به نام خدا

ایمان علیپور

۹۸۱۰۲۰۲۴

پروژه ۱ هوش مصنوعی

پیاده سازی الگوریتم ژنتیک

استاد: دکتر آرش عبدی هجراندوست

مفروضات من:

- اولا فرض کردم sin و cos فقط برگ های درخت میتوانند باشند.(چرا که اگر نود میانی باشند مشکلاتی در اضافه کردن نود جدید و خالی بودن یکی از نود های راست و چپ به وجود می‌آمد)
- فرض کردم تابع تک متغیره است.(در واقع n متغیره بودن کمی مشکل ایجاد میکرد که تلاش کردم اول مسئله را ساده کنم و بعد اگر شد پیچیدگی ها را بیشتر کنم.)
- متاسفانه دیر متوجه شدم که وقتی عملگر توان چندین بار پشت سر هم می‌آید یا توان یک عدد مختلط می شود الگوریتم من دچار مشکل میشود(برای محاسبات) و بنابراین این فرض را کردم که چنین اتفاقاتی نباید بیفتد و برای بسیاری از رخداد ها مقدار بینهایت برگرداندم.

مشکلات راه:

- بسیاری از وقت ها والدین تکراری برای تولید نسل بعدی انتخاب می شوند که من این شرط را گذاشتم که این اتفاق نیفتد اما بخش انتخاب والدین بسیار بسیار زمان‌بر شد.
- داشتن چندین متغیر مسئله را سخت میکند و من ایده ای برای حل این مشکل نداشتم.
- انتخاب والدین به صورتی که مورد های تکراری زیاد نباشند بسیار زمان‌بر بود و در بسیاری از موارد در این مرحله کد من زمان بسیاری میبرد و هر فکری کردم زمان را نتوانستم بهبود ببخشم.
- در بسیاری از مواقع بخاطر تولید تصادفی جمعیت اولیه الگوریتم به زمان زیاد و iteration های زیاد نیاز دارد تا بتواند تابعی نزدیک به تابع خوب را پیدا کند، برای همین من یک سقف برای تعداد iteration ها گذاشتم که بتوانم گزارشم را بنویسم.
- در مواقعی که بوسیله زاد و ولد یا crossover و والدین، فرزند خوبی تولید نشود و با جهش هم فایده‌ای حاصل نشود، الگوریتم جواب بهینه را پیدا نمیکند اما اگر زمان کافی بدهیم ممکن است این اتفاق بیفتد.
- یکی از اشتباهات من این بود که بجای اینکه والدین بهینه را نگه دارم، آن را احتمالاتی کردم تا بقیه والدین هم امکان ماندن را داشته باشند اما حس میکنم شاید یکی از دلایل نتایج نه چندان خوب همین بوده.
- در کل زمان اجرای این پیاده سازی بسیار زیاد است و به کمی hyper parameter tuning نیاز دارد که با اینکه من آزمایش های بسیار زیادی کردم، نتوانستم حالت بهینه‌ای بیابم.
- یکی دیگر از مشکلا دخیل بودن زیاد شانس برای الگوریتم برای یافتن جواب است که شاید اگر کمی ذکاوت به خرج میدادم نتایج بهتری حاصل میشد، اما تعداد پارامتر های مجهول و متغیر مسئله به حدی زیاد بود که مجبور شدم به نتیجه فعلی بسنده کنم.

پیاده سازی:

نگاشت مسئله به درخت:

برای این کار فرض کردم هر عبارت به صورت یک درخت ذخیره شده است، حال این درخت را با استفاده از تابعی مقدارش را در نقطه x محاسبه میتوان کرد، همچنین پیمایش درخت به صورت inorder به ما عبارت ریاضی را میدهد.

