بسم الله الرحمن الرحیم

پروژه ۱ درس هوش مصنوعی

دکتر فدایی و دکتر یعقوب زاده

مهدی وجهی

۸۱۰۱۰۱۵۵۸

فهرست

مقدمه

هدف از این پروژه حل مسئله Problem Knapsack با استفاده از الگوریتم ژنتیک است. ما در این مسئله باید حالت بهینه از غذا ها را که شرایط ورودی را داشته باشد پیدا کنیم.
برای این کار ما ابتدا باید ساختار کروموزوم های خود را تعریف کنیم که دارای قابلیت تلفیق با بقیه و جهش را داشته باشد. سپس باید تعدادی کروموزوم اولیه تولید کنیم و با پیاده سازی یک تابع فیتنس برای آنها میزان خوب بودن آنها را بسنجیم این تابع باید به شرایط مسئله حساس باشد و همچنین میزان خوب بودن ژن ها را نشان دهد. سپس باید تابعی برای تولید نسل بعدی نوشت که ابتدا کروموزوم ها را جفت کند سپس آنها را تلفیق کند و در نهایت آنها را جهش دهد. در آخر هم باید نتایج را بررسی کرد و جواب به دست آمده را بررسی کنیم.

تعریف ژن و کروموزوم ها

در این مسئله با توجه به این که طول کروموزوم باید ثابت باشد و همچنین تعداد غذا ها ثابت است می توانیم کروموزوم را فهرست تمام غذا ها بگذاریم که با یک متغیر مشخص شود که آیا آن غذا را می بریم یا خیر و اگر می بریم چه مقدار از آن را می بریم. پس در این مسئله ژن ها همان غذا ها هستند که می تواند فعال یا غیر فعال باشند.

تعریف ژن ها

کلاس ژن در واقع نماینده یک غذا است که در آن نام، حداکثر مقدار موجود، فعال بودن آن، چگالی ارزش آن و وزن الان آن است. توابع آن شامل توابع مقدار دهی، توابع خروجی، تابع جهش است که به دلیل سادگی باقی توابع تنها تابع جهش توضیح داده می شود.

تابع جهش

در تابع جهش وزن ژن با احتمالی که برای آن تعیین شده (WIGHT_MUTATION_CHANCE) و به اندازه تعیین شده (WIGHT_MUTATION_RATE) به طور شانسی کم یا زیاد می شود.

```
    def mutation(self):
        if IS_WIGHT_MUTATION_IF_DEACTIVATE or self.is_active:
            if WIGHT_MUTATION_CHANCE > random.random():
                self.weight = max(MIN_WEIGHT,
                                  min(self.max_weight, self.weight
                                      + (self.max_weight *
                                         WIGHT_MUTATION_RATE
                                         * random.choice([1,-1]))))
```

البته می توان این تابع را به صورت دیگری نیز نوشت به طوری که ژن با یک توزیع نرمال حول مقدار قبلی خود تغییر می کند. توزیع به گونه ای تنظیم شده که انحراف معیار به اندازه یک ثابت بین ۰ تا ۱ در مقدار وزن باشد که تغییر معقول باشد. همچنین شرطی گذاشته شده که آیا در موقع غیر فعال بودن ژن وزن آن جهش کند یا خیر.

```
    def weight_mutation(self):
        std = self.weight * WIGHT_MUTATION_RATE
        self.weight = np.random.normal(self.weight, std, 1)[0]
        # fix number in range and round
        self.weight = round((max(MIN_WEIGHT, min(self.max_weight,
self.weight))),1)
```

این روش با طبیعت سازگاری بیشتری دارد اما به دلیل ایجاد تابع توزیع زمان اجرا به طور قابل توجهی افزایش می یابد به طوری که زمان اجرا برنامه حدودا ۱۰۰ برابر می شود.

تعریف کروموزوم ها

کروموزوم ها در واقع تجمیع ژن ها هستند. این کلاس شامل توابع جهش، تلفیق، تنظیم مقادیر، خروجی ها می شود که در ادامه به توضیح توابع جهش و تلفیق می پردازیم.

