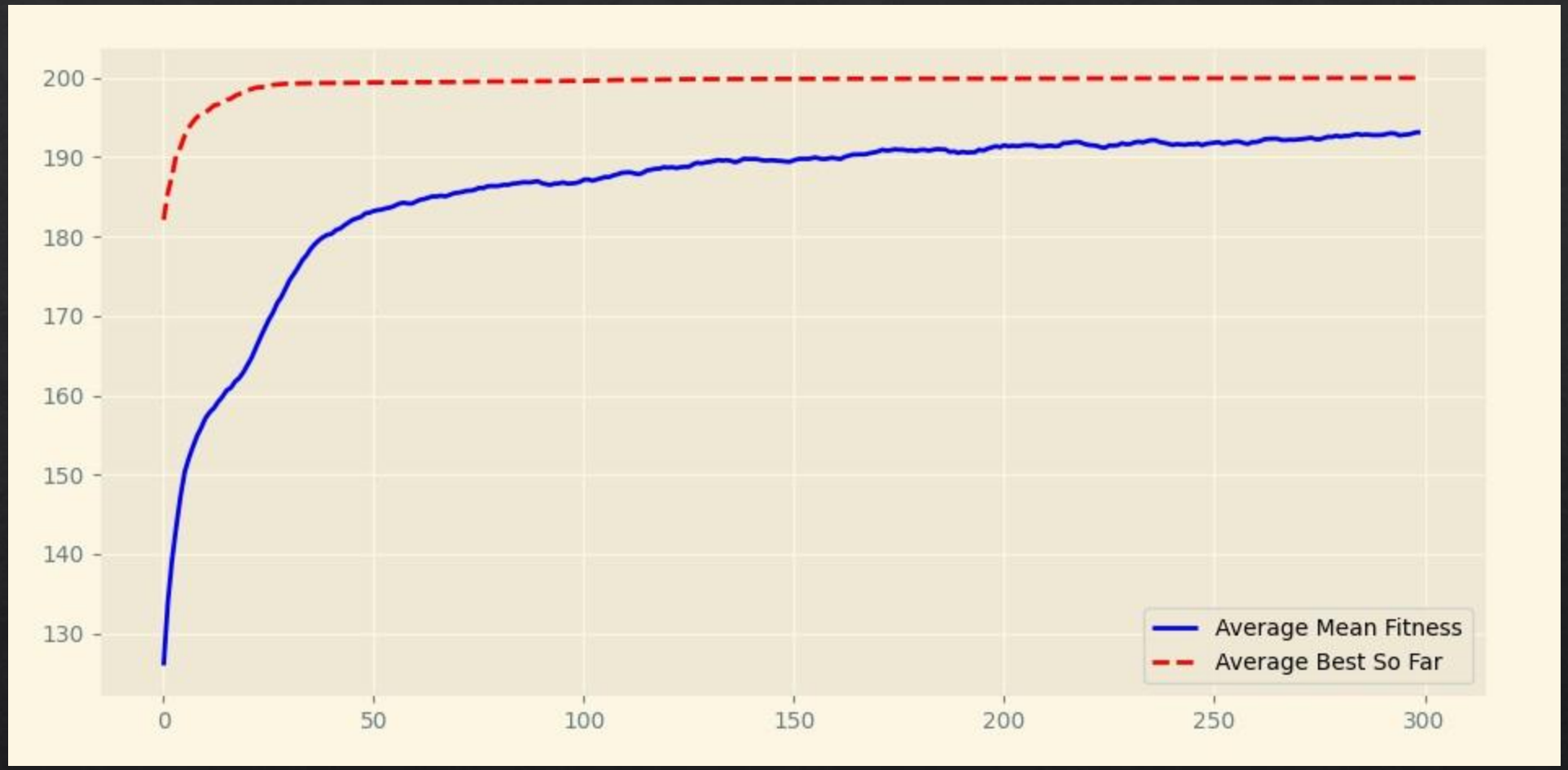
اجرای برنامه



نتایج به دست آمده

```
_____________________ Results _____________________
Average Best Solution     :       (-0.0063984284051720 65, -0.0031840548449787 46, 0.0028784854564645472, -0.0027623902835195216)
Optimum Value             :       0.013289955096261963
Last Average Best So Far  :       199.95512849832357
Last Average Mean Fitness :       193.11082031223626
```

