8

به نام خدا

دانشگاه تهران

پردیس دانشکده های فنی

دانشکده مهندسی برق و کامپیوتر

تمرین شماره‌ی 1

بخش عملی

استاد درس:

دکتر فدائی

دکتر یعقوب‌زاده

نگارش:

فاطمه محمدی 810199489

بهینه سازی "کوله‌پشتی"

الگوریتم‌های ژنتیک

بهینه سازی "کوله‌پشتی"

الگوریتم‌های ژنتیک

اهداف:

آشنایی و استفاده از الگوریتم‌های ژنتیک برای حل مسائل بهینه سازی

مقدمه:

در این پروژه با روش‌های برگرفته از طبیعت و انتخاب طبیعی و یا همان الگوریتم‌های ژنتیک آشنا میشویم؛ و قصد داریم با استفاده از این الگوریتم‌ها، یکی از مسائل بهینه سازی را پیدا سازی کنیم.

در مسائل بهینه سازی، هدف یافتن بهترین راه حل که با مجموعه از معیارها و اهداف تعریف میشود، از مجموعه بزرگی از راه حل‌های ممکن است.

حل این نوع مسائل میتواند مزایای فراوانی داشته باشد، از جمله افزایش کاراییف کاهش هزینه ها بهبود عملکرد و موارد دیگر.

تعریف پروژه:

در این پروژه به حل مسئله تغییر یافته از مسئله knapsack یا همان کوله‌پشتی میپردازیم.

در این مسئله فرض کرده ایم که تعدادی خوراکی و تنقلات (که objectهای مسئله هستند) با مقدار محدودی در دسترس داریم که دارای ارزش خاصی میباشند.میخواهیم برخی از این خوراکی‌ها را انتخاب کنیم و در کوله‌پشتی قرار بدهیم، به صورتی که وزن خوراکی‌هایی که انتخاب میکنیم کمتر از یک مقدار خاص باشند، و در عین حال تنوع آنها در بازه خاصی قرار داشته باشد و همچنین مجموع ارزش خوراکی‌ها بیشتر از یک مقدار خاص باشد.

داده‌ها و پارامترهای مسئله:

- داده‌های مورد استفاده برای این مسئله شامل لیستی از خوراکی‌ها (و یا object)های مسئله به همراه وزن موجود و ارزش کل آن‌ها است، که ازین داده‌ها به عنوان objectهایی که میخواهیم در کوله‌پشتی قرار بدهیم استفاده میکنیم. این داده‌ها به صورت یک فایلcsv به مسئله داده میشود.
- پارامترهای ورودی مسئله عبارت اند از: *حداکثر وزن مجاز، حداقل ارزش مجاز، بازه تنوع خوراکی‌های انتخاب شده.*

همچنین همانطور که در ادامه توضیح میدهیم دو ورودی سایز جمعیت اولیه و حداکثر تعداد ران کردن برنامه برای رسید به جواب نیز به عنوان ورودی‌های مسئله در نظر گرفته شده اند.

پیاده سازی گام به گام پروژه:

در ادامه به توضیح هر گام از پروژه به همراه علت انجام آن و تحلیل خروجی توابع صدا زده در هر گام میپردازیم:

برای تست برنامه از همان مثال داده شده استفاده شده است.(ران شده آن به صورت فایل HTML به نام success-12_10_2_4 در کنار گزارش کار پیوست شده است.)

بخش صفر: مروری بر ورودی ها، کتابخانه ها و داده های ضروری:

تعریف کتابخانه‌های مورد نیاز:

در این پروژه به چهار کتابخانه مهم نیاز داریم:

1. Pandas

2. Numpy
3. Random
4. matplotlib.pyplot

```
import pandas as pd
import numpy as np
import random as random
import matplotlib.pyplot as plt
```

در ادامه علت استفاده از هریک از کتابخانه ها را بیان میکنیم.

تعریف ورودی‌ها:

همانطور که اشاره شد، لازم است برای مسئله یک سری محدودیت‌ها قرار بدهیم، که آنها را به صورت ورودی‌های مسئله در نظر میگیریم:

1. min_v: حداقل ارزش قابل قبول
2. max_w: حداکثر وزن قابل قبول
3. min_n: حداقل تنوع قابل قبول
4. max_n: حداکثر تنوع قابل قبول

```
min_v = float(input("Enter the minimum value (min_v) you want to achieve: "))
max_w = float(input("Enter the maximum weight (max_w) allowed: "))

min_n = int(input("Enter the minimum number of snack types (min_n): "))
max_n = int(input("Enter the maximum number of snack types (max_n): "))
```

در ادامه دو ورودی دیگر نیز دیگر تعریف کرده ایم:

1. Population_size: با توجه به اینکه نمیخواهیم الگوریتم را تنها برای یک سری محدودیت و تنها یک لیست از خوراکی ها اجرا کنیم، ثابت در نظر گرفتن این متغیر میتواند در بعضی موارد موجب استفاده بیش از اندازه از فضا شود و در برخی موارد که محدودیت‌ها و یا تنوع خوراکی‌ها بسیار است ممکن از اندازه کم جمعیت موجب اجرای بیش از حد برنامه باشد، حتی اگر برنامه به صورتی نوشته شود که تنوع بسیاری از جمعیت ها را ایجاد کند، اما باتوجه به ثابت ماندن اندازه جمعیت باز هم نیاز به اجراهای طولانی مدت ویا بیشمار میشود.
مقداری که به بنظر میاد خوب باشه که درنظر بگیریم برای جمعیت باید متناسب با فضای جواب، طول کروموزوم ها باشد، که باتوجه به بزرگ بودن فضای حالت حتی برای تعداد خوراکی کم بسیار است، اندازه جمعیت را به صورت زیر انتخاب میکنیم:

Number_of_snacks * max_n *(max_n – min_n + 1)

البته ازین مقدار گاها میتواند بسیار کمتر باشد برای همین به صورت ورودی در نظر گرفته ایم.

- در صورتی که لازم باشد این مقدار را ثابت در نظر بگیرم، میتوان مسئله را با میزان محدودیت‌های متوسط و تنوعی از خورکی‌ها در حدود 20 تا در نظر گرفت؛ در این صورت جمعیت را 200 تایی در نظر میگیریم. جمعیت با این تعداد جمعیت مناسبی برای تعداد اجرای مناسب میباشد.

```
population_size = 200 #if it shouldn't be considered as an input!
```

```
population_size = int(input("Enter population size: "))
```

2. max_run: همانطور که قبلتر اشاره کردیم، نمیخواهیم الگوریتم را تنها برای یک سری محدودیت و تنها یک لیست از خوراکی ها اجرا کنیم، ثابت در نظر گرفتن این متغیر میتواند در بعضی موارد ممکن از کمتر از میزان اجرا لازم برای رسیدن به جواب ایده‌آل برای مسئله ای با جمعیت اولیه کم و یا تنوع خوراکی بالا و یا محدودیت های فراوان بشود، حتی اگر برنامه به بهترین نحو ممکن نوشته شود، و در مواردی که جمعیت بسیار باشد و محدودیت‌ها به گونه‌ای باشند، که مسئله جواب نداشته باشد، موجب اجراهای طولانی مدت و با تعداد بیش از اندازه میباشد و در این موارد که مسئله جواب نداشته باشد، اگر شرط پایان را تعریف نکرده باشیم مسئله تا بینهایت بار اجرا شود. که این عمر در تناقض با اصلی و علت روی‌آوردن به الگوریتم‌های ژنتیک است، چرا که یکی از اهداف استفاده از این الگوریتم‌ها محدودیت زمانی است که داریم.

- در صورتی که لازم باشد این مقدار را ثابت بگیریم، با توجه به همان شروط گفته شده(برای population_size) میتوان این مقدار را 1000 گرفت که البته برنامه بیشتر از تعداد بار های لازم نیز حتی اجرا میشود.

```
max_run = 1000 #if it shouldn't be considered as an input!
```

```
max_run = int(input())
```

در اخر نیز برای بررسی نهایی مقادیر تعریف شده پرینت شده اند.

```
print(f"min value: {min_v},\nmax weight: {max_w},\nrange: {min_n}-{max_n},\npopulation size: {population_size},\nmax run: {max_run}")
```

```
min value: 12.0,
max weight: 10.0,
range: 2-4,
population size: 10,
max run: 200
```

*** یک راه مناسب این است به مرور تعداد اجراها را افزایش بدهیم.

خواندن **CSV** فایل و ذخیره داده ها در دیتافرم:

در ادامه به کمک کتابخانه pandas که قبلتر، import کرده ایم، داده‌ها را از فایل snacks.csv خوانده و در یک دیتافرم ذخیره میکنیم؛ سپس برای بررسی نهایی یک بار دیتافرم را پرینت کرده ایم:

```
file_path = 'snacks.csv'
df = pd.read_csv(file_path)
```

```
print(df)
```

```
           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
```

بخش یک: مشخص کردن مفاهیم اولیه:

در الگوریتم‌های ژنتیک باید دو تعریف ارائه بدهیم، یک تعریف برای ژن که سپس به کمک آن، بتوان یک کروموزوم ساخت.