توابع جهش

جهش در کروموزوم ها به این صورت یک جهش وزنی دارم و یک جهش تعدادی.
در جهش وزنی صرفا روی ژن ها پیمایش می شود و تابع جهش آنها صدا زده می شود.

```
    def weight_mutation(self):
        for i in self.gens:
            i.mutation()
```

در جهش تعدادی ما ژن ها را فعال یا غیر فعال می کنیم که به دو صورت است. در حالت اول به به صورت شانسی تعداد ژن های خود را تغییر می دهیم مثلا اگر ۳ ژن فعال داریم آنها را به ۴ ژن یا ۱ ژن تغییر می دهیم فقط ما بررسی می کنیم که تعداد ژن های فعال از محدودیت تعداد مسئله خارج نشود.

```
    def add_or_remove_food_mutation(self):
        new_count = np.random.randint(self.count_range[0],
self.count_range[1] + 1)
        diff = new_count - len([i for i in range(len(self.gens)) if
self.gens[i].is_active == True])
        if diff == 0:
            return
        elif diff > 0:
            disable_food = np.random.choice([i for i in range(len(self.gens))
if self.gens[i].is_active == False], int(diff), replace=False)
            for i in disable_food:
                self.gens[i].set_gen_status(True)
        else:
            active_food = np.random.choice([i for i in range(len(self.gens))
if self.gens[i].is_active == True], int(abs(diff)), replace=False)
            for i in active_food:
                self.gens[i].set_gen_status(False)
```

حالت دوم به این صورت است که ما ژن های فعال خود را تغییر می دهیم مثلا اگر ژن های فعال ما a,b,c باشد بعد اجرای آن به a,d,h تغییر می کند.

```
    def change_active_food_mutation(self):
        disable_gen = [i for i in range(len(self.gens)) if
self.gens[i].is_active == False]
        active_gen = [i for i in range(len(self.gens)) if
self.gens[i].is_active == True]
        change_count = np.random.randint(min(len(active_gen),
len(disable_gen)) + 1)
        selected_disable_gen = np.random.choice(disable_gen, change_count,
```

```
replace=False)
        selected_active_gen = np.random.choice(active_gen, change_count,
replace=False)
        for i, j in zip(selected_disable_gen, selected_active_gen):
            self.gens[i].set_gen_status(True)
            self.gens[j].set_gen_status(False)
```

البته قبلا فعال شدن ژن ها درون ژن ها انجام می شد و به محدودیت های مسئله توجهی نمی کرد اما با پیاده سازی به این روش بازدهی الگوریتم بالاتر رفت.

توابع تلفیق

تلفیق به این صورت انجام می گیرد که ابتدا به طور شانسی تعداد ژن های فعال یک کروموزوم انتخاب می شود و سپس به همان تعداد از مجموع ژن ها فعال ۲ کروموزوم ژن انتخاب می شود و برای مابقی ژن ها به صورت شانسی از یکی از کروموزوم ها گرفته می شود. احتمالی هم وجود دارد که دو کروموزوم مستقیم به نسل بعد منتقل شوند.

برای این کار اپراتور جمع بازنویسی که با احتمالی خود دو کروموزوم را بر می گرداند و با احتمالی دیگری با روش ذکر شده دو کروموزوم جدید می سازد و به نسل بعد می دهد.

```
    def __add__(self, pair):
        if random.choices([True, False],
                          [CROSSOVER_CHANCE,1 - CROSSOVER_CHANCE])[0]:
            return self.crossover(pair)
        return self, pair
```

```
    def crossover(self, pair):
        all_active_gen, count = self.get_active_gen_and_number(pair)
        child_1 = self.create_gens_for_new_chromosome(all_active_gen, count,
pair)
        child_2 = self.create_gens_for_new_chromosome(all_active_gen, count,
pair)
```

تابع create_gens_for_new_chromosome

این تابع به این صورت کار می کند به طور شانسی و به تعداد ژن های فعال یکی از کروموزوم ها از کل ژن های فعال دو کروموزوم، ژن انتخاب می کند و آن ژن ها را فعال می کند و وزن آن را به وزن ژن در کروموزوم مبدا می گذارد اگر هر دو کروموزوم آن ژن را داشته وزن به صورت شانسی از یکی برداشته می شود.

```
    def create_gens_for_new_chromosome(self, active_gen, count, pair):
        new_active_gen = np.random.choice(list(active_gen.keys()),
                                          count[np.random.randint(2)],
replace=False)
        new_gens = self.create_gens_from_pair(pair)

        for gen_index in new_active_gen:
```

```
            parent = [self,
pair][np.random.randint(len(active_gen[gen_index]))]
            new_gens[gen_index].set_gen_weight(parent.gens[gen_index].weight)
            new_gens[gen_index].set_gen_status(True)

        return new_gens
```

