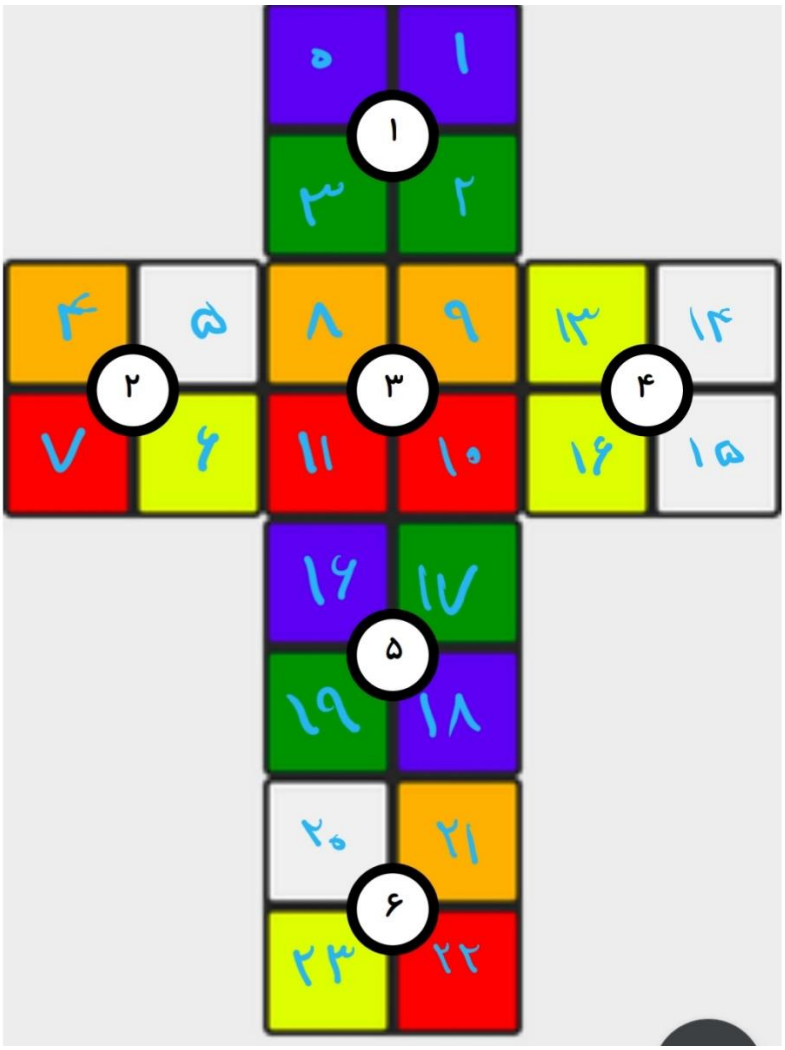
حسین محمدی ۹۵۳۳۰۸۱

.1

برای حل مسئله روبیک ابتدا باید آن را مدل کرد سپس متناسب با مدل سازی حرکات قابل انجام روی یک روبیک واقعی را برای آن پیاده سازی کرد.

نوع مدل کردن در جواب بدین صورت بوده است.همه را در یک آرایه integer ذخیره میکنیم.

a. فایل IDS \ first_question



که به اقتضای آن نوع ورودی نیز به شرح زیر خواهد بود

۶ ۵ ۱ ۳ ۲ ۴ ۲ ۴ ۶ ۳ ۳ ۶ ۵ ۵ ۱ ۱ ۵ ۶ ۳ ۱ ۲ ۲ ۴ ۴

گفتنی است که با ۲*۲ بودن روبیک برای مثال چرخیدن صفحه ۱ در جهت ساعت گرد معادل چرخیدن صفحه ۵ در جهت پاد ساعت گرد است. بنابراین کافیست تنها سه وجه ۱ و ۲ و ۳ را حرکت دهیم

که پاسخ در عمق ۱۰ با الگوریتم IDS بدست می آید

۱۲۸۵۳۵۸۸ تعداد نود تولید شده

۱-,۲ + ۱-,۳ + ۱'۱ + ۳'۱ + ۳'۱ + ۱-,۱ + ۳'۱ + ۲'۱ + ۱'۱ + ۱'۱

منظور از ۱'۱ چرخش وجه یک در جهت ساعت گرد میباشد.

