پروژه ژنتیک

نحوه نگاشت مسئله به درخت  
برای نگاشت هر عبارت به صورت کلی از لینک لیست استفاده شده و هر درخت اتریبیوت های زیر را دارد:

* ریشه (root)
* لیست برگ ها (LeafList)
* تعداد گره ها (numberOfNodes)

همچنین دارای تابع های زیر جهت استفاده در ادامه روند الگوریتم و یا تست کردن آن میباشد:

* محاسبه کردن تعداد پدر های یک گره (calculateFather)
* اضافه کردن گره در سطح (addNodeInSurf)
* اضافه کردن گره در عمق(addNodeInDepth)
* اطمینان از کامل بودن درخت(isComplete)

اطمینان از کامل بودن درخت : همه آن هایی که در لیست برگ ها هستند، برگ باشند و گره نباشند .

همچنین برای ساخت هر درخت، ما نیازمند شئ (object) دیگری به اسم گره هستیم؛ که نماینگر هرگره در درخت ما میباشد. اتریبیوت های هر گره :

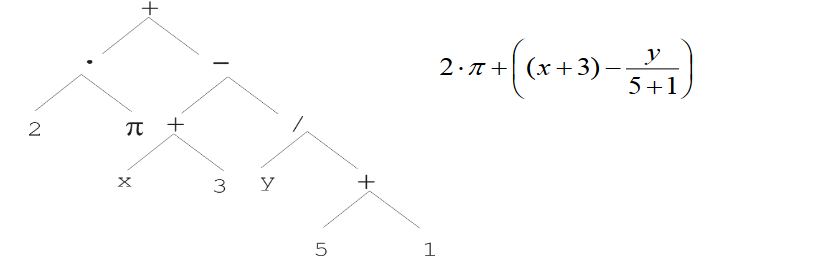
* لینک به پدر خود (Father)
* لینک به گره سمت راست خود (rightnode)
* لینک به گره سمت چپ خود (leftNode)
* عملگر یا مقدار قرار گیرنده در گره (operator)
* مقدار بولین تعیین کننده که ایا گره برگ هست یا خیر (leaf)

و دارای متود هایی نیز میباشد که بعدا استفاده میشوند :

* اضافه کردن اپراتور(assignOperator)
* اضافه کردن گره(addNode)

اضافه کردن اپراتور صرفا یک راه امن برای اضافه کردن عملگر هست، و در خود الگوریتم کاربردی ندارد.

نحوه محاسبه و نمایش درخت  
به صورت کلی همه جمله های ریاضی را میتوان به صورت نوعی درخت تصور کرد، به این صورت که کافی هست درخت را از بالا به پایین به صورت بازگشتی بنویسیم، دقت کنید که قرار نیست عبارات ریاضی را به درخت تبدیل کنیم؛ بلکه درخت های تولید شده توسط برنامه را به جملات ریاضی تبدیل میکنیم. به طور مثال درخت رو به رو قابل تبدیل به عبارات زیر میباشد:



به این صورت که کافی هست صرفا عبارات فرزند را به صورت دو عملوند در دو طرف عملگر هر گره(Node) بنویسیم، به طور مثال برای گره اول داریم : (مقدار سمت راست + مقدار سمت چپ)  
حال اگر بخواهیم عبارت ریاضی کل درخت را به نمایش دربیاریم کافی هست این روند را ادامه دهیم:   
((مقدار سمت راست – مقدار سمت چپ ) + (مقدار سمت راست x مقدار سمت چپ)  
و به همین ترتیب ادامه دهیم تا به برگ ها درخت برسیم. قابل اشاره هست که مقدار عبارت هم به ازای مقادیر از پیش تعیین شده x و y ، به همین صورت حساب میشود، به صورت بازگشتی مقادیر از پایین درخت حساب شده و تا ریشه(root) درخت ادامه میابد .