تابع get_active_gen_and_number

این تابع ژن ها فعال را در قالب (دیکشنری{شماره ژن: در کدام کروموزوم ها فعال است}) ذخیره می کند همچنین تعداد ژن های فعال هر کروموزوم را می شمارد.

```
    def get_active_gen_and_number(self, pair) -> tuple[dict[int, list[int]],
list[int]]:
        counter = [0, 0]
        active_gen = dict[int, list[int]]()
        chromosomes = [self, pair]
        for i in range(len(chromosomes)):
            for j in range(len(self.gens)):
                if chromosomes[i].gens[j].is_active == True:
                    counter[i] += 1
                    if j in active_gen:
                        active_gen[j] += [i]
                    else:
                        active_gen[j] = [i]

        return active_gen, counter
```

تابع create_gens_from_pair

این تابع یک لیست از ژن ها می سازد که هر ژن به صورت شانسی از یکی از کروموزوم ها، انتخاب می شود. به علت این که ژن های چند کروموزوم یکسان نباشند از ژن ها کپی عمیق گرفته می شود.

```
    def create_gens_from_pair(self, pair) -> list[Gen]:
        new_gen = Gen
        new_gens = list[Gen]()
        for i in range(len(self.gens)):
            if np.random.randint(2):
                new_gen = copy.deepcopy(self.gens[i])
            else:
                new_gen = copy.deepcopy(pair.gens[i])
            new_gen.set_gen_status(False)
            new_gens.append(new_gen)
        return new_gens
```

تولید نسل اولیه

در این قسمت باید نسل اول را تولید کنیم برای این کار لازم است گام های زیر را طی کنیم:

1. خواندن داده غذا ها
2. تولید ژن ها خام از غذا ها
3. تولید کروموزم با مقادیر شانسی

خواندن داده غذا ها

در این قسمت فایل snack.csv را بارگذاری می کنیم که حاوی فهرستی از غذا ها و محدودیت های آن است.

```
snacks = pd.read_csv('./snacks.csv')
```

	Snack	Available Weight	Value
0	MazMaz	10	10
1	Doogh-e-Abali	15	10
2	Nani	5	5
3	Jooj	7	15
4	Hot-Dog	20	15
5	Chips	8	6
6	Nooshaba	12	8
7	Shokolat	6	7
8	Chocoroll	9	12
9	Cookies	11	11
10	Abnabat	4	4
11	Adams-Khersi	14	9
12	Popcorn	16	13
13	Pastil	3	7
14	Tordilla	10	9
15	Masghati	5	6
16	Ghottab	7	10
17	Saghe-Talaei	9	11
18	Choob-Shoor	13	12

تولید ژن ها خام

در این قسمت ژن های خام اولیه را می سازیم که در قسمت های بعد آن ها را مقدار دهی و استفاده کنیم. تنها کافیست مقادیر را از جدول به کلاس منتقل کنیم.

```
row_gens = list[Gen]()
for i in snacks.iterrows():
   name, max_weight, value = i[1]
   row_gens.append(Gen(name, max_weight, value/max_weight))
```

نمونه ای از ژن های تولید شده به صورت زیر هستند:

```
------------
name: MazMaz
status: False
weight: 1.00
max weight: 10.00
vpw: 1.00
value: 1.00
------------
name: Doogh-e-Abali
status: False
weight: 1.00
max weight: 15.00
vpw: 0.67
value: 0.67
```

تولید کروموزوم های نسل اول

حال باید ژن های خام نسل قبل را به صورت شانسی مقدار دهی کنیم و در کروموزوم بریزیم و سپس با یک حلقه به تعداد پارامتر مربوطه کروموزوم بسازیم و نسل اول خود را تشکیل دهیم.

تابع create_random_chromosome

این تابع ابتدا از ژن های خام کپی می گیرد و سپس وزن شانسی روی آنها می گذارد و سپس به تعداد شانسی که در بازه مجاز، ژن ها را به صورت شانسی فعال می کند و در آخر یک کروموزوم می سازد.

```
def create_random_chromosome(count_range) -> Chromosome:
   gens = list[Gen]()
   for i in row_gens:
       new_gen = copy.deepcopy(i)
       new_gen.set_random_gen()
       gens.append(new_gen)
   active_count = np.random.randint(count_range[0],count_range[1]+1)
   for i in np.random.choice([i for i in range(len(gens))], active_count,
replace=False):
       gens[i].set_gen_status(True)
   return Chromosome(gens,count_range)
```