و پاسخ روبیک بدین شکل خواهد بود

۲ , ۲ , ۲ , ۶ , ۶ , ۶ , ۵ , ۵ , ۵ , ۳ , ۳ , ۳ , ۴ , ۴ , ۴ , ۱ , ۱ , ۱ , ۱ ,

```
found
2 , 2 , 2 , 2 , 6 , 6 , 6 , 6 , 5 , 5 , 5 , 5 , 3 , 3 , 3 , 3 , 4 , 4 , 4 , 4 , 1 , 1 , 1 , 1 ,
12853588
2,-1 + 3,-1 + 1,1 + 3,1 + 3,1 + 1,-1 + 3,1 + 2,1 + 1,1 + 1,1 +
```

b. فایل USC \ first_question

برای استفاده از الگوریتم UCS از آنجا که هزینه همه حرکتها یکسان است پس مسئله تبدیل به جست و جوی اول سطح میشود.

برای این موضوع در هر نود مسیر پیموده شده تا آن گره و وضعیت روبیک در آن گره را ذخیره میکنیم اما چون سطح به سطح پیش میرویم در عمق ۹ ، ۱۰۰۷۷۶۹۶ گره در صف خواهیم داشت. شش به توان نُه گره!

برای کاهش حجم هر گره ، فقط حرکات (مسیر) را به صورت String ذخیره میکنیم و از وضعیت روبیک صرفنظر میکنیم.

برای چک کردن رسیدن به هدف باید کل مسیر از ریشه تا گره را اعمال کنیم.

با این حال از آنجا که حداکثر حافظه در اختیار بنده ۶ گیگابایت است موفق به حل مسئله تا عمق ۹ نشدم.

```
7
8
9

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main"
```

.2

برای حل مسئله رنگ آمیزی نیز از روش ژنتیک استفاده میکنیم.

مسئله قابل توجه در اینجا تورنمت است.

که تعریف آن در صورت پروژه متناقض است.

مرحله ۳ (انتخاب والدین): در این مرحله از روش tornument selection استفاده می‌کنیم. در این روش تعداد k عضو را به صورت random انتخاب کرده و بهترین آن‌ها را برمی‌گزینیم. مثلا اگر جماعت اولیه برابر ۱۰۰ باشد و k را برابر ۴ قرار دهیم، ۲۵ tornument اتفاق می‌افتد و ۲۵ عضو برگزیده می‌شوند. توجه کنید که k از پارامترهای قابل تنظیم می‌باشد و tornumentSize نام دارد. جماعت برگزیده، والدین نامیده می‌شوند.