حال کافی است تعدادی درخت اولیه ایجاد کنیم و هر بار آنها را با هم ترکیب کنیم و بعضی وقت ها یک جهش در یکی از فرزندان ایجاد کنیم.

```
45    def inorder(node):
46        if node:
47            print('(', end = '')
48            inorder(node.left)
49            print(node.data, end = '')
50            inorder(node.right)
51            print(')', end = '')
```

```
34    class Node:
35        def __init__(self, data):
36            self.left = None
37            self.right = None
38            self.data = data
39
```

```python
def generate_random_leaf_node():
    probability = generate_random_number(1, 100)
    if probability <= 70:
        return generate_random_number(0, 10)
    elif probability <= 90:
        return 'x'
    elif probability <= 95:
        return 'sin(x)'
    else:
        return 'cos(x)'
```

```python
def generate_random_inner_node():
    probability = generate_random_number(1, 100)
    if probability <= 20:
        return '+'
    elif probability <= 40:
        return '-'
    elif probability <= 60:
        return '*'
    elif probability <= 80:
        return '/'
    else:
        return '^'

```

ساخت جمعیت اولیه:

برای این کار، از یک تابع استفاده کردم که ابتدا بصورت رندوم ارتفاع درخت را بدست می‌آورد، سپس با اساین کردن احتمال برابر برای اپراتور های مختلف، داده نودهای میانی را یک اپراتور، و داده نود های برگ را یک عدد ثابت، x یا sin یا cos قرار دادم که بیشترین احتمال برای عدد ثابت رندوم است و پس از آن x و احتمال اضافه شدن sin و cos را هم ۵ درصد قرار دادم.

حال به تعداد دلخواه با این تابع می توان جمعیت دلخواه ساخت و در ادامه هم همین کار را کردم، توجه کنید شاید ساخت هوشمندانه تر جمعیت اولیه باعث بهبود عملکرد الگوریتم میشد که حس میکنم شاید من تا انتها به آن زیاد فکر نکردم که میتوان این جمعیت اولیه را هوشمندانه تر تولید کرد، مثلا ترکیبات توان های مختلف x و ضرب ضرایب مختلف در sin و cos و ...

```python
def create_initial_generation(size_of_generation, max_tree_length = 5):
    initial_generation = []

    for i in range(size_of_generation):
        length_of_tree = generate_random_number(1, max_tree_length)
        tree_leaves = []
        root = Node(generate_random_inner_node())
        tree_leaves.append(root)
        initial_generation.append(root)

        for _ in range(length_of_tree - 1):
            tmp_tree_leaves_copy = tree_leaves
            for i in range(len(tmp_tree_leaves_copy)):
                left_child = Node(generate_random_inner_node())
                right_child = Node(generate_random_inner_node())
                parent = tree_leaves.pop(0)
                parent.right = right_child
                parent.left = left_child
                tree_leaves.append(parent.right)
                tree_leaves.append(parent.left)

        tmp_tree_leaves_copy = tree_leaves
        for _ in range(len(tree_leaves)):
            left_child = Node(generate_random_leaf_node())
            right_child = Node(generate_random_leaf_node())
            parent = tree_leaves.pop(0)
            parent.right = right_child
            parent.left = left_child
    return initial_generation
```

تابع شایستگی:

برای تابع شایستگی این کار را کردم که در هر نقطه x مقدار درخت را حساب میکنم و قدر مطلق فاصله را از مقدار اصلی می‌گیرم و جمع میکنم، حال مقدار fitness را برابر با

1000/distance

قرار میدهم، دلیل آن هم این است که اگر این مقدار برابر 1000 باشد یعنی فاصله بسیار کم است و هرچه بیشتر باشد بهتر است،هر چقدر هم فاصله بیشتر باید مقدار شایستگی کمتر میشود.

```python
def fittness_score(generation, x_values, expectation):
    fittness_values = []

    for i in range(len(generation)):
        difference = 0
        for j in x_values:
            try:
                difference += abs(evaluate_expression_tree(generation[i], j))
            except:
                difference = float('inf')
        try:
            distance = abs(expectation[i] - difference)
        except:
            distance = float('inf')
        try:
            if(not math.isnan(distance)):
                fittness_values.append(1000/distance)
        except:
            fittness_values.append(0)
    return fittness_values

```

انتخاب والدین:

برای انتخاب والدین هر بار دو عدد رندوم انتخاب میکنم و چک میکنم کدامین درخت ها شایستگی نزدیک تری به آن دارند و اینگونه دوتایی هایی انتخاب می شوند و اگر دوتایی انتخاب شده جدید نباشد، دوباره گردش انجام میشود تا والدین جدیدی انتخاب شوند. کدی که در اینجا زدم خیلی جالب نبود، چالش های زیادی داشتم و بارها به باگ برخورد کردم و دلیل آن هم این بود که میخواستم بر اساس شایستگی بیشتر نود ها را انتخاب کنم و اینکه عدم انتخاب جفت والدین تکراری زمان این بخش از کد را بسیار بسیار زیاد کرد.