تابع create_first_generation

این تابع صرفا با حلقه زدن روی تابع قبلی به تعدادی که پارامتر FIRST_GENERATION_COUNT تعیین می کند کروموزوم می سازد که نسل اول را تشکیل می دهد.

```
def create_first_generation(count_range) -> list[Chromosome]:
    first_chromosomes = list[Chromosome]()
    for _ in range(FIRST_GENERATION_COUNT):
        first_chromosomes.append(create_random_chromosome(count_range))
    return first_chromosomes
```

عملکرد آن را آزمایش می کنیم:

```
FIRST_GENERATION_COUNT = 2
test = create_first_generation([2,4])
for i in test[:2]:
    print(r'###############')
    print(i)
    print(r'__________')
    for j in i.gens[:1]:
        print(j)
```

```
###############

__________
active_count: 2
weight: 10.99
value: 7.71
vpw: 0.70

__________
------------
name: MazMaz
status: False
weight: 3.42
max weight: 10.00
vpw: 1.00
value: 3.42
###############

__________
active_count: 3
weight: 9.39
value: 11.02
vpw: 1.17

__________
------------
name: MazMaz
status: False
weight: 0.89
```

```
max weight: 10.00
vpw: 1.00
value: 0.89
```

طراحی تابع فیتنس

برای طراحی تابع فیتنس ما ۴ مورد داریم که می توانیم بررسی کنیم:

- ارزش غذایی
- وزن غذا ها
- تعداد غذا ها
- چگالی ارزش غذا ها

شرایط مسئله ما را به صورت زیر محدود می کند:

- حداقل ارزش
- حداکثر وزن
- بازه ای برای تعداد غذا ها

بنابراین تصمیم می گیریم که تابع فیتنس خود را روی ارزش، وزن، تعداد غذا ها تنظیم کنیم که هر کدام در ضریب خود جمع می شوند و به عنوان عدد نهایی برگردانده می شوند. البته که می توان برای سنگین تر کردن جریمه اعداد را به توان رساند.

```
def calculate_fitness(chromosome:Chromosome, conditions:dict):
   value_point =   calculate_value_fitness(chromosome.get_value(),
conditions["value"])
   weight_point =  calculate_weight_fitness(chromosome.get_weight(),
conditions["weight"])
   count_point =   calculate_count_fitness(chromosome.get_active_count(),
conditions["count"])
   return  value_point  *  FITNESS_VALUE_RATE +\
           weight_point *  FITNESS_WEIGHT_RATE +\
           count_point  *  FITNESS_COUNT_RATE
```

حال برای هر کدام از این موارد تابع مربوطه را می نویسیم.

تابع calculate_value_fitness و calculate_weight_fitness

این تابع اختلاف تا حداقل ارزش را حساب می کند. اگر منفی بود آن را بر میگرداند. اگر مثبت بود در یک ضریبی ضرب می کند و آن را بر می گرداند. (هردو به یک شکل هستند)

```
def calculate_value_fitness(value:float, condition:float):
   diff = value - condition
   if diff < 0:
       return diff
   return diff * POSITIVE_VALUE_RATE
```

تابع calculate_count_fitness

این تابع نیز بررسی می کند که در تعداد در بازه مجاز هست یا نه و اگر نیست منفی اختلاف را برمی گرداند و اگر بود یک پارامتر را بر می گرداند.

```
def calculate_count_fitness(count:int, condition:list[int]):
    if count < condition[0]:
        return count - condition[0]
    if count > condition[1]:
        return condition[1] - count
    else:
        return POSITIVE_COUNT_RATE
```

تابع calculate_generation_fitness

این تابع صرفا یک حلقه روی کروموزوم های نسل می زند و نتایج را بر می گرداند.

تولید نسل های بعدی

برای الگوریتم ژنتیک لازم است روشی برای تولید نسل بعد ارائه کنیم. برای این کار مراحل زیر را طی میکنیم:

1. تبدیل فیتنس ها به احتمال
2. جفت کردن کروموزوم ها
3. تولید کروموزوم های جدید با استفاده از جفت ها
4. جهش کروموزوم های جدید

تبدیل فیتنس به احتمال

تابع فیتنس ها از منفی بینهایت تا صفر است. (بدون در نظر گرفتن امتیاز مثبت) بنابراین نمی توان آنها را به عنوان احتمال استفاده کرد. برای رفع این مشکل به این صورت زیر عمل می کنیم:

1. پیدا کردن کوچک ترین عدد
2. تبدیل یکی کمتر از آن به مبدا
3. محاسبه احتمالات با اعداد مثبت به دست آمده