- هر کروموزوم مجموعه از ژن‌ها است و این مجموعه(کروموزوم)، یک راه حل پیشنهادی برای مسئله مورد نظرمان میباشد.
- در الگوریتم‌های ژنتیک ممکن است فضای حالت بسیار بزرگ باشد و پیدا کردن شرطی که تمام محدودیت‌ها را برقرا ر سازد بسیار دشوار است، در نتیجه باید اکثر کارها را با استفاده از تصادفی کردن وقایع انجام دهیم و همچنین تعریف کروموزوم‌ها دارای اهمیت ویژه‌ای میشود و باید به گونه‌ای باشد که اعمال توابع مختلفی از جمله تابع تناسب بر روی آن فراهم باشد.

در ادامه به تعریف ژن و سپس کروموزوم میپردازیم:

ژن:

برای تعریف، کلاس Gene را تعریف کرده ایم که بیانگر یک نوع خوراکی به همراه وزن انتخاب شده از آن است که در کل چهار ویژگی دارد:

1. name: نام
2. weight: وزن (مقدار) انتخاب شده

علاوه بر دو ویژگی بالا که لازم هستند، برای راحتی محاسبات دو ویژگی دیگر نیز تعریف کرده ایم که میتوانستیم تعریف نکنیم و در ازای فضای کمتر نیاز به تعریف ها و محسبات بیشتر در ادامه را ایجاد میکند:

3. max_weight: بیشتر وزن مجاز برای این ژن در یک کروموزوم (مقدار موجود از خوراکی)
4. value_per_weight: ارزش کروموزوم به ازای وزن (برای بهبود جمعیت این معیار بسیار کمک کننده است.)

در اخر برای پرینت اطلاعات ژن (چه مستقیم و چه در یک کروموزوم دو تابع تعریف کردیم که تنها مشخصات مورد نیازمان را برای ژن مربوطه در قالب مناسب Return میکنند.

```python
class Gene:
    def __init__(self, name, max_weight, value_per_weight, weight = 0):
        self.name = name
        self.weight = weight
        self.max_weight = max_weight
        self.value_per_weight = value_per_weight

    def __repr__(self):
        return f"Gene(name={self.name}, weight={self.weight}, max_weight={self.max_weight}, value_per_weight={self.value_per_weight}\n)"

    def __str__(self):
        return f"{self.name}: Weight={self.weight}, Value per Weight={self.value_per_weight}"
```

کروموزوم:

برای تعریف کروموزم از کلاس Chromosome استفاده کرده ایم که دارای ویژگی های زیر میباشد:

1. genes: لیستی از ژن های کروموزوم که هنگام تعریف کروموزوم به صورت رندوم این لیست را میدهیم و درصورتی که کروموزوم متعلق به نسل اول باشه، وزن ژن های انتخاب شده رو به صورت رندوم مشخص میکنیم و در غیر این صورت لیستی از ژن ها با وزن هایی متناسب به parents کروموزوم به آن داده میشود.

باقی ویژگی ها را میتوان هرموقع نیاز داشتیم، محاسبه کنیم، اما در حال حاضر برای راحتی کار فضای بیشتری مصرف کردیم که لزوما کار مناسبی نیست و ویژگی های زیر را در همان کلاس کروموزوم تعریف کرده ایم:

2. total_weight: که مجموع وزن های کروموزم است و ما نیاز داریم این مقدار را به مرور کمتر کنیم تا وزن کروموزوم کمتر از max_w شود.

3. total_value: که مجموع ارزش ژن های کروموزوم است و ما میخواهیم به مرور این مقدار را افزایش بدهیم تا از min_v بیشتر شود.
4. variety_of_snacks: که باید در یک بازه خاص قرار بگیرد.
5. fitness: که معیاری است برای سنجش کروموزوم ها و نسل ها.(در ادامه نحوه محاسبه آن را گفته ایم.)

در اخر برای پرینت اطلاعات کروموزوم(چه مستقیم و چه در یک جمعیت دو تابع تعریف کردیم که تنها مشخصات مورد نیازمان را برای کرووزوم مربوطه در قالب مناسب Return میکنند.

```
class Chromosome:
    def __init__(self, geness, new_born = False):
        self.genes = []
        for g in geness:
            self.genes.append(Gene(g.name, g.max_weight, g.value_per_weight, g.weight))
        if (new_born == True):
            for gene in self.genes:
                while gene.weight == 0:
                    gene.weight = random.uniform(0, gene.max_weight)
        self.total_weight = sum(gene.weight for gene in self.genes)
        self.total_value = sum(gene.weight * gene.value_per_weight for gene in self.genes)
        self.variety_of_snacks = len(self.genes)
        self.fitness = 0


    def __repr__(self):
        return f"\nChromosome(genes=\n{self.genes}, fitness={self.fitness}) \nTotal Weight: {self.total_weight}\nTotal Value: { self.total_value}\nRange: {self.variety_of_snacks}\n{"---"*25} \n"

    def __str__(self):
        genes_str = '\n'.join(str(gene) for gene in self.genes)
        return f"\nChromosome Details:\nGenes:\n{genes_str}\nFitness: {self.fitness} \nTotal Weight: {self.total_weight}\nTotal Value: { self.total_value}\nRange: {self.variety_of_snacks}\n{"---"*25} \n"
```

مخزنی از ژن‌ها:

با توجه به ویژگیای جدید که برای ژن تعریف کرده ایم، از جمله Value per Weigth، یک دیتافرم جدید از ژن ها ایجاد میکنیم که تنها شامل ویژگی هایی باشد که برای تعریف یک ژن نیاز داریم:

```
genes_pool = df.copy()
genes_pool.rename(columns={'Snack': 'Name'}, inplace=True)
genes_pool.rename(columns={'Available Weight': 'Maximum Weight'}, inplace=True)
genes_pool['Value per Weight'] = genes_pool['Value'] / genes_pool['Maximum Weight']
genes_pool = genes_pool[['Name', 'Maximum Weight', 'Value per Weight']]
```

```
genes_pool
```

	Name	Maximum Weight	Value per Weight
0	MazMaz	10	1.000000
1	Doogh-e-Abali	15	0.666667
2	Nani	5	1.000000
3	Jooj	7	2.142857
4	Hot-Dog	20	0.750000
5	Chips	8	0.750000
6	Nooshaba	12	0.666667
7	Shokolat	6	1.166667
8	Chocoroll	9	1.333333
9	Cookies	11	1.000000
10	Abnabat	4	1.000000
11	Adams-Khersi	14	0.642857
12	Popcorn	16	0.812500
13	Pastil	3	2.333333
14	Tordilla	10	0.900000
15	Masghati	5	1.200000
16	Ghottab	7	1.428571
17	Saghe-Talaei	9	1.222222
18	Choob-Shoor	13	0.923077

سپس به کمک این دیتافرم جدید لیستی از ژنهای اولیه ممکن برای کروموزوم ها تعریف کرده ایم تا بعد به صورت رندوم از این لیست تعداد ژن برای تعریف یک کروموزوم استفاده کنیم:

```
gene_objects = [Gene(row['Name'], row['Maximum Weight'], row['Value per Weight']) for index, row in genes_pool.iterrows()]
```

```
gene_objects
```

```
[Gene(name=MazMaz, weight=0, max_weight=10, value_per_weight=1.0
 ),
 Gene(name=Doogh-e-Abali, weight=0, max_weight=15, value_per_weight=0.6666666666666666
 ),
 Gene(name=Nani, weight=0, max_weight=5, value_per_weight=1.0
 ),
 Gene(name=Jooj, weight=0, max_weight=7, value_per_weight=2.142857142857143
 ),
 Gene(name=Hot-Dog, weight=0, max_weight=20, value_per_weight=0.75
 ),
 Gene(name=Chips, weight=0, max_weight=8, value_per_weight=0.75
 ),
 Gene(name=Nooshaba, weight=0, max_weight=12, value_per_weight=0.6666666666666666
 ),
 Gene(name=Shokolat, weight=0, max_weight=6, value_per_weight=1.1666666666666667
 ),
 Gene(name=Chocoroll, weight=0, max_weight=9, value_per_weight=1.3333333333333333
 ),
 Gene(name=Cookies, weight=0, max_weight=11, value_per_weight=1.0
 ),
 Gene(name=Abnabat, weight=0, max_weight=4, value_per_weight=1.0
 ),
 Gene(name=Adams-Khersi, weight=0, max_weight=14, value_per_weight=0.6428571428571429
 ),
 Gene(name=Popcorn, weight=0, max_weight=16, value_per_weight=0.8125
 ),
 Gene(name=Pastil, weight=0, max_weight=3, value_per_weight=2.3333333333333335
 ),
 Gene(name=Tordilla, weight=0, max_weight=10, value_per_weight=0.9
 ),
 Gene(name=Masghati, weight=0, max_weight=5, value_per_weight=1.2
 ),
 Gene(name=Ghottab, weight=0, max_weight=7, value_per_weight=1.4285714285714286
 ),
 Gene(name=Saghe-Talaei, weight=0, max_weight=9, value_per_weight=1.2222222222222223
 ),
 Gene(name=Choob-Shoor, weight=0, max_weight=13, value_per_weight=0.9230769230769231
 )]
```

بخش دو: تولید جمعیت اولیه:

پس از تعریف مفاهیم اولیه لازم است نسل اول (جمعیت اولیه) را نیز به صورت کاملا تصادفی ایجاد کنیم.

