گزارش تمرین پنجم هوش محاسباتی

نام تهیه‌کننده: ملیکا نوبختیان

شماره دانشجویی: ۹۷۵۲۲۰۹۴

نسخه: ۱

۱- بخش تئوری

۱-۱- سوال اول

① برای حل مسئله graph coloring با کمترین تعداد رنگ به این صورت عمل می‌کنیم که ابتدا با یک upper bound از تعداد رنگ، رنگ‌آمیزی را شروع کرده و اگر هر بار با GA به بیشترین میزان fitness رسیدیم، جواب را نگه داشته به سراغ رنگ‌آمیزی با کمتری تعداد رنگ یعنی یکی کمتر از بار قبل می‌رویم و روند گفته‌شده را تا زمانی که برای یک تعداد رنگ دیگر به fitness دلخواه نرسیم ادامه می‌دهیم.

کروموزوم های ما به این شکل خواهند بود که لیستی از 15 عدد خواهند بود که 15 همان تعداد رأس هاست.

upper bound برای تعداد رنگ ها برابر بیشترین درجه یک node به علاوه یک خواهد بود. پس تعداد رنگ های ما در ابتدا برابر 8 خواهد بود و لیست های ما اعداد 1 تا 8 را در بر خواهند گرفت.

population اولیه ما که به طور random تولید شده است به شکل زیر خواهد بود:

C1 = [6, 4, 4, 7, 4, 4, 6, 2, 2, 3, 5, 7, 7, 7, 1]

C2 = [1, 8, 5, 5, 1, 1, 1, 8, 3, 8, 8, 7, 2, 8, 1]

C3 = [5, 7, 5, 8, 7, 2, 2, 2, 6, 7, 3, 8, 5, 2, 1]

C4 = [4, 3, 1, 2, 1, 7, 5, 6, 2, 3, 1, 4, 3, 2, 5]

C5 = [5, 4, 2, 2, 7, 4, 3, 4, 7, 8, 2, 4, 1, 2, 4]

C6 = [1, 6, 6, 7, 3, 2, 6, 6, 8, 4, 2, 7, 7, 6, 1]

برای محاسبه fitness به این روش عمل می‌کنیم که به ازای هر یالی که دو node دو طرف آن هم رنگ هستند، منفی به اندازه 1- در نظر خواهیم گرفت. در این صورت بیشترین مقدار fitness ای که می‌توانیم داشته باشیم 0 خواهد بود.

در صورتی که به چنین مقداری برسیم به یک جواب n-coloring رسیده‌ایم و حالا می‌توانیم به سراغ پیدا کردن جواب n-1 رنگ برویم

ادامه ①

مقادیر fittness برای اعضای population به صورت زیر خواهد بود:

$f(c1) = -4 \quad f(c2) = -7 \quad f(c3) = -2 \quad f(c4) = -1 \quad f(c5) = -6$

$f(c6) = -1$

$$f(c4) = f(c6) > f(c3) > f(c1) > f(c5) > f(c2)$$

ترتیب fittness بودن عضوها به شکل بالا است که در آن c4 و c6 بهترین مقدار fittness را دارند ولی با هم به بهترین مقدار یعنی صفر نرسیده اند که نشان می دهد هنوز به جواب نرسیده ایم و باید کار را ادامه دهیم.

مقادیر fittness به ما نشان می دهند که چقدر به جواب مسئله نزدیک شده ایم.

برای انجام crossover و mutation باید یک rate برای هر کدام تعیین کنیم. برای این کار نرخ crossover را 50 % در نظر می گیریم و نرخ mutation را 10% درصد در نظر می گیریم.

برای cross-over به این شکل عمل می کنیم که ابتدا برای هر عضو population یک عدد رندوم بین 0 تا 1 به دست می آوریم. اگر این عدد کمتر 0.5 بود آن عضو در cross-over شرکت خواهد کرد.

برای مثال در اینجا عضوهای c1، c4 و c6 در cross-over شرکت خواهند کرد. هم چنین لازم است یک مکان رندوم برای هر crossover انتخاب کنیم تا cross-over در آنجا اتفاق بیفتد:

$C1 >< C4 \rightarrow$ در مکان 7

$C4 >< C6 \rightarrow$ در مکان 5

$C6 >< C1 \rightarrow$ در مکان 9

$C1 >< C4 \Rightarrow C1' = [6, 4, 4, 7, 4, 4, 6, 6, 2, 3, 1, 4, 3, 2, 5]$

$C4 >< C6 \Rightarrow C4' = [4, 3, 1, 2, 1, 2, 6, 6, 8, 4, 2, 7, 7, 6, 1]$

$C6 >< C1 \quad C6' = [1, 6, 6, 7, 3, 2, 6, 6, 8, 3, 5, 7, 7, 7, 1]$

حالا به سراغ mutation می رویم. ابتدا باید تعداد کل gen هایی که داریم را بدانیم تا بفهمیم چه تعداد mutation باید انجام دهیم. $15 \times 6 = 90$ عدد gen داریم که 10% درصد آن برابر 9 خواهد بود. پس باید به طور رندوم در 9 gen از این 90 gen mutation انجام دهیم. به طور رندوم gen های زیر برای mutation انتخاب خواهند شد:

78، 42، 14، 69، 30، 41، 5، 22، 49

سپس برای هر کدام از این جایگاه یک عدد رندوم بین 1 تا 8 انتخاب خواهد شد و جایگزین خواهد شد.

