گزارش پروژه میانترم هوش مصنوعی مجید ادیبیان 9631003

1. **نحوه مدل سازی:**

در این پروژه مسئله گفته شده را به شکل مناسبی مدل سازی می کنیم.
کلاس RubicProblem ساخته شده است که بخش های مختلف آن جنبه های مختلف مسئله را مدل سازی می کند.
در ابتدا رنگ های مکعب روبیک از کاربر گرفته میشود و رنگ های هر سطح آن در یک آرایه قرار می گیرد و مجموعه ای از این آرایه ها در یک ارایه قرار می گیرند که کل رنگ های مکعب را تشکیل می دهند.
این آرایه به عنوان حالت اولیه مسئله به آن داده میشود. همچنین رابطه هر خانه از هر سطح مکعب با هر خانه از سطح دیگر ساخته می شود و در مسئله ذخیره میشود که در جابه جایی رنگ ها در هنگام یک حرکت استفاده میشود.
مجموعه ای از اقداماتی که می توان در هر مرحله انجام داد نیز ساخته میشود و در مسئله ذخیره میشود که این اقدامات شامل حرکت ساعتگرد یا پاد ساعت گرد هر سطح می باشد و به صورت آرایه ای از جفت مقدار ها ذخیره میشود که در هر کدام مقدار اول سطح چخش داده شده و مقدار دوم اگر1 باشد به معنی ساعت گرد بودن و اگر 0 باشد به معنی پاد ساعت گرد بودن چرخش می باشد.
همچنین در مسئله ساخته شده باید برای هر حالت patent آن را نیز ذخیره کنیم که برای این منظور یک دیکشنری ساخته شده که کلید آن آرایه حالت کنونی است و value آن که به معنی parent آن است آرایه ای است که از آن با یک اقدام به حالت فعلی رسیده ایم به این ترتیب دسترسی به parent هر حالت از مکعب در اوردر یک انجام میشود.
توابعی که در این کلاس تعریف شده است:

- get_child: تابعی است که با گرفتن یک حالت از مکعب (آرایه دوبعدی گفته شده در بالا) و یک اقدام که یک ارایه دوتایی است که سطح روبیک و ساعتگرد بودن یا نبودن آن را تعیین می کند، تغییرات رنگ ها را با استفاده ار رابطه ساخته شده در ابتدای ساخت مسئله اعمال میکند و حالت جدید مکعب را برمی گرداند.
- get_action: برای برگردندن اقدماتی است که در حالت از مسئله می توان انجام داد که در بالا نحو ذخیره سازی آن گفته شد.
- goal_test: تابعی است برای بررسی آن که حالتی از مکعب که دریافت کرده است آیا حالت هدف است یا خیر که برای این بررسی تمام سطح های مکعب بررسی می شود و اگر تمام رنگ های هر سطح یکسان بود true برمی گرداند در غیر این صورت false برمی گرداند.

- set_parent: تابعی است که فرزند و پدر (آرایه ای از رنگ های مکعب) و اقدامی که از فرزند به پدر می رسیم را می گیرد و در دیکشنری گفته شده در ابتدای این بخش ذخیره می کند تا هر گاه لازم بود در O(1) بتوان به پدر یک حالت دسترسی داشت.
- get_heuristic: تابعی است که آرایه رنگ های مکعب را می گیرد و با توجه به تعداد رنگ های متفاوتی که در هر سطح وجود دارد عددی را برمی گرداند. برای ان منظور برای هر سطح آرایه شش عضوی با مقدار دهی اولیه false داریم و هرگاه در آن سطح رنگ مربوط به آن عدد را دیدیم آن خانه را true می کنیم و در پایان حلقه برای آن سطح بررسی می کنیم که با توجه به تعداد true ها رنگ متفاوت در آن سطح خواهیم داشت.

نحوه پیاده سازی الگوریتم IDS:

در پیاده سازی الگوریتم IDS کلاس IDS ساخته شده است که هنگام ساخت آبجکتی از آن، مسئله ای که می خواهیم این الگوریتم را روی آن پیاده کنیم و limit اولیه الگوریتم را به عنوان ورودی می گیریم و در آن ذخیره می کنیم و تعداد گره های تولید شده و بسط داده شده در ابتدا را صفر می گذاریم.

