به نام خداوند بخشنده‌ی مهربان

گزارش پروژه‌ی اول هوش مصنوعی

تقریب تابع با کمک برنامه‌نویسی ژنتیک

نیم‌سال اول ۱۴۰۰-۱۴۰۱

سید پارسا نشایی - ۹۸۱۰۶۱۳۴

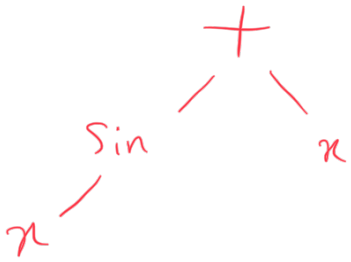
دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

مقدمه

در این پروژه، برنامه‌ای به زبان Python نوشته شده تا ضابطه‌ی تابعی که ضابطه‌ی آن مشخص نیست و تنها در نقاط مشخصی، از آن نمونه‌برداری شده است، به کمک برنامه‌نویسی ژنتیک، تخمین زده شود. در کد این برنامه، مجموعه‌ای از توابع، از پیش تعریف شده‌اند که برنامه ابتدا از آن‌ها نمونه‌برداری کرده و سپس سعی می‌کند به کمک برنامه‌نویسی ژنتیک، تخمین مناسبی از ضابطه‌ی تابع را با معیارهایی که در ادامه بیان خواهند شد، اعلام کند. برنامه، افزون بر نمایش فرمول متناظر با تابع تخمین زده شده، میزان شایستگی حاصل، تعداد نسل‌های طی شده، تعداد دفعات محاسبه‌ی شایستگی و نیز زمان اجرای عملیات را برمی‌گرداند. کتاب‌خانه‌های استفاده شده، numpy برای عملیات مقدماتی برداری ریاضی (به اسم np در کد وارد شده) و نیز matplotlib.pyplot برای رسم نمودار (به اسم pplot در کد وارد شده) هستند. الگوریتم کلی، بر اساس منابع شکل گرفته و سپس بر اساس آن الگوریتم کلی، کد را نوشتم و البته مواردی که صریحا در منابع مشخص نبود، به همراه حل چالش‌هایی که همگی آن‌ها لزوما در منابع پیش‌بینی نشده بودند، توسط خودم اضافه شده‌اند.

نحوه‌ی دقیق نگاشت مسئله به درخت و محدودیت‌های احتمالی

می‌دانیم که توابع ریاضی را می‌توان به صورت یک درخت نمایش داد، به طوری که هر راس میانی، یک عملگر است که اگر عملگر دوگانی (مانند جمع و تفریق) باشد، مقدار آن برابر حاصل اجرای عملگر روی مقدار دو فرزندش و اگر عملگر یگانی (مثلا سینوس) باشد، مقدار آن برابر حاصل اجرای عملگر روی تنها فرزندش است. به طور مثال، نمایش $sin(x) + x$ به شکل زیر است:



در برگ‌های درخت‌های مانند فوق، متغیر تابع (یعنی x – چون در این پروژه، توابع تک‌متغیره‌اند) قرار دارد. در این پروژه، کروموزوم‌ها، همین درخت‌ها خواهند بود که به صورت مداوم تغییر کرده و برای هر درخت، مقدار عبارت نشان داده شده توسط درخت در نقطه‌های ورودی با مقدار نمونه‌برداری شده از تابع مقایسه می‌شود تا تابع شایستگی هر کروموزوم به دست آید؛ این تابع شایستگی، توانایی مشخص کردن بقا و یا عدم بقای این کروموزوم / درخت / عبارت را در نسل‌های بعدی دارد.

به عنوان عملگرهای درخت، موارد زیر قرار داده شده‌اند:

- عملگرهای دوگانی (تعریف شده در BINARY_FUNCTIONS): جمع، ضرب، تفریق، تقسیم و توان
- عملگرهای یگانی (تعریف شده در UNARY_FUNCTIONS): لگاریتم، سینوس، کسینوس، تابع exponential و رادیکال

برخی از توابع فوق ممکن است به خطا برخورند (مانند تقسیم بر صفر و موارد مشابه) که با try-except های قرار داده شده در نسخه‌های my این توابع که در کد تعریف شده و به عنوان پوششی روی نسخه‌های استفاده کننده از np نوشته شده‌اند، این خطا برطرف می‌شود.

لازم به ذکر است که توابع فوق قادر به نمایش تمام عبارات ریاضی – حداقل به راحتی – نیستند؛ به همین منظور و به دلیل غلبه بر محدودیت‌های احتمالی، دو عملگر یگانی دیگر نیز اضافه شده است:

- عملگر addwithone – این عملگر، مقدار فرزند خود را با یک جمع می‌کند. اگر فرزند این عملگر، x-x باشد (یعنی عملگر دوگانی منفی با دو فرزند x و x باشد)، خروجی این عملگر، مقدار ثابت یک خواهد بود و این عملگر، نیاز به مقدار ثابت در بیان عبارات را برقرار می‌کند.

- عملگر inv – این عملگر، فرآیند معکوس شدن عبارات را تسهیل می‌کند، به گونه‌ای که لازم نباشد ابتدا ثابت ۱ ساخته شده و سپس توسط عملگر تقسیم، بر عبارت مدنظر تقسیم شود.