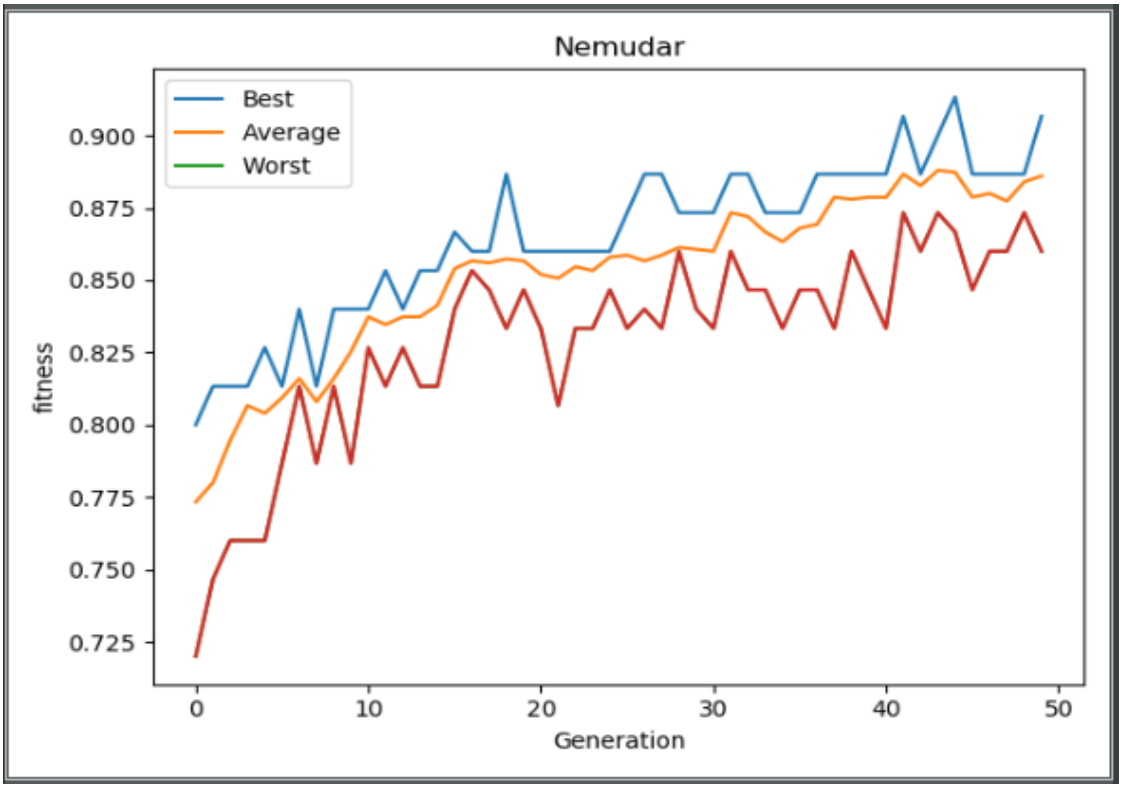
میخواهیم ۴ عضو برداریم اما ۲۵ عضو برمیداریم.

اما چیزی که پیاده شده بدین صورت است که اگر اندازه جمعیت ۱۰۰ و عدد تورنمنت ۴ باشد ابتدا ۲۵ عضو به صورت تصادفی انتخاب میکنیم سپس از بین آنها ، ۴ بهترین را انتخاب کرده و به عنوان والدین در نظر میگیریم. که حدس میزنیم منظور تعریف پروژه نیز همین باشد.