بعد از ساخت آبجکتی از این کلاس باید تابع iterative_deepening_search را فراخوانی کنیم. در این تابع حلقه ای از limit اولیه تا یک عدد بسیار بزرگ داریم که باعث میشود مطابق الگوریتم IDS، الگوریتم DFS به صورت تکراری و تا سطحی مشخص تکرار شود. در هر بار اجرای این حلقه الگوریتم DLS تا سطحی که متغیر حلقه تعیین می کند اجر میشود و اگر در این اجرا به هدف رسیدیم از حلقه خارج میشویم وگرنه حلقه را ادامه می دهیم.

در الگوریتم DLS به صورت بازگشتی عمل می کنیم در هر بار اجرای آن ابتدا تعداد نود های تولید شده را یکی افزایش می دهیم و سپس بررسی می کنیم که اگر نود فعلی نود هدف است آن را به آرایه مسیر رسیدن به هدف اضافه می کنیم و اگر به سطح limit رسیده ایم cutoff برمی گردانیم، در غیر این با استفاده از آبجکت problem که ورودی این کلاس بود اقدامات ممکن را میگیریم و child های آن حالت را می سازیم و تعداد نود های بسط داده شده را یکی افزایش می دهیم و به ازای هر فرزند تولید شده همین تابع را با ورودی همین فرزند فراخوانی می کنیم. در نتیجه باعث میشه تا رسیدن به هدف یا عمق cutoff به اولین فرزند تولید شده برویم و با برگشت به عقب سایر فرزندان را تولید کنیم.

نحوه پیاده سازی الگوریتم Bidirectional:

در پیاده سازی الگوریتم دو طرفه ابتدا کلاس BidirectionalSearch را می سازیم و آبجکتی از مسئله را به عنوان ورودی به آن می دهیم و در آن ذخیره می کنیم تا بتوانیم از توابع آن استفاده کنیم.دو set به نام های visited1 و visited2 را در آن میسازیم که هر گره را که از گره ابتدایی تولید میشود در visited1 قرار می دهیم و هر یک از گره هایی که از گره هدف ساخته میشود در visited2 قرار می دهیم تا نود های تکراری تولید نشود و در واقع explored set را با استفاده از set پیاده سازی می کنیم تا در O(1) تکراری بودن یا نبودن آن را متوجه شویم.

گره ابتدا و انتهایی را ذخیره می کنیم و از هر دو طرف به صورت BFS پیش می رویم. به این صورت که هر یک از این گره ها را به یک صف مجزا اضافه می کنیم و در هر مرحله از ابتدای صف ها یکی گرفته و آن را بسط می دهیم و فرزندان را به ابتدای صف اضافه می کنیم.

در نتیجه در یک حلقه تا زمانی که هر دو صف خالی نباشند باید هر دفعه از ابتدای صف یک گره برداریم و و اگر این گره در گره های موجود در صف طرف دیگر قرار داشته باشد مسیر پیدا شده است و از حلقه خارج میشویم در غیر این صورت تا خالی شدن حداقل یک صف پیش می رویم.

نحوه پیاده سازی الگوریتم UCS:

برای پیاده سازی این الگوریتم کلاس UCS ساخته شده است که ورودی آن در هنگام ساخت آبجکتی از آن مسئله ای است که می خواهیم این الگوریتم را روی آن اجرا کنیم.

در این کلاس یک min heap داریم که در آن هر آیتم آن یک tuple است که مورد اول آن g است و مورد دوم آن نودی است که آن g را دارد و مرتب سازی این آیتم ها در min heap بر اساس g می باشد. در نتیجه در O(log n) می توان نود جدید تولید شده را به مجموعه نود ها اضافه کرد و در در O(1) می توان به نودی با کمترین g دسترسی داشت.

هنگام ساخت آبجکتی از این الگوریتم تابع run_search را از آن اجرا می کنیم و در یک حلقه تا به گره هدف نرسیده ایم (که این بررسی رسیدن به هدف از طریق صدا زدن تابعی از مسئله داده شده با آن انجام میگیرد که در توضیح توابع مسئله گفته شد) هر دفعه به روش گفته شده گره با کمترین g را بدست می آوریم و به ازای هر یک از فرزندانش مقدار g گره پدر را با یک جمع کرده به عنوان g برای فرزند درminheap ذخیره می کنیم.