تابع شایستگی

برای به دست آوردن مقدار عبارت نشان داده شده توسط درخت در ساختار درختی فوق، کافیست مقدار متغیر را با مقدار داده شده جای‌گزین کرده و به جای هر راس، مقدار آن را قرار دهیم، یعنی هنگام بررسی مقدار هر راس، از طریق بررسی مقدار فرزندان به شکل بازگشتی و سپس اعمال عملیات لازم، مقدار راس کنونی را به دست آورده تا مقدار خروجی ریشه که همان مقدار عبارت در نقطه‌ی داده شده است، به دست آید. تابع شایستگی باید تابعی باشد که با نزدیک‌تر شدن مقدار تابع تخمین زده شده (مقدار درخت) به خروجی داده شده (نمونه‌برداری شده) در یک نقطه، مقدار آن بیش‌تر شود؛ از این رو، چون اگر فاصله (قدر مطلق) میان مقدار تخمین زده شده و مقدار نمونه‌برداری شده کم شود، تخمین دقیق‌تر است، می‌توان مقدار «۱ بر روی فاصله (قدر مطلق) میان مقدار تخمین زده شده و مقدار نمونه‌برداری شده» را به عنوان تابع برازش (fitness) در نظر گرفت. می‌توان به جای فاصله‌ی قدر مطلق، از مربع اختلاف مقادیر نیز استفاده کرد که رفتاری که در برنامه صورت می‌پذیرد، توسط متغیر fitness_evaluation_method قابل تنظیم است. هم‌چنین، مقدار بی‌نهایت را به عنوان fitness تخمینی که دقیقا برابر تابع اصلی است، در نظر می‌گیریم.

تولید جمعیت اولیه

جمعیت اولیه، به شکل تعدادی درخت (یک جنگل از درخت‌های عبارت) خواهد بود که به صورت تصادفی، به شکل full یا partial (روش grow) پر می‌شود، یعنی به شکل تصادفی و به احتمال ۰.۳، درخت به شکل کامل و به نوعی بالانس ساخته می‌شود. به دلیل تصادفی بودن الگوریتم، طبیعتا در هر دو اجرای پیاپی، جواب یکسانی نخواهیم داشت.
تعداد درخت‌هایی که در جمعیت اولیه ساخته می‌شوند، توسط متغیر population_size و عمق اولیه‌ی درخت‌های جمعیت اولیه نیز توسط متغیر initial_tree_depth در کد قابل تنظیم است. در constructor هر درخت، میزان accuracy (که به شکل یک تقسیم بر تابع برازش در بخش قبل تعریف شد)، در ابتدا برابر صفر تنظیم شده و سپس nodeهای اولیه، بسته به حالت تعیین شده (full یا غیر آن)، پر می‌شوند. در حالت پر، تا هنگامی که به عمق مجاز نرسیده‌ایم، برای هر راس، یکی از دو احتمال یکسان وجود یک عملگر یگانی یا دوگانی وجود دارد. در برگ‌ها نیز متغیرها

قرار می‌گیرند. در حالت غیر full، رئوس میانی نیز ممکن است به احتمال یک دوم به برگ منجر شوند (و برگ‌ها لزوما تنها در سطر (سطح) آخر درخت، موجود نیستند) و به احتمال یک دوم نیز مشابه full پر می‌شوند.

نحوه‌ی انتخاب والدین و تولید نسل بعد

انتخاب والدین در الگوریتم نوشته شده به شکل مرسوم خود انجام شده است، یعنی در یک متغیر در کد که مقدار آن قابل تنظیم است، number_to_choose_each_iteration که به معنای تعداد کاندیدهای لازم برای والد بودنی است که در هر مرحله انتخاب می‌شوند، مشخص شده و سپس در تابع findTree در کد، به تعداد number_to_choose_each_iteration از لیست تمام کروموزوم‌ها (درخت‌ها) نمونه‌برداری انجام شده و سپس بهترین کروموزوم این نمونه، به عنوان والد منتخب برگردانده می‌شود. سپس به تعداد iteration ها و به ازای هر یک از آن‌ها، دو والد انتخاب شده و عملیات crossover و نیز mutation که جلوتر جزئیات آن‌ها شرح داده می‌شود، روی آن دو والد انجام شده و سپس درخت جدید به جمعیت به این شکل در تابع addNewTreeToPopulation اضافه می‌شود که جای‌گزین درخت با کم‌ترین میزان fitness در جمعیت می‌شود. در این روش از برنامه‌نویسی ژنتیک که در انجام این پروژه در پیش گرفته شده است، در هر جهش، والدین حفظ می‌شوند (که اگر صفت خوبی داشتند، در نسل‌های بعدی به جا بماند و هم‌چنان بتوانند تولید مثل کنند) و از دو والد انتخاب شده در آن نسل نیز یک فرزند شکل می‌گیرد (همه‌ی خانواده‌ها در هر نسل، تک‌فرزنده هستند).

نحوه‌ی ترکیب متقاطع

نحوه‌ی crossover که در تابع performCrossover انجام می‌شود، به این شکل است که یک بازه از درون درخت یکی از والدین، جایگزین بخشی از درخت والد دیگر می‌شود و در نتیجه درختی تشکیل می‌شود که بخشی از عبارت آن، از عبارت یکی از والدین و باقی عبارت (عبارات حول و حوش آن عبارت)، از والد دیگر آمده است. در الگوریتم نوشته شده، این crossover اگر عمق فرزند تولید شده بیش‌تر از حداکثر عمق ممکن برای درخت باشد، به احتمال یک چهارم صورت می‌گیرد و به احتمال سه چهارم، مستقیما یکی از والدین باقی خواهند ماند (به نوعی می‌توان گفت فرزند والد، برابر خودش خواهد بود)، زیرا وقتی از برنامه‌نویسی ژنتیک برای تخمین توابع استفاده می‌کنیم، جایگزین شدن تنها بخشی از میانه‌ی عبارت از دو والد که به پاسخ نزدیک‌اند اتفاقا ممکن است اثر معکوس داشته و سبب دورتر شدن تخمین از پاسخ شوند (بالاخص اگر عمق درخت آن‌قدر زیاد شود که به معنی پیچیده شدن بیش از حد تابع باشد که احتمالا منظور کاربر