اندازه این جمعیت در صورتی که بسیار کم و یا بسیار زیاد باشد مشکلاتی ایجاد میکند (که برای جلوگیری از طولانی تر شدن گزارشکار در بخش پرسش ها علت این موضوع را بیان کرده ایم.) همچنین همانطور قبلتر اشاره کرده ایم تعریف اندازه جمعیت میتواند بسیار وابسته به میزان محدودیت های مسئله و یا محدودیت های زمانی یا فضا و یا حتی تنوع ژن ها باشد و تعریف اندازه آن به صورت ثابت برای تمامی تست کیس ها نمیتواند کار معقولی باشد و ما آن را به صورت ورودی مسئله تعریف کرده ایم (البته به صورت پیشفرض آن را 200 گرفته ایم.)

برای تعریف نسل اولیه به اندازه population_size کروموزوم های متنوع با انتخاب رندوم از ژن ها (به تعدادی که تنوع ژن ها در بازه تعریف شده از قبل قرار بگیرد.) ایجاد میکنیم. (new_bornبودن این کروموزوم ها True است که منجر میشود ژن های کروموزوم به صورت رندوم در هنگام تعریفِ کروموزوم یک وزن رندوم بگیرند.)

در اخر یک بار نسل اولیه را پرینت کرده ایم که به علت طولانی نشدن گزارش فقط قسمتی از آن را نمایش میدهیم:

```
initial_population = []
for _ in range(population_size):
    x = random.randint(min_n, max_n)
    genes_temp = random.sample(gene_objects, x)
    c = Chromosome(genes_temp, True)
    initial_population.append(c)
    c = None
```

```
initial_population
```

```
[
 Chromosome(genes=
 [Gene(name=Adams-Khersi, weight=5.976563821670894, max_weight=14, value_per_weight=0.6428571428571
429
 ), Gene(name=MazMaz, weight=0.5601905110966576, max_weight=10, value_per_weight=1.0
 ), Gene(name=Pastil, weight=2.7507211326529255, max_weight=3, value_per_weight=2.3333333333333335
 )], fitness=0)
 Total Weight: 9.287475465420476
 Total Value: 10.820616563122869
 Range: 3
 ---------------------------------------------------------------------------- ,

 Chromosome(genes=
 [Gene(name=Cookies, weight=8.817288078196738, max_weight=11, value_per_weight=1.0
 ), Gene(name=Abnabat, weight=3.628692899318139, max_weight=4, value_per_weight=1.0
 ), Gene(name=Jooj, weight=5.832040604365799, max_weight=7, value_per_weight=2.142857142857143
 )], fitness=0)
 Total Weight: 18.278021581880676
 Total Value: 24.943210844013016
 Range: 3
 ---------------------------------------------------------------------------- .
```

بخش سه: پیاده سازی و مشخص کردن تابع معیار سازگاری:

پس از تولید جمعیت اولیه، نیاز داریم تا تابع معیاری تعریف کنیم که بتواند برای شناسایی کروموزوم های برتر که شرایط و محدودیت های مسئله را بهتر مدل می کنند استفاده شود. ابتدا یک تعریف مناسب برای این تابع بیان و سپس به پیاده سازی آن میپردازیم، در ادامه به کمک این تابع سعی میکنیم تابعی برای نمایش میزان سازگاری و مناسب بودن جمعیت بیان کنیم که بعد کمک آن بتوانیم عملکرد الگوریتم خود را بررسی کنیم و درنهایت تابعی برای پیدا کردن بهترین جواب موجود و تعیین کروموزوم برنده تعریف میکنیم:

تعریف تابع معیار و fitness:

برای تعریف نحوه محاسبه fitness، چندین معیار و طراحی میتونستیم استفاده کنیم:

1. اولین معیاری که میتوانستیم اضافه کنیم، total value / total weight یک کروموزوم است که چرا که هدف نهایی ما کم کردن وزن و بیشتر کردن ارزش یک کرروموزوم یا همان کوله پشتی است.
2. معیار قبلی به محدودیت تنوع غذایی نپرداخته است و تلاش دارد بیشترین وزن را به کمترین تعداد تنوع خوراکی هایی که بیشترین value per weight را دارند بدهد و ممکن است نتواند به باقی خوراکی ها شانس خوبی بدهد چرا که ما لازم

داریم تنوع خوراکی هم داشته باشیم که گاها نیاز است از خوراکی های کم ارزش هم برداریم تا تنوع داشته باشیم و این باعث میشود fitness کمتر شود و گمان برود الگوریتم به خوبی کار نمیکند.
معیار دومی که میتوانیم تعریف کنیم به این صورت است که در صورتی وزن بیشتر از min_w بود fitness عدد منفی min_w – total_weight باشد و در غیر این صورت fitness برابر با total_value میشود.

در این پروژه ما از معیار دوم استفاده کرده ایم:

```
def calculate_fitness(chromosome):
    fitness = chromosome.total_value
    penalty_weight =max_w - chromosome.total_weight
    penalties = 0
    if penalty_weight < 0:
        penalties = penalty_weight
    if penalties < 0 :
        fitness = penalties
    return fitness
```

در اخر نیاز داریم تابعی بنویسیم که برای کل کروموزم های نسل، fitness را اپدیت کند و سپس همین کار را برای نسل اولیه انجام میدهیم و خروجی نسل را بررسی میکنیم تا از عملکرد توابع تعریف شده مطمئن شویم:

```
def update_fitness(population):
    for chromosome in population:
        chromosome.fitness = calculate_fitness(chromosome)
    return population
```

```
initial_population = update_fitness(initial_population)
```

```
initial_population
```

```
[
 Chromosome(genes=
 [Gene(name=Adams-Khersi, weight=5.976563821670894, max_weight=14, value_per_weight=0.6428571428571
429
 ), Gene(name=MazMaz, weight=0.5601905110966576, max_weight=10, value_per_weight=1.0
 ), Gene(name=Pastil, weight=2.7507211326529255, max_weight=3, value_per_weight=2.3333333333333335
 )], fitness=10.820616563122869)
 Total Weight: 9.287475465420476
 Total Value: 10.820616563122869
 Range: 3
 ---------------------------------------------------------------------------- ,

 Chromosome(genes=
 [Gene(name=Cookies, weight=8.817288078196738, max_weight=11, value_per_weight=1.0
 ), Gene(name=Abnabat, weight=3.628692899318139, max_weight=4, value_per_weight=1.0
 ), Gene(name=Jooj, weight=5.832040604365799, max_weight=7, value_per_weight=2.142857142857143
 )], fitness=-8.278021581880676)
 Total Weight: 18.278021581880676
 Total Value: 24.943210844013016
 Range: 3
 ---------------------------------------------------------------------------- ,
```

نمایش سازگاری جمعیت:

برای بررسی هر نسل و عملکرد الگوریتم نیاز است که اطلاعاتی مفید و خلاصه از جامعه داشته باشیم که به کمک آنها بتوانیم نسل ها را مقایسه کنیم، در اینجا چهار ویژگی را انتخاب کرده ایم که مهم ترین آنها Avg Fitness و Max Fitness هر نسل میباشد که به ترتیب میانگین و ماکسیسم fitness های موجود در نسل را به ما نشان میدهد، اما جهت بررسی دو معیاری که میتوانستیم برای Fitness داشته باشیم از Avg Value per Weight و Max Value per Weight نیز استفاده کرده ایم که نه تنها دو معیاری که میتوانسیتم استفاده کنیم را مقایسه کنیم بلکه با ترکیبی از این ها میتوان دید بهتری نسبت به نه تنها جمعیت بلکه به الگوریتم داشته باشیم. (الگوریتم خوب هر دو معیار رو تا حد مناسبی باید در سطح جامعه افزایش بدهد.)

```python
def print_population_compatibility(population):
    print("Population compatibility:")
    print(f"Avg Fitness: {sum(c.fitness for c in population) / len(population)}")
    print(f"Max Fitness: {max(c.fitness for c in population)}")
    print(f"Avg Value per Weight: {sum(c.total_value / c.total_weight for c in population) / len(population)}")
    print(f"Max Avg Value per Weight: {max(c.total_value / c.total_weight for c in population)}")
    print("--" * 25)
```

```python
print_population_compatibility(initial_population)
```

```
Population compatibility:
Avg Fitness: 1.9847199128673623
Max Fitness: 10.907907922844776
Avg Value per Weight: 1.1099622953343615
Max Avg Value per Weight: 1.36465594661184
--------------------------------------------------
```