قطعا کمی refactoring کد این بخش به بهبود سرعت الگوریتم کمک خواهد کرد که من بخاطر باگ های پیاپی زیاد موفق به بهبود آن نشدم.

```python
def parent_selector(generation, fittness_values, next_generation_size = 100): # This ugly code tr
    next_generation_parents = []
    next_gen_size = next_generation_size//2
    for i in range(next_gen_size):
        chosen_fittness = random_number_between_0_and_1()*1000
        parent_1_index = return_closest_item_in_list(fittness_values, chosen_fittness)
        parent_1 = generation[parent_1_index]
        chosen_fittness = random_number_between_0_and_1()*1000
        parent_2_index = return_closest_item_in_list(fittness_values, chosen_fittness)
        while parent_2_index == generation[parent_1_index]:
            chosen_fittness = random_number_between_0_and_1()*1000
            parent_2_index = return_closest_item_in_list(fittness_values, chosen_fittness)
        parent_2 = generation[parent_2_index]
        while (parent_1, parent_2) in next_generation_parents:
            chosen_fittness = random_number_between_0_and_1()*1000
            parent_1_index = return_closest_item_in_list(fittness_values, chosen_fittness)
            parent_1 = generation[parent_1_index]
            chosen_fittness = random_number_between_0_and_1()*1000
            parent_2_index = return_closest_item_in_list(fittness_values, chosen_fittness)
            while parent_2_index == generation[parent_1_index]:
                chosen_fittness = random_number_between_0_and_1()*1000
                parent_2_index = return_closest_item_in_list(fittness_values, chosen_fittness)
            parent_2 = generation[parent_2_index]
        next_generation_parents.append((parent_1, parent_2))
    return next_generation_parents
```

تولید نسل بعد:

برای تولید نسل بعد، با توجه به مقداری که ورودی داده میشود، عددی رندوم تولید می شود تا ببینیم ایا والدین را نگه داریم یا فرزندان جدید را جایگزین کنیم(در واقع زمانی که این کار را کردم هدفم انجام آزمایش بود تا ببینم آیا با زیاد نکردن تاثیر درخت های خوب آیا نتیجه بهبود می‌یابد یا خیر که بنظر جواب بله است اما با افزایش توان محاسباتی بسیار بسیار زیاد.)

حال اگر بر اساس عدد رندوم تولید شده تصمیم به ساخت نود های فرزند گرفته شود، دو فرزند جدید طی crossover میشوند و با تولید یک عدد تصادفی دیگر با احتمال ۵ درصد جهش می‌یابند.

برای جهش به احتمال ۵۰ درصد(با توجه به عدد تصادفی تولید شده) این کار را میکنیم:

یک مقدار رندوم بین ۱ تا ۵ برای عمق جهش انتخاب می‌شود و به صورت رندوم با احتمال برابر به راست یا چپ می‌رویم، حال اگر در نود های میانی باشیم operand را عوض میکنیم(باز هم بصورت رندوم) و اگر در برگ ها باشیم یا یک مقدار ثابت رندوم قرار میدهیم یا x یا sin و cos.

همچنین با احتمال ۲۵ درصد جهش بزرگ انجام میدهیم که این یا موجب از بین رفتن تمام فرزندان و کات شدن این نود میشود و یا شامل تغییر کلی و در مواردی extend شدن آن.

همچنین برای crossover فقط فرزند راست و چپ دو والد را جابجا میکنم و بیشتر اتفاقات مهم دیگر و حالت های دیگر را ب رخ دادن جهش واگذار میکنم.

