سنا ساری نوایی

شماره دانشجویی : 8110199435

هوش مصنوعی

پروژه دوم(ژنتیک)

هدف پروژه : آشنایی با الگوریتم ژنتیک در هوش مصنوعی از طریق حل مساله EP

توضیح کلی : در این پروژه تعدادی عملگر و عملوند داده می‌شود و با استفاده از الگوریتم ژنتیک باید پاسخ درست مساله را به‌دست بیاوریم به طوری‌که طرفین معادله با یکدیگر برابر باشند.

بخش یک: مشخص کردن مفاهیم اولیه

ژن: هر یک از اعدادی که به عنوان ورودی اولیه دریافت می‌کنیم و می‌خواهیم با استفاده از آنان خروجی مورد نظر را به دست آوریم، به عنوان ژن در اینجا استفاده می‌شوند.

کروموزوم: کروموزوم در واقع همان جواب مساله می‌باشد. به عنوان مثال ترتیبی که ورودی ها را در نظر می‌گیریم تا با خروجی مورد نظر برابر شود، یک کروموزوم محسوب می‌گردد. پس در واقع اینجا هم نیز می‌توان به این نتیجه رسید که کروموزوم مجموعه ای از ژن ها می‌باشد.

در این قسمت ورودی ها دریافت می‌شود و ژن نیز تعیین می‌گردد. کد مربوطه در شکل زیر آمده است. در اینجا operators و operands ژن های مسئله محسوب میشوند.

```
def __init__(self, operators, operands, equationLength, goalNumber):
    self.operators = operators
    self.operands = operands
    self.equationLength = equationLength
    self.goalNumber = goalNumber
    self.population = self.makeFirstPopulation()
```

بخش دو: تولید جمعیت اولیه

در این قسمت جمعیت اولیه ای از کروموزوم ها به صورت تصادفی می‌سازیم. تعداد جمعیت را 100 در نظر گرفتیم که این مقدار نیز می‌تواند هر مقدار دلخواهی داشته باشد. پس یعنی 100 جواب ایجاد می‌کنیم که می‌توانند درست یا غلط باشند.

در شکل زیر به تعداد جمعیت، یک لیست به اندازه عددی که در ورودی دریافت کرده ایم، درست می‌شود. به این صورت که یکی در میان آن را با عدد و عملگر پر میکنیم. خروجی کد زیر نیز در شکل بعدی آمده است.

```
def makeFirstPopulation(self):
    population = []
    for _ in range(populationSize):
        chromosome = []
        for j in range(self.equationLength):
            if j % 2 == 0:
                chromosome.append(random.choice(self.operands))
            else:
                chromosome.append(random.choice(self.operators))
        population.append(chromosome)
    return population
```

بخش سه: پیاده سازی و مشخص کردن تابع معیار سازگاری

پیاده سازی در این قسمت به این صورت میباشد که یک تابع به اسم calaFintness داریم. در این تابع به ازای هر کروموزوم که در جمعیتمان قرار دارد، ابتدا بررسی میکنیم میزان تفاوت آن با goalNumber را بدست می‌آوریم. سپس مقدار 1/(difference + 1) را ریترن میکنیم. دلیل گذاشتن یک در مخرج آن است که اگر میزان تفاوت صفر بود، مخرج کسر صفر نشود.

```
def calcFitness(self, chromosome):
    fitness = 0
    strr = ' '.join(chromosome)
    difference = abs(eval(strr) - goalNumber)
    fitness = 1/(1 + difference)
    return fitness
```

در قسمت پایین، علاوه بر محاسبه fitness، بررسی میکنیم که اگر فیتنس برابر یک باشد، همان جواب را به عنوان جواب اصلا برگرداند و برنامه تمام میشود.

```
    # Calculate fitness
    fitnesses = []
    total_fitness = 0
    for i in range(populationSize):
        fitnesses.append(self.calcFitness(self.population[i]))
        if fitnesses[i] == 1:
            return self.population[i]
        total_fitness = total_fitness + fitnesses[i]
```