یافتن بهترین جواب و درصورت وجود اعلام برنده:

در اخر نیاز است تابعی بنویسیم که بهترین کروموزوم موجود (با بیشترین fitness) را پیدا کند و سپس به کمک این تابع دو تابع دیگر مینویسم که اگر بهترین جواب، تمام محدودیت ها را پاسخگو بود، جواب اخر برگدانده شود و در غیر این صورت جوابی بر گردانده نشود و همچنین یک تابع برای نوشتن اطلاعات مورد نظر به فرمت خواسته شده برای جواب نهایی پیاده سازی کرده ایم:

```python
def find_winner(population):
    max_fitness = float('-inf')
    winner = population[0];
    for chromosome in population:
        if (chromosome.fitness > max_fitness):
            max_fitness = chromosome.fitness
            winner = chromosome
    return winner
```

```python
def check_for_answer(population):
    winner = find_winner(population)
    if winner.fitness >= min_v:
        return winner
    return None
```

```python
def print_winner(winner):
    for Gene in winner.genes:
        print(f"{Gene.name}: {Gene.weight}")
    print(f"Total Weight: {winner.total_weight}")
    print(f"Total Value: {winner.total_value}")
```

بخش چهار: پیاده سازی **crossover** و **mutation** و تولید نسل بعدی:

در نهایت برای اینکه به یک پاسخ مطلوب از مسئله نزدیک شویم، نیاز است که در هر نسل، جمعیت جدیدی با استفاده از جمعیت نسل قبل آن تولید گردد. برای این کار، باید از روش های crossover و mutationاستفاده گردد.

تعریف تابع "انتخاب یک موجودیت از بین دو موجودیت با احتمال معین":

قبل از پرداختن به پیاده سازی و توضیحات دو تابع crossover وmutation لازم است که یک تابع تعریف کنیم که از میان دو چیز با احتمال p یکی از آنها را انتخاب کند، که در ادامه علت تعریف این تابع و استفاده اش در دو تابع mutation و crossover خواهیم دید:

```python
def decide_with_probability(p, thing1, thing2):
    if random.random() < p:
        return thing1
    else:
        return thing2
```

تعریف تابع crossover:

تابع crossover بر روی دو کروموزوم اعمال می شود، و آن ها را ترکیب می کند تا به کروموزوم هایی از ترکیب آن دو که در حالت ایده آل بهترین ویژگی های دو ژن اولیه را دارند برسد.

این تابع سعی شده بسیار ساده نوشته شود، همانطور که قبلتر مشاهده کرده ایم، هنگام تولید یه نسل اولیه کروموزم هایی تولید کرده ایم که تعداد تنوع خوراکی آنها در یک بازه خواسته شده قرار دارد، در ادامه نیز سعی میکنیم همین مورد را رعایت کنیم و برای تولید نسل جدید به کمک پدران آنها، تنها یک ژن از پدران را تغیر میدهیم و از پدر دیگر منتقل میکنیم، به شرطی که این ژن موجب تکرار نام یک ژن یا یک خوراکی در فرزند نشود.

در نهایت به کمک تابع decide_with_probabitity نسل بهتر را با احتمال بالاتر به نسل بعد میبریم و درصورتی نسل جدید بهتر نبود نسل پدران را با احتمال بالاتری به نسل بعد میبریم.

```python
def crossover(prob , parent1, parent2, min_n, max_n):
    x = min(len(parent1.genes), len(parent2.genes))
    if x <= 1:
        return [parent1, parent2]
    if parent1 == parent2:
        return [parent1, parent2]

    child1 = Chromosome(parent1.genes, False);
    child2 = Chromosome(parent2.genes, False);

    possible1 = [g for g in child1.genes if g.name not in [k.name for k in child2.genes]]
    possible2 = [g for g in child2.genes if g.name not in [k.name for k in child1.genes]]
    if min(len(possible1), len(possible2)) >= 1 :
        i = random.randint(0, min(len(possible1), len(possible2)) - 1)
        child1.genes.remove(possible1[i])
        child2.genes.remove(possible2[i])
        child1.genes.append(possible2[i])
        child2.genes.append(possible1[i])

    def select_new_generation(prob, child1, child2, parent1, parent2):
        if (max_w - child1.total_weight - child2.total_weight > max_w - parent1.total_weight - parent2.total_weight):
            return decide_with_probability(prob,  [parent1, parent2],  [child1, child2])
        return decide_with_probability(prob, [child1, child2], [parent1, parent2])

    return select_new_generation(prob, child1, child2, parent1, parent2)
```

در اخر نیز یک تابع نوشته ایم تا یک نسل پیشنهادی (و نه نهایی) به کمک crossover روی نسل جدید ایجاد کند.

نکته قابل اهمیت این است هر بار باید جمعیت اولیه را Shuffle کنیم تا جفت پدران متفاوتی هر بار انتخاب شود و این تنوع پدران را در این تابع و نه تابع crossover هندل کرده ایم:

```python
def generate_new_population_crossover(prob, population):
    np.random.shuffle(population)
    new_population = []
    for i  in range(0, round((population_size)/2)):
        parent1 = population[i]
        parent2 = population[-i]
        new_generation = crossover(prob , parent1, parent2, min_n, max_n)
        new_population.append(new_generation[0])
        new_population.append(new_generation[1])
    return new_population
```

تعریف تابع **mutation**:

تابع mutation بر روی یک کروموزوم اعمال می شود، و آن را mutation و یا تغییر می دهد؛ به این امید که بتواند به کروموزوم بهتری جهش پیدا کند. می توانید درصد معقولی از ژن های برتر را نیز برای انتقال مستقیم به نسل های آینده در نظر بگیرید. (این درصد را بعدا تعریف کرده ایم)

تابع mutation که در این مسئله خاص (استثناءً) نقش پر اهمیت تری دارد به این صورت نوشته است که هر بار برای تمام کروموزم ها آن را اجرا میکنیم و در نهایت با احتمال prob_m کروموزوم ایجاد شده را به جا پدر قرار میدهیم(همه نسل نباید جهش داشته باشند) . میتوانستیم به صورت رندوم نیز به جای اعمال تابع روی تمام کروموزوم ها تعدادی از انها را انتخاب کنیم و سپس با احتمال prob_m فرزند را به جای پدر قرار بدهیم، اما با توجه به اینکه این تابع سعی شده تا حد بسیار خوبی نسل را بهبود ببخشد، روش دوم باعث نیاز به تعداد اجراهای بیشتر برای یافتن نسل جدید میشود.

*** توجه شود این تابع به صورت ادغام شده تابع mutation روی یک کروموزوم و تابع اجرای mutation بر روی کروموزوم های متعدد و ایجاد نسل جدید است.

توضیح نحوه اعمال mutation بر روی یک کروموزوم:

در این تابع سعی میشود هر بار کم ارزش ترین ژن را حذف کنیم و به جای آن یک ژن پر ارزش تر قرار بدهیم و در صورتی که پر ارزش ترین ژن ها در این کروموزوم وجود داشتند تلاش میشود وزن کروموزوم را درصورتی که بیش از اندازه است، را با کاهش وزن کم ارزش ترین ژن کمتر کنیم و در غیر این صورت برای بهبود value کروموزوم وزن ژن پر ارزش تر را افزایش بدهیم.