یکی از کارهایی که انجامش خوب بود این بود که بهترین والدین را نیز نگه داریم، در ابتدا این کار را کرده بودم، اما چون تعداد فرزندان تولید شده کم میشد آنرا تغییر دادم و یک دلیل دیگر هم این بود که صفات والدین را فرزندانشان دارند و نگه داشتن دوباره آنها امکان ایجاد جهش و تولید چیزهایی که در رابطه درست وجود دارند را کم می کرد، اما این نیز باز یک آزمایش بود و من بخاطر توان محاسباتی کم، با وجود آزمایش های مکرر موفق به گرفتن نتیجه گیری خوبی نشدم.

```
207  def make_next_generation(selected_parents, new_generation_rate = 0.7):
208      new_generation = []
209      for i in selected_parents:
210          parent_1 = i[0]
211          parent_2 = i[1]
212          probability = random_number_between_0_and_1()
213          if probability <= new_generation_rate:
214              child_1, child_2 = cross_over(parent_1, parent_2)
215              mutation_probability = random_number_between_0_and_1()
216              if mutation_probability > 5:
217                  new_generation.append(child_1)
218              else:
219                  mutated_child_1 = mutation(child_1)
220                  new_generation.append(mutated_child_1)
221              mutation_probability = random_number_between_0_and_1()
222              if mutation_probability > 5:
223                  new_generation.append(child_2)
224              else:
225                  mutated_child_2 = mutation(child_2)
226                  new_generation.append(mutated_child_2)
227          else:
228              new_generation.append(parent_1)
229              new_generation.append(parent_2)
230      return new_generation
```

```python
def cross_over(parent_1, parent_2):
    probability = generate_random_number(1, 100) # Better crossover?
    copy_of_parent_1 = copy.deepcopy(parent_1)
    copy_of_parent_2 = copy.deepcopy(parent_2)
    try:
        right_node_of_parent_1 = parent_1.right
        copy_of_parent_1.right = parent_2.right
        copy_of_parent_2.right = right_node_of_parent_1
    except:
        copy_of_parent_1 = copy_of_parent_1
        copy_of_parent_2 = copy_of_parent_2
    return copy_of_parent_1, copy_of_parent_2

def mutation(child_node):
    probability = generate_random_number(1, 100)    # Better mutation?
    node = child_node
    if probability <= 25:   # Cut the node
        node = Node(generate_random_leaf_node())
    elif probability <= 50: # Extend the node
        new_node = create_initial_generation(1, 4)
        node = new_node[0]
    else:   # Change nodes data
        how_far_to_go = generate_random_number(1, 5)
        while node.right != None and how_far_to_go != 0:
            if random_number_between_0_and_1() < 0.5:
                node = node.right
            elif node.left != None:
                node = node.left
        if node.right == None:
            node.data = generate_random_leaf_node()
        else:
            node.data = generate_random_inner_node()
    return node
```

شرط های خاتمه الگوریتم:

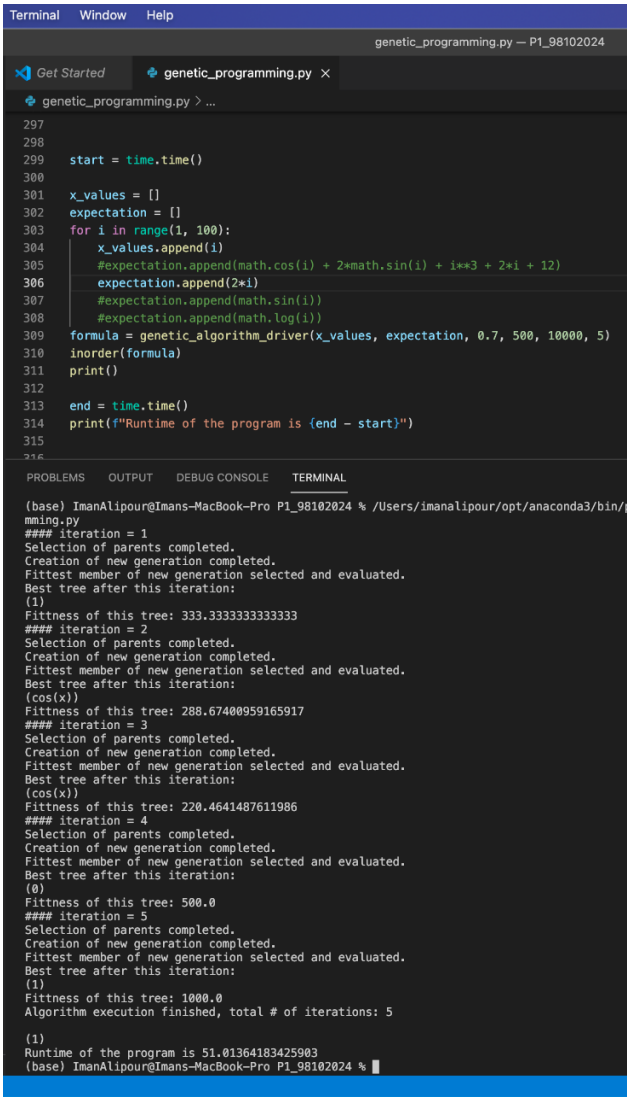
در ابتدا تنها شرطی که گذاشته بودم این بود که شایستگی به بینهایت میل کند این یعنی تابع خروجی الگوریتم رفتاری بسیار شبیه به نقاط داده شده و تابع احتمالی داده شده داشته باشد، اما متاسفانه هیچ وقت با این استراتژی الگوریتم من کامل نشد و خروجی‌ای نداشتم، برای همین در روز آخر سقف تعداد iteration را به کدم اضافه کردم تا حداقل نتایجی تولید و گزارش کنم و آزمایش هایم را انجام بدهم و