نمودار


200
190
180
170
160
150
140
130
0
50
100
150
200
250
300
Average Mean Fitness
Average Best So Far

Griewank Function

Formula : $f(X) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$

Global Minimum : $f(X^*) = 0$ Where $X^* = (0, \dots, 0)$

Domain : $x_i \in [-600, 600]$

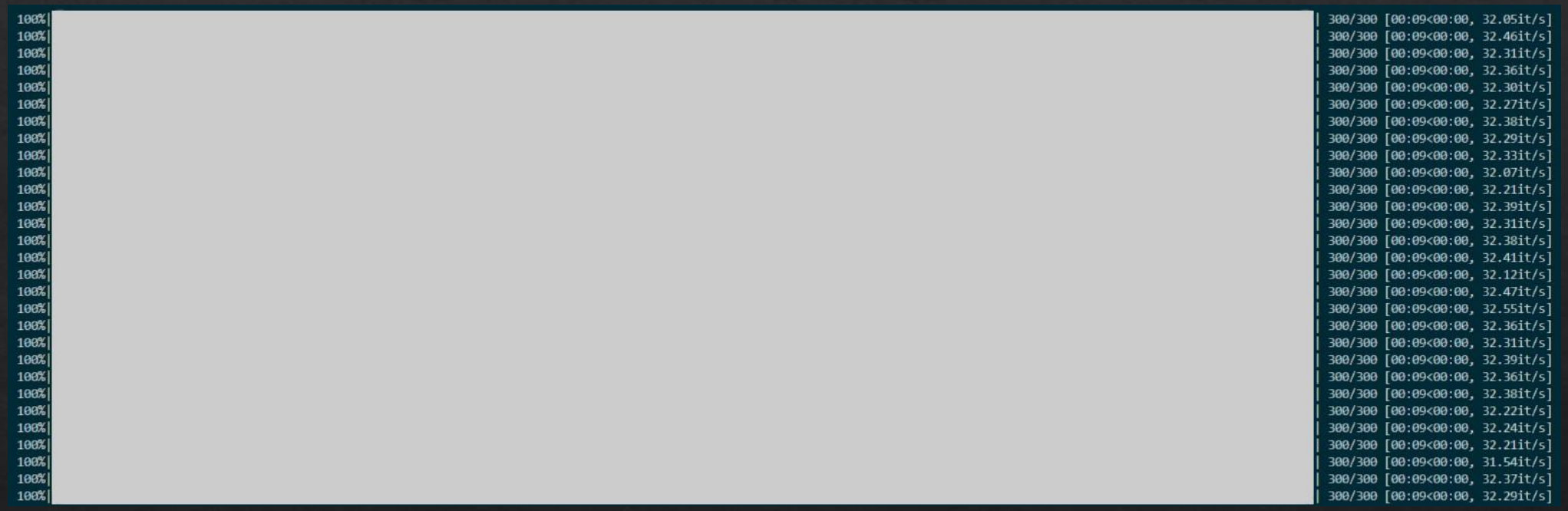
پارامتر ها

```
rga_griewank = RGA(target_function=theorem.griewank, fitness_function=lambda x: 350 - theorem.griewank(x),
                   function_config=[{'low': -600, 'high': 600}, {'low': -600, 'high': 600},
                                    {'low': -600, 'high': 600}, {'low': -600, 'high': 600}],
                   crossover_rate=0.5, mutation_rate=0.01, max_gen=300, population=700, run_rga=30)

rga_griewank.Run()
```

```
==================================================
  Running Real-Valued Genetic Algorithm
==================================================
```

Parameter	Value
Algorithm Runs	5
Maximum Generation	300
Population Size	700
CrossOver Rate	0.5
Mutation Rate	0.01
Function Input Shape	4