در صورتی کروموزم شامل پر ارزش ترین ژن ها نبود، میتوانیم یا در صورت امکان یا تعداد ژن ها را کم تر کنیم با حذف کم ارزش ترین ژن یا (درصورتی که کمتر از min_n نشود) و یا تعداد ژن ها را بیشتر کنیم که این مورد با احتمال کمتری انجام میشود چرا که به طور معمول داشتن کمترین تعداد از پر ارزش ترین خوراکی ها میتواند با وزن کمتر ارزش بیشتری به همراه بیاورد.

```python
def mutation(prob_m, population):
    np.random.shuffle(population)
    new_population = []
    for i  in range(0, population_size - 1):
        parent = population[i]
        genes = []
        for g in parent.genes:
            genes.append(g)

        genes.sort(key=lambda x: x.value_per_weight)

        pre_gene = genes.pop(0)

        sorted_genes = gene_objects
        sorted_genes.sort(key=lambda x: x.value_per_weight, reverse=True)

        new_gene = pre_gene

        flag1 = False
        flag2 = True
        p = False
        for g in sorted_genes:
            if pre_gene.name == g.name:
                p = True
                genes.append(pre_gene)
                break
            if any(x.name == g.name for x in genes) == True:
                continue
            else:
                new_gene = g
                break
```

```python
        if p:
            n = random.randint(0, len(genes) - 1)
            if (parent.total_weight > max_w):
                genes[n].weight = random.uniform(0, genes[n].weight)
                while genes[n].weight == 0:
                    genes[n].weight = random.uniform(0, genes[n].weight)
            else:
                genes[n].weight = random.uniform(genes[n].weight, genes[n].max_weight)

        else:
            if parent.variety_of_snacks > min_n:
                flag1 = True
            if pre_gene.weight == pre_gene.max_weight:
                flag2 = True
            if flag1:
                if random.random() > 0.1:
                    genes.append(new_gene)
                else:
                    if len(genes) < min_n:
                        genes.append(new_gene)
            else:
                genes.append(new_gene)
            if flag2:
                not_in_parent = [obj for obj in sorted_genes if obj.name not in [objj.name for objj in parent.genes]]
                if random.random() > 0.9 and len(not_in_parent) != 0:
                    new_new = random.choice(not_in_parent)
                    new_new.weight = random.uniform(0, new_new.max_weight)
                    if any(x.name == new_new.name for x in genes) == False and len(genes) < max_n:
                        genes.append(new_new)
            for g in genes:
                if g.name == new_gene.name:
                    g.weight = random.uniform(min(g.max_weight, pre_gene.weight), new_gene.max_weight)
                    break
        if len(genes) == 0:
            return population
        child = Chromosome(genes, False)
        #print(f"parent: {parent}")
        #print (f"child: {child}")
        #print("__" * 10)
        genes.clear()
        new_generation = decide_with_probability(prob_m, parent, child)
        new_population.append(new_generation)
    return new_population
```

تعریف تابع **generate_new_generation**:

در اخر لازم است بتوانیم با احتمال های prob_c و prob_m که به ترتیت برای crossover و mutation استفاده میشوند، نسل جدید را با توابع generate_new_population_crossover و mutation ایجاد کنیم و در اخر نیز fitness کروموزوم ها را اپدیت کنیم که همه این کار ها را در یک تابع generate_new_geneation انجام میدهیم:

```python
def generate_new_geneation(population, prob_c, prob_m):
        new_population = population
        new_population = generate_new_population_crossover(prob_c, new_population)
        new_population = mutation(prob_m, new_population)
        new_population = update_fitness(new_population)
        return new_population
```

بخش پنج: ایجاد الگوریتم ژنتیک روی مسئله:

در نهایت لازم است با استفاده از توابع بالا الگوریتمی ارائه بدهیم که نسل های مختلفی را ایجاد کند و به دنبال جواب بگردد، به این منظور تابع genetic_algorithm را تعریف کرده ایم:

این تابع پس از دریافت نسل اولیه، به دنبال جواب میگردد، درصورتی که جوابی وجود نداشت، تا زمانی که یا جواب مطلوبی را پیدا کند و یا به حداکثر تعداد ران های خود برسد، تابع generate_new_geneation را با احتمال ثابت prob_c = 0.25 که به صورت فرضی انتخاب شده و جواب های مناسبی هم برگردانده و احتمال متغیر prob_m صدا زده میشود.

احتمال prob_m به این دلیل ثابت نیست که در نسل های اولیه که بسیار رندوم و احتمالا نا مطلوب هستند، نیاز به جهش های بزرگتری داریم و در عین حال در نسل های اخری نیازی به جهش های فراوان نیست، در نتیجه این احتمال به تعداد اجراهای بیشتر، کمتر میشود اما این موضوع رو مشخص کرده ایم که کمتر از 0.1 نشود چرا که ممکن است 1000 بار برنامه را اجرا کنیم، و برای اجراهای اخر نمیخواهیم احتمال خیلی به صفر میل کند.

سپس پارامتر هایی که برای ارزیابی نسل ها و الگوریتم را نیاز داریم (که قبلتر با این 4 پارامتر اشنا شده ایم)را در 4 ارایه global نگه داری و ذخیره میکنیم. در اخر در صورت جواب مطلوب ok با تعداد اجرا ها را نمایش میدهیم در غیر این صورت اعلام میکنیم جوابی پیدا نشده است (fail) و در اخر جواب(درصورت پیدا شدن) و نسل نهایی را برمیگردانیم:

```
def genetic_algorithm(initial_population):
    initial_population = update_fitness(initial_population)
    population = initial_population
    winner = check_for_answer(population)
    counter = 0
    cur_fit = sum(c.fitness for c in population) / len(population)
    value_per_weight = sum(c.total_value / c.total_weight for c in population) / len(population)

    while winner == None and counter < max_run:
        fitness_arr.append(cur_fit)
        val_weight_arr.append(value_per_weight)
        max_fitness_arr.append(max(c.fitness for c in population))
        max_val_weight_arr.append(max(c.total_value / c.total_weight for c in population))
        counter = counter + 1
        prob_m = max (1 / (counter + 1), 1/10)
        prob_c = 0.25
        population = generate_new_geneation(population, prob_c, prob_m)
        cur_fit = sum(c.fitness for c in population)/ len(population)
        value_per_weight = sum(c.total_value / c.total_weight for c in population) / len(population)
        winner =check_for_answer(population)

    fitness_arr.append(cur_fit)
    val_weight_arr.append(value_per_weight)
    max_fitness_arr.append(max(c.fitness for c in population))
    max_val_weight_arr.append(max(c.total_value / c.total_weight for c in population))

    if winner != None:
        print(f"OK -#run: {counter + 1}")
    else:
        print("fail")
    return winner, population
```

بخش شش: ارزیابی نتایج:

این بخش به دو قسمت کلی تقسیم میشود:

1. اجرا و نمایش نتیجه نهایی
2. بررسی الگوریتم و روند بهبود نسل ها به همراه نمایش جزییات بیشتر:

اجرا برنامه و نمایش نتیجه نهایی:

ابتدا لازم است ارایه های مورد نیاز که به صورت global در توابع استفاده کرده ایم را تعریف و در نهایت به اجرا برنامه بپردازیم، سپس بهترین جواب ممکن را نمایش میدهیم:

```
fitness_arr = []
val_weight_arr = []
max_fitness_arr = []
max_val_weight_arr = []
winner, population = genetic_algorithm(initial_population)
```

```
OK -#run: 2
```

```
if winner != None:
    print_winner(winner)
else:
    print("No answer found:\n")

    print("The best possible answer: ")
    print(find_winner(population))
```

```
Abnabat: 3.628692899318139
Jooj: 5.832040604365799
Total Weight: 9.460733503683938
Total Value: 16.12592276581628
```

بررسی الگوریتم و روند بهبود نسل ها به همراه نمایش جزییات بیشتر:

در نهایت برای نمایش بهتر و بررسی راحت تر اطلاعات تو تابع تعریف میکنیم:

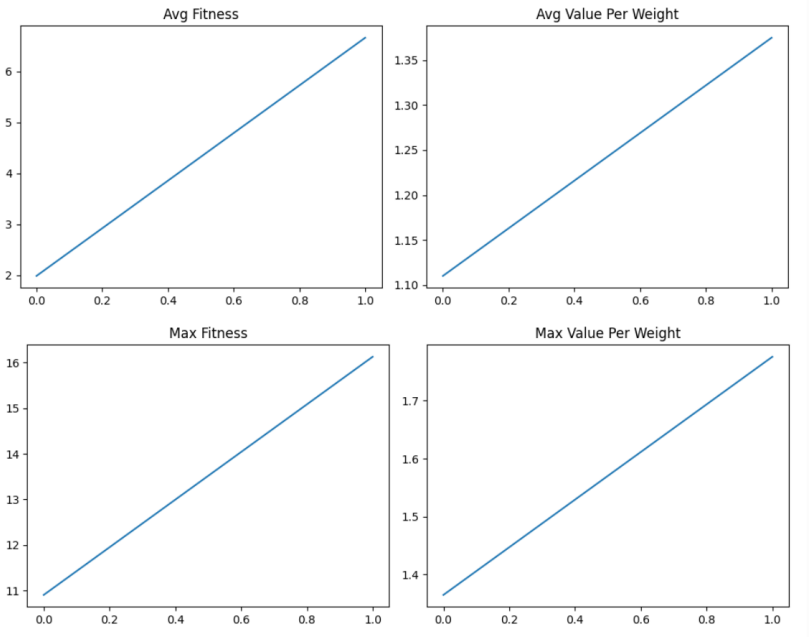
1. plot_details

از این تابع برای نمایش 4 نمودار استفاده میکنیم تا روند بهبود 4 ویژگی مهم در نسل ها را مشاهده کنیم که عبارت اند از میانگین و ماکسیمم fitness نسلها و میانگین و ماکسیمم total value / total weight کروموزوم ها یا همان value per weight کروموزوم ها.

```
def plot_details(val_weight_arr, fitness_arr, max_val_weight_arr, max_fitness_arr):
    def plot_detail(y, title, axes, i):
        x = [i for i in range(0, len(y))]
        axes[i].set_title(title)
        axes[i].plot(x,y)
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
    plot_detail(fitness_arr, "Avg Fitness", axes, 0)
    plot_detail(val_weight_arr, "Avg Value Per Weight", axes, 1)
    fig.tight_layout()
    plt.show()
    fig2, axes2 = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
    plot_detail(max_fitness_arr, "Max Fitness", axes2, 0)
    plot_detail(max_val_weight_arr, "Max Value Per Weight", axes2, 1)
    fig2.tight_layout()
    plt.show()
```

```
print(f"#run = {len(max_fitness_arr)}")
plot_details(val_weight_arr, fitness_arr, max_val_weight_arr, max_fitness_arr)
```

#run = 2



در این تست کیس با تنها دو اجرا جواب پیدا شده و خروجی به صورت صعودی است که نشانه خوبی است(اما لزوما با این تک تست کیس ساده نمیتوان الگوریتم را بررسی کرد و این کار را در ادامه برای تست کیس های متنوع دیگر انجام میدهیم.)