ادامه ①

برای مثال اعداد زیر در این جایگاه ها قرار خواهند گرفت:

78 ، 42 ، 14 ، 69 ، 30 ، 41 ، 5 ، 22 ، 49

↓ 2 ↓ 4 ↓ 2 ↓ 8 ↓ 8 ↓ 4 ↓ 3 ↓ 7 ↓ 2

در نهایت کروموزوم های نسل جدید به صورت زیر خواهد بود:

C1' = [6, 4, 4, 7, 3, 4, 6, 6, 2, 3, 1, 4, 3, 2, 5]

C2' = [1, 8, 5, 5, 1, 1, 7, 8, 3, 8, 8, 7, 2, 8, 8]

C3' = [5, 7, 5, 8, 7, 2, 2, 2, 6, 7, 4, 4, 5, 2, 1]

C4' = [4, 3, 1, 2, 1, 2, 6, 6, 8, 4, 2, 7, 7, 6, 1]

C5' = [5, 4, 2, 2, 7, 4, 3, 4, 8, 8, 2, 4, 1, 2, 4]

C6' = [1, 6, 3, 7, 3, 2, 6, 6, 8, 3, 5, 7, 7, 7, 1]

حالا میزان fitness کروموزوم های جدید را محاسبه می کنیم:

f(C1') = -2 f(C2') = -7 f(C3') = -2

f(C4') = -1 f(C5') = -5 f(C6') = 0

مجموع fitness جمعیت قبلی 21- بود و مجموع fitness جمعیت جدید 17- است. این مقایسه نشان می دهد که نسل جدیدی که تولید شده است نسبت به نسل قبلی بهبود یافته است و به جواب مناسب تری دست پیدا کرده است و به جواب مناسب تر نزدیک تر شده است. هر چقدر میزان fitness برابر صفر نشان می دهد این گراف با 8 رنگ قابل رنگ آمیزی است و حالا می توانیم به سراغ رنگ کردن با تعداد رنگ کمتر برویم.

۲-۱- سوال دوم

(۲)

ما با استفاده از الگوریتم Genetic Programming می‌توانیم مدل ریاضی تابع متناظر با این نقاط را به دست آوریم.

ابتدا از تعریف مجموعه‌های T و F کار را شروع می‌کنیم

مجموعه T می‌تواند شامل x و y و هم چنین برای مثال اعداد حقیقی رندوم در بازه (30 , 30-) باشد

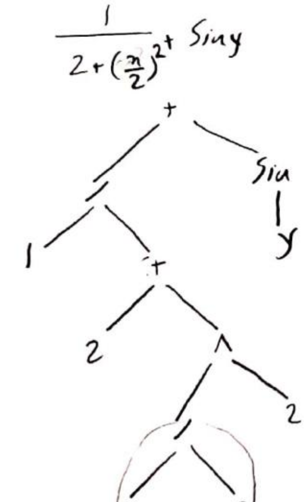
برای مجموعه F هم می‌توانیم این موارد را داشته باشیم:

$\{\times, /, -, +, \sin, \cos, \log, \wedge\}$

برای init کردن جمعیت اولیه هم از روش half-half استفاده می‌کنیم.

حداکثر عمق را برابر 5 می‌گیریم پس 20% درخت‌ها عمق 1 و 20% عمق 2 و ... 20% عمق 5 خواهند داشت و 50 درصد در هر عمق با روش full method و 50 درصد با روش grow method ساخته خواهند شد.

برای مثال tree-representation یک نمونه می‌تواند به صورت زیر باشد:



برای به دست آوردن fitness به شکل زیر عمل می‌کنیم:

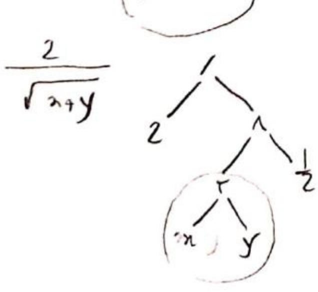
$$f = \frac{1}{1+\sum_{i=1}^{9}\left(f(x_i)-y_i\right)^2}$$

که در واقع میزان تفاوت با تابع اصلی را نشان می‌دهد

هر چه میزان تفاوت کمتر باشد fitness بزرگتری خواهیم داشت.