مثلا ما با داشتن [-1,-2,-3] ابتدا عدد منفی ۳ را پیدا می کند و عدد ۴- را مبدا قرار می دهیم پس لیست به این شکل می شود [1,2,3] و سپس به آسانی به احتمال تبدیل می شوند.

```
def
convert_fitness_to_probability_of_choice(generation_fitness:list[float]):
    minimum = min(generation_fitness)
    if minimum <= 0:
        generation_fitness = [i + abs(minimum) + 1 for i in
generation_fitness]
```

```
    sum_all = sum(generation_fitness)
    return [i/sum_all for i in generation_fitness]
```

جفت کردن کروموزوم ها

برای این کار به صورت زیر عمل می کنیم:

1. محاسبه احتمال انتخاب شدن هر کروموزوم
2. انتخاب کروموزوم با توجه به احتمال آنها و به تعداد نسل
3. تولید و جهش کروموزوم های جدید

```
def create_next_generation(generation:list[Chromosome],
generation_fitness:list[float],
                           size:int) -> list[Chromosome]:
    probability_of_choice =
convert_fitness_to_probability_of_choice(generation_fitness)
    rand_choice = np.random.choice(generation, size, p=probability_of_choice)
    next_generation = list[Chromosome]()
    for i in range(math.floor(size/2)):
        children = create_children((rand_choice[i*2], rand_choice[i*2+1]))
        next_generation.extend(children)

    return next_generation
```

تولید و جهش کروموزوم های جدید

برای این کار تنها کافیست از توابع تعریف شده در کلاس استفاده کنیم که به صورت زیر می شود. پارامتر هایی هم برای فعال و غیر فعال کردن آن ها قرار می دهیم.

```
def create_next_generation(generation:list[Chromosome],
generation_fitness:list[float],
                           size:int) -> list[Chromosome]:
    probability_of_choice =
convert_fitness_to_probability_of_choice(generation_fitness)
    rand_choice = np.random.choice(generation, size, p=probability_of_choice)
    next_generation = list[Chromosome]()
    for i in range(math.floor(size/2)):
        children = create_children((rand_choice[i*2], rand_choice[i*2+1]))
        next_generation.extend(children)

    return next_generation
```

ایجاد الگوریتم ژنتیک

قبل از ایجاد الگوریتم نهایی لازم است چند تابع کوچک تعریف کنیم.

- تابع check_chromosome که برقرار بودن شرایط مسئله را در یک کروموزوم بررسی می کند.
- تابع get_generation_size که اندازه نسل را مشخص می کند. (برای بخش سوالات)
- تابع find_best_chromosome که با توجه به مقادیر تابع فیتنس بهترین کروموزوم را پیدا می کند.

برای ساخت تابع ژنتیک به ترتیب زیر عمل می کنیم

1. تولید نسل اول و محاسبه فیتنس آن
2. تولید نسل جدید و محاسبه فیتنس آن
3. بررسی پیدا شدن جواب
4. تکرار این روند

```
def solve_with_genetic(conditions):
   generation = create_first_generation(conditions["count"])
   generation_fitness = calculate_generation_fitness(generation, conditions)

   for i in range(GENERATION_STEP):
       generation = create_next_generation(generation, generation_fitness,
get_generation_size(i))

       generation_fitness = calculate_generation_fitness(generation,
conditions)

       best_chromosome = find_best_chromosome(generation,
generation_fitness)

       if check_chromosome(best_chromosome, conditions):
           return best_chromosome, i
   raise Exception("doesn't find")
```

ارزیابی نتایج

با اندازه جامعه ۲۰ مسئله را حل می کنیم.

مسئله:

```
conditions = {"weight": 10,
              "value": 12,
              "count": [2,4]}
```

و مشاهده می کنیم در کمتر از صدم ثانیه مسئله حل می شود.

```
step:0, best:-0.89, avg:-2.98, size:20
step:1, best:-0.66, avg:-2.74, size:20
```

```
step:2, best:-0.11, avg:-2.96, size:20
step:3, best:0.09, avg:-1.70, size:20
step:3, best:0.09, avg:-1.70, size:20
------------
name: Nani
status: True
weight: 4.86
max weight: 5.00
vpw: 1.00
value: 4.86
------------
name: Jooj
status: True
weight: 2.20
max weight: 7.00
vpw: 2.14
value: 4.71
------------
name: Pastil
status: True
weight: 1.41
max weight: 3.00
vpw: 2.33
value: 3.29
__________
active_count: 3
weight: 8.47
value: 12.86
vpw: 1.52
-------
step: 3
size: 20
```