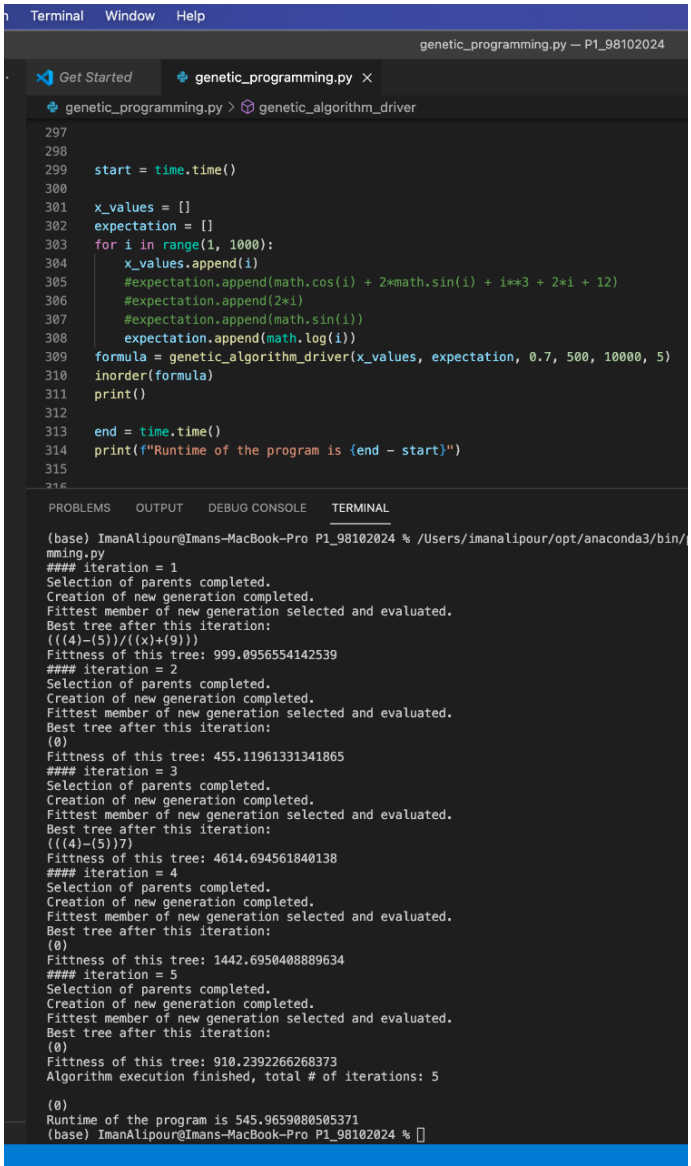
نمیدانم آیا این کار درستی بود یا نه. اما در بعضی از مواقع میتوانستم نتیجه ای بگیرم که بعضا توجیه پذیر باشد.

