دانشگاه شهید بهشتی

دانشکده مهندسی و علوم کامپیوتر

گزارش تمرین دوم درس یادگیری ماشین

پیاده‌سازی و تحلیل الگوریتم ژنتیک

استاد درس: دکتر احمدعلی آبین

نام دانشجو: طراوت پارت

شماره دانشجویی: 400443040

بهار 1401

مقدمه:

در این پروژه مراحل الگوریتم ژنتیک پیاده سازی می‌شود و عملکرد آن روی توابع مختلف بررسی می‌شود. عملکرد الگوریتم ژنتیک بدین شکل است که از یک جمعیت اولیه شروع می‌کند و هر بار سعی می‌کند نسل‌های بهتری تولید کند. مراحل الگوریتم ژنتیک برای تولید یک نسل عبارت است از انتخاب بهترین والدین، تقاطع[1] و جهش[2]. این مراحل به دفعات تکرار می‌شوند تا نسل‌های متوالی تولید شوند. در نهایت بهترین جواب از آخرین نسل انتخاب می‌شود.

نتایج نشان می‌دهد که الگوریتم‌های ژنتیک روی توابع unimodal به سادگی به نقطه بهینه می‌رسند. اما چون در توابع multi-modal تعداد زیادی نقطه بهینه وجود دارد، الگوریتم‌های ژنتیک به سادگی نقطه بهینه را پیدا نمی‌کنند. با انجام بهبودهایی روی الگوریتم ژنتیک، موفق می‌شویم که روی هر دو نوع تابع به جواب‌های مناسبی برسیم.

توضیح الگوریتم ژنتیک:

الگوریتم‌های ژنتیک زیر دسته‌ای از الگوریتم‌های تکاملی هستند که می‌توانند نقطه بهینه یک تابع هدف را برای ما پیدا کنند. مراحل الگوریتم ژنتیک به شرح زیر است:

1- تولید جمعیت اولیه‌ای از مسئله با رمزگذاری مناسب[3]: در این پروژه هر کروموزوم شامل 30 تا ژن است و هر ژن یک عدد حقیقی[4] است.
2- محاسبه برازش[5] هر کدام از اعضای جمعیت: برای محاسبه مقدار برازش هر کدام از کروموزم‌ها مقدار تابع هدف را بدست می‌آوریم. چون در این پروژه مقدار تابع هدف کمتر برای ما مطلوب‌تر است، مقدار برازش را برابر معکوس مقدار قدر مطلق تابع هدف کروکوزوم در نظر گرفته شده است.
3- انتخاب والد ها: روش‌های مختلفی برای انتخاب والدها ممکن است. از جمله این روش‌ها، انتخاب رندوم والدها یا انتخاب برا اساس fitness هر کدام از والدها است. در این پروژه از روش roullete wheel جهت انتخاب والدها استفاده شده است. مطابق این روش، والدی با fitness بالاتر شانس بیشتری برای انتخاب شدن دارد.

[1] Crossover
[2] Mutation
[3] Encoding
[4] Real number
[5] Fitness

4- تقاطع: روش‌های مختلفی جهت انجام تقاطع وجود دارد. در این پروژه از دو روش تقاطع تک نقطه‌ای و تقاطع ترکیبی استفاده شده است. در تقاطع تک نقطه‌ای، یک نقطه به صورت تصادفی روی دو والد انتخاب می‌شود و ژن‌های سمت راست آن نقطه بین دو والد جابه‌جا می‌شوند. در تقاطع ترکیبی ژن‌های دو والد را با وزنی رندوم ترکیب میکنیم. برای مثال ژن شماره یک از والد اول را با وزن α ژن شماره یک والد دوم را با وزن $1-\alpha$ جمع می‌کنیم و به یک ژن جدید می‌رسیم.

5- جهش: روش های مختلفی برای انجام جهش وجود دارد. در این پروژه با احتمال 0/2 هر کدام از ژن‌ها ممکن است دچار جهش شوند. در صورتی که یک ژن جهش یابد، مقدار آن ژن با یک عددی که در بازه مجاز قرار دارد جابه‌جا می‌شود.

جزئیات پیاده‌سازی:

توابع و ماژول‌های پیاده‌سازی شده در این تمرین به شرح زیر است:

ماژول chromosome:

هر کروموزوم نماینده یک عضو از جمعیت است. یک کروموزم دارای ویژگی‌های زیر است:

- **Cost_func**: یک رشته‌ای است که نوع تابع هزینه را نشان می‌دهد. تابع هزینه می‌توایند sphere، bentcigar، rastrigins و Ackley باشد. هدف در این پروژه پیدا کردن نقطه بهینه این توابع است.
- **Num_genes**: تعداد ژن‌هایی که در یک کروموزوم قرار می‌گیرد.
- **Lower_bound**: کران پایین تابع هزینه که مطابق تعریف هر تابع تنظیم می‌شود.
- **Upper_bound**: کران بالای تابع هزینه که مطابق تعریف هر تابع تنظیم می‌شود.
- **Chromosome**: آرایه‌ای از اعداد حقیقی است.

توابع استفاده شده در ماژول chromosome به شرح زیر است:

- **get_bounds**: مطابق با تابع هدف استفاده شده، کران پایین و کران بالا را برمی‌گرداند.
- **Get_chromosome**: یک آرایه از جنس numpy array بر میگرداند که با مقادیر رندوم در بازه‌ی [lower_bound,upper_bound] مقداردهی شده است.
- **Set_chromosome**: ویژگی کروموزم را با آرایه‌ای مقداردهی می‌کند.
- **Get_cost_value**: با توجه به نوع تابع هزینه استفاده شده، مقدار هزینه یک کروموزوم را برمی‌گرداند.