حال مسئله را سخت تر می کنیم.

```
conditions = {"weight": 10,
              "value": 21,
              "count": [3,19]}
```

بعد از نیم ثانیه حل می شود.

```
step:0, best:-10.05, avg:-30.10, size:20
step:100, best:-2.22, avg:-17.43, size:20
step:169, best:0.00, avg:-32.67, size:20
------------
name: Jooj
status: True
weight: 6.30
```

```
max weight: 7.00
vpw: 2.14
value: 13.50
------------
name: Pastil
status: True
weight: 2.70
max weight: 3.00
vpw: 2.33
value: 6.30
------------
name: Masghati
status: True
weight: 1.00
max weight: 5.00
vpw: 1.20
value: 1.20
__________
active_count: 3
weight: 10.00
value: 21.00
vpw: 2.10
-------
step: 169
size: 20
```

تغییر پارامتر ها

احتمال های جهش

با کاهش و افزایش احتمال جهش زمان حل مسئله را بررسی می کنیم.

حالت اولیه:

```
conditions = {"weight": 10,
              "value": 21,
              "count": [3,19]}

# Generation parameter
FIRST_GENERATION_COUNT = 20
GENERATION_STEP = 100000
INCREASE_GENERATION_RATE = 0
PRINT_RATE = 100

# Enable feature
PRINT_ACTIVE = False
HAVE_CROSSOVER = True
```

```
HAVE_MUTATION = True

# Feature rate
WIGHT_MUTATION_RATE = 0.2
CHROMOSOME_MUTATION_RATE = 0.05
WIGHT_MUTATION_CHANCE = 0.3
CROSSOVER_CHANCE = 1/3
```

در زمان ۲.۱ ثانیه مسئله حل می شود.

کاهش احتمال:

```
# Feature rate
WIGHT_MUTATION_RATE = 0.1
CHROMOSOME_MUTATION_RATE = 0.02
WIGHT_MUTATION_CHANCE = 0.1
CROSSOVER_CHANCE = 1/3
```

زمان به حدود ۳.۵ ثانیه می رسد.

افزایش احتمال:

```
# Feature rate
WIGHT_MUTATION_RATE = 0.5
CHROMOSOME_MUTATION_RATE = 0.4
WIGHT_MUTATION_CHANCE = 0.6
CROSSOVER_CHANCE = 1/3
```

زمان بیشتر از ۱ دقیقه می شود.

در نهایت می توان نتیجه گرفت که همان مقدار اولیه این پارامتر خوب بوده و اگر آن را کمی کاهش هم دهیم آسیب زیادی به الگوریتم نمی رسد.

احتمال تلفیق

شرایط اولیه مانند بخش قبل است. جدول زیر نتایج تغییر پارامتر است:

مقدار پارامتر	۱	۱/۲	۱/۳	۱/۶	۱/۱۵
زمان اجرا	۶.۸	۲.۱	۱.۸	۲	۳۷

می توان نتیجه گرفت که مقادیر اگر به ۱ یا صفر میل کنند باعث کند شدن الگوریتم می شوند.