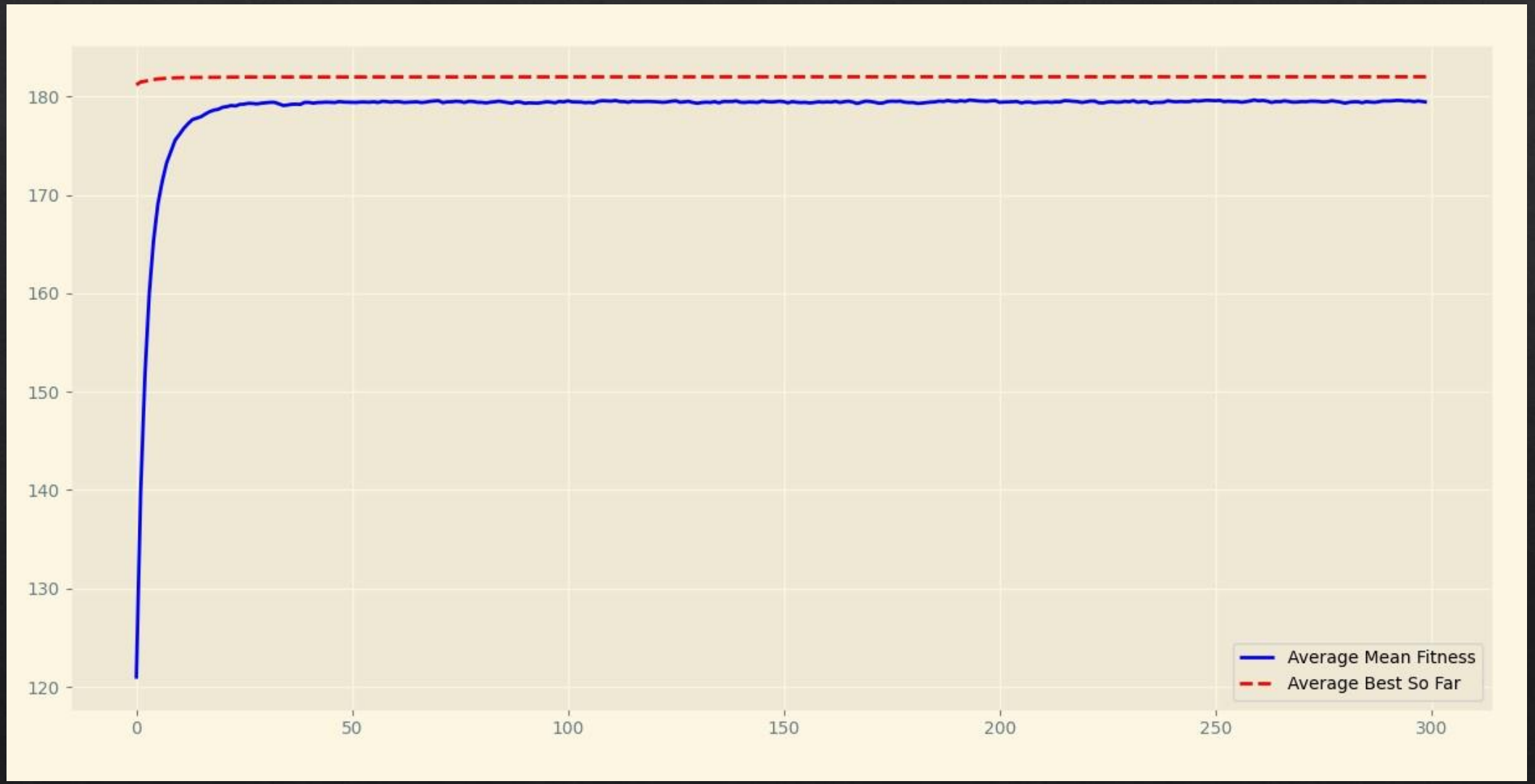
اجرای برنامه



نتایج به دست آمده

```
_____________________ Results _____________________
Average Best Solution     :       (-1.6852678631296365, -0.050198976729758O2, 1.0754803618298645, -1.9539964777438523)
Optimum Value             :       0.6307518530964873
Last Average Best So Far  :       181.99866850871453
Last Average Mean Fitness :       179.3325866919226
```

نمودار


180
170
160
150
140
130
120
0
50
100
150
200
250
300
Average Mean Fitness
Average Best So Far