- **Sphere**: هزینه کروموزم را با استفاده از تابع sphere بازمی‌گرداند.

$$f\left(x_1, x_2, \ldots, x_n\right) = \sum_{i=1}^{n} {x_i}^2$$

- **Bentcigat**: هزینه کروموزم را با استفاده از تابع Bentcigat بازمی‌گرداند.

$$f(X) = x_1^2 + 10^6 \sum_{i=2}^{n} x_i^2$$

- **Rastrigins**: هزینه کروموزم را با استفاده از تابع Rastrigins بازمی‌گرداند.

در رابطه‌ی زیر معمولا مقدار A=10 قرار داده می‌شود.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i)\right]$$

- **Ackley**: هزینه کروموزم را با استفاده از تابع Ackley بازمی‌گرداند.

در این رابطه معولا a=20، b=0.2 و $c=2\pi$ قرار داده می‌شود.

$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

تابع initialization:

این تابع تعدادی کروموزم تولید می‌کند و آن‌ها را در یک لیست ذخیره می‌کند.

```
def initialization():
    solutions = []
    for n in range(Npopulation):
        solutions.append(Chromosome(cost_function))
    return solutions
```

تابع roulette_wheel:

در این تابع ابتدا fitness هر کدام از کروموزم‌های موجود در جمعیت فعلی محاسبه می‌شود. سپس احتمال انتخاب شدن هر کروموزم برای تولید نسل بعدی بدست می‌آید. هرچقدار fitness یک کروموزوم بیشتر باشد، احتمال انتخاب شدن آن نیز بیشتر خواهد بود. در نهایت احتمال تجمعی هر کدام از کرومزوم‌ها محاسبه می‌شود. در این حالت حاصل جمع احتمال انتخاب شدن هر کدام از کروموزوم‌ها برابر یک می‌شود. سپس یک اندیس تصادفی در بازه‌ی [0,1] تولید می‌کنیم. اولین کروموزومی که مقدار احتمال تجمعی آن کمتر از اندیس تولید شده باشد، به عنوان والد مناسب انتخاب می‌شود.

```
def roulette_wheel(population):
    fitness = []
    chromosome_probabilities = []

    for p in population:
        fitness.append(1 / p.get_cost_value())

    sum_fitness = sum(fitness)
    for fit in fitness:
        chromosome_probabilities.append(fit / sum_fitness)

    chromosome_cumulative_probabilities = np.cumsum(np.array(chromosome_probabilities))
    pointer_value = sum(chromosome_probabilities) * np.random.rand()
    index = np.argwhere(pointer_value <= chromosome_cumulative_probabilities)
```

تابع crossover:

این تابع دو والد را ورودی می‌گیرد. یک نقطه به صورت تصادفی روی دو والد انتخاب می‌شود و ژن‌های سمت راست آن نقطه بین دو والد جابه‌جا می‌شوند. سپس دو فرزند جدید تولید شده بازگردانده می‌شوند.

```
def corssover(parent1, parent2):
    child1 = Chromosome(cost_function)
    child2 = Chromosome(cost_function)

    split_point = int(np.random.randint(0, parent1.chromosome.shape))
    child1_chromosome = np.concatenate((parent1.chromosome[0: split_point], parent2.chromosome[split_point:]))
    child1.set_chromosome(child1_chromosome)

    child2_chromosome = np.concatenate((parent2.chromosome[0: split_point], parent1.chromosome[split_point:]))
    child2.set_chromosome(child2_chromosome)

    return child1, child2
```

تابع Mutation:

در این تابع هرکدام از ژن‌های موجود در کروموزوم با احتمال mu جهش میابند و با احتمال 1-mu حفظ می‌شوند. بنابراین به ازای هر کدام از ژن‌های موجود در کرومزوم یک عدد تصادفی در بازه [0,1] تولید می‌شود. در صورتی که این عدد از 0.2 کوچکتر باشد، مقدار آن ژن با یک عدد تصاویر که در بازه‌ی مجاز قرار دارد، جایگزین می‌شود.

```
def mutate(solution, mu):
    sol = copy.deepcopy(solution)
    flag = np.random.rand(*solution.chromosome.shape) <= mu
    ind = np.argwhere(flag)
    sol.chromosome[ind] = np.random.uniform(solution.lower_bound, solution.upper_bound)
    return sol
```

تابع best_solution:

بهترین جواب را در یک جمعیت پیدا می‌کند. بهترین جواب، کروموزومی است که کمترین مقدار cost_funtion را دارد.

```
def best_solution(population):
    best_cost = math.inf
    for sol in population:
        if sol.get_cost_value() < best_cost:
            best_cost = sol.get_cost_value()
    return best_cost
```

لوپ اصلی برنامه:

در لوپ اصلی این برنامه الگوریتم ژنتیک به دفعات تکرار می‌شود. چون این الگوریتم رویکردی رندوم دارد، هربار بهترین جواب را بدست می آوریم و در نهایت میانگین بهترین جواب‌ها محاسبه می‌شود.