2. در نهایت برای بررسی دستی نیز تابعی نوشته ایم که همان مقادیر را برای تمامی نسل ها پرینت کند:

```
def print_details(val_weight_arr, fitness_arr, max_val_weight_arr, max_fitness_arr):
    for i in range(0, len(fitness_arr)):
        print(f"i: {i}")
        print(f"avg fitness: {fitness_arr[i]}")
        print(f"max fitness: {max_fitness_arr[i]}")
        print(f"avg value per weight: {val_weight_arr[i]}")
        print(f"max value per weight: {max_val_weight_arr[i]}")
        print("_" * 50)
```

```
print_details(val_weight_arr, fitness_arr,  max_val_weight_arr, max_fitness_arr)
```

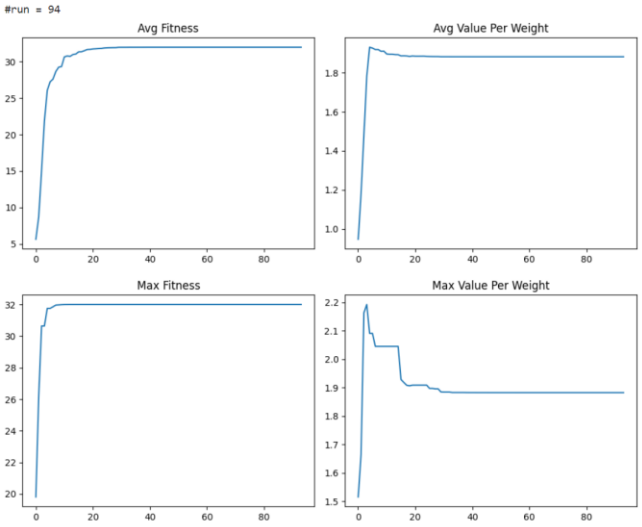
```
i: 0
avg fitness: 1.9847199128673623
max fitness: 10.907907922844776
avg value per weight: 1.1099622953343615
max value per weight: 1.36465594661184
__________________________________________________
i: 1
avg fitness: 6.661904581778223
max fitness: 16.12592276581628
avg value per weight: 1.3750185117872162
max value per weight: 1.7759548808108667
__________________________________________________
```

نتیجه گیری نهایی:

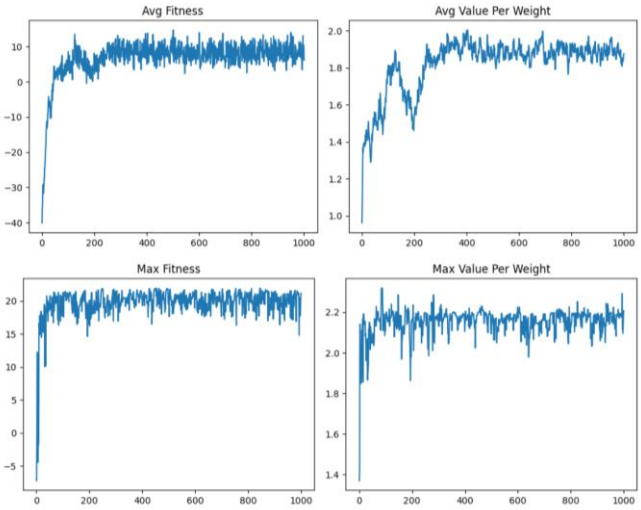
در ادامه چندین نمونه از خروجی های تست کیس های مختلف را بررسی کرده ایم تا صحت الگوریتم و کیفیت آن را مورد ارزیابی قرار بدهیم.

نکته قابل توجه این است صحت الگوریتم را با همیشه جواب را پیدا کردن نمیتوان فهمید چرا که در کل الگوریتم ژنتیک و الگوریتم های محلی لزوما به جواب رسیدن آنها اثبات نمیشود و تنها با یک سری ملاحظات باید کیفیت آن ها را بررسی کرد:

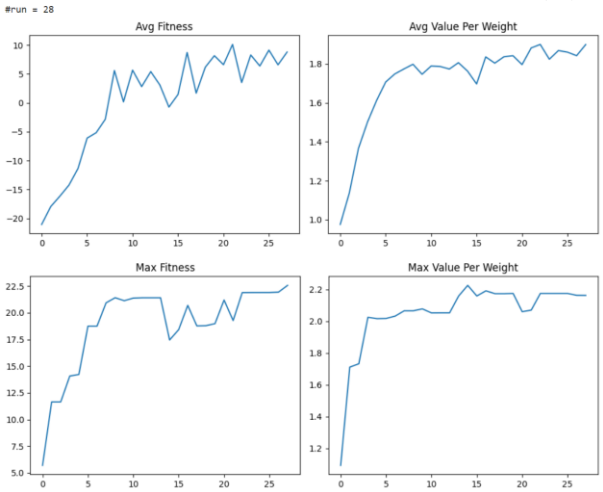
1. تست کیس 3_3_17_32: (جوابی وجود دارد و پیدا میکند.)



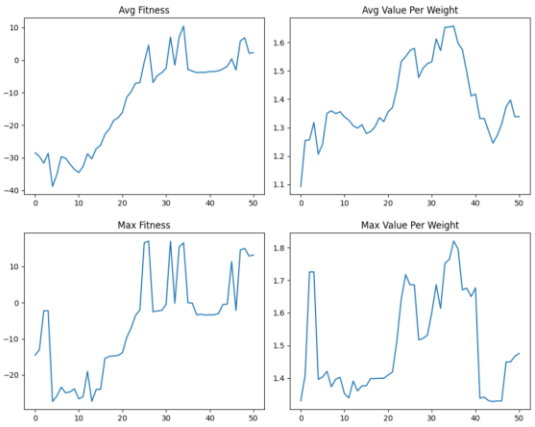
2. تست کیس 19_3_10_22:(جواب وجود نداشته)



3. تست کیس 22.5_10.5_3_10(جواب وجود داشته و پیدا کرده):



4. تست کیس 10_3_10.5_22.5(جواب وجود داشته ولی پیدا نکرده):
به علت هم جمعیت بسیار کوچک الگوریتم نتوانسته جواب را پیدا کند



در تمامی این اجراها میتواند عملکرد رندوم الگوریتم را مشاهده کرد.

همچنین علت برتری دومین معیار fitness به خوبی قابل مشاهده است. (چرا که هم پایدار تر است و هم معیار اول برای پیدا کردن جواب بهینه با range بیشتر که منجر به استفاده از تعداد بیشتر خوراکی با ارزش کمتر است، کاهش پیدا میکند و میتواند به اشتباه این حس را ایجاد کند که الگوریتم مناسب نیست.)

راهکار برای توسعه و بهبود پروژه:

1. برای مصرف کمتر فضا میتوانستیم بسیاری از ویژگی های کلاس ها را حذف کنیم که در همان گام طراحی به آنها پرداخته شد.
2. میتوانسیتم در تابع mutation فقط به ازای prob_m *population الگوریتم را اجرا کنیم تا الگوریتم سریع تر شود.
3. تعریف شرط اتمام قوی تر (که در بخش پرسش ها بحث میشود، مثلا تغیر نکردن کیفیت جامعه یا کم تر تغیر کردن ان نسبت به یک معیار و ...)