برنامه نبوده است) و در این حالت، crossover انجام نمی‌شود. (مشخصا بر اساس توضیحات فوق، حداکثر عمق ممکن برای درخت، در این برنامه، یک حد تخمینی است که در متغیری به برنامه داده شده است و کران بالای اکیدی نیست)

نحوه‌ی جهش

نحوه‌ی mutation که در تابع performMutation انجام می‌شود، به این شکل است که اولا به احتمال ۰.۱ کلا جهشی انجام نمی‌شود، اما اگر تصمیم گرفته شود که جهشی انجام شود، یک راس از راس‌های درخت به صورت تصادفی انتخاب شده و بسته به نوعش (عملگر یگانی یا دوگانی)، با یک عملگر هم‌نوع دیگر به صورت تصادفی عوض می‌شود تا ساختار درخت تغییر نکند. در این کد، در صورت انتخاب شدن راس‌های حاوی متغیر، جهشی ایجاد نمی‌شود. در انتها، درخت جهش‌یافته برگردانده می‌شود. ورودی این تابع، درخت crossover شده است.

شرط خاتمه‌ی الگوریتم

الگوریتم نوشته شده، در یکی از دو حالت زیر خاتمه می‌یابد:

- تابع fitness برای درخت جدید تولید شده در یک نسل، بیش‌تر از 10^9 شود – که به معنای اختلاف کم‌تر از 10^{-9} تخمین و واقعیت است و به این دلیل این شرط گذاشته شده که اگر به تخمین خوبی رسیدیم، دیگر لازم نباشد الگوریتم را ادامه دهیم تا با جهش‌ها و سایر عملیات احتمالی، از جوابی که به آن رسیده بودیم، دورتر نشویم و هم‌چنین الگوریتم نیز در بسیاری از حالات، سریع‌تر به پاسخ برسد
- تعداد نسل‌های به‌پیش‌رفته، از حداکثر تعداد نسل‌های مشخص شده (max_count_of_iterations) بیش‌تر شود که به این شرط گذاشته شده که اجرای برنامه تا ابد ادامه نیافته و اگر از جایی به بعد تخمین بهتر نشد، بهترین تخمین تا آن لحظه که هم‌اکنون در جمعیت وجود دارد، برگردانده شود

چالش‌های مواجه شده و روش حل آن‌ها

در مسیر راه توسعه‌ی این کد، به چالش‌هایی برخوردم که برای حل آن‌ها سعی کردم یا از منابع کمک گرفته یا در مواردی که در منابع موجود در اینترنت صریحا مشخص نشده بودند، حالت‌هایی را تست کنم تا به حالتی که به جواب‌های قابل قبولی برای برنامه منجر شود، برسم. چند چالش اصلی‌تری که به آن‌ها برخوردم، موارد زیر بودند:

- در ابتدا که برنامه را نوشتم، تنها به عملگرهای مرسوم بسنده کرده بودم و پشتیبانی از عدد ثابت وجود نداشت و در صورت ورودی دادن تابعی مانند $\sin(x) + 2$، خروجی نه چندان مطلوبی داده می‌شد. با افزودن عملگر addwithone، این چالش حل شد (البته راه دیگری هم آن بود که مقدار ثابت را در برگ‌ها تعریف کنیم). وقتی این عملگر را اضافه کردم، تصمیم گرفتم عملگر inverse را هم اضافه کنم که توابعی مانند $\frac{1}{x}$ و خانواده‌ی آن، به راحتی به دست آیند.

- وقتی عملگرها را ساختم، از ابتدا دقت داشتم که اعتبارسنجی ورودی‌های تابع divide را از طریق try-except انجام دهم تا تقسیم بر صفر صورت نگیرد (و نیز توابعی دیگر مانند sqrt که قدر مطلق ورودی همواره به آن داده می‌شود تا ورودی منفی به sqrt خود نام‌پای داده نشود)، اما این دقت را برای تابع exp و نیز عملگر توان به کار نبردم و پس از اجرای برنامه با امتحان کردن چند ورودی به خطای overflow در به توان رسانی برخوردم و در نتیجه این چک و کنترل را به آن دو تابع نیز افزودم.

- در ابتدا برای تخمین توابع مختلف هر بار مجبور بودم کد توابعی ریاضی که از قبل نوشته بودم را کپی کنم و برای حل این چالش، آرایه‌ای از lambda ها ساختم و هر بار برای اجرا تنها اندیس تابع ریاضی انتخاب شده از تابع را عوض می‌کنم تا تابع متفاوتی تست شود.

- ورودی صفر منجر به مشکلات متعددی در توابعی مانند لگاریتم می‌شد که در آن نقاط خوش‌تعریف نیستند و یا تعریف‌نشده‌اند؛ مشابها ممکن بود برای ورودی ۱ نیز در توابعی مانند $log(1 - x)$ مشکل پیش آید. برای حل این چالش، دامنه‌ی نمونه‌برداری را به جای ۰ تا ۱ که در ابتدا نوشته بودم، بازه‌ی ۰.۱ تا ۰.۹ در نظر گرفتم که هر ۰.۰۱ فاصله، یک نمونه گرفته شده است.

توضیح بخش‌های توضیح داده نشده از کد

- تابع check_for_khatkhati_value: این تابع، شامل مقادیر نمونه‌برداری شده از یک تابع خطخطی شده است که در دستور کار به آن اشاره شده است. یک تابع برای کار برداری با این تابع به نام khatkhati_function نیز در ادامه آمده است.