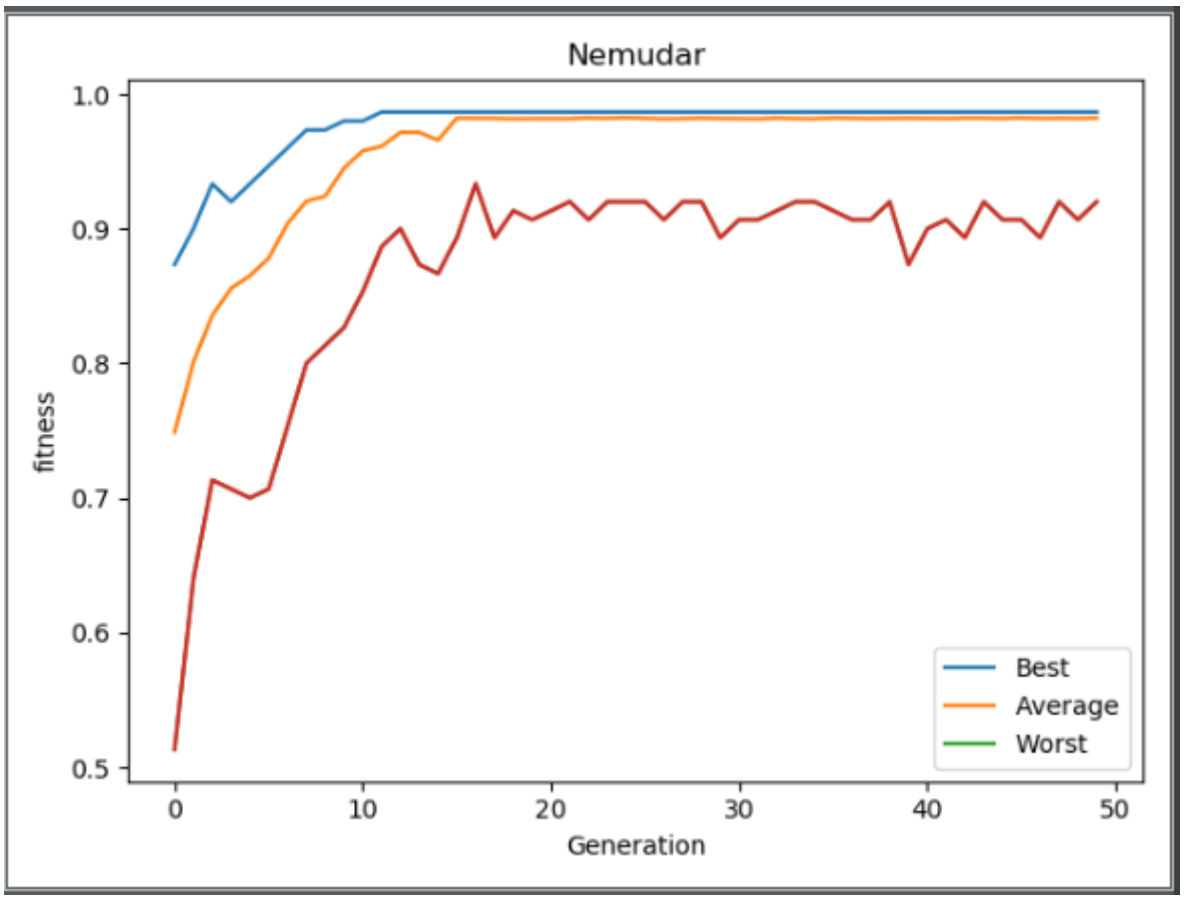
Q_2_Genetic Algorithm.py

به هر صورت با اعداد زیر نتایج زیر حاصل میشود.

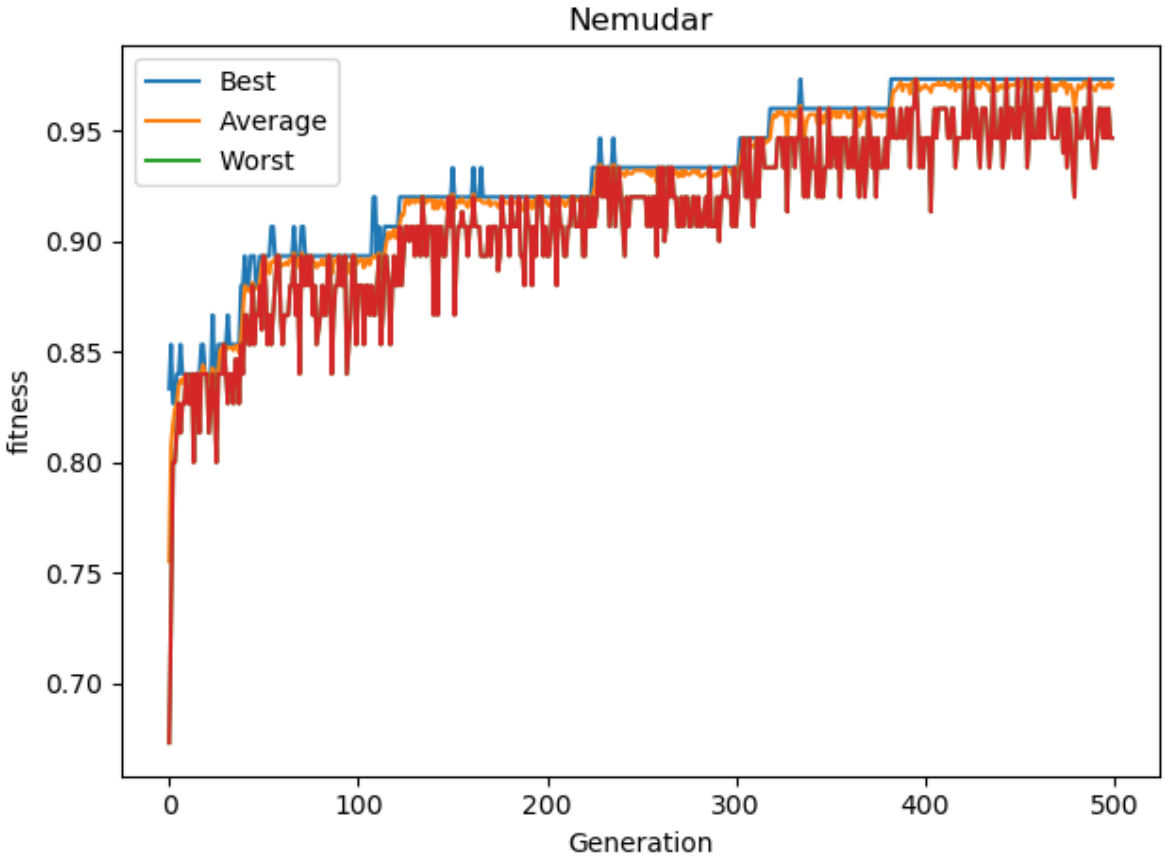
```
numberOfGenerations = 50
populationSize = 10
tornumentSize = 2
mutationRate = 0.02
```