آزمایشات:

آزمایش اول:

در اولین آزمایش، تابع 2x را در بازه ۱ تا ۱۰۰ به الگوریتم دادم و تعداد گردش ها را به ۵ محدود کردم تا بتوانم خروجی داشته باشم. متاسفانه خروجی برای این ورودی زیاد جالب نبود و فکر میکنم دلیل آن تولید درخت های اولیه نه چندان خوب بود که من چند بار این را آزمایش کردم اما یا تابع خروجی بسیار طولانی و عجیب بود یا مثل این خروجی یک عدد ثابت که البته شایستگی این تابع ثابت بالاست و این نشان از این است که بقیه توابع اولیه ساخته شده طی جهش ضابطه خوبی تولید نکردند و باید تعداد گردش ها بیشتر باشد تا نتیجه بگیریم.



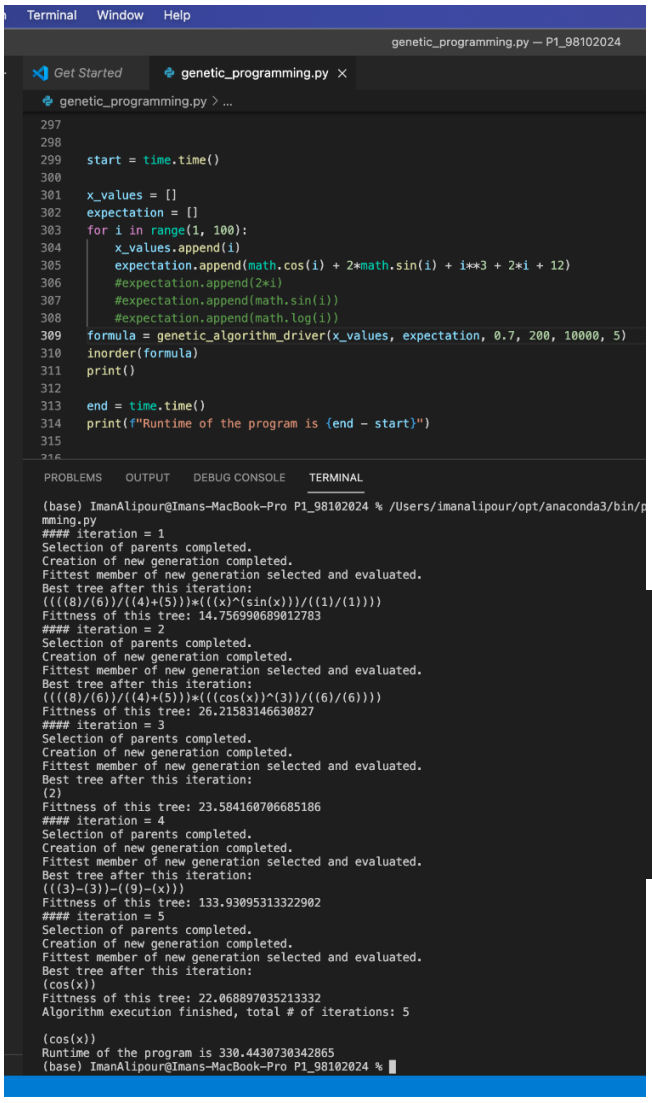


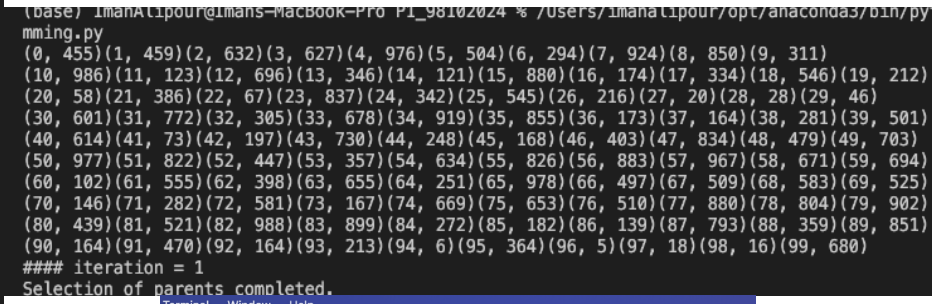
آزمایش دوم:

در این آزمایش تابع logx را در بازه ۱ تا ۱۰۰ به الگوریتم دادم و خروجی آن مقدار ۰ شد که باز هم بخاطر تولید درخت های اولیه نه چندان خوب بود، اما این نتیجه بسیار گویاست چرا که مقدار logx در این بازه به ۰ بسیار نزدیک است.