محدودیت ها  
به صورت کلی پیاده سازی عبارات، محدودیت هایی برای ما به وجود می آورد، که اکثرا قابل حل میباشند:  
1- محدودیت در عملگر های ریاضی : صرفا با پیاده سازی و برخورد با آن ها مثل عملگر جمع موضوع حل میشود .  
2- محدودیت ایجاد شده توسط تقسیم بر صفر: با اضافه کردن مقداری کم (epsilon) در مواقعی که مخرج صفر هست، این موضوع حل میشود و در تقسیم هم مشکلی به وجود نمی آورد. (هایپر پارامتر epsilon)  
3- محدودیت ایجاد شده توسط عملگر ها تک عملوندی : با قرار دادن گره راست به صورت NULL ، و رعایت کردن این موضوع در تمامی الگوریتم های مربوط به اضافه کردن گره و ... ، میتوان آن را حل کرد.  
4- اعداد ثابت موجود در عبارات ریاضی: این مشکل با توجه به این نکته که در هنگام مقدار دهی به اتریبیوت اپراتور در گره حواسمان باشد، حل میشود، به این صورت که مانند متغیر باآن برخورد شود و از طرفی این مشکل وجود دارد که اعداد بازه بینهایت دارند و نمیتوان این بازه را بررسی کرد، در نتیجه بازه معلومی برای اعداد در نظر میگیریم. (هایپر پارامتر CoefficientsRange)  
5- پیوسته بودن اعداد: این خود ویژگی میباشد و به نوعی باعث پیشرفت ما در اجرای الگوریتم اصلی میشود(در ادامه توضیح داده میشود)، در تولید اعداد کافی هست به این نکته توجه داشته باشیم که قابلیت اعشار داشته باشند و از طرفی هم مقداری که میتوان اعشار بگیرند، به صور مثال تا دو دهم، از مقادیر و هایپرپارامتر هایی میباشد که میتوان آن را تغییر داد. ( هایپر پارامتر preecicionInCeo )  
6- مشکل محاسبات توان در درخت های بیشتر: به صورت کلی این مشکل وجود دارد که گاها مقادیر توان بسیار بسیار زیاد شده و از مقدار نهایی که متغیر میتواند بگیرد خارج میشوند، برای همین مجبور به حذف توان

هستیم از مجموعه اپراتور ها، ولی به صورت کلی الگوریتم برای آن هم کار میکند .

تابع های کمکی در الگوریتم:

1-نحوه ایجاد درخت به صورت رندوم

ابن الگوریتم به صورت کلی دو مرحله دارد:

1. مرحله ایجاد گره های درخت
2. مرحله کامل کردن درخت با برگ ها

این گونه عمل میکند که مقداری برای ماکسیموم عمق درخت در نظر میگیرد، حال عددی رندوم تولید کرده و یکی از گره های موجود در لیست گره را برمیگزیند، حال برای انتخاب اپراتور هم به صورت رندوم عمل کرده و مواظب عملگر های تک عملوند هم میباشد، در صورتی که نتواند در عمق اضافه کند به علت مقدار ماکسیموم در نظر گرفته شده، گره دیگری را انتخاب کرده و تلاش میکند که مقدار برای گره راست یا چپ آن انتخاب کند.  
در مرحله دوم به این صورت عمل میکند که تمامی مقادیری که در لیست برگ ها میباشند، قرار هست که برگ باشند ، پس فقط مقداری در بازه تعریف شده توسط هایپرپارامتر و یا یکی از متغیر های تعریف شده را به صورت رندوم انتخاب میکند، دقت کنید تمامی انتخاب ها در این مرحله و مرحله قبل(از جمله انتخاب اپراتور، انتخاب زیر گره برای مقدار دهی ، انتخاب مقادیر ثابت و ... ) به صورت کامل رندوم میباشد .

2-تابع calculateValue :

تابع مقادیر ورودی را گرفته به ازای تک تک آن ها مقدار درخت را درآن نقاط حساب کرده با استفاده از تابع caculate ، و بعد نتایج را به صورت لیست برمیگرداند.

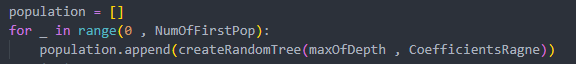
3-تابع randomSelection :

به این صورت عمل کرده که به هرکدام از مقادیر ورودی، با توجه به مقادیر fitness آن ها، وزنی داده و با تولید عدد رندوم، یکی از آن ها را انتخاب کرده و درخت مربوط به آن مقدار شایستگی را بر میگرداند.