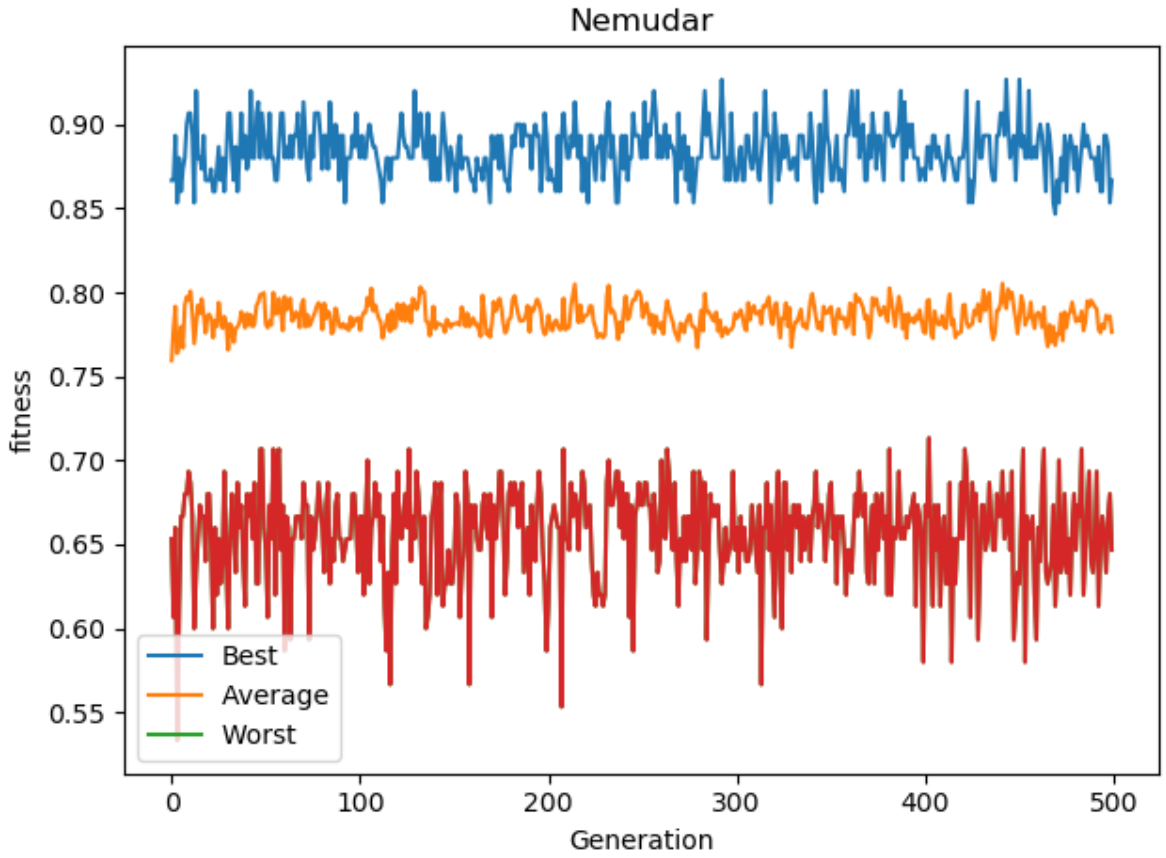
```
numberOfGenerations = 50
populationSize = 1000
tornumentSize = 10
mutationRate = 0.01
```