مقایسه الگوریتم ها: در توضیحات زیر بعد از توضیح کلی الگوریتم و تعداد گره های خواسته شده در هر یک از آن ها، خروجی آن الگوریتم را برای یه نمونه ازورودی ها نیز مشخص شده است. این نمونه به شکل زیر است:

```
input:
    2 3 5 2
    1 2 6 6
    6 1 5 5
    6 4 3 5
    4 4 4 3
    1 2 3 1
```

این مثال با اقدامات زیر قابل حل است که در هر یک از این اقدامات مقدار اولی سطح مورد نظر برای چرخش است و مقدار دومی اگر یک باشد به معنی چخش ساعت گرد است و اگر صفر باشد به معنی چرخش پادساعتگرد است:

```
actions to solve:
    [1,1],  [3,1], [4,0],  [2,1],  [3,0],  [5,0],  [1,1]
```

- در الگوریتم IDS چون به صورت بازگشتی پیاده سازی شده است در هر سطح تنها یکی از فرزندان ساخته میشود و تا انتها پیش می رود تا برگردد و فرزندان دیگر را بسازد و بسط دهد. به ازای limit صفر یک گره و limit = 1، 12^0+12^1 گره و... و در limit = n، $12^n+...+12^1+12^0$ گره در بدترین حالت تولید میشود که جمع این موارد تعداد گره های تولید شده در بدترین حالت است که از $O(12^n)$ می باشد.

تعداد گره های بسط داده شده همین تعداد خواهد بود و فقط سطر نهایی در هر مرحله بسط پیدا نمی کند و در بدترین حالت تعداد گره های بسط داده شده $O(12^{n-1})$ است.

در این الگوریتم بیشترین گره برای ذخیره سازی گره های مسیری است که از ابتدا تا گره کنونی رفته ایم که بیشترین این مقدار تعداد گره های مسیر از ابتدا تا آخرین سطح است.

خروجی این الگوریتم جز اقدامات خروجی و حالت روبیک بعد از هر یک از این اقدامات به ازای مثال گفته شده به شکل زیر است:

depth of result: 5
number of nodes in memory: 6
number of created node: 102027
number of expanded node: 8505

- در الگوریتم bidirectional که از دو طرف BFS میرویم تعداد گره های تولید شده در بدترین حالت آن است که از هر دو طرف نصف عمق را برویم و گره تکراری نداشته باشیم که در این صورت تعداد گره های تولید شده برابر است با: $2*(12^0+ 12^1 +...+ 12^{n/2})=O(12^{n/2})$

تعداد گره های بسط داده شده تعداد تمام گره های تولید شده به جز گره هایی است که در frontier است که در پیاده سازی انجام شده این frontier ها در صف قرار دارند. این تعداد در بدترین حالت از $O(12^{n/2} + 12^{n/2}) = O(12^{n/2})$ می باشد.

حداکثر تعداد گره های موجود در حافظه در این الگوریتم برابر است با تعداد اعضای explored و تعداد اعضای موجود در صف که حداکثر ای مقدار در بدترین حالت از $O(12^{n/2} + 12^{n/2}) = O(12^{n/2})$ است.

خروجی این الگوریتم جز اقدامات خروجی و حالت روبیک بعد از هر یک از این اقدامات به ازای مثال گفته شده به شکل زیر است:

depth of result: 5
number of_nodes in memory: 904
number of created node: 904
number of expanded node: 97

- در الگوریتم UCS هزینه هر یک از گره ها برابر هزینه گره پدر بعلاوه 1 است که این مورد باعث میشود گره های هر سطح از درخت ساخته شده هزینه یکسانی داشته و از گره های سطح بالاتر هزینه بیشتری داشته باشند و در نتیجه ابتدا تمام گره های یک سطح بسط داده میشوند و سپس به سراغ گره های سطح بعدی میرود. به همین دلیل الگوریتم UCS در این مسئله بسیار شبیه BFS عمل می کند. پس اگر جواب بهینه در عمق n وجود داشته باشد تعداد گره های تولید شده برابر$(12^0+ 12^1 +...+ 12^n +12^{n+1})=O(12^{n+1})$ می باشد و تعداد گره های بسط داده شده نیز برابر $12^0+ 12^1 +...+ 12^n)=O(12^n)$ می باشد.