- تابع check_for_not_continuous_value: این تابع، شامل مقادیر نمونه‌برداری شده از یک تابع دارای ناپیوستگی است که در دستور کار به آن اشاره شده است. یک تابع برای کار برداری با این تابع به نام not_continuous_function نیز در ادامه آمده است.
- مقدار number_to_choose_each_iteration به طور مستقیم تنظیم نمی‌شود، بلکه مقدار درصدی از کل جمعیت که باید نمونه‌برداری شود – یعنی percent_to_choose_each_time – تنظیم می‌شود و number_to_choose_each_iteration از روی percent_to_choose_each_time به دست می‌آید.
- تابع calculateValueOnVariableValue، با پیمایش درخت، مقدار هر راس را به شکل بازگشتی به دست می‌آورد تا مقدار ریشه به دست آید؛ توضیحات بیش‌تر در خصوص این تابع، پیش‌تر گفته شده است.
- تابع findExpression، با پیمایش درخت، مقدار expression (عبارت) مربوط به هر راس از روی عملگر آن راس و نیز فرزند(ان)ش (و یا اگر متغیر باشد، خود متغیر) به شکل بازگشتی به دست می‌آید که برای چاپ حاصل به دست آمده، این تابع مورد نیاز است.
- تابع calculateDepth به شکل بازگشتی عمق درخت را به وضوح محاسبه می‌کند.
- تابع calculateAccuracy مقدار وارون fitness درخت را از طریق اعمال تابع روش مشخص شده به ازای تمام ورودی‌ها، محاسبه می‌کند.
- تابع findAllAccuracies، مقدار وارون fitness درخت را برای تمام درخت‌های داخل جمعیت کنونی با صدا زدن calculateAccuracy محاسبه می‌کند.
- تابع bestInSample، درخت (کروموزوم) با بیش‌ترین fitness (و در نتیجه کم‌ترین وارون fitness، زیرا fitness نامنفی است) را در نمونه درخت‌های داده شده، برمی‌گرداند.
- تابع findVariableInTree، به وضوح به دنبال مکان متغیر در درخت می‌گردد.
- تابع estimate: این تابع مراحل ایجاد هر نسل از روی نسل قبلی را تا برقراری شرط خاتمه به پیش برده و سپس بهترین کروموزوم (درخت) را برمی‌گرداند.

پس از تعریف توابع، ابتدا `np.seterr(all='raise')` نوشته شده تا در صورت ایجاد خطا در محاسبات، بخش‌های except فعال شوند، سپس توابع نمونه تعریف شده، یکی از آن‌ها انتخاب شده، دامنه‌ی ورودی ساخته شده و خروجی‌های نمونه‌برداری‌شده در y ریخته می‌شوند و در انتها، دو عملیات ساخت جمعیت اولیه و نیز پیش‌برد

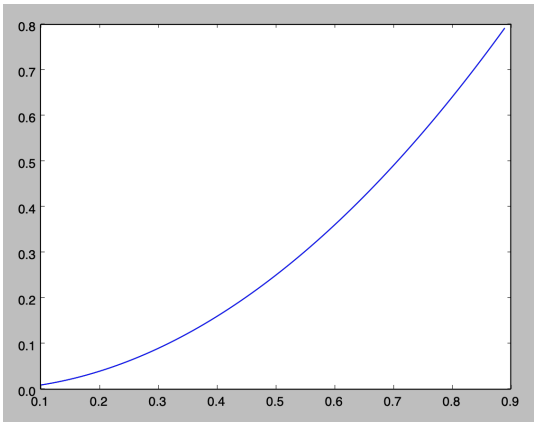
نسل‌ها، با اندازه‌گیری زمان، انجام می‌شوند. در نهایت، عبارت حاصل، میزان شایستگی و زمان چاپ شده و برای کسب شهود بهتر از میزان دقت و کارایی کد، در یک نمودار به کمک matplotlib، تابع اصلی به رنگ سبز و تابع تخمین زده شده به رنگ آبی رسم می‌شود.

آزمایش‌های انجام شده

در ادامه، نتیجه‌ی آزمایش برنامه برای ۱۲ تابع متمایز ورودی، آمده است:

- تابع $y(x) = x^2$:

```
Count of generations: 38
Count of fitness calculations: 1539
Expression: y(x) = (x * x)
Fitness: infinity
Time: 1.67586803436 seconds
```

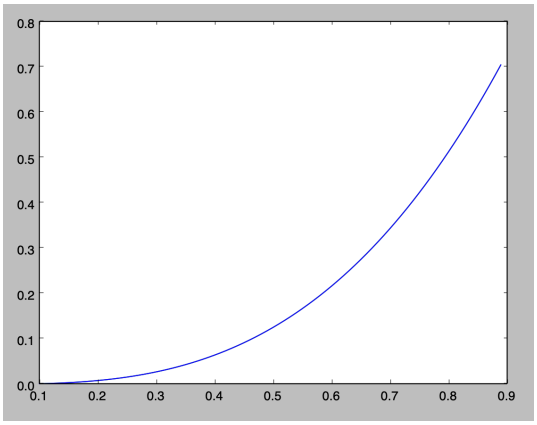


تابع فوق کاملا به درستی تخمین زده شده است؛ که طبیعی است زیرا خیلی راحت از روی درختی با ریشه‌ی * و نیز دو فرزند هر دو متغیر به دست می‌آید. پس از تنها ۳۸ نسل، برنامه به پاسخ رسیده است.

- تابع $y(x) = x^3$:

```
Count of generations: 1
Count of fitness calculations: 1502
Expression: y(x) = (((x * x) - (x - x)) *  sqrt((x * x)) )
Fitness: 1.39801016095e+17
```