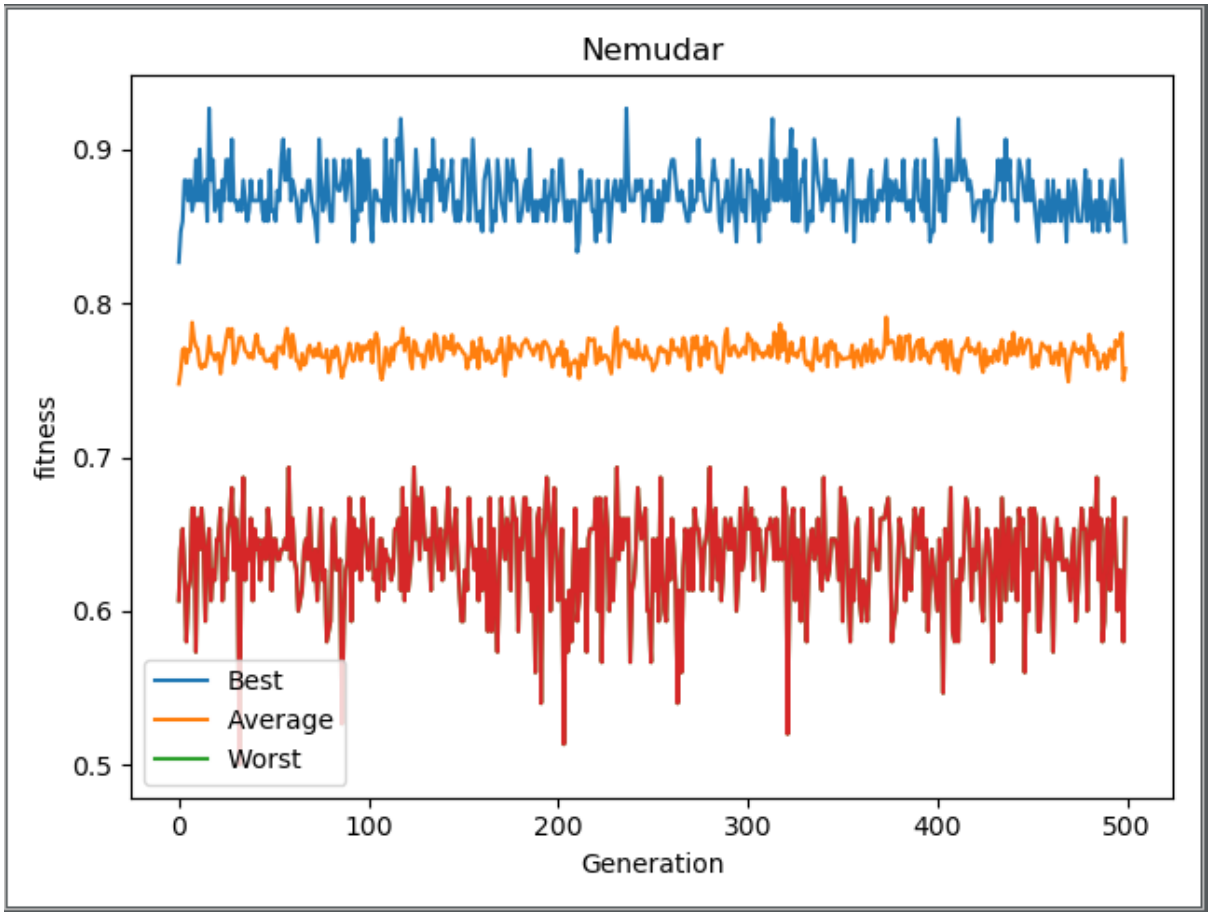
```
numberOfGenerations = 500
populationSize = 10
tornumentSize = 2
mutationRate = 0.01
```



```
numberOfGenerations = 500
populationSize = 100
tornumentSize = 3
mutationRate = 0.5
```



```
numberOfGenerations = 500
populationSize = 100
tornumentSize = 6
mutationRate = 0.5
```



این نمودار ها را اینگونه میتوان تفسیر کرد که با افزایش میزان جهش ، درست است که میزان تنوع افزایش می یابد اما ما را از رسیدن به هدفمان باز میدارد.