پارامتر های تابع فیتنس

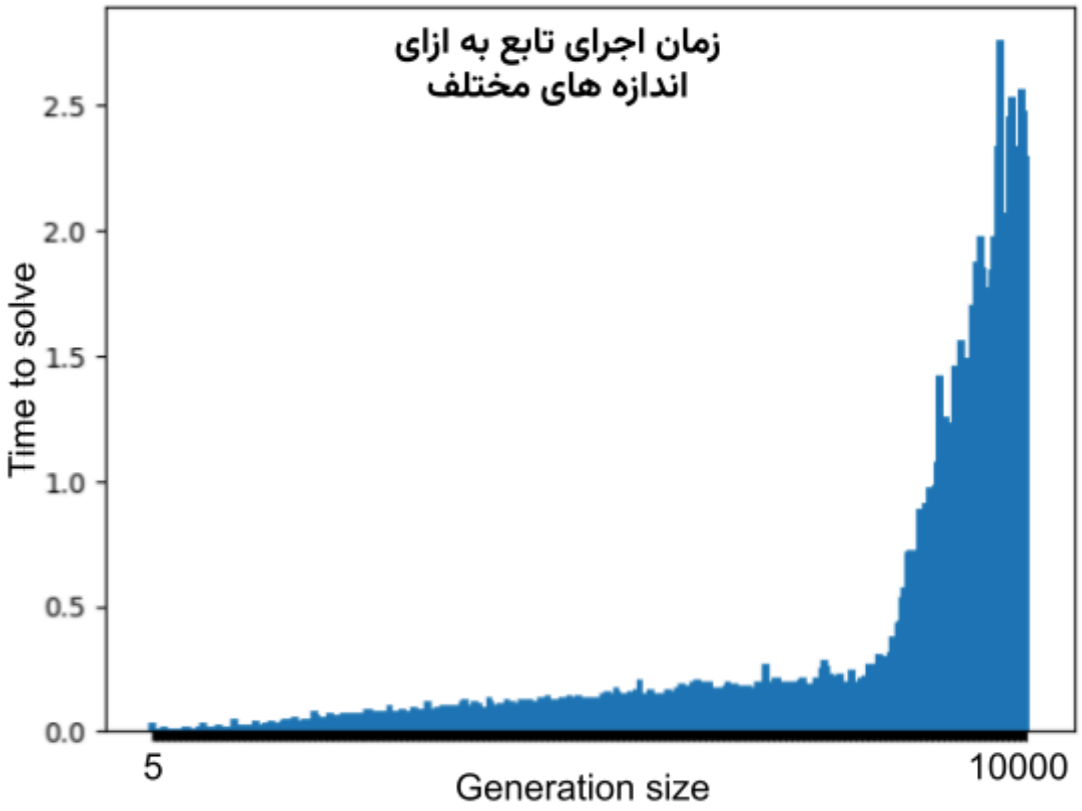
ابتدا پارامترهای نمره مثبت را دستکاری می کنیم و می بینیم که با افزایش آن تا حدود ۰.۷ تاثیری روی جواب ندارد ولی مقدار بزرگتر از ۱ روی جواب تاثیر منفی می گذارد به طوری که الگوریتم گیر می کند.

پارامتر های ضرایب تاثیر زیادی در الگوریتم ندارد.

پاسخ به سوالات

سوال ۱: اندازه جمعیت

جمعیت زیاد باعث می شود بار پردازشی زیادی به سیستم تحمیل شود و زمان اجرای برنامه را بالا می برد و باعث کاهش بازدهی آن می شود. جمعیت کم نیز با این که سرعت تولید نسل های ما را بالا می برد اما تنوع ژنتیکی ما را کاهش می دهد و باعث می شود که نمونه ها یکسان شوند و بعد از تعدادی مرحله عملا تابع تلفیق دیگر سودی نداشته باشد. اما طبق بررسی ای که در این مسئله انجام دادم مشاهده شد که زمان اجرا الگوریتم با جمعیت بسیار کم برای مسئله ذکر شده در صورت پروژه بسیار کمتر است. نمودار زیر میانگین زمان اجرا به ازای تعداد مختلف نسل را نشان می دهد.



سوال ۲: افزایش دوره ای جمعیت

متوجه منظور سوال از دقت الگوریتم نمی شوم ولی طبق بررسی هایی که انجام دادم با افزایش دوره ای جمعیت سرعت برنامه به شدت کاهش پیدا می کند. یعنی اگر ما در جمعیت را روی ۱۰۰ تنظیم کنیم و افزایشی نداشته باشیم مسئله در حدود ۶ ثانیه حل می شود اما با افزایش ۰.۱ جمعیت این زمان به ۱۳ ثانیه و در ۰.۳ به ۲۸ ثانیه میرسد.

سوال ۳: تاثیر تابع جهش و تابع تلفیق

تابع جهش از یکسان شدن نسل جلوگیری می کند و همچنین اگر به جواب نزدیک باشیم باعث می شود که جواب را پیدا کنیم. تابع تلفیق باعث می شود که نسل جدید از نسل قبلی بهتر شود و کروموزوم های خوب با با باقی کروموزوم ها تلفیق شوند تا شاید به کروموزوم بهتری برسیم. همانطور که گفته شد جهش برای ایجاد تنوع است و تلفیق برای بهتر کردن نسل.
شدنش که می شود از یکی از آنها استفاده کرد اما طبق بررسی هایی که بالاتر انجام دادیم اگر هر کدام از این موارد زیاد یا کم انجام شوند به الگوریتم آسیب می زنند و بهره آن را کاهش می دهند حتی در بعضی مواقع می تواند باعث قفل شدن الگوریتم شود و مانع پیشرفت آن.