حداکثر گره هایی که در حافظه قرار می گیرند برابر تمام گره هایی است که در min heap ذخیره شده اند که این مقدار برابر تعداد گره هایی است که تولید شده اند ولی بسط نیافته اند که بیشترین حالت را هنگام رسیدن به جواب دارد و برابر $O(12^{n+1})$ می باشد.

خروجی این الگوریتم جز اقدامات خروجی و حالت روبیک بعد از هر یک از این اقدامات به ازای مثال گفته شده به شکل زیر است:

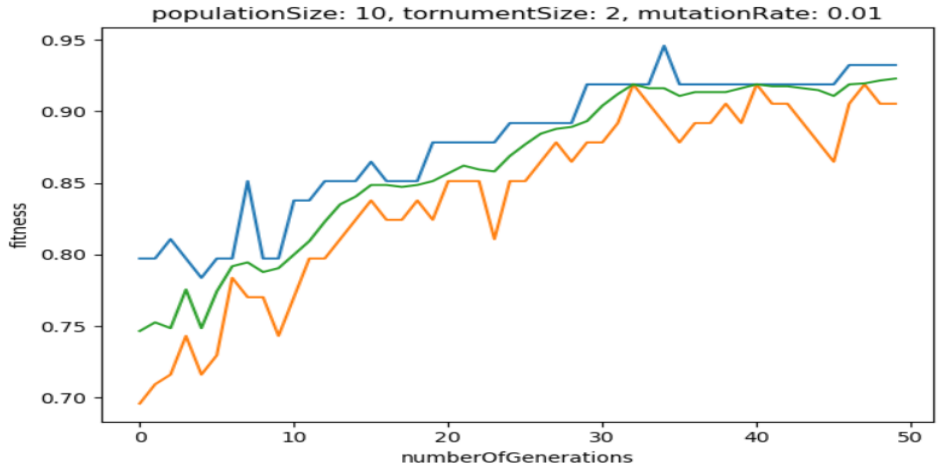
depth of result: 5
number of_nodes in memory: 248831
number of created node: 271453
number of expanded node: 22621

2. در این قسمت مسئله مورد نظر را به صورت کلاس Problem پیاده سازی کرده ایم. در این کلاس رابطه همسایگی بین استان ها را به صورت یک آرایه و بعدی پیاده سازی کرده ایم که هر index از آرایه برای یک استان است و آرایه ای که برای آن قرار دارد شماره استان های همسایه آن استان است و شماره استان بر اساس ترتیب وسعت استان ها است که از صفر شروع شده است.
رنگ آمیزی استان ها به صورت یک آرایه 31 تایی است که برای هر خانه یکی از اعداد 1 تا 3 در نظر گرفته میشود که بیانگر رنگ های متفاوت می باشد.
تابع first_state تابعی است که برای هر استان به صورت رندوم رنگی را در نظر می گیریم.
تابع get_next_state تابعی است که یک استان را رندوم انتخاب می کند و رنگ آن را به صورت رندوم تغییر می دهد و رنگ آمیزی جدید را برمی گرداند.