نرخ crossover را برابر 90% قرار می‌دهیم و نرخ mutation را برابر 10%.

برای مثال تغییر crossover بین دو نمونه نشان داده شده می‌تواند به این شکل باشد که زیر درخت نمونه دوم با اولی جابه‌جا شود

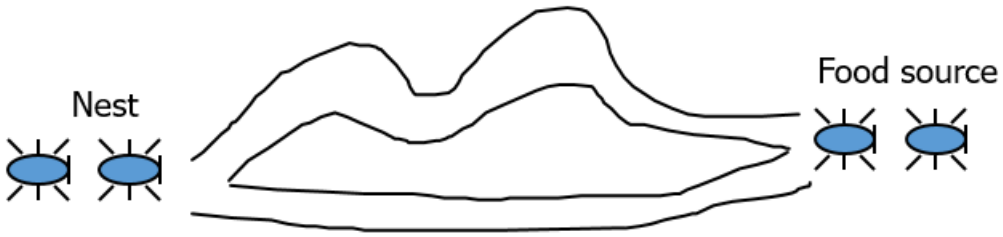


برای mutation هم می‌توانیم یک زیر درخت انتخاب شده را با دیگری که به طور رندوم انتخاب شده جایگزین کنیم.

این کار را تا رسیدن به error کمتر از 0.1 یا تا 1000 نسل ادامه خواهیم داد.

۳-۱- سوال سوم

می‌توانیم این مسئله را مانند شکل زیر در نظر بگیریم:



هر مورچه در هر مسیری که می‌رود مقداری فرومون از خودش به جای می‌گذارد و این فرومون‌ها پس از مدتی با یک نرخ مشخص تبخیر خواهند شد. مورچه دوم که از مسیر طولانی تر می‌رود، چون به مراتب زمان سفرش هم طولانی‌تر است پس مقدار فرومون بیشتری در مسیری که دارد تبخیر خواهد شد و این مسیر فرومون کمتری خواهد داشت.

همچنین می‌دانیم احتمال انتخاب یک مسیر از فرمول زیر محاسبه می‌شود:

$$P_{ij}(t) = \frac{\tau_{ij}(t)^{\alpha}\left(\frac{1}{d_{ij}}\right)^{\beta}}{\sum_{j \in \text{allowed nodes}} \tau_{ij}(t)^{\alpha}\left(\frac{1}{d_{ij}}\right)^{\beta}}$$

در این فرمول میزان احتمال انتخاب یک مسیر با مقدار فرومون موجود در آن رابطه مستقیم دارد و با طول آن مسیر ارتباط عکس دارد. با توجه به اینکه فرومون موجود در مسیر دوم کمتر و هم چنین طول آن بیشتر است احتمال این مسیر کمتر خواهد بود و احتمال انتخاب مسیر کوتاه‌تر بیشتر خواهد بود. پس مورچه دوم مسیر کوتاه‌تر را انتخاب خواهد کرد.

۲- بخش عملی

۲-۱- سوال اول

ابتدا به توضیح روش encoding می‌پردازم. جواب‌های معادله می‌توانند اعداد اعشاری باشند پس دارای دو بخش صحیح و اعشاری خواهند بود. Encoding ای که من استفاده کردم به این شکل است که دارای دو قسمت است، یک قسمت برای نشان دادن قسمت صحیح و یک قسمت برای نشان دادن قسمت اعشاری. قسمت اعشاری یک عدد حداکثر تا ۵ رقم حفظ خواهد شد و بقیه قسمت‌ها حذف خواهند شد. هر دو قسمت را به صورت binary نشان می‌دهیم. برای نشان دادن هر عدد به این شکل کلاس FractionalNumber به شکل زیر پیاده‌سازی شده‌است:

```
class FractionalNumber:
    def __init__(self, number: float):
        new_number = abs(round(number, 5))
        self.real_number = new_number
        self.decimal_number = int(new_number)
        self.float_number = int((new_number - int(new_number)) * (10**5))
        self.sign = -1 if number < 0 else 1

    @classmethod
    def from_binary_str(cls, binary_decimal: str, binary_float: str, sign : int):
        decimal = str(int(binary_decimal, 2))
        floating = str(int(binary_float, 2))
        return FractionalNumber(float(f'{decimal}.{floating}') * sign)

    def binary_form(self):
        return (str(bin(self.decimal_number).replace("0b", "")) , str(bin(self.float_number).replace("0b", "")))

    def __repr__(self):
        return str(self.real_number * self.sign)
```

چون دو قسمتی که نگه می‌داریم باید مثبت باشند و علامتی نداشته باشند تا در شکل binary تاثیر بگذارد، ابتدا از عدد قدرمطلق می‌گیریم و سپس عملیات مناسب را روی آن انجام می‌دهیم. هم چنین علامت عدد را برای محاسبات دیگر ذخیره خواهیم کرد. چون به هردو شکل binary و real اعداد نیازمندیم با استفاده از متدها و پارامترهای مختلف می‌توانیم به آن‌ها دست پیدا کنیم. هم چنین می‌توانیم با متد from_binary_str هر گاه دو بخش یک عدد را به صورت string داشته باشیم، FractionalNumber آن را به دست آوریم.