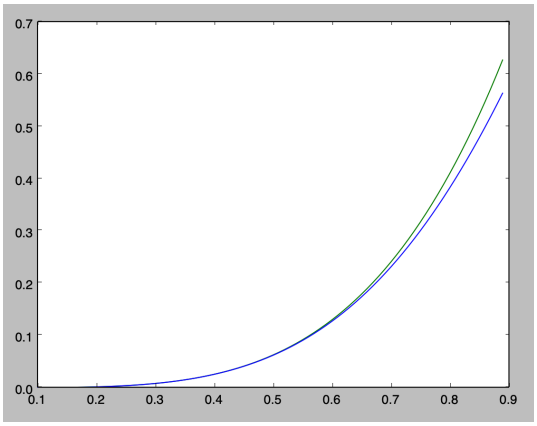
Time: 1.54880619049 seconds



تابع فوق نیز به درستی تخمین زده شده است و نکته‌ی جالب آن است که این تابع تنها در یک نسل به دست آمده – که البته این اتفاق تصادفی رخ داده است. تابع به دست آمده همان x^3 است، ولی چند جمله‌ی اضافه‌تر بی‌تاثیر نیز دارد، یعنی در اصل x^2 با $x - x$ که همان صفر است جمع شده و سپس ضربدر $\sqrt{x * x}$ شده تا x^3 به دست آید، و دلیل دقیقا بی‌نهایت نشدن fitness و صرفا بسیار بزرگ شدن آن نیز همین خطای مربوط به عملیات sqrt است؛ ولی تخمین انجام شده، از نظر ریاضی درست است.

- تابع $y(x) = x^4$:

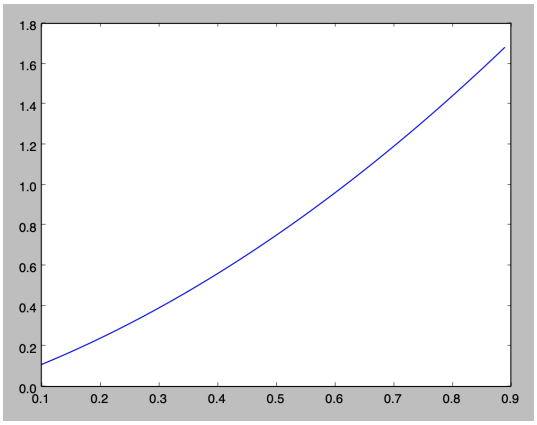
Count of generations: 1000
Count of fitness calculations: 2500
Expression: y(x) = ((x * sin((x / inv(x)))) / inv(x))
Fitness: 119.915463052
Time: 4.9568810463 seconds



تخمین نزدیکی به تابع به دست آمده، اما دقیق نیست، با این‌که می‌توانست در تئوری مقدار دقیق نیز به دست آید، اما به تصادف تابع بهتر دیگری به دست آمده است که در حدی نزدیک به واقعیت بوده که شرط خاتمه‌ای که تعریف کرده بودم را برقرار کرده است. یک مورد $x/inv(x)$ وجود دارد که می‌توانست با x^2 جای‌گزین شود.

- تابع $y(x) = x^2 + x$:

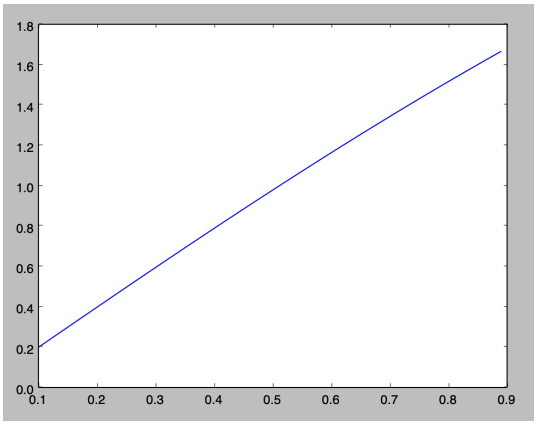
Count of generations: 33
Count of fitness calculations: 1534
Expression: y(x) = ((x * x) + x)
Fitness: infinity
Time: 1.60835790634 seconds



تابع فوق نیز کاملا به درستی تخمین زده شده است و جمله‌ی اضافی نیز ندارد که می‌تواند به دلیل سادگی تابع باشد؛ البته هم‌چنان ۳۳ نسل طول کشیده تا به این تابع برسیم. شایستگی نیز بی‌نهایت است، زیرا تابع دقیقا منطبق بر ورودی‌های داده شده است.

- تابع $y(x) = \sin(x) + x$:

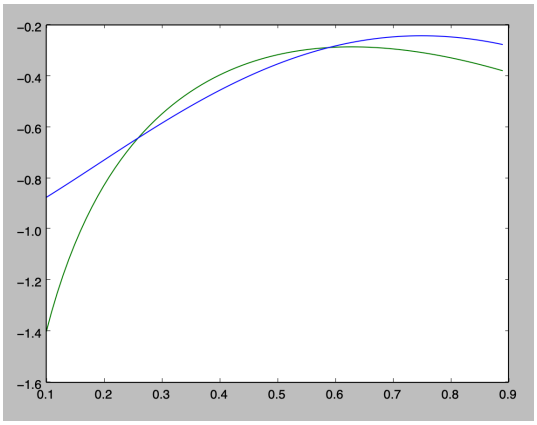
Count of generations: 506
Count of fitness calculations: 2007
Expression: y(x) = (((x + x) + sin(x)) - sqrt((x * x)))
Fitness: 1.69547280089e+16
Time: 2.22720813751 seconds



تـابـع فـوق نیز بـه درسـتی تخـمین زده شـده اسـت؛ در اصـل $x + x$ جـمله‌ی $2x$ را سـاخـته و عـبارت بـرابـر $2x + \sin(x) - \sqrt{x^2} = 2x + \sin(x) - x = x + sin(x)$ (در دامنه‌ی داده شده) است که دقیقا مطابق تابع داده شده است و دلیل بی‌نهایت نشدن fitness و صرفا خیلی بزرگ شدن آن نیز وجود خطای محاسبات در sqrt است.