در حالی که با کاهش میزان جهش نیز تنوع کم شده و زودتر به همگرایی میرسیم.

مسئله دیگر اندازه تورنمت است. در این روشی که پیاده شده این فاکتور نقش مهمی دارد. واضح است از بین نمونه های بیشتر تصادفی میتوان کروموزومی با fitness بالاتر پیدا کرد و زودتر به هدف رسید.

هم چنین میتوانیم نتیجه بگیریم با نسل و جمعیت پایین نیز میتوان به جواب رسید.

البته این مسئله کمی خاص است.

زیرا همان گونه که در نمودارها پیداست ، در تولید جمعیت اولیه ، میزان ارزش اطراف ۰/۷۵ دیده میشود که نشانگر آن است که به صورت تصادفی نیز ما بیشتر از نصف راه را رفته ایم!

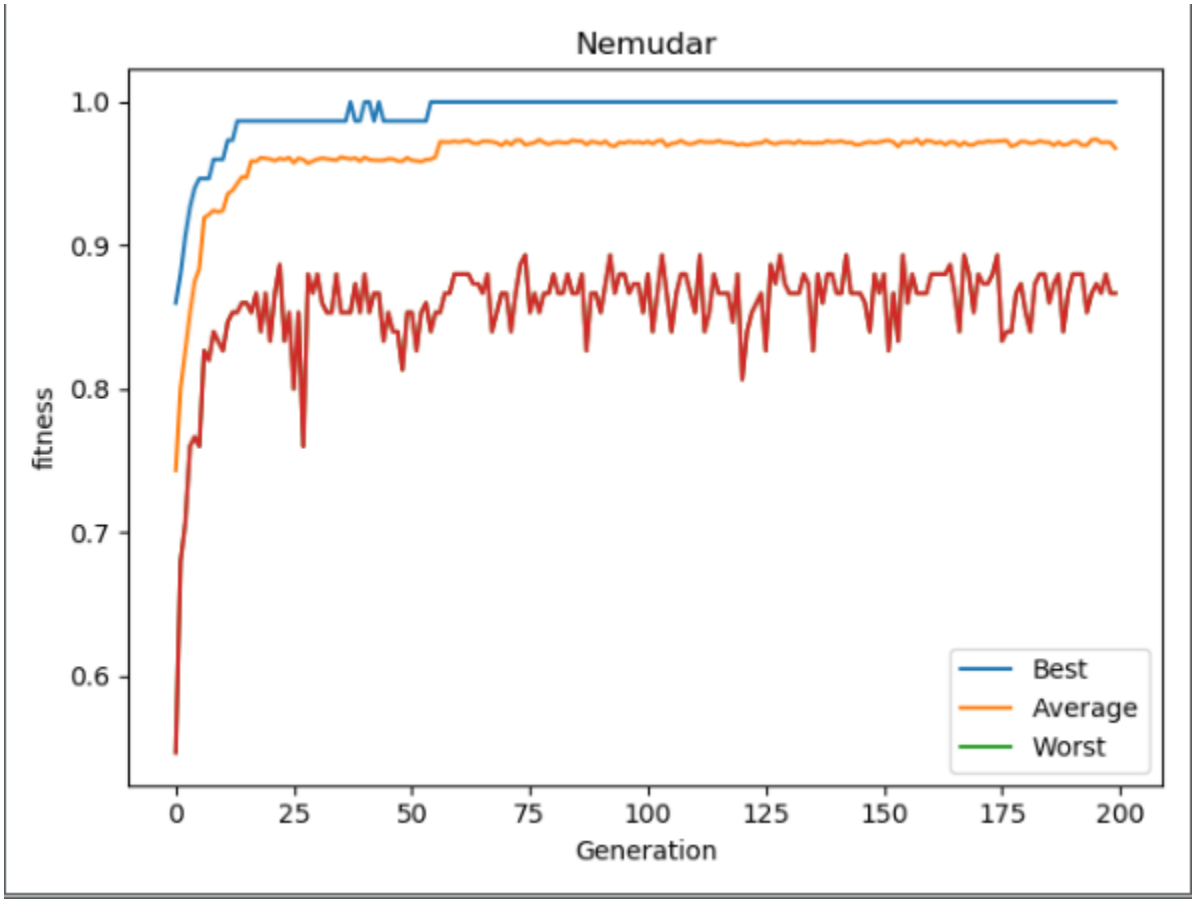
شاید اگر در جمعیت اولیه میزان ارزش بدین صورت نباشد و بسیار پایینتر باشد آنگاه تاثیر جمعیت و نسل بهتر مشخص شود.