بخش چهار: پیاده سازی crossover و mutation و تولید جمعیت بعدی

قبل از اینکه به بخش crossover و mutation برسیم، از جمعیت اولیه تعدادی کروموزوم را که نسبت به بقیه وضع بهتری دارند را به طور مستقیم انتخاب میکنیم تا در جمعیت جدید حضور داشته باشند. برای این کار، نسبت هر fitness را به جمع کل fitness محاسبه میکنیم و سپس سورت میکنیم. آن کروموزوم هایی که نسبت بالاتری نسبت به بقیه دارند، انتخاب میشوند. کد این بخش در شکل زیر آمده است.

```
# Calculate probability
prob_fitness = []
for i in range(len(fitnesses)):
    prob_fitness.append(fitnesses[i] / total_fitness)

# Carry over the best chromosomes
fit = [(fitnesses[i], chromo) for i, chromo in enumerate(self.population)]
sorted_fitness = sorted(fit, reverse=True)
carriedChromosomes = []
carry = int(carryPercentage * populationSize)
for i in range(carry):
    carriedChromosomes.append(sorted_fitness[i][1])
```

سپس matingPool را محاسبه میکنیم. در این تابع، با استفاده از تابع رندم، یک جمعیت جدید به اندازه همان جمعیت قبلی ولی با وضعیت بهتر میسازیم.

```
def createMatingPool(self, prob_fitness):
    matingPool = []
    cum_sum = np.cumsum(prob_fitness)
    for _ in range(populationSize):
        r = random.random()
        for j in range(len(cum_sum)):
            if r < cum_sum[j]:
                matingPool.append(self.population[j])
                break
    return matingPool
```

تابع crossover در شکل زیر آمده است. در crossover بین دو کروموزوم مقداری ژن جابجا میشود تا به حالت مطلوب تری برسیم. برای این کار هربار یک عدد رندوم که دامنه آن همان اندازه کروموزوم است، بدست می‌آوریم. سپس وقتی مکان crossover مشخص شد، کروموزوم i و کروموزوم i+1 از آن بخش مشخص شده، با یکدیگر جابجا میشوند و این حلقه به اندازه کل جمعیت تکرار میشود.

```
def createCrossoverPool(self, matingPool):
    crossoverPool = []
    crossoverPool = copy.deepcopy(matingPool)
    for i in range(0, populationSize, 2):
        if i != populationSize - 1:
            r = random.random()
            if r < crossoverProbability:
                crossoverPoint = random.randint(1, self.equationLength - 1)
                crossoverPool[i][crossoverPoint:] = matingPool[i +1][crossoverPoint:]
                crossoverPool[i +1][crossoverPoint:] = matingPool[i][crossoverPoint:]
    return crossoverPool
```

در تابع mutate ، هر کروموزوم را بررسی میکنیم و بعضی از ژن های آن را به قصد بهتر کردن کروموزوم تغییر میدهیم. به ازای هر کروموزوم، تابع mutate صدا زده میشود. یک عدد رندوم انتخاب میکنیم و اگر این عدد از mutationProbabilty کمتر باشد، یک جای رندوم در کروموزوم را تغییر میدهیم.

```
def mutate(self, chromosome):
    r = random.random()
    if r < mutationProbability:
        mutationPoint = random.randint(0, self.equationLength - 1)
        if mutationPoint % 2 == 0:
            chromosome[mutationPoint] = random.choice(self.operands)
        else:
            chromosome[mutationPoint] = random.choice(self.operators)
    return chromosome
```

بخش پنج: ایجاد الگوریتم ژنتیک روی مسئله

در نهایت جمعیت جدیدمان را می‌سازیم. مقداری را از طریق mutation و مقدار دیگر را از طریق همان کروموزوم هایی که تعیین کرده بودیم تا به صورت مستقیم به جمعیت بعدی بفرستیم، ترکیب میکنیم و جمعیت جدید را میسازیم تا وارد حلقه شود. این مراحل تا زمان رسیدن به جواب ادامه پیدا میکند.

```
    self.population.clear()
    for i in range(populationSize - int(populationSize*carryPercentage)):
        self.population.append(self.mutate(crossoverPool[i]))

    self.population.extend(carriedChromosomes)
```

جوابی که با توجه به ورودی داده شده در پی دی اف بدست می‌آید:

```
21
1 2 3 4 5 6 7 8
+ - *
18019
5*7+6*2*8*6*8*4-7*8*8
```