در هر دور اجرای این الگوریتم، تعدادی نسل می‌سازیم(num_generations). برای ساخت هر نسل جدید به تعداد نصف اندازه جمعیت، عملیات crossover و.. را انجام می‌دهیم. چون با هر بار انجام cross over دو تا فرزند تولید می‌شود و اینطوری جمعیت نسل بعدی برابر با نسل فعلی خواهد بود. بنابراین در این لوپ دو تا والد با تابع roulette_wheel انتخاب می‌شوند، crossover روی آنها اعمال می‌شود و دو فرزند تولید میشود و در نهایت روی دو فرزند تولید شده mutation اعمال می‌شود. فرزندان تولید شده به نسل جدید منتقل می‌شوند. در نهایت پس از اینکه نسل جدید به طور کامل ساخته شد، بهترین جواب را می‌توانیم از آخرین نسل ساخته شده انتخاب کنیم.

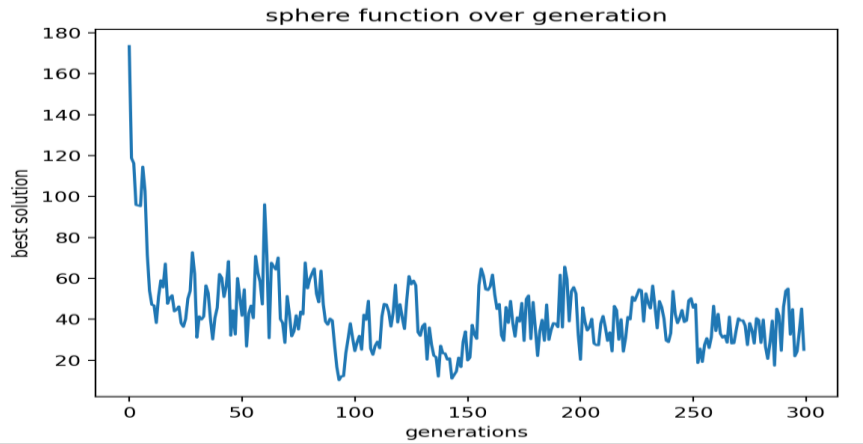
نتایج حاصل از الگوریتم ژنتیک کلاسیک:

در الگوریتم اجرا شده برای هر کروموزوم 30 تا ژن در نظر گرفته شده است. جمعیت هر نسل شامل 50 تا کروموزوم است. 300 تا نسل تولید می‌شود بهترین جواب هر نسل را ذخیره میکنیم. نمودارهای زیر روند خطا را با تولید نسل‌های جدید نشان می‌دهد.

نتایج تابع Sphere:

همانطور که دیده میشود، با تولید نسل‌های بیشتر این الگوریتم روند نزولی دارد و مقدار خطای آن کاهش میابد.

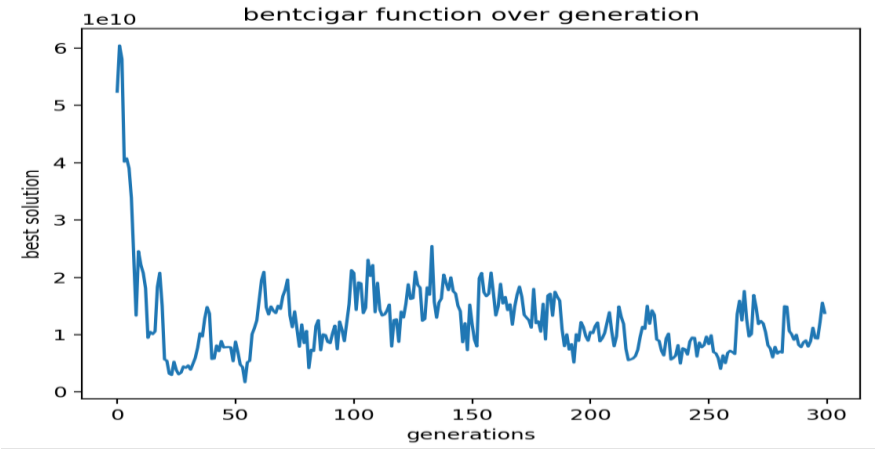
مقدار میانگین خطا: 38.35



نتایج تابع Bentcigar :

همانطور که دیده میشود، با تولید نسل‌های بیشتر این الگوریتم روند نزولی دارد و مقدار خطای آن کاهش میابد.

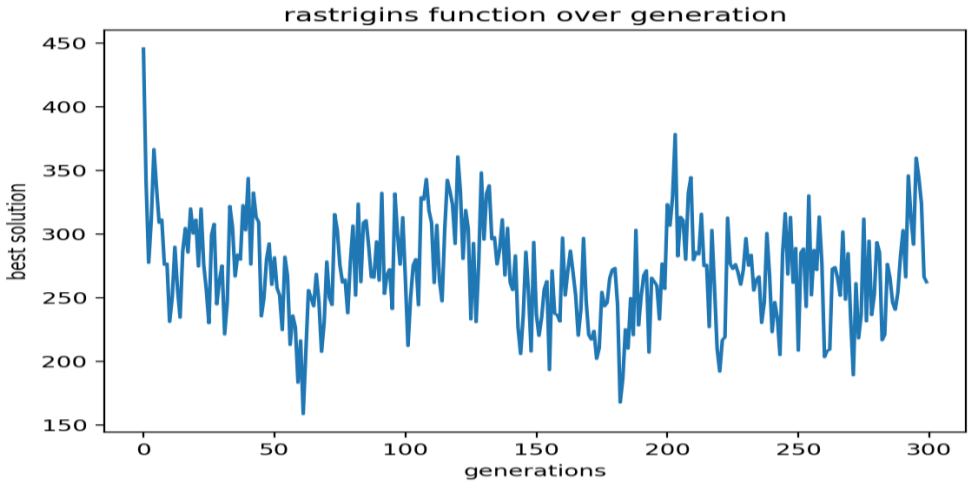
میانگین خطا: 12024394872.41



تابع Rastrigins:

تابع Rastrigins یک تابع Multimodal است. الگوریتم پیاده‌سازی شده با جست‌وجو در فضا نمی‌تواند نقطه بهینه را پیدا کند. همانطور که در نمودار زیر مشاهده می‌شود، نمودار روند نزولی ندارد.