4-تابع iterate :  
با توجه به مقداری که در ورودی برای ان تعریف شده ، بر روی درختی که به آن داده شده شروع به گردش میکند و به ازای دیدن هر گره جدید، یک واحد به میزان گردش خود اضافه میکند، تا زمانی که به مقدار تعریف شده برای چرخیدن بر روی درخت برسد. (این تابع برای انتخاب رندوم یک گره توسط الگوریتم جهش مورد استفاد میگیرد)

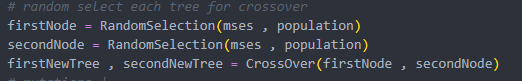
نحوه تولید جمعیت اولیه

به صورت کلی با استفاده از تابع کمکی درست کردن درخت رندوم، با توجه به مقادیری که درخت رندوم میخواهیم، این تابع صدا شده (با مقادیر ورودی که حداکثر عمق و بازه اعداد ثابت را معلوم میکند) و به لیست جمعیت اولیه ما اضافه میشود .



نحوه انتخاب والدین

با صدا کردن تابع کمکی randomSelection ، با توجه به مقدار شایستگی هر درخت ، دو درخت انتخاب شده جهت تولید دو فرزند در نسل بعد .



نحوه تولید نسل بعد

تابع جابجایی(crossOver)

برای تولید نسل بعد به صورت کلی باید دو گره را از دو والد انتخاب شده به صورت تصادفی انتخاب کنیم و ادامه درخت ها را از آن نقطه به بعد جابجا کنیم ، بنا براین با استفاده از تابع iterate و دادن عددی تصادفی بورودی آن، از هر درخت گره به صورت تصادفی انتخاب میکنیم، حال از خود درخت های اصلی یک deepCopy گرفته تا بتوان از از آن برای تولید گره های بعدی استفاده کرد، حال باید دو گره تصادفی انتخاب شده را در در دو درخت کپی گرفته شده عوض شود، که صرفا گرفتن دیپ کپی دیگری از یکی از نود ها و جابجایی رفرنس ها میباشد ( به این علت دوباره دیپ کپی گرفته شد که وقتی جابجا میکنیم، رفرنس های ماهم جابجا شده و برای جابجا شدن نیاز به داشتن رفرنس های اولیه میباشیم) ، همچنین باید نود های بالاتر و والد آن ها نیز این دو گره را بشناسند ، بنابراین به درخت های والد هم لینک این گره های جابجا شده را میدهیم و در نهایت کپی ها را حذف میکنیم .

حال بعد از تولید شدن دو فرزند جدید، نیازمند جهشی با درصد بسیار کم میباشیم، بنابراین عددی را به صورت رندوم تولید کرده و با گرفتن باقیمانده آن، به نوعی درصد کم را تولید کرده و جهش را انجام میدهیم. ( برای هردو گره این عمل تکرار میشود)

تابع جهش (Mutatation)  
مانند تابع crossover ، یک گره را به صورت تصادفی انتخاب کرده و حال اگر اپراتور باشد به صورت تصادفی مقداری از اپراتور ها به آن میدهیم، در صورتی که اپراتور نباشد هم مقداری به صورت تصادفی از اعداد در بازه تعیین شده و یا یکی از متغیرهای موجود را به آن انتساب میکنیم .

حال دو فرزند آماده هستند و به جمعیت جدید اضافه میشوند، این کار را تا آن جایی که مقدار جمعیت جدید به جمعیت اولیه برسد تکرار میکنیم، و در نهایت برای جمعیت بعدی همه این مراحل را تکرار میکنیم .

زمان پایان پذیری الگوریتم  
الگوریتم وقتی پایان میابد که تعداد نسل به میزانی که ماتعیین کردیم را تولید کرده باشد، همچنین در نهایت

تمامی نسل ها، تمامی مقادیر تابع شایستگی ، بهترین درخت تولید شده با توجه به شایستگی و مقدار شایستگی 20% برتر هر نسل را بر میگرداند.