سوالات:

مشکلات جمعیت اولیه بسیار کم و یا بسیار زیاد:

جمعیت اولیه بسیار کم:

1. کاهش تنوع:
در همین مثال فرض کنید جمعیت اولیه 10 تایی باشد و range خوراکی ها بین 2 تا 19 باشد، باعث میشود 7 تا از حالت های ممکن را در نسل ها به صورت همزمان نداشته باشیم که شاید یکی از همین ها بتواند به ما جواب را بدهد.
2. افزایش خطا (ایستایی) و درنتیجه همگن شدن جمعیت
همان مثال بالا را در نظر بگیرد، جمعیت ده تایی کم باعث میشود حالت ها بسیار کم بشود حتی با crossover و جهش نتوان جمعیت های متنوع ایجاد کرد و در نهایت نمیتوان به جواب به راحتی رسید حتی با اجراهای بالا جمعیت به مرور همگن میشود و نمیتوانیم با crossover نیز تغیر مناسبی ایجاد کرد و به جهش های خیلی پیچیده تر و بیشتر نیاز است که حتی بازهم ممکن است نتوانند کارساز باشند.
3. افزایش حساسیت به جمعیت اولیه:
از همان مثال قبل و توضیحات قبل میتوان فهمید جمعیت های بعدی بسیار شبیه جمعیت اولیه میشود و تنوع زیادی نمیتوان ایجاد کرد. این مورد زمانی که جمعیت اول به هر دلیل از جمله رندوم بودن از راه حل مناسب بسیار فاصله داشته باشد بیشتر خودش را نشان میدهد.
4. افزایش حساسیت به پارامتر های الگوریتم:

عملکرد الگوریتم های ژنتیک با جمعیت های کوچک اغلب به انتخاب پارامترهای الگوریتم مانند نرخ جهش و استراتژیcrossover بسیار حساس است. تنظیم دقیق این پارامترها می تواند زمان بر باشد و نیاز به درک عمیق رفتار الگوریتم دارد که ممکن است در همه کاربردها عملی نباشد.

به منظور نشان دادن همین عیب دوبار الگوریتم را برای مقدار های 10-3-10.5-22.5 اجرا کرده ایم که بالاتر در گزارش اورده ایم و دیدیم به خاطر جمعیت کم نتوانسته به جواب برسد.

جمعیت اولیه بسیار زیاد:

1. افزایش هزینه محاسباتی:
با افزایش جمعیت، واضحا نیاز به اجرای عملیات های crossover و mutation و ارزیابی fitness بیشتری است و این مورد متناسب با افزایش زمان احرا میباشد به خصوص اگر هر یک از این عملیت ها پیچیده و زمان بر باشند.
2. نیاز به حافظه بیشتر:
اندازه جمعیت بزرگتر به حافظه بیشتری برای ذخیره اطلاعات در مورد هرکروموزوم، از جمله fitness و یا ویژگی های آنها نیاز دارد. این می تواند یک عامل محدود کننده برای جمعیت های بسیار بزرگ، به ویژه در محیط هایی با منابع حافظه محدود باشد.
3. خطر همگرایی زود رس:
در حالی که تنوع به طور کلی خوب است، تنوع بیش از حد بدون فشار انتخاب کافی می تواند منجر به همگرایی زودرس شود. این زمانی اتفاق می‌افتد که جمعیت دارای راه‌حل‌های بسیار متنوع اما غیربهینه باشد، که تمرکز الگوریتم بر تکامل به سمت بهترین راه‌حل را دشوار می‌کند.
4. کاهش بازدهی نسبت به هزینه:
فراتر از یک نقطه خاص، افزایش بیشتر جمعیت ممکن است باعث بهبود قابل توجهی در کیفیت راه حل نشود. مزایای تنوع بیشتر و پتانسیل برای همگرایی سریع‌تر کاهش می‌یابد، و توجیه هزینه‌های محاسباتی اضافی دشوارتر می‌شود.

تاثیر افزایش تعداد جمعیت در هر دوره بر روی دقت و سرعت الگوریتم:

دقت:

1. انطباق با پیچیدگی مسئله:
برای مسائل پیچیده، ممکن است جمعیت بزرگتری برای کشف فضای جستجو به اندازه کافی لازم باشد. افزایش تدریجی جمعیت به الگوریتم اجازه می‌دهد تا با پیچیدگی مسئله سازگار شود و به طور بالقوه دقت را بدون متحمل شدن هزینه‌های محاسباتی غیر ضروری از همان ابتدا بهبود بخشد
2. کاهش همگرایی زودرس:
یک مشکل رایج در الگوریتم های ژنتیک، همگرایی زودهنگام به راه حل های غیربهینه است. افزایش اندازه جمعیت در طول زمان می‌تواند ژن جدیدی را به جامعه وارد کند، خطر گیر افتادن در راه حل های بهینه‌های محلی را کاهش دهد و شانس یافتن راه‌حل‌های بهتر را افزایش دهد..
3. کاوش و بهره برداری بهبود یافته:
در ابتدا، الگوریتم ممکن است بر کاوش فضای جستجو تمرکز کند. با افزایش جمعیت، تنوع افزوده می‌تواند به کاوش و بهره‌برداری بهتر از فضای جستجو کمک کند، و به طور بالقوه منجر به دقت بالاتر در یافتن راه‌حل‌های بهینه یا تقریباً بهینه می‌شود.
4. تاخیر در دستیابی به دقت:
در حالی که هدف استراتژی بهبود دقت در طول زمان است، استفاده اولیه از جمعیت های کوچکتر ممکن است دستیابی به راه حل های با دقت بالا را به تاخیر بیندازد. این تاخیر در سناریوهایی که تکرارهای اولیه برای مفید بودن نیاز به دقت بالاتری دارند می تواند مشکل ساز باشد.
5.

سرعت الگوریتم:

1. سرعت اولیه بالا:
شروع با جمعیت کمتر، اجرای اولیه الگوریتم را سریعتر می کند زیرا افراد کمتری نیاز به ارزیابی دارند. این برای درک سریع فضای مشکل و شناسایی مناطق امیدوار کننده مفید است.
همچنین اگر جواب بسیار ساده باشد سریع بدون هزینه زیاد برای حافظه به جواب میرسیم.
2. کاهش سرعت در طول زمان:
با افزایش جمعیت با هر بار اجرا، هزینه محاسباتی افزایش می یابد. ارزیابی افراد بیشتر به قدرت پردازش و زمان بیشتری نیاز دارد که منجر به کندی تدریجی می شود. افزایش اندازه جمعیت به طور مستقیم بر زمان لازم برای تکمیل هر نسل تأثیر می گذارد.
3. پتانسیل برای موازی سازی:
در حالی که جمعیت رو به رشد الگوریتم را کند می کند، همچنین فرصت های بهتری برای موازی سازی ارائه می دهد. ارزیابی یک جمعیت بزرگتر می تواند به طور موثر در بین چندین پردازنده توزیع شود. با این حال، مقیاس پذیری توسط منابع محاسباتی موجود و هزینه های سربار مدیریت وظایف موازی محدود می شود.

تاثیر هر یک از عملیات های crossover و mutation:

Crossover:

Crossover یک عملگر حیاتی در الگوریتم‌های ژنتیک (GAs) است که یک کلاس از الگوریتم‌های تکاملی است که برای بهینه‌سازی و مشکلات جستجو استفاده می‌شود.

1. کاوش در فضای راه حل: crossover با ترکیب بخش هایی از دو یا چند راه حل والدین برای تولید فرزندان جدید به کاوش کارآمدتر فضای راه حل کمک می کند. این فرآیند به GA ها اجازه می دهد تا مناطق جدیدی از فضای راه حل ها را که ممکن است تنها از طریق جهش قابل دسترسی نباشد، کشف کنند
2. حفظ تنوع: تنوع ژنتیکی را به جمعیت راه حل ها معرفی می کند. با اختلاط اطلاعات ژنتیکی از والدین مختلف، crossover تضمین می‌کند که جمعیت پیش از موعد به راه‌حل‌های غیربهینه همگرا نمی‌شوند.
3. سرعت همگرایی: عملگر crossover می تواند به طور قابل توجهی بر سرعت همگرایی الگوریتم ژنتیک تأثیر بگذارد. یک مکانیسمcrossover خوب طراحی شده می تواند به الگوریتم کمک کند تا با ترکیب موثر صفات مفید از افراد مختلف، سریعتر به راه حل بهینه همگرا شود.
4. انطباق خاص مشکل: اثربخشیcrossover می تواند به مشکلی که حل می شود بستگی داشته باشد. برای برخی از مشکلات، تکنیک هایcrossover خاص می تواند با حفظ ویژگی ها یا ساختارهای مهم در راه حل ها منجر به عملکرد بهتر شود.