- تابع $y(x) = \log(x) + \cos(x) - x$:

Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = ((sin(x) / (x ^ x)) - ((x / sin(x)) ^ sqrt(x)))
Fitness: 12.5336119871
Time: 4.53057217598 seconds

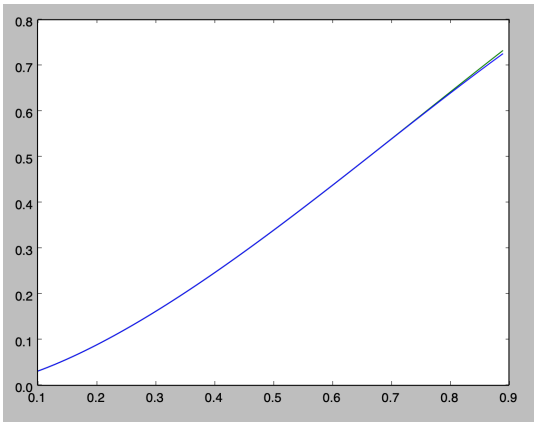


تابع فوق با خطا تخمین زده شده، ولی شکل کلی تابع تخمین زده شده، بالاخص در نقاط غیر نزدیک به صفر (که در آن‌ها تابع لگاریتم شتاب می‌یابد و وجود خطا تا حدی طبیعی‌تر است) تقریبا شکل کلی تابع اصلی را رعایت کرده است. البته،

ممکن است به دلیل وجود عنصر تصادف در الگوریتم، در دفعات دیگر اجرا، جواب‌های بدتر و یا گاه بسیار بهتر از برنامه دریافت کنیم که به تابع اصلی از این خروجی نزدیک‌تر باشد.

- تابع $y(x) = \sqrt{x}\,\sin(x)$:

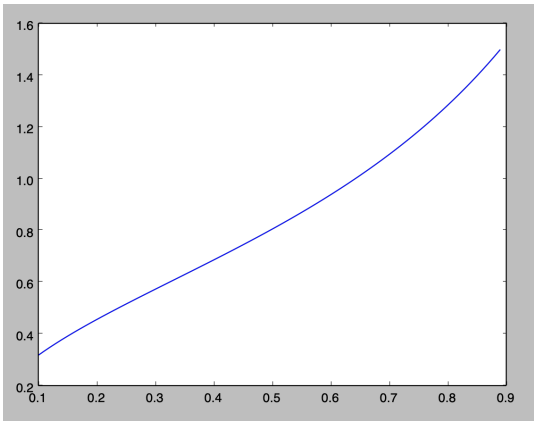
Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = sqrt((sin((sin(sin(x)) * (x * x))) * (x / x)))
Fitness: 1275.21592169
Time: 4.09384202957 seconds



تابع فوق با خطا تخمین زده شده، ولی خطا بسیار کم است. البته، در یکی از دفعاتی که برنامه را با این ورودی اجرا کردم، جواب دقیقا درست را گرفتم (که صرفا حاوی جملات اضافی با مقدار صفر بوده و در اصل برابر خود تابع اصلی بود) اما متاسفانه به اشتباه VS Code را بستم و دفعه‌ی بعدی که اجرا کردم، به دلیل وجود عنصر تصادف، به این تخمین رسیدم که البته باز هم از نظر مقداری تخمین دقیقی است، اما از نظر ضابطه‌ای، نادقیق‌تر است.

- تابع $y(x) = \dfrac{\sqrt{x}}{\cos(x)}$:

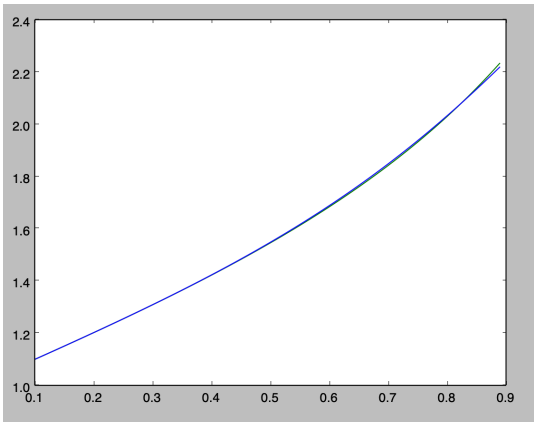
Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = ((sqrt(x) / cos(x)) * ((x / x) ^ log(x)))
Fitness: infinity
Time: 3.26536488533 seconds



تابع فوق به درستی تخمین زده شده، اما یک ضرب اضافی در $(\frac{x}{x})^{\log x}$ دارد که به وضوح برابر یک است و در نتیجه، بی‌تاثیر است. شایستگی آن نیز بی‌نهایت شده است، اما برنامه – احتمالا به دلیل پیچیدگی بیهوده‌ی term دوم – به سختی و در نسل انتهایی به این تابع رسیده است.

- تابع $y(x) = \tan(x) + 1$:

```
Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = (( sin(x)  +  addwithone(x) ) - ((( cos(x)  ^ x) ^ x) *
sin( sin( sin(x) ) ) ))
Fitness: 365.59159717
Time: 6.21893119812 seconds
```

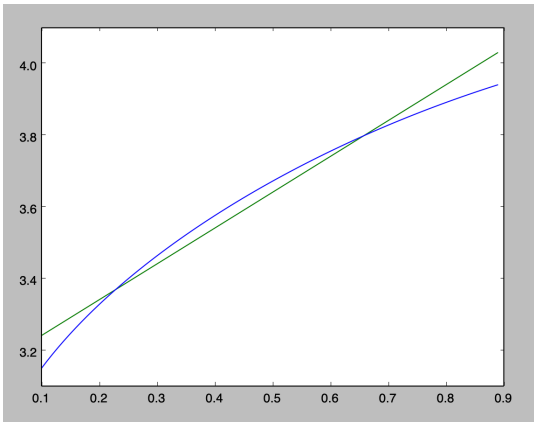


ویژگی خاص تابع فوق آن است که tan در عملگرهای پشتیبانی شده توسط برنامه نیست، از این رو، برنامه سعی کرده آن را به کمک سایر عملگرها تخمین بزند و اتفاقا تقریب خوبی از نظر داده‌ای پیدا کرده است (البته، ممکن بود در اجرای

تصادفی دیگری، مستقیما به تعریف تانژانت یعنی حاصل تقسیم سینوس بر کسینوس برسیم که خطای آن از این تابع نیز کم‌تر می‌بود). چون fitness این تابع بسیار بالا نیست، برنامه تا نسل آخر منتظر مانده و سپس وقتی از رسیدن به تابعی بسیار بسیار دقیق ناامید شد، میان جمعیت کنونی، بهترین تابع را انتخاب کرد.