آزمایش سوم:

در این آزمایش تابع نسبتا پیچیده ای که ترکیب یک سینوس و یک کسینوس و x و x^2 بود را به الگوریتم دادم که خروجی آن cosx شد اما روند تغییر تابع خروجی در این ورودی جالب است که ابتدا یک تابع پیچیده است و کم کم به cos تبدیل میشود و شایستگی آن زیاد میشود اما باز هم شایستگی مقدار کمی دارد و این نتیجه بخاطر تعداد گردش های کم بوده، زمان اجرای الگوریتم روی این ورودی بسیار زیاد بود.(۵ دقیقه، در شکل هم آمده)

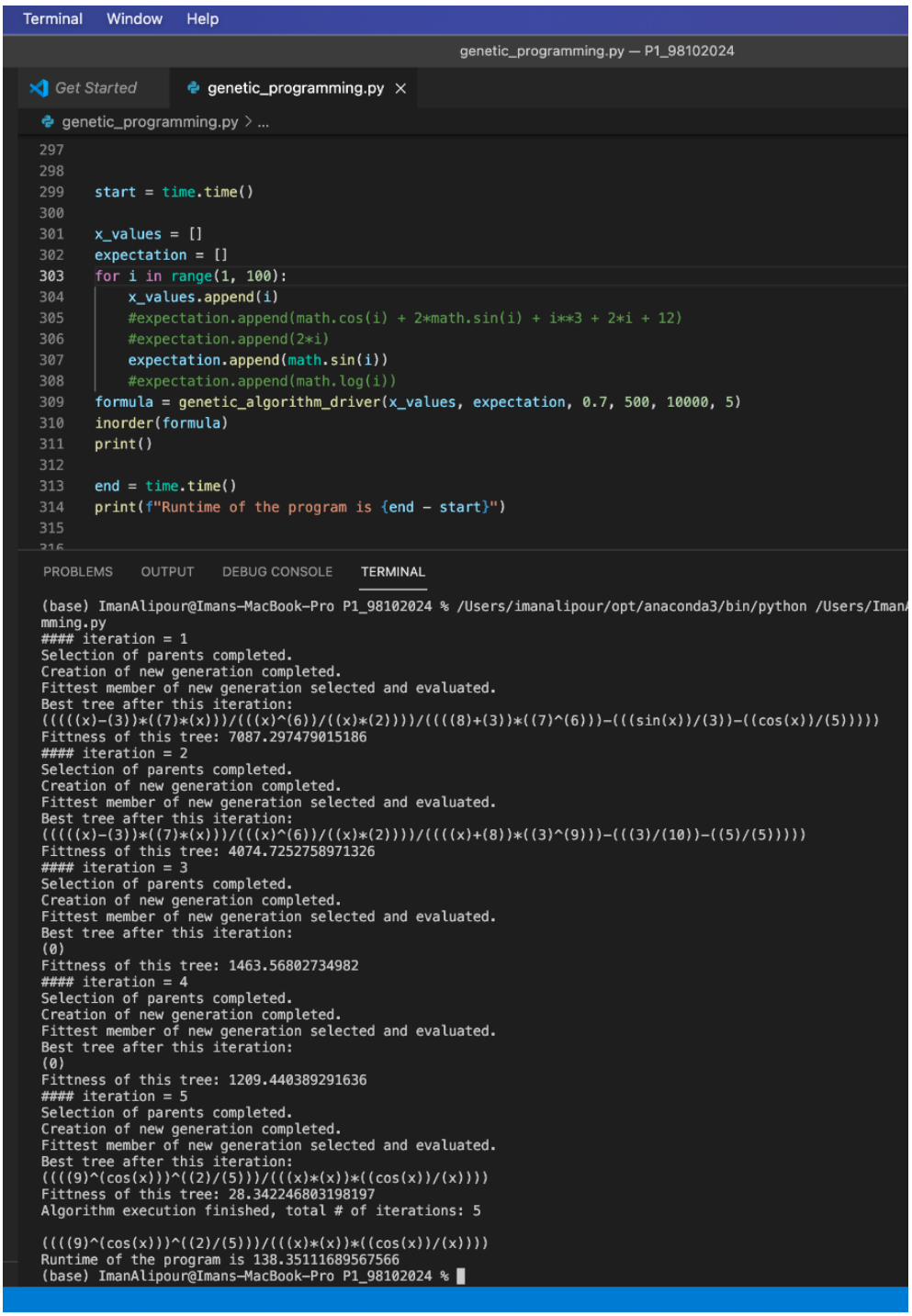




آزمایش چهارم:

تابع خط خطی را این بار به عنوان ورودی به الگوریتم دادم که مقدار آن را در بالا سمت راست میبینید. باز هم خروجی مقداری ثابت است و دلیل آن این است که تلورانس شایستگی بقیه توابع را بسیار کم کرده است، یک نکته جالب انتخاب sinx در یکی از گردش ها است که بسیار جالب بنظر میرسد، یعنی الگوریتم این نوسانی بودن ورودی را فهمیده اما بعدا متوجه شده شاید یک تابع ثابت از یک تابع نوسانی بهتر باشد که شایستگی بیشتر گواه این نکته است.

آزمایش پنجم:
در این آزمایش من ورودی الگوریتم را sinx دادم و آن را اجرا کردم و محدودیت ها را نیز کمتر کردم تا کمی نتیجه با قبلی ها متفاوت تر باشد، در خروجی های قبلی تابع خروجی کوچک بود و خیلی اشتباه انداز، همانطور که میبینید برای این ورودی تابع خروجی پیچیده تر است و شکل آن بنظر می‌رسد با sinx مشابهت هایی داشته باشد.



جمع بندی و نتایج

اولا میخواهم بگویم بنظرم دلیل اصلی نتایج نه چندان جالب بخاطر این بود که کاملا خودم کد را پیاده سازی کردم، و خب بسیاری از جاها پیاده سازی من بهینه نبوده است و بعضی جا ها کارهایی کرده ام که شاید منطقی نبوده باشند و به همین دلیل که کاملا خودم کد را نوشتم نتایج زیاد جالب نبودند.

مجددا با اینکه به شخصه از نتایج پیاده سازی خودم خرسند نیستم، اما حس میکنم بیشتر دلیل این نتایج نه چندان مطلوب بخاطر عوامل تصادفی زیاد در پیاده سازی من بود، دلیل دیگر نیز طول کشیدن اجرا شدن شدن الگوریتم بود. زمانی که کمی محدودیت ها را کمتر میکردم، بعضا به جواب نمی رسیدم، اما فکر میکنم با کم شدن محدودیت ها این کد من هم نتایج خوبی داشته باشد.

در کل نتیجه گیری من از الگوریتم های ژنتیک، نیاز به میزان محاسبه و سعی و خطای بسیار بالاست که حس میکنم با واقعیت هم سازگار است چرا که موجودات در طی سالیان بسیار طولانی این تغییرات ژنتیکی در آنها رخ داده است و نمونه های بسیار بسیار زیادی هم از هر کدام بوجود آمده و از بین رفته است، یکی از مسائلی که در این پیاده سازی به آنها دقت نشد این بود که در طبیعت اعضای ضعیف زنده نمی‌مانند اما در پیاده سازی من به آنها این امکان داده می شود، حس میکنم باید احتمال اجازه به آن اعضا را بسیار بسیار بسیار کمتر کنم، مسئله دیگر هم محدودیت محاسبات بود که به نظرم برای پیاده سازی من مشکل ایجاد کرد.

به نظر خودم نتیجه گیری من با واقعیت هم سازگار است و بنظر می آید برای نتیجه گرفتن از این الگوریتم ها زمان و محاسبات زیادی نیاز است و همچنین هوشمندانگی بیشتر، قطعا با تغییر در پیاده سازی بعضی از قسمت های کد من، نتایج بسیار بهتر خواهند شد.

در کل الگوریتم های ژنتیکی بخاطر سادگی نسبی بنظر می‌آید بتوانند سوالات را حل کنند یا برای آنها جواب تقریبی خوبی بیابند اما این کار به زمان و توان محاسباتی بالایی نیاز دارد.