حالا به سراغ توضیح کلاس GeneticAlgorithmSearch می‌رویم که با استفاده از آن الگوریتم ژنتیک روی مسئله مورد نظر ما انجام خواهد شد.

ابتدا کار را با معرفی init این کلاس شروع می‌کنیم:

```
class GeneticAlgorithmSearch:

    def __init__(self, equation, boundary, num_generations, population_size):
        self.keep_best = True
        self.num_generations = num_generations
        self.current_population = []
        self.best_so_far = None
        self.crossover_rate = 50  # 50%
        self.mutation_rate = 10  # 10%
        self.equation = equation
        self.lower_bound, self.upper_bound = boundary
        self.population_size = population_size
```

پارامترهای ورودی برای ساختن یک نمونه از این کلاس معادله موردنظر، محدوده متغیر x، تعداد دفعات generation و اندازه population اولیه خواهد بود. Keep_best اگر true باشد بهترین کروموزوم را همیشه نگه خواهد داشت. میزان crossover و mutation هم مشخص است و در تصویر واضح است.

برای ساختن population اولیه از تابع زیر استفاده می‌کنیم:

```
def generate_initial_population(self):
    return [FractionalNumber(random.uniform(self.lower_bound, self.upper_bound)) for _ in range(self.population_size)]
```

این تابعی لیستی از FractionalNumbetr ها در محدوده‌ای که تعیین کردیم و با میزان جمعیت مشخص شده خواهد ساخت.

تابع handle_crossover_between به صورت زیر crossover را میان دو کروموزوم انجام می‌دهد:

```
def handle_crossover_between(self, chromosome1, chromosome2):
    offspring_genes_decimal = ''.join(gene1 if random.randint(0, 100) < self.crossover_rate else gene2
                              for gene1, gene2 in zip(chromosome1.binary_form()[0], chromosome2.binary_form()[0]))
    offspring_genes_float = ''.join(gene1 if random.randint(0, 100) < self.crossover_rate else gene2
                              for gene1, gene2 in zip(chromosome1.binary_form()[1], chromosome2.binary_form()[1]))
    return FractionalNumber.from_binary_str(offspring_genes_decimal, offspring_genes_float, chromosome1.sign * chromosome2.sign)
```

برای هر دوبخش صحیح و اعشاری crossover به صورت جداگانه انجام خواهد شد. crossover به این صورت انجام می‌شود که برای ژن موجود در کروموزوم یک عدد رندوم در نظر می‌گیریم. اگر از نرخ crossover کمتر بود آن بخش از کروموزوم اول انتخاب خواهد شد ولی در غیر این صورت از کروموزوم دوم انتخاب خواهد شد.

Mutation نیز به صورت زیر انجام می‌شود:

```
def handle_mutation_in(self, chromosome):
    flip_bit = lambda x: '0' if x == '1' else '1'
    binary_generator_decimal = (flip_bit(bit) if random.randint(0, 100) < self.mutation_rate else bit
                        for bit in chromosome.binary_form()[0])
    binary_generator_float = (flip_bit(bit) if random.randint(0, 100) < self.mutation_rate else bit
                        for bit in chromosome.binary_form()[1])
    return FractionalNumber.from_binary_str(''.join(binary_generator_decimal), ''.join(binary_generator_float), chromosome.sign)
```

Mutation هم روندی شبیه به crossover دارد و با توجه به نرخ crossover و عدد رندوم بیت موردنظر flip خواهد شد.

برای ارزیابی میزان fitness یک کروموزوم به شکل زیر عمل می‌کنیم:

```
def evaluate_chromosome(self, chromosome):
    return  - abs(self.equation(chromosome.real_number * chromosome.sign))
```

میزان fitness ما به این صورت محاسبه می‌شود که مقدار معادله با عدد ورودی محاسبه می‌شود و مقدار منفی آن به عنوان fitness در نظر گرفته می‌شود. بهترین fitness برای ما صفر خواهد بود.