- تابع $y(x) = x + \pi$:

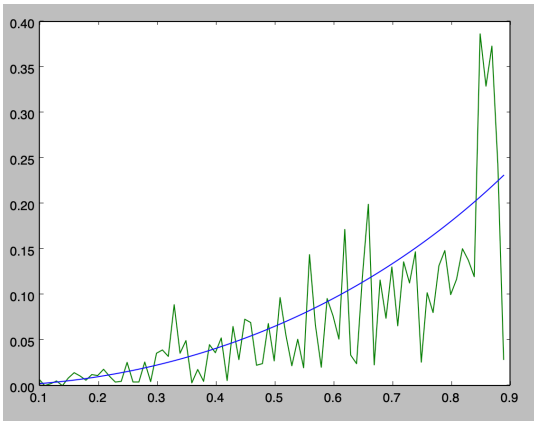
Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = exp(sqrt(addwithone(sqrt(sin(x)))))
Fitness: 31.3900113451
Time: 5.56650495529 seconds



عدد پی (و به طور کلی این چنین اعداد اعشاری) نیز از مواردی هستند که به عمد در نمونه‌ها آمده اند تا پاسخ کد که مستقیما آن‌ها را پشتیبانی نمی‌کند به آن معلوم شود. یک راه ساختن پی، ساختن آن به کمک تقریبی از جمع و تفریق و ضرب و تقسیم‌های صحیح بود و راه دیگر، یافتن تابعی کاملا متفاوت بود که تقریب خوبی از تابع اصلی باشد که برنامه در اجرای فوق، راه دوم را برگزیده است. طبیعتا fitness حاصل از این روش، چندان بالا نیست. البته، می‌توانستیم یک عملگر addwithpi تعریف کرده و به این روش، پشتیبانی از عدد پی را به کد اضافه کنیم، ولی برای نشان دادن عملکرد کد در صورت عدم تعریف یک ثابت، تصمیم گرفتم که کد را بدون وجود این عملگر – که تعریف آن دقیقا مشابه addwithone ولی با یک عدد ثابت متفاوت است – بررسی کنم. چون fitness این تابع بسیار بالا نیست، برنامه تا نسل آخر منتظر مانده و سپس وقتی از رسیدن به تابعی بسیار بسیار دقیق ناامید شد، میان جمعیت کنونی، بهترین تابع را انتخاب کرد.

- تابع خطخطی شده:

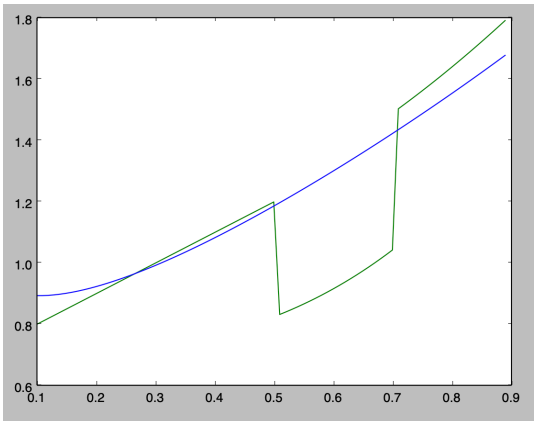
Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = log(inv(sqrt(cos(x))))
Fitness: 27.4840289402
Time: 4.33811497688 seconds



این تابع خطخطی شده، ضابطه‌ی مشخصی نداشته و به همین دلیل، از کد – که یک ضابطه‌ی یکسان برای کل نمودار مشخص می‌کند – انتظار نمی‌رود که بتواند ضابطه‌ای برای این تابع بیابد، در نتیجه کد سعی کرده منحنی‌ای بیابد که تا حد امکان با سیر کلی نمودار خطخطی شده – که افزایشی است – هم‌خوانی داشته باشد که در این کار موفق شده و البته طبیعتا موفق به fit شدن روی تمام نقاط نشده است. چون fitness این تابع بسیار بالا نیست، برنامه تا نسل آخر منتظر مانده و سپس وقتی از رسیدن به تابعی بسیار بسیار دقیق ناامید شد، میان جمعیت کنونی، بهترین تابع را انتخاب کرد.

- تابع غیر پیوسته (یا همان «دارای گسستگی»، که به ازای x های کم‌تر از ۰.۵ برابر $x + 0.7$ برای x های بین ۰.۵ (با حساب خود ۰.۵) و ۰.۷ برابر $x^3 + 0.7$ و برای x های برابر ۰.۷ یا بالاتر برابر $x^2 + 1$ است که به وضوح ناپیوستگی دارد):

Count of generations: 1500
Count of fitness calculations: 3000
Expression: y(x) = (inv((x / x)) * ((x ^ x) + sin(x)))
Fitness: 7.79510643179
Time: 5.16050601006 seconds



کد نوشته شده، یک ضابطه‌ی یکسان برای کل نمودار مشخص می‌کند و در نتیجه از آن انتظار نمی‌رود که بتواند ضابطه‌ی این تابع را بیابد بیابد، در نتیجه کد سعی کرده منحنی‌ای بیابد که تا حد امکان با سیر کلی نمودار ناپیوسته – که افزایشی است – هم‌خوانی داشته باشد که در این کار موفق شده و البته طبیعتا موفق به fit شدن روی تمام نقاط نشده است (به طور ویژه، ضابطه‌ی میانی را تقریبا کامل از دست داده، ولی تقریب خوبی روی بازه‌های طولانی‌تر ابتدا و انتها دارد). چون fitness این تابع بسیار بالا نیست، برنامه تا نسل آخر منتظر مانده و سپس وقتی از رسیدن به تابعی بسیار بسیار دقیق ناامید شد، میان جمعیت کنونی، بهترین تابع را انتخاب کرد.