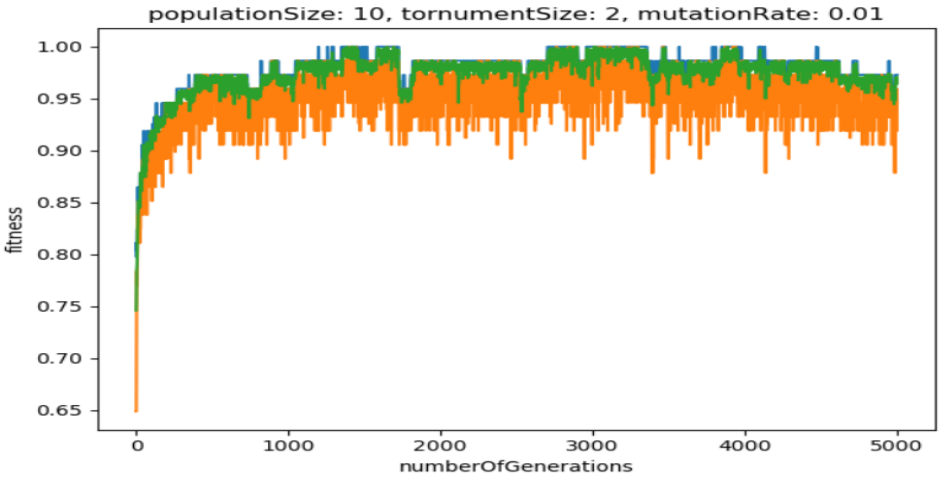
الگوریتم ژنتیک: در این الگوریتم هر یک پارامتر های زیر را در همگرایی بررسی می کنیم:
numberOfGenerations: هرچه تعداد ساخت نسل ها بیشتر شود مراحل الگوریتم بیشتر طی میشود و در نتیجه در انتها رنگ آمیزی استان ها به یکدیگر نزدیکتر میشوند و همگرایی بیشتر میشود.
populationSize: هرچه جمعیت اولیه بیشتر شود پراکندگی رنگ آمیزی استان ها بیشتر میشود و در زمان بیشتری رنگ آمیزی استان ها به یکدیگر نزدیک میشوند و در نتیجه هرچه تعداد جمیت اولیه بیشتر باشد همگرایی سخت تر می شود.
tornumentSize: هر چه این عدد بزرگتر باشد تعداد parent بیشتری انتخاب میشود و در نتیجه پراکندگی رنگ آمیزی بیشتر میشود و همگرایی سخت تر میشود.
mutationRate: هرچه این درصد بیشتر باشد تعداد حالت هایی که یک استان آن ها به صورت رندوم تفاوت رنگ رندوم میگیرد بیشتر می شود و در نتیجه پراکندگی رنگ ها بیشتر میشود و همگرایی سخت تر میشود.

خروجی این الگوریتم را به ازای مقادیر مختلف این معیار ها در زیر دیده میشود که با توضیحات داده شده تطابق دارد:

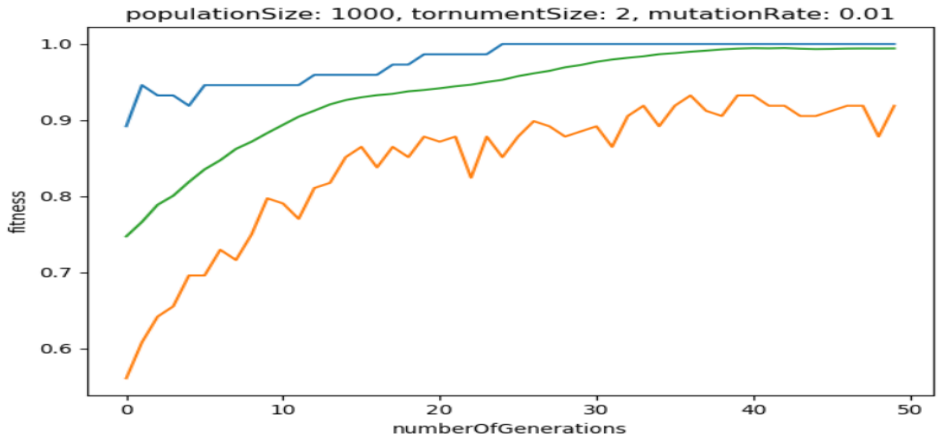
کمترین مقدار برای همه ی معیار ها:



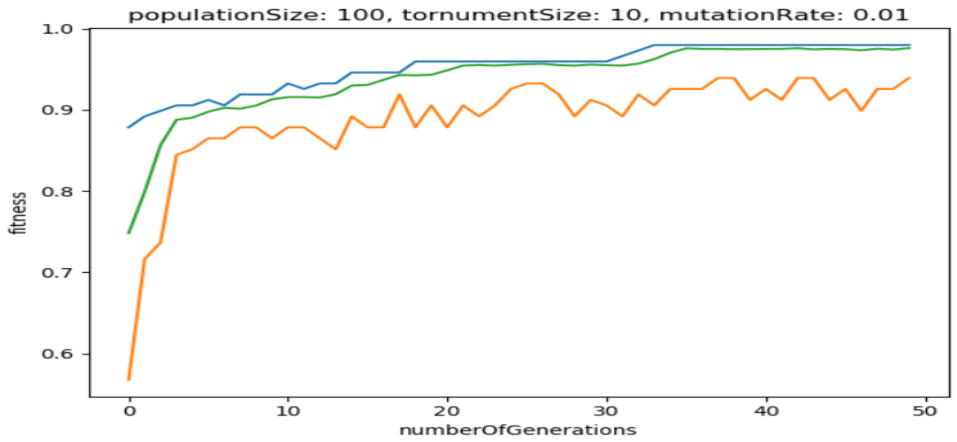
مقدار number of generations را بیشترین قرار داده ایم و بقیه را کمترین مقدار قرار داده ایم:



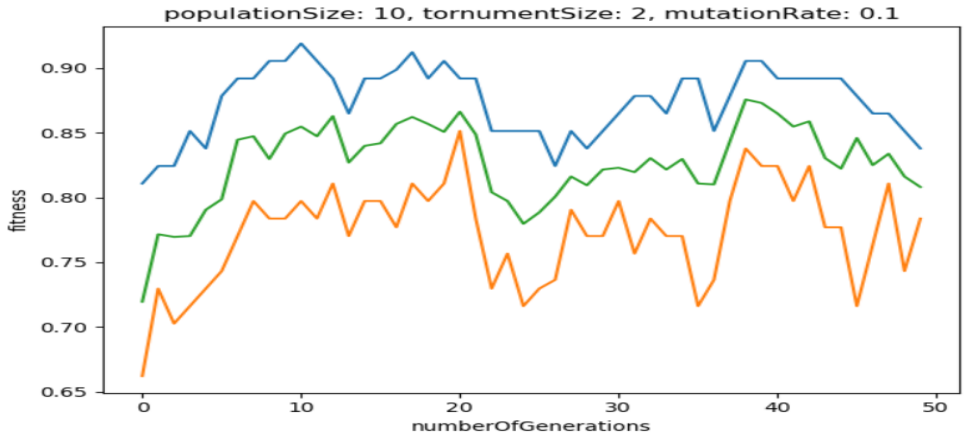
مقدار population size را بیشترین قرار داده ایم و بقیه را کمترین مقدار قرار داده ایم:



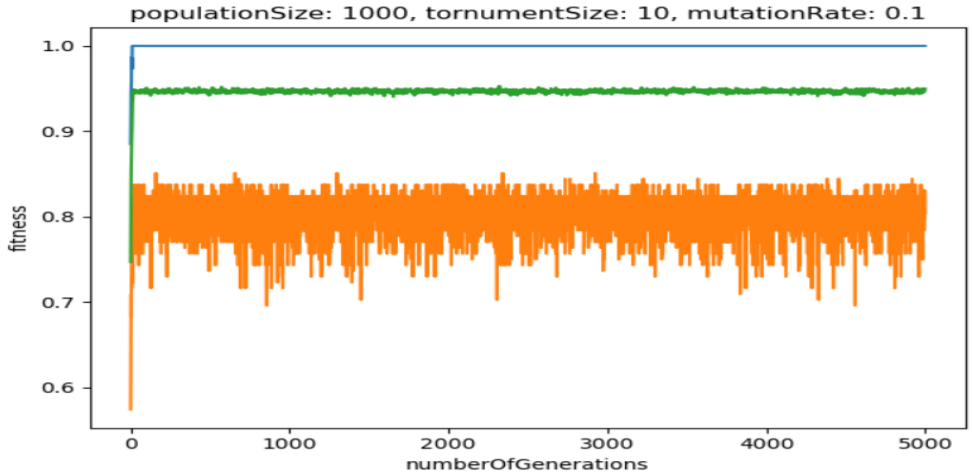
مقدار tornument size را بیشترین قرار داده ایم و بقیه را کمترین مقدار قرار داده ایم:



مقدار mutation rate را بیشترین قرار داده ایم و بقیه را کمترین مقدار قرار داده ایم:



بیشترین مقدار برای همه ی معیار ها:

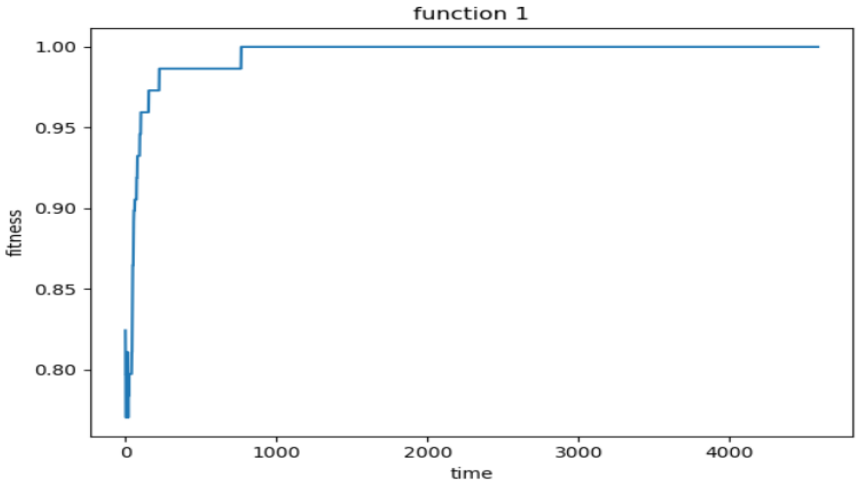


الگوریتم شبیه سازی ذوب فلزات:

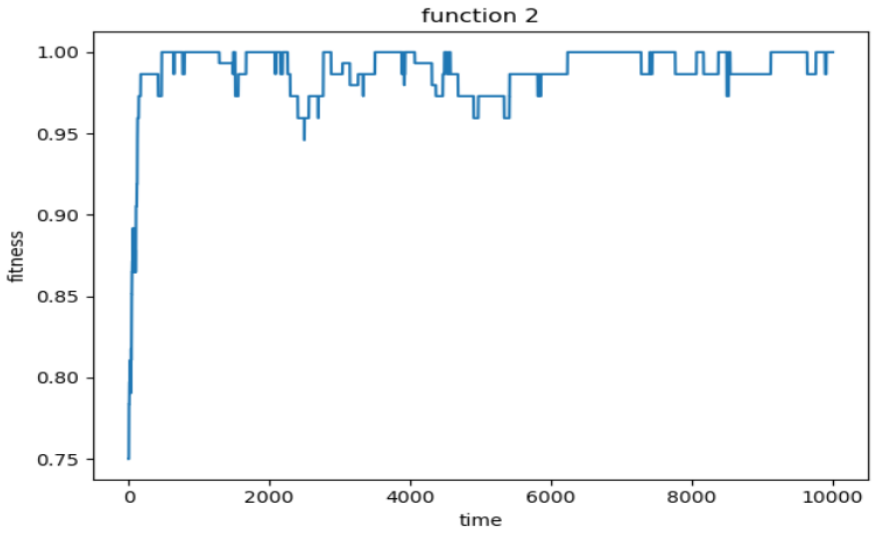
برای این الگوریتم کلاسی به نام SimulatedAnnealing ساخته شده است که ابتدا تابع مورد نظر و مسئله را که در قسمت قبل توضیح داده شد را می گیرد و ذخیره می کند. سپس در تابع run در هر مرحله state فعلی را می گیرید و با استفاده از مسئله گرفته شده state بعدی را بدست می آورد و مقدار fitness انها را مقایسه میکند و اگر بهتر بود state را تغییر می دهد و اگر بدتر بود با احتمالی که با استفاده از تابع بدست می آید به آن می رویم

- در تابع اولی در برنامه ی نوشته شده مقدار الفا را برابر 0.85 در نظر گرفته ایم و مقدار دمای اولیه را یک در نظر می گیریم.در این تابع به صورت نمایی و با سرعت بالایی دما به صفر میل میکند و تعداد حرکات در جهت کاهش fitness کم است.