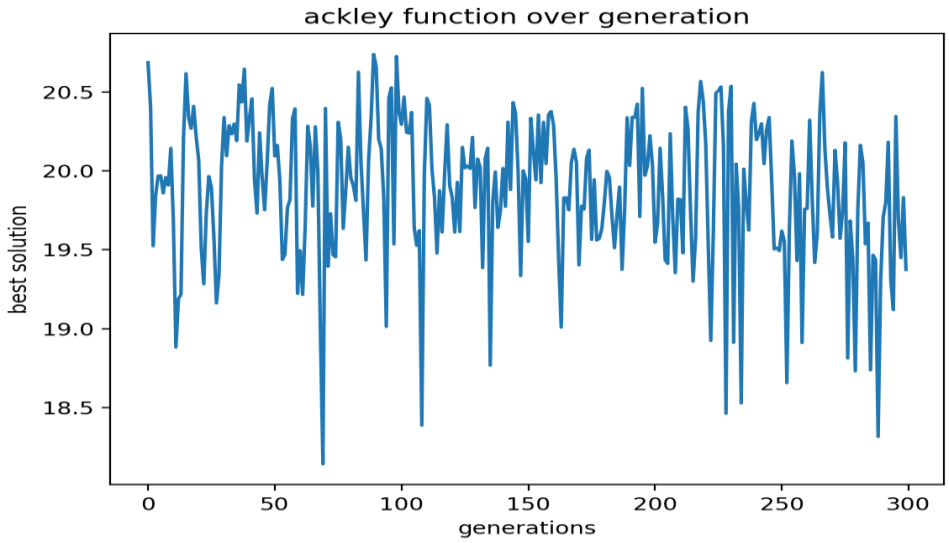
میانگین خطا: 269.23



نتایج تابع Ackley:

تابع Ackley یک تابع Multimodal است. الگوریتم پیاده‌سازی شده با جست‌وجو در فضا نمی‌تواند نقطه بهینه را پیدا کند. همانطور که در نمودار زیر مشاهده می‌شود، نمودار روند نزولی ندارد.

میانگین خطا: 19.75



نسخه اول - بهبود الگوریتم ژنتیک کلاسیک:

همانطور که مشاهده می‌شود، الگوریتم ژنتیک پیاده‌سازی شده نمیتواند به خوبی نقطه بهینه توابع Multimodal را پیدا کند چون با تولید نسل‌های بیشتر، نمیتواند هزینه را کاهش دهد. برای حل این مشکل، اندکی تغییرات در الگوریتم پیاده‌سازی شده اعمال می‌شود:

1- تغییر اول:

هر بار 20٪ از بهترین کروموزوم‌های یک نسل را وارد نسل بعدی می‌کنیم. این کار باعث می‌شود که کروموزم‌های خوب در اثر crossover و mutation گم نشوند و مستقیم به نسل بعدی وارد شوند.

2- تغییر دوم:

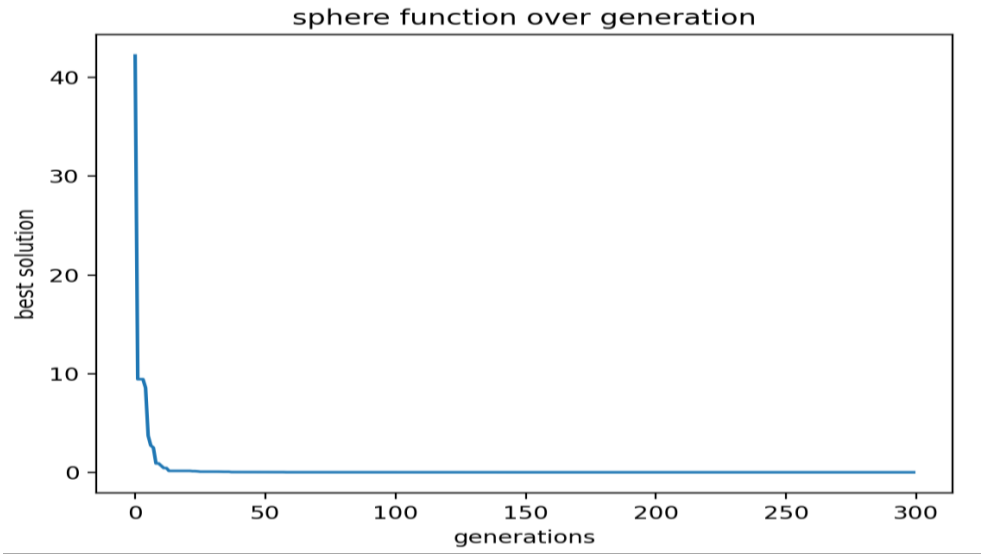
از یک تابع crossover ترکیبی استفاده شده است. (البته عدم استفاده از آن نیز تغییر زیادی در جواب نهایی ایجاد نمی‌کرد). در تقاطع ترکیبی ژن‌های دو والد را با وزنی رندوم ترکیب میکنیم. برای مثال ژن شماره یک از والد اول را با وزن α ژن شماره یک والد دوم را با وزن $1-\alpha$ جمع می‌کنیم و به یک ژن جدید می‌رسیم.

```
def corssover(parent1, parent2, gamma=0.1):
    child1 = Chromosome(cost_function)
    child2 = Chromosome(cost_function)
    alpha = np.random.uniform(-gamma, 1 + gamma, *parent1.chromosome.shape)

    child1_chromosome = alpha * parent1.chromosome + (1 - alpha) * parent2.chromosome
    child2_chromosome = alpha * parent2.chromosome + (1 - alpha) * parent1.chromosome

    child1.set_chromosome(child1_chromosome)
    child2.set_chromosome(child2_chromosome)

    return child1, child2
```