جمع‌بندی و نتایج عملی

همان طور که مشاهده شد، برنامه‌نویسی ژنتیک در بسیاری از مثال‌های فوق و تقریبا اکثر مثال‌هایی که عملگرهای مربوط به آن‌ها در لیست عملگرهای برنامه موجود بود، جواب بسیار خوب و در تعداد زیادی از موارد، جواب کاملا درست را به عنوان خروجی در کم‌تر از ۱۰ ثانیه با متغیرها و پارامترهای تنظیم شده برای مسئله داده است که بسیار قابل قبول محسوب می‌شود و در نتیجه می‌توان برنامه‌نویسی ژنتیک را روشی کارا برای تقریب توابع دانست.

برای کاراتر کردن روش، می‌توان عملگرهای بیش‌تری (مانند tan یا tanh یا...) را نیز افزود که البته برخی از آن‌ها به عمد و برای تست ورودی‌هایی که عملگرهای آن‌ها در لیست عملگرهای برنامه نیست، به این کد افزوده نشده‌اند. به طور کلی، می‌توان نتیجه گرفت که در اکثر حالات، در صورتی که به تصادف به جمعیتی نرسیم که همه‌ی اعضای آن دور از تابع اصلی باشند، به جوابی بسیار خوب رسیده و حتی اگر به جمعیتی نامطلوب برسیم، احتمال اصلاح آن به کمک جهش‌ها کاملا فراهم است.

چند نقطه ضعف اصلی حل مسئله‌ی تقریب توابع با الگوریتم ژنتیکی که پیاده‌سازی شده است، موارد زیر هستند:

- در بسیاری از مواقع، ضابطه‌ی تابع به درستی پیدا می‌شود، اما شامل term هایی بیهوده است که مقادیر ۱ را در ضرب و تقسیم و یا مقادیر ۰ را در جمع و تفریق دارند که عملا تاثیری بر جواب نگذاشته، اما ضابطه‌ی خروجی توسط برنامه را

کمی پیچیده می‌کنند. برای حل این مشکل می‌توان الگوریتمی نوشت که یک ضابطه را گرفته، جملات بیهوده‌ی آن که همواره مقدار ثابتی دارند را مشخص کرده و آن مقدار ثابت را به جای آن‌ها جای‌گزین کند که البته این الگوریتم از حوزه‌ی برنامه‌نویسی ژنتیک خارج است و در واقع برنامه‌ی دیگری است که به عنوان پوشش روی برنامه‌ی نوشته شده قرار گرفته و به جای جملات بیهوده‌ی خروجی این برنامه، مقدار ثابت آن‌ها را جای‌گزین کرده و حاصل را به عنوان خروجی کل، اعلام می‌کند. الگوریتم‌های ساده‌سازی بسیاری برای عبارات ریاضی وجود دارد که می‌توان از آن‌ها به این منظور کمک گرفت.

- در بسیاری از مواقع – بالاخص مواردی که ضابطه‌ی دقیق پیدا نمی‌شود – ضابطه‌ای بسیار طولانی و گاه چند خطی برای تابع پیدا می‌شود؛ البته، هنگام آزمایش برای نوشتن گزارش این اتفاق به شکل شدید رخ نداد، اما در حین توسعه‌ی برنامه به موارد این‌چنینی برخوردم که یکی از راه‌های آن می‌تواند گذاشتن یک جریمه روی عبارات طولانی باشد و یا راه دیگر این باشد که در صورت برخورد به یک عبارت بسیار طولانی، دوباره الگوریتم اجرا شود و به عبارتی که حتی ممکن است fitness آن کمی کم‌تر باشد، ولی خود عبارت بسیار ساده‌تر است، اکتفا کرد که بستگی به شرایط استفاده دارد.
- اگر دامنه‌ی مورد بررسی از بازه‌ی میان صفر و یک فراتر رفته یا ضابطه‌ی یک تابع پس از یک یا قبل از صفر تفاوت کند، برنامه‌ی نوشته شده و بخش نمونه‌گیری، به گونه‌ای است که متوجه این تفاوت نمی‌شود و لازم است بسته به دامنه‌ی مورد بررسی از تابع، دامنه‌ی نمونه‌گیری نیز تغییر یابد. هم‌چنین ممکن است لازم باشد برای دستیابی به دقت بیش‌تر در چنین توابعی، پارامترهای کارکرد کد نیز در عمل تغییر یابد.
- کد نوشته شده، توابع تک‌متغیره (بر اساس دستور کار) و نیز عملگرهای شامل حداکثر ۲ عملوند را پشتیبانی می‌کند که می‌توان برای دستیابی به دقت بیش‌تر برای توابع بیش‌تر، به این پارامترها افزود.

در نهایت، با توجه به زمان اجرای نسبتا مناسب و نیز نتایج در کل قابل قبولی که از این روش برای تخمین توابع به دست آمد، کارایی عملی این روش، مناسب ارزیابی می‌شود و می‌تواند در مواقعی که داده‌های نمونه‌برداری شده از یک تابع را داشته و به دنبال مقدار تابع در نقطه‌ای که در آن به طور مستقیم داده‌ای نمونه‌برداری نشده نیز هستیم، ابتدا با این برنامه تخمینی برای تابع یافته و سپس مقدار جدید را در تابع یافته شده جای‌گذاری کنیم.

منابع

- research.IAUN.ac.ir
- PSU.edu
- GitHub