و در آخر می آموزیم یافتن پارامترها به صورتی که بتوان در کمترین تعداد نسل و جمعیت به جواب رسید بسیار مهم است.

در مواردی با افزایش نسل به ۵۰۰۰ و یا جمعیت به ۱۰۰۰ هیچ پیشرفتی مشاهده نشد!

و در پایان حالت مورد علاقه بنده حقیر

```
numberOfGenerations = 200
populationSize = 300
tornumentSize = 4
mutationRate = 0.06
```



.3

در مسئله ذوب فلزات نیز متاسفانه علی رغم امتحان موارد بسیار و به حالتی برای حل مسئله دست یافته نشد.

Q_3_Simulated Annealing.py

نهایت ارزشی که به آن رسیدم ۰/۹۰ بود.

به نظر می آید الگوریتم نمیتواند از ماکسیمم محلی فرار کند. به نظر من علت آن است که همسایه ها تا یک قدمی بررسی میشوند.

فرقی هم نمیکند از کدام تابع دما استفاده شود.

در واقع کاربرد این توابع آن است که وقتی هر چه همسایه تولید میکنیم و در ارتفاع بالاتر نیستند ، احتمال پذیرش همسایه ای با ارتفاع پایینتر برای فرار از ماکسیمم محلی بیشتر شود. اما جواب نداد که نداد.

مکانیزمی پیاده سازی شد تا بتوان گیر افتادن در ماکزیمم محلی را بهتر رفع کرد.

بدین صورت که یک متغیر تعریف میکنیم تا بتوانیم حساب کنیم در دور بعد ، تا چه مقدار از حالت فعلی دور شویم و به همسایه دورتری برویم.

هر چه این متغیر بیشتر شود میزان تغییرات برای همسایه تولیدی بیشتر خواهد شد.

این متغیر مثلا x را در هربار تولید همسایه جدید به اندازه ۴۰ تا زیاد میکنیم.

و هربار که ماکسیمم ارزش جدیدی پیدا کردیم (توانستیم به قله بالاتری برویم) آنرا ریست میکنیم.

همچنین در هربار ، میزان تغییرات اعمال شده آنرا کم میکند.مثلا اگر در تولید همسایه جدید ، ۵ تغییر نسبت به حالت فعلی داشته باشیم ، ۵*۲۰ تا از x کم میکنیم.

این متغیر در کد با نام rep به معنی تکرار وجود دارد.

و میزان اینکه چقدر در هربار زیاد شود توسط متغیر stan قابل تنظیم است.

و پس از رسیدن به فاصله ۰/۰۶ از قله اصلی میزان تغییرات ثابت خواهد ماند.چون به احتمال زیاد به قله اصلی نزدیک شده ایم.

یا حتی میتوان هربار که به ارتفاع بالاتر رفتیم آن را کم کنیم و هربار به ارتفاع پایینتر رفتیم آنرا زیاد کنیم تا تغییرات بیشتر شود.

پس از این تغییرات ، ماکسیمم ارزش به ۰/۹۴ افزایش یافت.

اما به نظر میرسد باز هم کافی نبود و نتوانستیم جواب مسئله را پیدا کنیم.