برای شانس بیشتری برای نمونه‌های با fitness بیشتر قائل شویم. از تابع زیر استفاده می‌کنیم. این تابع کروموزوم‌های مرتب شده بر اساس مقادیر fitness را بر اساس جایگاهشان به همان تعداد تکرار خواهد کرد و شانس بیشتری برای کروموزوم‌های مناسب‌تر خواهد بود:

```
def create_probabalistic_population_for_pick(self):
    """ best last """
    to_return = []
    for position, chromosome in enumerate(self.current_population):
        to_return.extend([chromosome]*position)
    return to_return
```

در نهایت به توضیح کلی متد run_search می‌رسیم که تمام مراحل الگوریتم را در بر می‌گیرد. ابتدا لازم است که جمعیت اولیه را با استفاده از تابع گفته‌شده بسازیم. سپس کروموزوم‌ها را بر اساس میزان fitness مرتب خواهیم کرد و بهترین نمونه را معرفی خواهیم کرد. سپس به سراغ ساختن جمعیت جدید می‌رویم. برای اینکار همیشه بهترین نمونه را حتما اضافه می‌کنیم. سپس با تابعی که پیش‌تر گفته شد جمعیت آماری رو متناسب با fitness می‌سازیم. سپس تا زمانی که به اندازه جمعیت مطلوب نرسیم crossover و mutation را تکرار خواهیم کرد و دوباره کارهای گفته شده را تا رسیدن به generation نهایی و اتمام کار انجام خواهیم داد.

تصویر زیر این متد را نشان می‌دهد:

```
def run_search(self):
    self.current_population = self.generate_initial_population()

    for i in range(self.num_generations):
        print('Starting generation {}'.format(i))

        # Evaluate
        self.current_population.sort(key=self.evaluate_chromosome)
        print("current population: ", self.current_population)
        self.best_so_far = self.current_population[-1]
        print('\tBest score so far = {}'.format(self.evaluate_chromosome(self.best_so_far)))
        print(f'best x so far : {self.best_so_far.real_number}')

        # Creating new population
        new_population = []

        # Copy best over if needed
        if self.keep_best:
            new_population.append(self.best_so_far)

        # Filling the rest
        probabilistic_population_for_mating = self.create_probabalistic_population_for_pick()
        while len(new_population) < len(self.current_population):
            parent1 = random.choice(probabilistic_population_for_mating)
            parent2 = random.choice(probabilistic_population_for_mating)

            # Performing crossover
            child = self.handle_crossover_between(parent1, parent2)

            # Performing mutation
            child = self.handle_mutation_in(child)

            # Ensuring child is good
            if self.should_exclude(child):
                continue
```

اجرای این الگوریتم با ۲۰ generation و جمعیت اولیه ۱۰ و پیداکردن جواب در بازه ۱۰- تا ۱۰ نتایج زیر را به دنبال خواهد داشت:

```
Starting generation 0
current population:  [-9.51073, -7.43652, 6.74634, -6.32827, -5.71989, -
5.39079, -4.47453, 4.38043, -3.9964, 2.05945]
      Best score so far = -1455.548117455599
best x so far : 2.05945
Starting generation 1
current population:  [5.51346, -4.40326, 3.49262, 2.68923, -2.5998,
2.57884, 2.53676, 2.05945, 0.7195, 0.5907]
      Best score so far = -28.063228186224006
best x so far : 0.5907
Starting generation 2
current population:  [3.4926, 2.42524, 1.35088, 0.62878, 0.61258, 0.5907,
0.5899, 0.44695, 0.30624, 0.34493]
      Best score so far = -1.8181061594236212
best x so far : 0.34493
```

```
Starting generation 3
current population:  [1.22331, 0.9774, 0.61914, 0.61035, 0.59102, 0.29735,
0.30624, 0.33981, 0.34493, 0.36535]
      Best score so far = -0.30792466214699843
best x so far : 0.36535
Starting generation 4
current population:  [1.30462, 0.61148, 0.18036, 0.22334, 0.43118,
0.29702, 0.34617, 0.35997, 0.36535, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 5
current population:  [1.43244, 0.5811, 0.56991, 0.52117, 0.16902, 0.21516,
0.4475, 0.29988, 0.35989, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 6
current population:  [0.52631, 0.1614, 0.17387, 0.17747, 0.20556, 0.22318,
0.44983, 0.40607, 0.40087, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 7
current population:  [0.17243, 0.1771, 0.21359, 0.22361, 0.44725, 0.2894,
0.29024, 0.3051, 0.36791, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 8
current population:  [0.7863, 0.18257, 0.18265, 0.24537, 0.44469, 0.29069,
0.30118, 0.40887, 0.36351, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 9
current population:  [1.13693, 0.17755, 0.19927, 0.20314, 0.20443,
0.44989, 0.25566, 0.3009, 0.3213, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 10
current population:  [0.5531, 0.19722, 0.19921, 0.19935, 0.20417, 0.25998,
0.26011, 0.29593, 0.30158, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 11
current population:  [1.21959, 0.18395, 0.19403, 0.25068, 0.25742,
0.25867, 0.26062, 0.27994, 0.32641, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 12
current population:  [1.19403, 0.18126, 0.19547, 0.22091, 0.24193,
0.25806, 0.25986, 0.26089, 0.27587, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 13
current population:  [0.14154, 0.17353, 0.22379, 0.24775, 0.25804,
0.25839, 0.26095, 0.30153, 0.3971, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 14
current population:  [1.9665, 1.26078, 1.2479, 0.17519, 0.18378, 0.22507,
0.25837, 0.26111, 0.3366, 0.36791]
```