چالش ها مواجه شده و راه حل های ارائه شده

تابع شایستگی

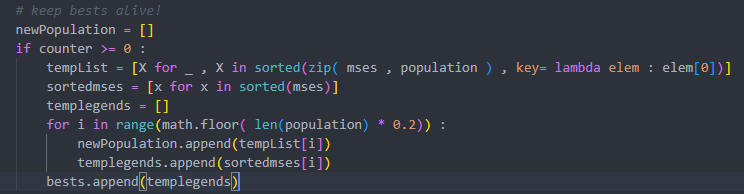
تابع شایستگی ما یکسان نیست ، به این علت که خروجی های الگوریتم در ورژن ها اولیه، به شدت در بخش اول خط ممتد بودند و همچنین بعد از متمایل شدن به سمت پایین هم، خط ممتد میشدند و پیشرفت نمیکردند،  
برای این که این مورد برطرف شود، چندین راهکار استفاده شده، که یکی از آن ها استفاده از تابع شایستگی متغیر میباشد.  
این تابع برای مقادیری که MSE آن ها بیشتر از 1000 واحد باشد(این مقدار با ازمون بدست امده)، از مقدار 1 برروی MSE حساب میکند ، به این علت که تابع به صورت نمایی هست و به مقادیر MSE بسیار بالا مقداری فیتنسی تقریبا مایل به صفر میدهد و سریعا همه درخت ها را به سمت قله ها متمایل میکند .  
در مقادیر MSE بین 1000 تا 1 ، تابع فیتنس به این صورت عمل میکند که به هر درخت مقدار منفی MSE آن و جمع شده با بیشترین مقدار مثبت MSE میدهد، این به نوعی مقدار شایستگی هر تابع را به نسبت خطی به بدترین تابع میسنجد ، در این مرحله همه توابع نزدیک قله های محلی هستند و لازم به حذف نیست، اما باید به آن هایی که به قله نزدیک ترند بیشتر توجه کرد .   
در MSE هایی با مقدار کمتر از صفر ، تابع به صورت توانی از 0.9 عمل میکند، زیرا برای این مقادیر، مقدار توان بیشتر از 1 مقدار MAE ها را زیاد میکند، پس تابع فیتنس همه MAE ها را به توان 0.9 رسانده و بعد منفی کرده و مانند قبل عمل میکند .  
نکته 1 : به این مورد توجه شود که منظور از MAE ، mean absolute error هستش، و برای مقادیری کمتر از یک عملا ما مقدار مثبت اختلاف ها را به توان 0.9 میرسانیم و ادامه آنچه گفته شد .  
نکته 2 : این تغییر تابع شایستگی برای یک نسل به صورت کامل رخ میدهد، و معیار آن هم این هست که 20% برتر نسل قبلی ، میانگین MSE به چه صورت داشته اند، و باتوجه به میانگین آن ها تابه شایستگی هر نسل عوض میشود .  