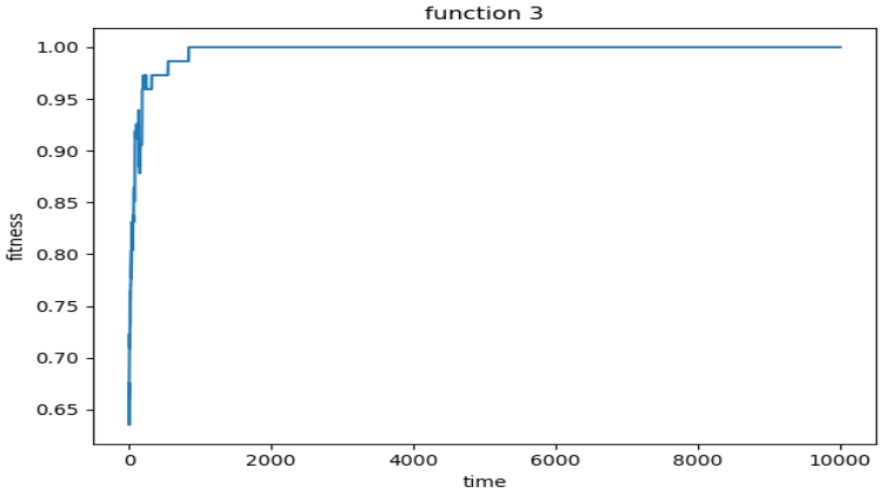
  خروجی این برنامه به شکل زیر می باشد:



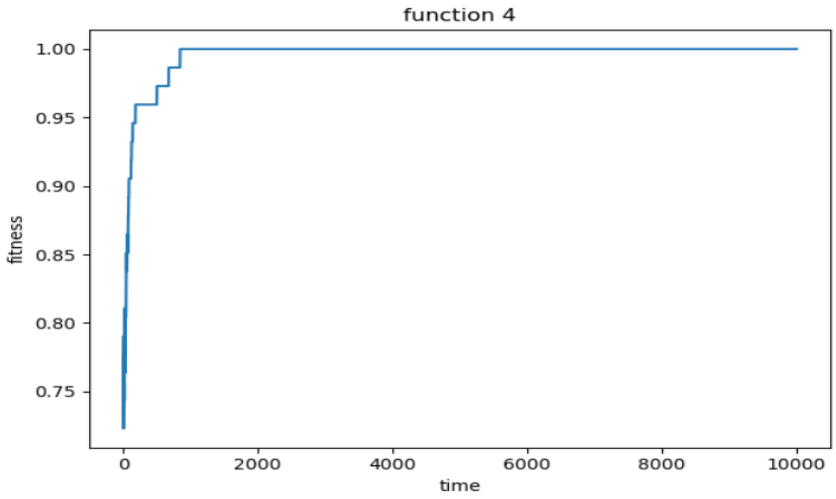
- در تابع دومی مقدار آلفا برابر 100 و مقدار دمای اولیه را 1 در نظر گرفته شده است. در این تابع به دلیل آن که مقدار k در داخل الگاریتم قرار گرفته تاثیر افزایش k در افزایش مخرج کسر بسیار کم است و در نتیجه سرعت کاهش دما در این رابطه بسیار پایین است. خروجی این تابع به شکل زیر است:



- در تابع سوم مقدار آلفا برابر یک و مقدار دمای اولیه هم برابر 1 در نظر گرفته شده است. در این تابع k در مخرج کسر قرار دارد و افزایش آن باعث کاهش دما میشود و این کاهش دما از سرعت کاهش دما در تابع اولی کمتر است و از سرعت کاهش دما در دومی بیشتر می باشد. خروجی تابع سوم به شکل زیر است:



- در تابع چهارم مقدار آلفا برابر 0.05 می باشد و مقدار دمای اولیه را یک در نظر می گیریم. در این تابع k با درجه دو در مخرج تابع قرار دارد و در نتیجه باعث میشود سرعت کاهش دما در این تابع از تابع سوم بیشتر باشد ولی با این وجود سرعت آن از تابع اول کمتر و از تابع دوم بیشتر خواهد بود.


function 4
1.00
0.95
0.90
0.85
0.80
0.75
fitness
0
2000
4000
6000
8000
10000
time