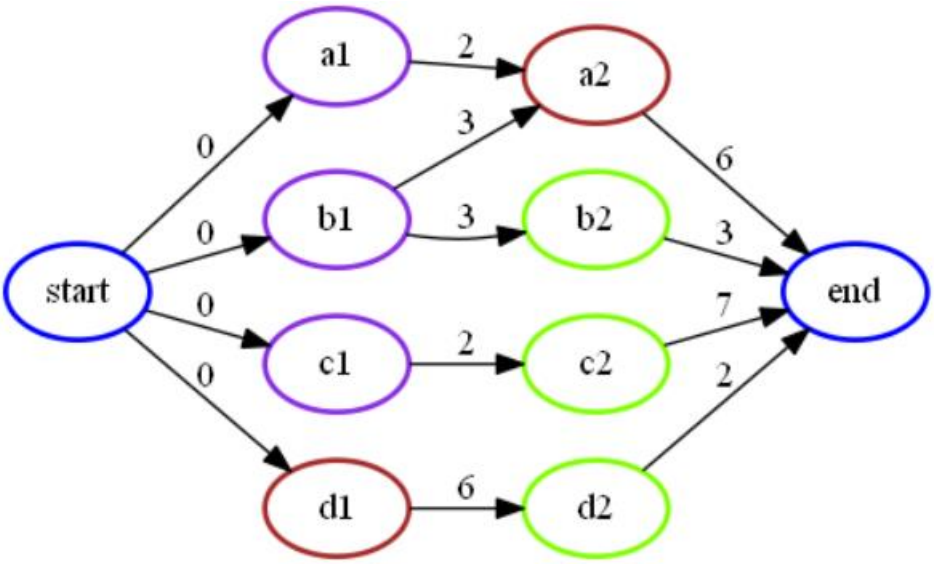
```
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 15
current population:  [1.32759, 1.30185, 0.1531, 0.16995, 0.2223, 0.26036,
0.26606, 0.26767, 0.33656, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 16
current population:  [1.10462, 0.727, 0.17122, 0.18371, 0.18386, 0.22212,
0.26074, 0.33656, 0.36156, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 17
current population:  [1.25502, 0.1368, 0.17882, 0.22523, 0.25755, 0.26347,
0.28632, 0.30683, 0.38715, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 18
current population:  [1.30714, 1.1227, 0.18173, 0.21978, 0.44989, 0.26507,
0.29851, 0.31197, 0.32666, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Starting generation 19
current population:  [0.14235, 0.1788, 0.21918, 0.24474, 0.29145, 0.2932,
0.2995, 0.31229, 0.31646, 0.36791]
      Best score so far = -0.10836604012127005
best x so far : 0.36791
Best Estimated X 0.36791
Best Equation Value -0.10836604012127005
```

بهترین مقدار به دست آمده برای x برابر با 0.36791 است و مقدار معادله نیز تقریبا نزدیک به صفر است که نشان تقریب مناسب الگوریتم ژنتیک برای این مسئله است.

۲-۲- سوال دوم

برای حل کردن مسئله jobShop با استفاده از الگوریتم ACO ابتدا لازم است یک گراف مبتنی بر job ها و task های موجود در هر کدام بسازیم که تا مورچه‌ها با طی کردن یک مسیر کلی که از یک نود آغازین شروع می‌شود و از تمام نودها می‌گذرد و به یک نود پایانی می‌رسد بهترین مسیر را پیدا کنند.

نحوه ساخت گراف به این صورت است که به ازای هر تسک موجود در هر job یک نود خواهیم داشت. اگر یک تسک وابسته به این باشد که تسک دیگری قبل از آن انجام شود، تسک پیشین به لیست نودهای وابسته تسک دوم اضافه خواهد شد و هم چنین تسک دوم به لیست تسک‌های ادامه دهنده تسک اول اضافه خواهد شد. یک نود source کلی داریم که گراف از آنجا شروع می‌شود. در لیست وابسته‌های تسک اول تمام job ها ، این نود source اضافه خواهد شد. هم چنین یک نود sink و پایانی نیز خواهیم داشت که پیش نیازهای آن تسک آخر تمام job ها خواهند بود. تصویر زیر یک نمونه از گراف برای task ها را نشان می‌دهد:



حالا به سراغ توضیح کد می‌رویم. ابتدا یک کلاس Node خواهیم داشت:

```
class Node:

    def __init__(self, name, machine, time_value, job, task_number):
        self.name = name
        self.job = job
        self.task_number = task_number
        self.machine = machine
        self.time_value = time_value
        self.successor_list = []
        self.predecessor_list = []
        self.pheromone_dict = {}
        self.start_time = None
        self.end_task = False

    def pheromones_get(self, _nodes):
        return {node: self.pheromone_dict[node] for node in _nodes if node in self.pheromone_dict}

    def nested_predecessors(self):
        nested_list = self.predecessor_list[:]
        if not nested_list:
            return []
        for node in nested_list:
            nested_list.extend(node.nested_predecessors())
        return nested_list
```

برای ساختن یک نمونه از این کلاس لازم است نام نود، ماشین، مقدار زمان لازم ، job و task را بدهیم. لیست successor ها و predecessor ها بعدا آپدیت خواهند شد. توابع موجود هم در فرآیند انجام الگوریتم مورد استفاده قرار خواهند گرفت.