به طور خلاصه، عملگرcrossover جزء اساسی الگوریتم‌های ژنتیک است که بر توانایی آن‌ها برای کاوش و بهره‌برداری کارآمد از فضای راه‌حل تأثیر می‌گذارد. طراحی و پارامترهای اپراتورcrossover باید به دقت در نظر گرفته شود و در برخی موارد، برای دستیابی به عملکرد مطلوب، با مشکل خاص در دست تطبیق داده شود

Mutation:

mutation یکی دیگر از عملگرهای اساسی در الگوریتم های ژنتیک (GAs) است که مکمل عملگر crossover در فرآیند تکامل است. در حالی که crossover مسئول ترکیب اطلاعات ژنتیکی از دو یا چند والدین برای تولید فرزندان است، mutation تغییرات تصادفی را در ژن‌های افراد ایجاد می‌کند و تنوع ژنتیکی را در جمعیت ارتقا می‌دهد. نقش mutation و قانون آن در الگوریتم های ژنتیک را می توان به صورت زیر بیان کرد:

1. ایجاد تنوع: mutation برای معرفی تغییرات ژنتیکی جدید به جمعیت ضروری است، که می تواند برای کاوش مناطق ناشناخته فضای راه حل ها حیاتی باشد. این تنوع می تواند منجر به کشف راه حل های بهینه تری شود که ممکن است تنها از طریق crossover ایجاد نشوند.

2. از همگرایی زودرس جلوگیری می کند: با معرفی مداوم ژن جدید به جمعیت، mutation به جلوگیری از همگرایی زودهنگام الگوریتم بر روی بهینه محلی کمک می کند. این امر تعادل اکتشاف و بهره برداری سالم تر را تضمین می کند و شانس یافتن بهینه جهانی را بهبود می بخشد.

3. تنوع ژنتیکی را حفظ می کند: به خصوص در مراحل بعدی الگوریتم، زمانی که جمعیت ممکن است شروع به همگن شدن کند، mutation خوب تضمین می کند که تنوع حفظ می شود و GA را قادر می سازد تا راه حل های جدید را به جای تکرار در اطراف زیر مجموعه ای از راه حل های احتمالاً کمتر از حد بهینه بررسی کند.

4. انطباق با محیط‌های پویا: مسائل بهینه‌سازی پویا، mutation به جمعیت کمک می‌کند تا با معرفی مداوم صفات جدید سازگار شوند و الگوریتم را در برابر تغییرات در فضای مسئله قوی‌تر کند.

در نتیجه، جهش برای اطمینان از تنوع ژنتیکی و سازگاری جمعیت در الگوریتم‌های ژنتیک حیاتی است. در کنارcrossover کار می کند تا اکتشاف و بهره برداری از فضای راه حل را متعادل کند، به جلوگیری از همگرایی زودرس و بهبود شانس یافتن راه حل های بهینه کمک می کند. اثربخشی جهش، مانند crossover، به تنظیم دقیق پارامترهای آن و انتخاب عملگرهای مناسب برای مشکل مورد نظر بستگی دارد.

آیا می توان فقط یکی از آن ها را استفاده کرد؟ چرا؟

این سوال را نمیتوان به صورت کلی و برای تمام مسائل بیان کرد.

شاید بتوان گفت از نظر تئوری پاسخ به این پرسش بله باشد، اما در عمل، ترکیب هر دو crossover وmutation ، رویکرد متعادل تری را برای کاوش و بهره‌برداری از فضای جستجو فراهم می‌کند. crossover امکان کاوش در مناطق جدید فضای راه حل را با ترکیب مجدد راه حل های موجود فراهم می کند، در حالی که جهش تصادفی و تنوع لازم را معرفی می کند، از همگرایی زودرس جلوگیری می کند و الگوریتم را قادر می سازد تا طیف وسیع تری از راه حل های بالقوه را بررسی کند.

استفاده از تنها یکی از این اپراتورها می تواند به عنوان یک مورد خاص یا یک نسخه ساده شده از GAها دیده شود، که ممکن است برای مشکلات خاص مناسب باشد، اما به طور کلی فاقد استحکام و کارایی ترکیب هر دو است. انتخاب باید بر اساس ماهیت مسئله، نمایش راه حل و آزمایش تجربی برای تعیین مؤثرترین استراتژی برای کار در دست انجام شود.

*** در مسئله خاصی که ما داشتیم و روش بهینه سازی به راحتی قابل پیاده سازی به صورت مستقیم بود این امکان داشت که تنها از mutation استفاده کنیم ولی در کل نقش هردو عمل بسیار مهم و حیاتی است.***

راهکارهایی جهت افزایش سرعت در رسید به جواب:

1. میتوانسیتم در تابع mutation فقط به ازای prob_m * population الگوریتم را اجرا کنیم تا الگوریتم سریع تر شود.
2. برای مصرف کمتر فضا میتوانستیم بسیاری از ویژگی های کلاس ها را حذف کنیم که در همان گام طراحی به آنها پرداخته شد که میتوانست سرعت اجرا را نیز افزایش بدهد.
3. تنها از یک mutation قویتر استفاده میکردیم و نه crossover

همانطور که در این پروژه ما نیز تابع mutation را نوشته ایم، به value per weight ژن ها اهمیت داده و براساس آن ژن های بهتر استفاده کنیم.

4. میتوانسیتم نسل اولیه را با ژن هایی با بیشترین value per weight ایجاد کنیم و نه به صورت رندوم.

رفع مشکل تغییر نکردن کروموزوم‌ها پس از چند مرحله:

دلایل ممکن برای بروز این اتفاق:

همگرایی زودرس یکی از مهم ترین دلالیل بروز این اتفاق است، گاهی اوقات، یک الگوریتم ژنتیک ممکن است خیلی زود در یک راه حل غیربهینه همگرا شود زیرا جمعیت به سرعت تنوع خود را از دست می دهد.

1. یک یا چند کروموزوم در اوایل جمعیت بر جمعیت مسلط شوند و منجر به همگرایی نسبت به ویژگی‌های آنها شود.
2. جمعیت بسیار کوچک و درنتیجه وابستگی نسل ها به نسل اول
3. تابع crossover ضعیف و یا احتمال crossover بسیار کم
4. تابع mutationضعیف و یا احتمال جهش بسیار کم

مشکلات ناشی از این اتفاق:

واضحا یکی از مهم ترین مشکلات میتواند نرسیدن به جواب بهینه پس از اجراهای بسیار باشد. که میتواند از مشکلات برآمده دیگری ناشی شود:

1. از دست دادن تنوع
2. همگرایی به راه حل های غیر بهینه

راه حل ها:

1. حفظ تنوع: برای مبارزه با همگرایی زودرس حفظ تنوع ژنتیکی در جمعیت بسیار مهم است. این را می توان از طریق تکنیک هایی مانند اشتراک گذاری fitness به دست آورد، که در آن افرادی که دارای ویژگی های مشابه هستند. Fitnessآنها کاهش می یابد و مجموعه ای متنوع تر از راه حل ها را تشویق می کند.
2. نرخ جهش تطبیقی: تنظیم نرخ جهش به صورت تطبیقی بر اساس مرحله الگوریتم یا عملکرد فعلی می تواند به حفظ تعادل بین اکتشاف و بهره برداری کمک کند.
3. ثبت بهترین راه حل های یافت شده تا کنون و اطمینان از گم نشدن آنها در نسل های بعدی می تواند به جلوگیری از از دست رفتن راه حل های خوب به دلیل نوسانات تصادفی یا جهش بیش از حد کمک کند.
4. رویکردهای ترکیبی: ترکیب الگوریتم ژنتیک با سایر تکنیک های بهینه سازی، مانند روش های جستجوی محلی، می تواند به اصلاح راه حل ها و فرار از بهینه محلی کمک کند. این رویکرد ترکیبی امکان اکتشاف گسترده‌تر و بهره‌برداری کامل‌تر از فضای راه‌حل را فراهم می‌کند.

پیشنهاداتی جهت اتمام برنامه درصورت جواب نداشتن مسئله:

1. تعین حداکثر تعداد اجرا (که در این پروژه از این روش استفاده شده است.)
2. تعریف سطح fitness رضایت بخش: اگر الگوریتم نتواند کروموزومی را تولید کند که پس از تعداد معینی از نسل ها این آستانه را برآورده کند یا از آن فراتر رود، الگوریتم می تواند متوقف شود.
3. بررسی همگرایی: جمعیت را بررسی کنیم که آیا به مجموعه ای از راه حل های مشابه همگرا شده است یا خیر. همگرایی ممکن است نشان دهد که جمعیت به حداکثر محلی رسیده است و تکرارهای بیشتر ممکن است راه حل های متفاوتی را به وجود نیاورد. اگر تنوع جمعیت زیر یک آستانه مشخص کاهش یابد یا اگر بهترین fitness طی چندین نسل بهبود نیابد، الگوریتم را می توان خاتمه داد.
4. خاتمه دستی
5. بررسی تغیرات کیفیت نسل: درصورتی که کیفیت نسل با معیار مشخصی از یک حد پس تعداد اجرا تغییر نکرد، الگوریتم میتواند خاتمه پیدا کند. (مثلا avg fitness پس از 10 ران کمتر از 0.01 واحد تغییر کند.)

منابع استفاده شده:

- Stackoverflow