Figure 6انتخاب 20% برتر و نگه داشتن آنها

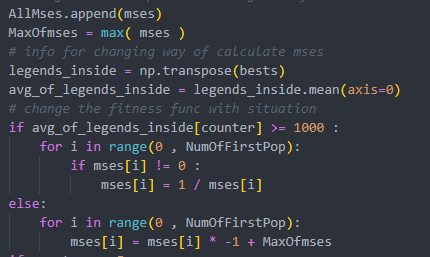


Figure 7تغییر کردن تابع fitness

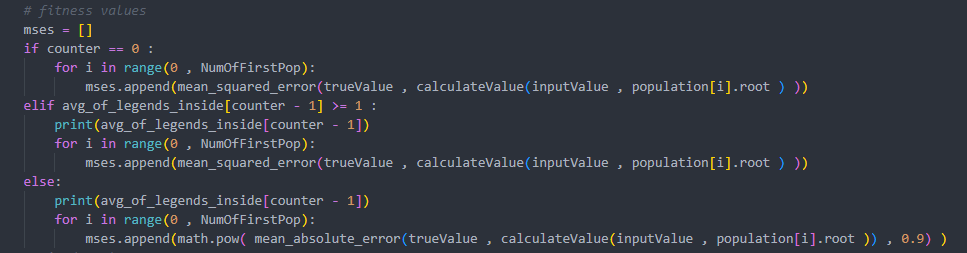


Figure 8 تغییر کردن MSE به MAE

نگه داشتن درصد بهتر یکی از مشکلاتی که بود این بود که الگوریتم در گام های اول به مقادیری میرسید که نزدیک به تابع اصلی بود، ولی آن درخت از جمعیت حذف میشد، درست هست که بچه های زیادی تولید میکرد، ولی هیچ لزومی نبود که بچه ها بهتر باشند و ممکن بود کلا آن ژن از بین برود، بنابراین در مرحله اول از نصف راه به بعد 20% برتر را در هر نسل نگه میداشتیم برای نسل بعدی ، اما با کمی بررسی به این رسیدیم که از همان اول 20 % برتر را نگه داریم، اگر مقادیری دورافتاده باشند در نهایت حذف میشوند، و این همان 20% میباشند که در روند تغییرات تابع شایستگی اثر گذارند.

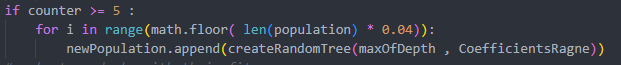
روش های دیگر برطرف کردن یکنواختی  
1 – تولید کردن درخت های تصادفی به صورت درصد کمی از هر نسل :  
در نسل های نهایی، مشاهده میشد که تابع به شدت به تابع اصلی نزدیک شده ولی به آن نمیرسد، یکی از دلایل به علت نبود اپراتور مورد نظر در بین درخت های مانده هست، این مشکل با اضافه کردن حدود 5 % از جمعیت از درخت هایی کاملا تصادفی ، تا حدودی برطرف میشود و احتمال آن که آن اپراتور مورد نیاز تولید شود و اضافه شود ایجاد میشود . (از نسل 5 به بعد انجام میشود، به این دلیل که اولویت به قله ها نزدیک شدن هست)   


Figure 9اضافه کردن جمعیت به هر نسل

2 – استفاده از annealing در احتمال جهش ها:  
یکی دیگر از کار هایی که در این بخش کمک میکند، این هست که احتمال جهش از اول الگوریتم بسیار باشد و بعد که به انتهای الگوریتم میرسیم این احتمال کمتر شود، باعث میشود که الگوریتم بتواند از اول طیف گسترده ای را در بر بگیرد، بنابراین این احتمال که اپراتوری در عبارات درخت های مانده وجود نداشته باشد و در تابع اصلی باشد، کمتر میشود، همچنین درصد آن باید رفته رفته کمتر شود تا بتوان به یک قله تخمینی رسید .



Figure 10نمونه استفادهاز annealing

3- بررسی فرزند های تولید شده:  
به صورت کلی وقتی به سمت قله در نمودار فیتنس حرکت میکنیم و الگوریتم به پایان خود نزدیک میشود، تابع های تولیدی بسیار شبیه بهم میشوند و در نقاطی عینا یکسان میشوند، برای این که جمعبت جدیدمان پویا باشد و همه تابع های آن یکی نباشند ، درصورتی که تابع فرزند تولید شده با یکی از والدین یکی باشد، حتما باید جهش پیدا کند و به جمعیت جدید اضافه شود تا حداقل داده تکراری نباشد و مفید باشد .



Figure 11قسمت جهش اجباری

راه حل اضافه کردن مقادیر پیوسته:

تابع های تولیدی وقتی که اعداد ثابتی که در اختیار داشتیم گسسته بودند، بسیار بسیار تخمین هایی بدتر به نسبت زمانی که اعدادمان پیوسته بودند میزدند، زیرا با ترکیب اپراتور ها به مقداری اعشاری میرسیدند که به این راحتی نمیتوان به آن ترکیب رسید، بنابراین با اضافه کردن اعداد پیوسته این مشکل هم حل شد ، گرچه ن هبه صورت کامل چون اعداد پیوسته ماهم به نوعی پیوسته کامل نیستند .

آزمایش های انجام شده :