برای ساختن لیست Node ها از تابع CreateNodelist استفاده می‌کنیم:

```
def CreateNodelist(jobTimesMach):

  Nodelist = []

  # first we create start node
  start_node = Node(1, 0, 0, -1, -1)
  Nodelist.append(start_node)

  predecessors = []
  predecessors.append([])

  #create node for jobs
  node_name = 2
  for i, job in enumerate(jobTimesMach):
    prev_node = None
    for j, task in enumerate(job):
      machine_num = task[0]
      duration = task[1]
      new_node = Node(node_name, machine_num, duration, i+1, j+1)
      Nodelist.append(new_node)
      predecessors.append([])
      if prev_node is not None:
        predecessors[-1].append(prev_node)
      node_name += 1
      prev_node = new_node
    Nodelist[-1].end_task = True

  # Add successors and predecessors
  for idx, node in enumerate(Nodelist):
    if len(predecessors[idx]) > 0:
      for pred in predecessors[idx]:
        node.predecessor_list.append(pred)
        pred.successor_list.append(node)

  # Start Node Config
  start_nodes = [node for node in Nodelist if len(node.predecessor_list) =
= 0 and node.name != 1]
  start_node.successor_list = start_nodes
  for node in start_nodes:
    node.predecessor_list.append(start_node)

  # Last Node Config
  last_node = Node(node_name, 0, 0, -1, -1)
  last_nodes = [node for node in Nodelist if node.end_task]
  for node in last_nodes:
    last_node.predecessor_list.append(node)
    node.successor_list.append(last_node)
```

```
  Nodelist.append(last_node)

  return Nodelist
```

یکی از کلاس‌هایی که خواهیم داشت کلاس Ant بود که به شکل زیر است:

```
class Ant:

    def __init__(self, _start_node):
        self.visited_list = [_start_node]
        self.visibility_list = []
        self.result_value = None
        self.visibility_list_update()

    def result_generate(self):
        while self.visibility_list:
            next_node = self.next_node_calculate()
            self.ant_move(next_node)
        self.result_value = result_value_calculate_as_makespan(self.visite
d_list)

    def next_node_calculate(self):
        pheromones_dict = self.visited_list[-
1].pheromones_get(self.visibility_list)
        next_node = weighted_choice_sub(pheromones_dict)
        return next_node

    def ant_move(self, _next_node):
        self.visited_list.append(_next_node)
        self.visibility_list.remove(_next_node)
        self.visibility_list_update()

    def visibility_list_update(self):
        for successor in self.visited_list[-1].successor_list:
            if set(successor.predecessor_list).issubset(self.visited_list)
:
                self.visibility_list.append(successor)


def weighted_choice_sub(_dict):
    rnd = random.random() * sum(_dict.values())
    for i, w in enumerate(_dict):
        rnd -= _dict[w]
        if rnd < 0:
            return w


def result_value_calculate_as_makespan(_operations):
    for i, operation in enumerate(_operations):
```

```
        machine_unload_time = get_machine_unload_time(_operations[:i+1])
        if operation.predecessor_list:
            predecessor_end_times = [predecessor.start_time + predecessor.
time_value for predecessor in
                                     operation.predecessor_list]
        else:
            predecessor_end_times = [0]
        operation.start_time = max([machine_unload_time] + predecessor_end
_times)
    return _operations[-1].start_time + _operations[-1].time_value
```

برای ساختن یک مورچه لازم است نود آغازین که مورچه از آنجا حرکت خود را آغاز خواهد کرد را داشته باشیم. Visited_list نودهایی که را که از آن عبور کردیم را به ما نشان خواهد داد. Visibility_list نیز نودهایی که قادر هستیم در آینده درآن‌ها را ببینیم در خود خواهد داشت. تابع result_generate یک مسیر کلی که از تمام نودها می‌گذرد به ما خواهد داد. تابع next_node_Calculate با توجه به مقدار فرومونی که در نودهای قابل دیدن داریم نود بعدی در مسیر را به ما خواهد داد. تابع ant_move با توجه به نود بعدی در مسیر پارامترها را آپدیت خواهد کرد. تابع result_value_calculate_as_makespan نیز طول و مدت این مسیر را محاسبه خواهد کرد.

