گزارش تمرین پنجم هوش محاسباتی

غزاله محمودی

Traveling Salesman problem

در این مسئله می خواهیم به کمک الگوریتم ژنتیک دور همیلتنی یک گراف را پیدا کنیم به طوری که مجموع اندازه یال هایی که از آن میگذریم مینیمم باشد .

کروموزوم تعریف شده در این مسئله رشته ای از حروف می باشد که در واقع یکی از دور های گراف مد نظر می باشد .

تابع fitness مجموع اندازه یال هایی است که در دور همیلتنی از آن میگذریم.

هدف این است که دور با مینیمم fitness را پیدا کنیم .

در ابتدا جمعیت اولیه را به صورت رندوم مقدار دهی می کنیم .

```
def generate_start_population(self):
    return [np.random.permutation(self.node) for i in range(self.population_size)]
```

سپس fitness مجموعه را با توجه به تعریفی که از آن یاد شد محاسبه می کنیم .

```
def calculate_fitness(self) :
    fitness = []

    for pop in self.population :
        dist = 0

        for i in range(len(pop) - 1):
            path = "".join(pop[i: i+2])
            dist += edge[path]

        path = pop[len(pop)-1] + pop[0]
        dist += edge[path]

        fitness.append(dist)

    return fitness
```

سپس جمعیت جدید را تولید میکنیم . نحوه تولید جمعیت جدید به این صورت است که تعدادی از نمونه های جمعیت قبلی را که fitness مناسبی داشتند نگه می داریم و همچنین برای تولید مابقی به طور رندوم از جمعیت فعلی parent ها را انتخاب می کنیم و با فرایند های mutation , crossover child جدید را می سازیم .

لازم به ذکر است برای اینکه با چه درصدی mutation یا crossover انجام دهیم راه ها و احتمالات گوناگونی وجود دارد . در این جا این طور در نظر گرفته شد که با احتمال 0.8 crossover و 0.2 mutation را اجرا کنیم .

و عملیات باز تولید نسل و محاسبه fitness را با تعداد epoch (که قابل تغییر است) مشخص کردیم، تکرار می کنیم تا به این اطمینان برسیم که بهترین جواب ممکن را به دست می آوریم.

در ادامه لازم به ذکر است که برای mutation برای یک نمونه دو ایندکس را به طور رندوم انتخاب کرده و با هم جا به جا می کنیم .

```
def mutation(self, parent) :
    rand =  np.random.randint(0, len(parent) - 1, 2)
    x = parent.copy()
    x[rand[1]], x[rand[0]] = x[rand[0]], x[rand[1]]
    return x
```

برای crossover هم طبق تابع زیر عمل می کنیم .

```
def cross_over(self, parent) :
    child = parent[0].copy()
    index = np.random.randint(0, len(parent[0]) - 1, 2)

    start = np.min(index)
    end   = np.max(index)

    prime = [item for item in parent[1] if item not in child[start : end + 1]]
    prime_index = 0

    for i in range(len(parent[0])):
        if i < start  or i > end:
            child[i] = prime[prime_index].copy()
            prime_index += 1

    return child
```

در نهایت با اجرا الگوریتم با تعداد epoch های مشخص مسیر های بهینه به صورت زیر به دست آمد .

```
best path :  DABCD  distance :  125
```

```
best path :  BADCB  distance :  125
```

که با توجه به مسیرهای به دست آمده خواهیم یافت یکسان هستند فقط نقاط شروع دور متفاوتی دارند.

Root of polynomial equation

در این مسئله می خواهیم به کمک الگوریتم ژنتیک ریشه معادله چند جمله ای را پیدا کنیم .

برای پیدا کردن ریشه بازه محدود (100 , 100-) را بررسی می کنیم . لازم به ذکر است که الگوریتم ژنتیک جواب نزدیک بهینه (و نه جواب دقیق) را پیدا می کند .

کروموزوم تعریف شده در این مسئله اعداد حقیقی هستند . سایز جمعیت را 50 در نظر میگیریم.

تابع fitness با محاسبه مقدار چند جمله ای در x مد نظر (کروموزوم) به دست می آید و هر چه به صفر نزدیکتر باشد به جواب اصلی نزدیک تر است .

در ابتدا جمعیت اولیه را به صورت رندوم مقدار دهی می کنیم .

```
def generate_start_population(self):
    return np.random.randint(self.lower_bound, self.upper_bound, self.population_size)
```

سپس fitness مجموعه را با توجه به تعریفی که از آن یاد شد محاسبه می کنیم .

```
def calculate_fitness(self) :
    return [np.abs(self.f(i)) for i in self.population]
```

```
f = lambda x : 9*(x)**5 - 194.7*(x)**4 + 1680.1*(x)**3 - 7227.2*(x)**2 + 15501.2* (x) -13257.2
```

سپس جمعیت جدید را تولید میکنیم . نحوه تولید جمعیت جدید به این صورت است که تعدادی از نمونه های جمعیت فعلی را که fitness مناسبی داشتند نگه می داریم و همچنین برای تولید مابقی به طور رندوم از جمعیت فعلی parent ها را انتخاب می کنیم و با فرایند های mutation , crossover child جدید را می سازیم .

و عملیات باز تولید نسل و محاسبه fitness را با تعداد epoch که قابل تغییر است مشخص کردیم تکرار می کنیم تا به این اطمینان برسیم که بهترین جواب ممکن را به دست می آورده ایم.

در ادامه به توضیح چگونگی crossover , mutation می پردازیم .

در ابتدا لازم به ذکر است برای اینکه با چه درصدی mutation یا crossover انجام دهیم راه ها و احتمالات گوناگونی وجود دارد . در این جا این طور در نظر گرفته شد که اگر parent های انتخابی مشابه باشند mutation و در غیر این صورت crossover انجام شود.

در mutation تنها parent های انتخاب شده را با اعداد رندومی جمع می کنیم .(در واقع می خواهیم جهشی در جمعیت ایجاد کنیم به این امید که به نمونه ها با fitness بهتر برسیم .)

```
def mutation(self, parent) :
    return parent + np.random.randint(1, 10), parent + np.random.randint(20, 50)
```

برای crossover ابتدا parent ها را به اعداد باینری تبدیل می کنیم و نقطه رندومی را به عنوان crossover point انتخاب کرده و از آن نقطه دو child جدید تولید می کنیم .

```
def cross_over(self, parents) :
    a = self.convert_decimal_to_binary(parents[0])
    b = self.convert_decimal_to_binary(parents[1])

    cross_over_point = np.random.randint(8, max(len(a)-1, len(b)-1))

    a , b = a[:cross_over_point] + b[cross_over_point:],  b[:cross_over_point] + a[cross_over_point:]
    return self.convert_binary_to_decimal(a) , self.convert_binary_to_decimal(b)

def convert_decimal_to_binary(self, x) :
    return format(x, '010b')


def convert_binary_to_decimal(self, x) :
    return int(x, 2)
```

در نهایت با اجرا الگوریتم با تعداد epoch های مشخص و تابع چند جمله ای سوال نتیجه زیر حاصل شد که بسیار نزدیک به جواب اصلی است .

```
root :  4 fitness :  11.600000000002183
```