بخش شش : سوالات

1. **جمعیت اولیه بسیار کم یا بسیار زیاد چه مشکلاتی را بوجود می‌آورد؟**

اگر جمعیت اولیه خیلی کم باشد، ممکن است پاسخ مورد نیاز ما هیچوقت تشکیل نشود. در اینصورت ممکن است زمان زیادی بگذرد ولی ما به جواب مسئله خودمان نرسیم. از طرف دیگر اگر جمعیت اولیه بسیار زیاد باشد، در اینصورت جواب مسئله به احتمال بالایی پیدا میشود اما سرعت برنامه به شدت پایین می اید و در مدت زمان کافی ممکن است به جواب نرسیم و از طرف دیگر

حافظه بیشتری برای ذخیره داده ها نیازمندیم. پس هردوی این دو نامناسب هستند و باید جمعیت اولیه یک مقدار معقول باشد.

2. اگر تعداد جمعیت در هر دوره افزایش یابد، چه تاثیری روی دقت و سرعت الگوریتم می‌گذارد؟

اگر جمعیت در هر دوره افزایش پیدا کند، حافظه زیادی نیاز داریم و این ممکن است مشکل ساز شود از طرف دیگر اگر مقدار مقدار جمعیت خیلی زیاد شود، همانطور که در قسمت قبل گفتیم، سرعت برنامه را به شدت پایین می‌آورد ولی احتمال افزایش دقت وجود دارد.

3. تاثیر هر یک از عملیات های crossover و mutation را بیان و مقایسه کنید. آیا میتوان فقط یکی از آنهارا استفاده کرد؟ چرا؟

Crossover : در اینجا بخشی از دو کروموزوم متوالی را با یکدیگر جابجا میکنیم تا کروموزوم های جدیدی درست شود. ابتدا یک عدد رندوم به اندازه طول کروموزوم در نظر میگیریم و سپس از ان بازه انتخاب شده، ژن های دو کروموزوم را جابجا میکنیم.

Mutation : در این بخش، تعدادی از ژن های درون یک کروموزوم را جابجا میکنیم. به این صورت عمل میکنیم که عدد رندوم به اندازه طول کروموزوم در نظر میگیریم و سپس ان را با یک مقدار تخمینی تغییر میدهیم.

پس طبعا نمیتوان از یکی از آنها استفاده کرد و هردوی آنها برای سریعتر رسیدن به جواب لازم میباشند.

4. به نظر شما چه راهکار هایی برای سریعتر به جواب رسیدن در این مسئله ی خاص وجود دارد؟

یکی از کارهایی که در کد استفاده شده و برای سریعتر به جواب رسیدن بسیار موثر است، این است که ایندکس های زوج را به اعداد و ایندکس های فرد را به اپراتور ها اختصاص دهیم و این را در تمامی مراحل کد رعایت کنیم. به این صورت بسیاری از جواب های غلط اصلا در نظر گرفته نمیشوند و سرعت رسیدن به جواب بیشتر میشود.

5. با وجود استفاده از این روش ها، باز هم ممکن است که کروموزوم ها پس از چند مرحله دیگر تغییر نکنند. دلیل این اتفاق و مشکلاتی که به وجود می آورد را شرح دهید. برای حل آن چه پیشنهادی میدهید؟

این اتفاق به دلیل این است که تنوع جمعیت پس از مدتی کم میشود و جمعیت در یک لوکال مینیموم گیر میکند و احتمال حل شدن مسئله در چنین شرایطی کم خواهد شد. برای حل آن میتوان وضعیت را در نظر گرفت که اگر پس از چند دور متوالی، فیتنس ثابت ماند، در آنصورت یک جمعیت جدید تولید شود و مسئله را با آن حل کنیم.

6. چه راه حلی برای تمام شدن برنامه در صورتی که مسئله جواب نداشته باشد، میدهید؟

یه متغیر به عنوان maxgeneration تعریف کردم و مقدار آن را برابر 1000 گذاشتم. در هر بار که حلقه را طی میکند، یک مقدار از این متغیر کم میکنیم. وقتی این مقدار به صفر رسید و ما هنوز به جواب نرسیدیم، میتوانیم بگوییم این مسئله جواب ندارد.