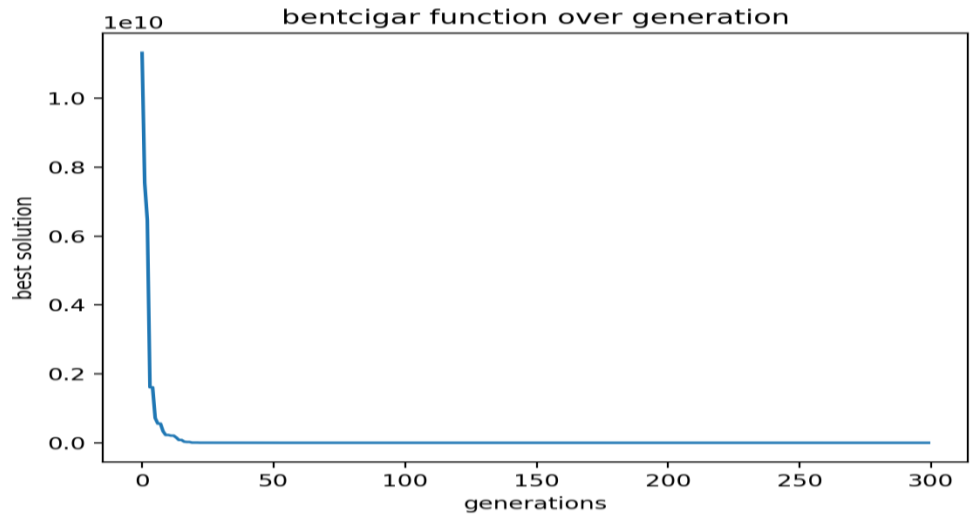
نتایج حاصل از الگوریتم ژنتیک بهبودیافته نسخه اول:

در الگوریتم اجرا شده برای هر کروموزوم 30 تا ژن در نظر گرفته شده است. جمعیت هر نسل شامل 50 تا کروموزوم است. 300 تا نسل تولید می‌شود بهترین جواب هر نسل را ذخیره میکنیم. نمودارهای زیر روند خطا را با تولید نسل‌های جدید نشان می‌دهد. همانطور که مشاهده می‌شود این الگوریتم نسبت به حالت کلاسیک عملکرد بسیار بهتری را دارد. در تمامی توابع با تولید نسل‌های بیشتر، مقدار خطا کم می‌شود. پس الگوریتم در فضای جست‌وجو به درستی در به سمت جواب بهینه در حرکت است.

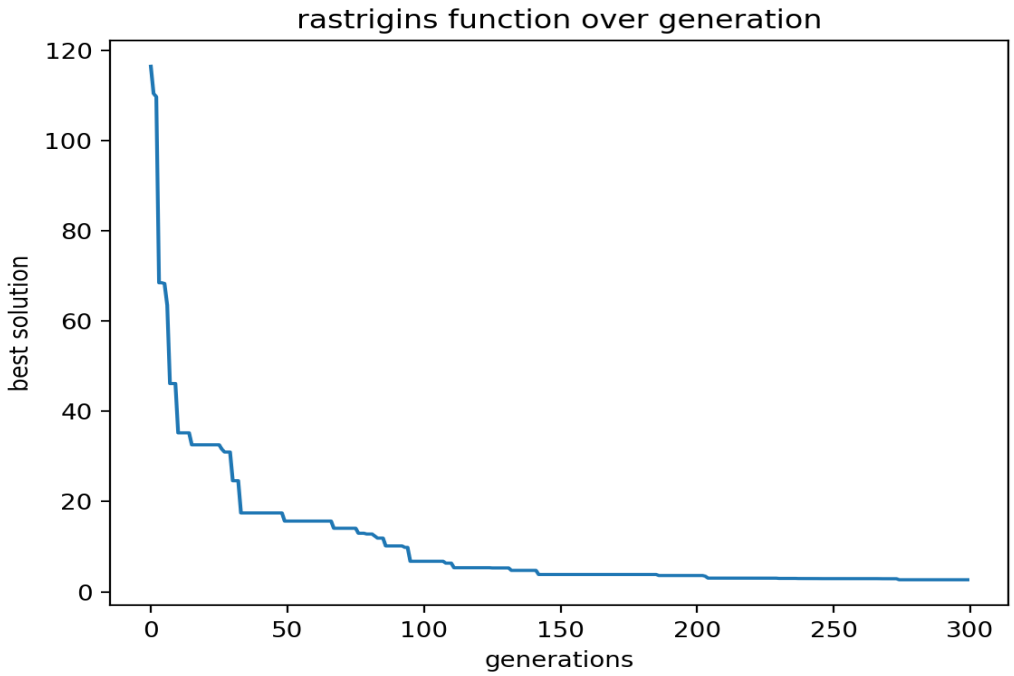
نتایج تابع Sphere:



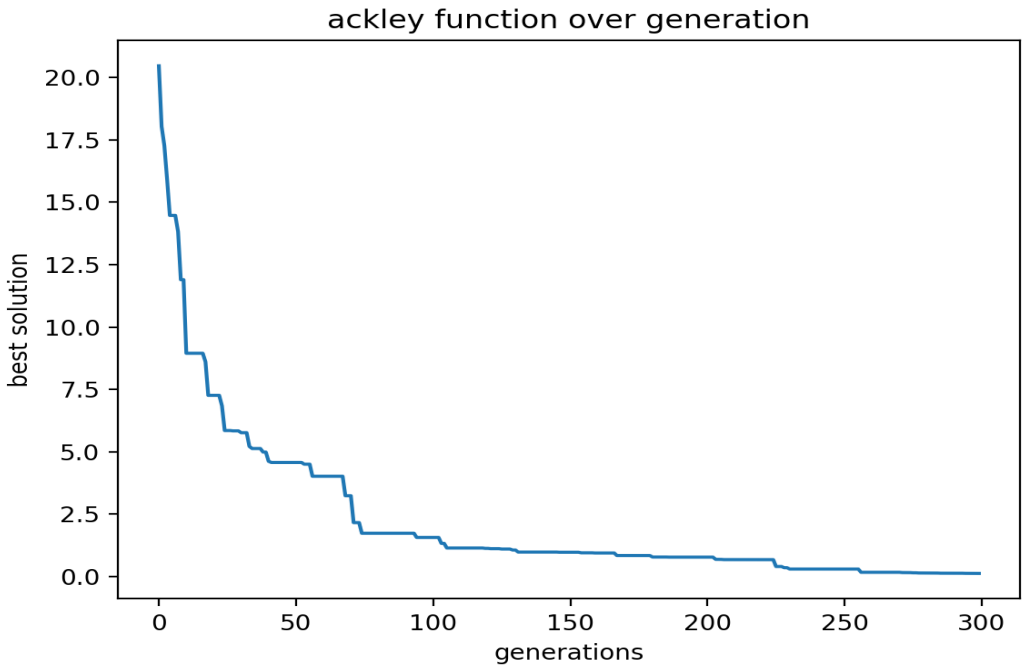
نتایج تابع Bentcigar :



تابع Rastrigins:



نتايج تابع Ackley:



نسخه دوم- الگوریتم ژنتیک بهبود یافته – PSO:

این الگوریتم از حرکت دسته‌جمعی پرندگان الهام گرفته است. برای مثال، وقتی دسته‌ای از پرندگان در حال حرکت و پیدا کردن غذا هستند، اگر یکی از آنها غذا پیدا کند، میتواند بقیه‌ی اعضای گروه را با خبر سازد. اعضای موجود در الگوریتم PSO سعی می‌کنند به سمت نقطه بهینه سراسری حرکت کنند. در نهایت بهترین کمینه پیدا شده به عنوان پاسخ بازگردانده می‌شود.

سه قانون کلی در این الگوریتم وجود دارد:

1- هر عضو بهترین نقطه‌ای که تاحالا مشاهده کرده است را ذخیره می‌کند.
2- بهترین جوابی که تمام اعضای الگوریتم مشاهده کرده‌اند ذخیره می‌شود.
3- هر عضو جهتی که باید در راستای آن حرکت کند را ذخیره می‌کند.

با استفاده از سه اطلاعات ذخیره شده‌ی بالا، هر عضو محاسبه محاسبه می‌کند که باید در چه جهتی حرکت کند. در رابطه زیر w ، c_1 و c_2 پارامترهایی هستند که تاثیرگذاری هرکدام از قسمت‌های این رابطه را کنترل می‌کنند. r_1 و r_2 پارامترهای رندوم در بازه [0,1] هستند و قابلیت عملکرد تصادفی به الگوریتم می‌دهند. t نماینده شماره مرحله‌ی الگوریتم و i نشان دهنده شماره عضو است.

$$v_i^{t+1} = wv_i^t + c_1r_1(p_i^t - x_i^t) + c_2r_2(G_t - x_i^t)$$

```
def update_velocity(self, best_global_position):
    w = 0.5  # constant intertia weight
    c1 = 2  # cognetive constant
    c2 = 2  # social constant

    r1 = np.random.uniform(0, 1, self.dims)
    r2 = np.random.uniform(0, 1, self.dims)
    cognitive_velocity = c1 * r1 * (self.best_position - self.current_position)
    social_velocity = c2 * r2 * (best_global_position - self.current_position)
    self.current_velocity = (w * self.current_velocity) + cognitive_velocity + social_velocity
```

بعد از محاسبه‌ی جهت حرکت، موقعیت یک عضو طبق رابطه‌ی زیر به روز رسانی می‌شود:

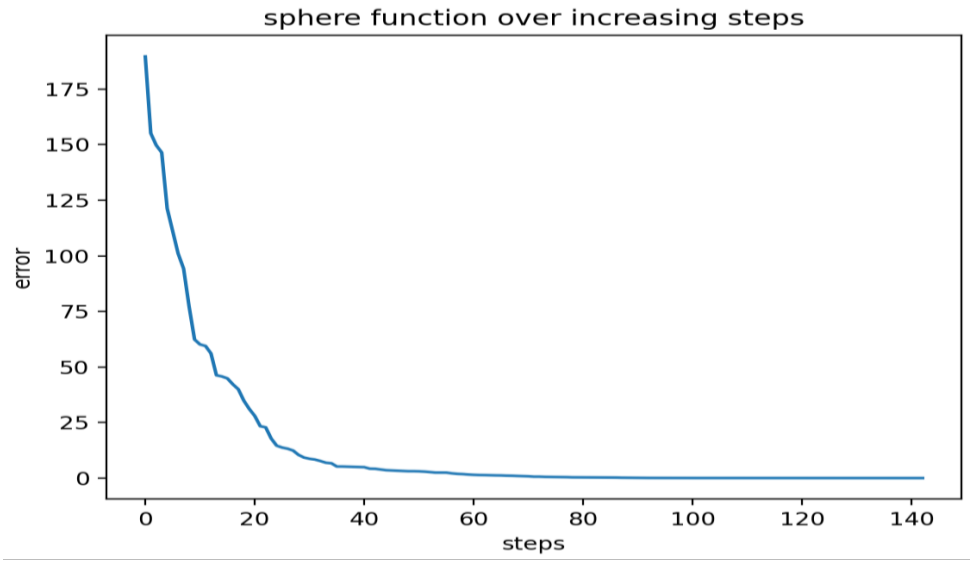
$$X_i^{t+1} = x_i^t + v_i^t$$

```
def update_position(self):
    self.current_position = self.current_position + self.current_velocity
    self.current_position = np.maximum(self.current_position, self.lower_bound)
    self.current_position = np.minimum(self.current_position, self.upper_bound)
```

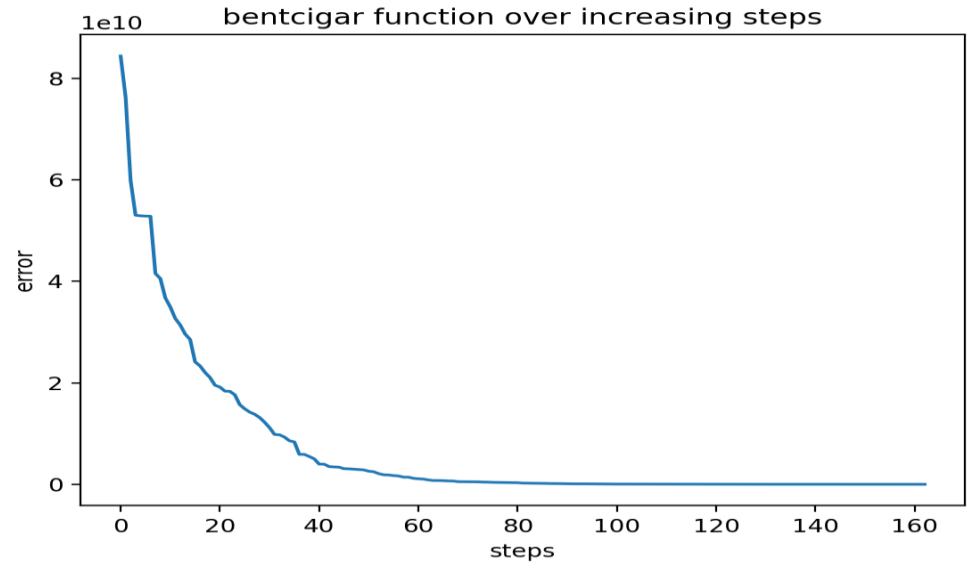
نتایج حاصل از الگوریتم PSO:

نمودارهای زیر روند خطا را در مراحل متوالی نشان می‌دهد. همانطور که مشاهده می‌شود، این الگوریتم نسبت به حالت کلاسیک عملکرد بسیار بهتری را دارد. در تمامی توابع، چه Multi-modal و چه unimodal، با گذر مراحل متولی مقدار خطا کم می‌شود. پس الگوریتم در فضای جست‌وجو به درستی در به سمت جواب بهینه در حرکت است. جمعیت این مسئله 50 در و توابع 30 بعدی در نظر گرفته شده‌اند.

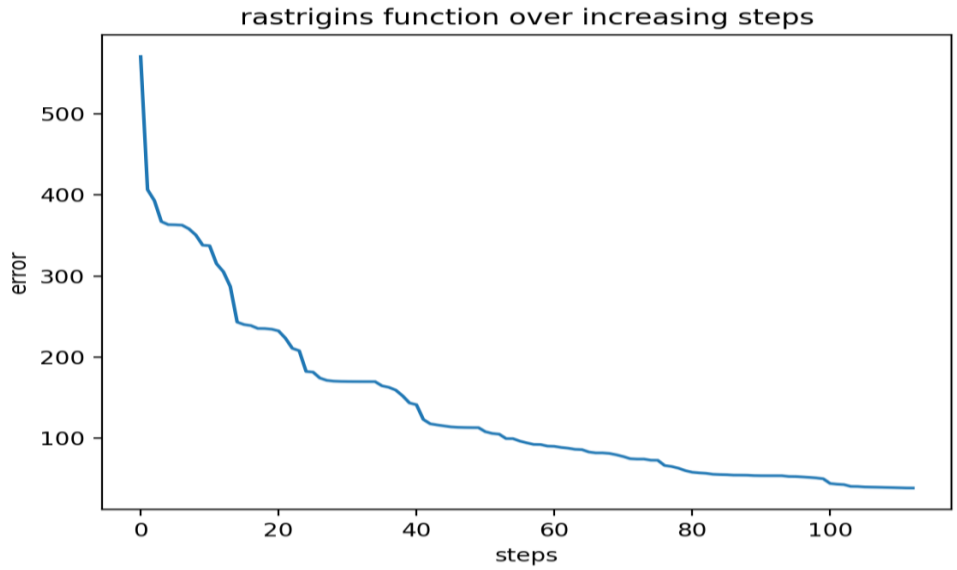
نتایج تابع Sphere:



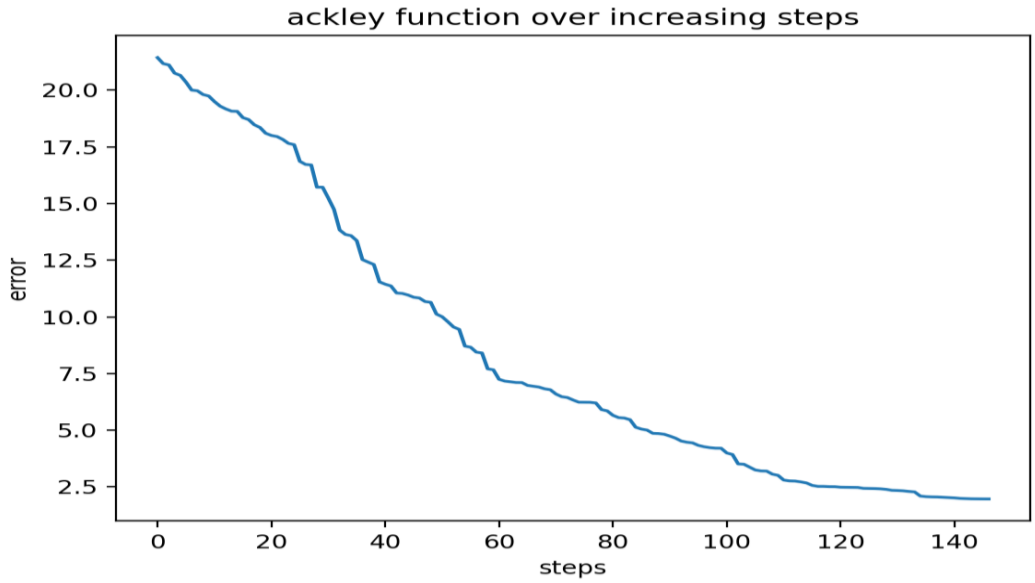
نتایج تابع Bentcigar :



نتایج تابع Rastrigins:



نتایج تابع Ackley:



جمع‌بندی:

در این پروژه الگوریتم ژنتیک برای پیدا کردن نقطه بهینه چهار نوع تابع متخلف پیاده‌سازی شده است. مشاهده می‌شود که الگوریتم ژنتیک هربار با تولید نسل‌های بهتر میتواند کروموزوم‌های بهتری را پیدا کند. منظور از کروموزوم بهتر، کروموزومی است که تابع هدف را کمینه می‌کند. نقاط بهینه در توابع unimodal ساده‌تر پیدا می‌شود. چون این توابع فقط یک نقطه بهینه دارند. اما پیدا کردن نقطه بهینه در توابع multimodal سخت‌تر است چون این توابع تعداد زیادی نقطه اکسترمم محلی دارند و الگوریتم ممکن است در نقاط اکسترمم محلی گیر کند و نتواند به نقطه اکسترمم سراسری برسد.

برای اینکه به الگوریتم بهتری برسیم، در این تمرین ابتدا سعی شد که هربار درصدی از بهترین کروموزم‌ها را مستقیما از نسل قدیم به نسل جدید منتقل شده است. این کار باعث می‌شود که ژن‌های خوب گم نشوند و توابع Multimodal نیز جواب‌های مناسبی را پیدا کنند.

در نسخه‌ی دوم سعی شد که از الگوریتم تکاملی دیگری به نام PSO جهت پیدا کردن نقطه بهینه استفاده شود. ای الگوریتم بسیار سریع‌تر از الگوریتم ژنتیک است. چون تولید نسل جدید در الگوریتم ژنتیک کلاسیک زمان‌بر تر است. اما در این الگوریتم به سادگی هر جواب ممکن(هر عضو) در راستایی مشخص آپدیت می‌شود. این الگوریتم به نوعی نقاط شروع متفاوت دارد. اعضای مختلف از نقاط مختلف شروع به جست‌وجو می‌کند و همدیگر را از بهترین جواب‌هایی که پیدا کرده‌اند با خبر می‌سازند. به همین دلیل این الگوریتم در توابع Multi-modal که دارای تعداد زیادی نقطه اکسترمم محلی است نیز به خوبی کار می‌کند و در نقطه اکسترمم محلی گیر نمی‌کند.

در الگوریتم pso هر کدام از اعضا بهترین نقطه در یک ناحیه را ذخیره می‌کنند و در راستای بهترین نقاط سعی می‌کنند که نقاط بهتری پیدا کنند. بنابراین این الگوریتم حتما می‌تواند به بهترین جواب برسد.

در جدول زیر مقایسه نهایی نتایج آورده شده است:

	Sphere	Bentcigar	Rastrigins	Ackley
Classic version	38.35	12024394872.41	269.23	**19.75**
Enhance version	0	9682	2.61	**0.17**
PSO – 30d	0.87	1000732554.78	83.33	**1.77**