سوال ۴: سریعتر به جواب رسیدن

واضح است که سریعترین راه رسیدن به جواب این مسئله استفاده از الگوریتم خودش است. اما اگر منظور همین الگوریتم هست می تواند با نوشتن تابعی که بهترین پارامتر هایی که برای حالت های مختلف مسئله به خوبی کار می کند را پیدا کند می تواند در بهبود الگوریتم موثر باشد. همچنین طبیعتا نوشتن برنامه با زبان هایی مانند c در این مسئله موثرند. یکی از راه هایی که به نظر می آید این است که ابتدا مسئله را برای یکسری مقادیر حل کنیم و سپس وقتی ورودی جدیدی به ما داده می شود چند نسلی که در جواب های قبلی به دست آوردیم و سوال های آن نزدیک به این مسئله بود را با هم تلفیق کنیم و به عنوان نسل اول به برنامه بدهیم.

سوال ۵: یکسان شدن نسل

دلیل این اتفاق این است که آن کروموزوم هایی که خوب هستند با یک احتمالی مستقیم به نسل بعد منتقل می شود یا فرزندانشان که شبیه به آنها هستند به نسل بعدی منتقل می شوند و ژن های بد از بین می روند یا با احتمال بسیار کمتری به نسل بعد می روند. مثلا در نظر بگیرید نسل به صورت [1,2,3,4,5] هست و ژن ۱ خیلی خوب است نسل بعدی داریم [3,5,"1,1] و سپس ["1,"1,1,1] و همینطور به نسل به سمت ۱ شدن می رود. این از این جهت خوب است که نسل به سمت بهتر شدن می رود اما بعد از یکسان شدن دیگر نمی تواند بهتر شود و این مشکل ساز است. مشکل اصلی همانطور که گفته شد این است که از نسلی که جامعه یکسان می شود دیگر تلفیق کاربردی ندارد و ما فقط به جهش متکی هستیم و جهش به دلیل احتمالاتی بودن قدرت مقابله ی زیادی با یکسان شدن نسل ندارد و این موضوع باعث توقف الگوریتم می شود.
چند راهکار به نظر می رسد:

- تزریق چند کروموزوم شانسی به نسل جدید
- ذخیره بهترین کروموزوم هر نسل و انتخاب شانسی از آنها و تزریق به نسل جدید
- افزایش احتمال جهش با یکسان شدن نسل
- شروع مجدد الگوریتم بعد از همگرایی در چند مرحله و نرسیدن به جواب

سوال ۶: مسئله بدون جواب

می توانیم در ابتدا یکسری شرایط را بررسی کنیم مثلا چگالی جواب نباید از چگالی بیشترین خوراکی بیشتر باشد همین موضوع بعد از کم کردن کل وزن و ارزش با ارزش ترین خوراکی از مسئله درباره دومین بزرگترین خوراکی صادق است و می توانیم تا چند مرحله این موضوع را بررسی کنیم یا مثلا بررسی کنیم که حداقل ارزش از کل ارزش های حداکثر خوراکی بیشتر نباشد و شرایط اینطوری را بررسی کنیم همچنین اگر الگوریتم بعد از مدت زمان مشخصی (مثلا چارک سوم توزیع اجرا) زمان بیشتری برد الگوریتم متوقف شود و با پارامتر های دیگری اجرا شود و بعد از تعداد مشخصی تلاش می توان گفت که مسئله احتمالا جوابی ندارد. همچنین می توان چند بار تلاش کرد و اگر دیدیم بعد از هر دفعه تلاش انگار که به یک سقفی رسیدیم و تابع فیتنس در تلاش های مختلف از یک حدی بیشتر نمی شود می توان گفت احتمالا جوابی وجود ندارد.

نتیجه گیری

استفاده از الگوریتم ژنتیک در مواردی که ما راه حل مستقیمی برای حل مسئله نداریم می تواند مفید باشد. همچنین تعریف درست و خوب ژن، کروموزوم، تابع جهش و تابع تلفیق بسیار در حل مسئله حیاتی است و توجه به رابطه بین آنها از اهمیت بالایی برخوردار است و تاثیر زیادی در بازدهی الگوریتم دارد.

راهکار هایی برای بهبود و توسعه

موارد زیر می تواند گامی مهم در توسعه پروژه باشد:

- نوشتن تابعی جهت بررسی امکان حل شدن مسئله (مانند آنچه توضیح داده شد)
- تاثیر تابع فیتنس در احتمال جهش کروموزوم
- جلوگیری از یکسان شدن نسل (مانند آنچه توضیح داده شد)
- بهبود تابع فیتنس و نوشتن تابعی که به ارتباط متغیر ها نیز امتیاز دهد
- اجرای برنامه و تولید نسل به صورت multithreading

منابع

- اسلاید های درس