کلاس کلی و بدنه الگوریتم ما از این کلاس خواهد بود:

```
class AntAlgorithm:
    def __init__(self, _nodes_list):
        self.nodes_list = _nodes_list
        self.ant_population = []
        self.result_history = []
        self.init_pheromone_value = 0.8
        self.pheromone_potency = 0.7
        self.pheromone_distribution = 0.5
        self.max_min_ants_promoted = 5
        self.iterations = 30
        self.ant_population = []
        self.ant_population_size = 20
        self.evaporation_rate = 0.8

    def run(self):
        pass

    def pheromone_init(self):
        for node in self.nodes_list:
            nested_predecessors = [node] + node.nested_predecessors()
            for successor in self.nodes_list:
                if successor not in nested_predecessors:
                    node.pheromone_dict[successor] = self.init_pheromone_v
alue
```

```
    @staticmethod
    def pheromone_trail_modify(_trail, _value, _operation):
        iterator = iter(_trail)
        node = next(iterator)
        for next_node in iterator:
            if _operation == MULTIPLY:
                node.pheromone_dict[next_node] *= _value
            elif _operation == ADD:
                node.pheromone_dict[next_node] += _value
            node = next_node
```

برای ساختن یک نمونه از آن لازم است که لیست نودها را داشته باشیم. حال به توضیح یک سری از پارامترهای مسئله می‌پردازیم. Init_pheromone_values نشان دهنده مقدار آغازین فرومون ما خواهد بود. Pheromone_potency میزان قدرت فرومون‌ها هنگام آپدیت میزان فرومون را نشان می‌دهد. تعداد iteration ها، تعداد مورچه‌ها و هم چنین نرخ تبخیر فرومون نیز مشخص است.

تابع pheromone_init برای نودهای موجود در لیست و براساس پیش نیازها و دنبال کننده‌ها میزان فرومون نودها را مشخص خواهد کرد.

تابع pheromone_trail_modify نیز میزان فرومون نودهای مسیر را را براساس عملیاتی که داده می‌شود را تغییر خواهد داد.

حالا به معرفی کلاس MaxMin که کلاس اصلی الگوریتم است می‌پردازیم و از کلاس قبلی ارث بری می‌کند:

```
class MaxMin(AntAlgorithm):

    def __init__(self, _nodes_list):
        AntAlgorithm.__init__(self, _nodes_list)
        self.history_best = Ant(self.nodes_list[0])
        self.history_best.result_value = 100000

    def run(self):
        self.pheromone_init()
        for iteration in range(self.iterations):
            self.ant_population = [Ant(self.nodes_list[0]) for _ in range(
self.ant_population_size)]
            for ant in self.ant_population:
                ant.result_generate()
                self.evaporate_pheromone_trail(ant)

            self.graph_update()
            print(
                "running iteration: {0}, best result_permutation is: {1}".
format(iteration,

       self.result_history[-1].result_value))
```

```
    def graph_update(self):
        self.ant_population.sort(key=lambda x: x.result_value)
        self.result_history.append(self.ant_population[0])
        self.history_best = min(self.history_best, self.ant_population[0],
 key=lambda x: x.result_value)
        self.prepare_and_modify_best_trails()

    def prepare_and_modify_best_trails(self):
        self.pheromone_trail_modify(self.history_best.visited_list, 1 + se
lf.pheromone_potency, MULTIPLY)
        for i in range(self.max_min_ants_promoted):
            value = (1 + self.pheromone_potency * (self.pheromone_distribu
tion ** (i + 1)))
            self.pheromone_trail_modify(self.ant_population[i].visited_lis
t, value, MULTIPLY)

    def evaporate_pheromone_trail(self, ant):
        self.pheromone_trail_modify(ant.visited_list, self.evaporation_rat
e, MULTIPLY)
        self.pheromone_trail_modify(ant.visited_list, 1 - self.evaporation
_rate, ADD)
```

این کلاس بهترین مسیری که یک مورچه طی می‌کند و هم چنین بهترین تایم پایانی که خواهیم داشت را در خود خواهد داشت. در تابع run روند کلی کار آغاز خواهد شد. ابتدا فرومون‌ها init خواهند شد. در هر iteration از الگوریتم ابتدا به تعداد population مورچه‌ها init خواهند شد. سپس برای هر مورچه یک مسیر تولید خواهیم کرد و پس از آن میزان فرومون مسیر آپدیت خواهد شد. پس ار آن graph_update را خواهیم داشت. در این تابع بهترین مسیر یک مورچه در iteration کنونی را به دست خواهیم آورد و با بهترین نتیجه مقایسه خواهیم کرد و بهترین را حفظ خواهیم کرد. سپس در تابع prepare_and_modify_best_trails بر اساس بیشترین تعداد مورچه هایی که مشخص خواهیم کرد میزان فرومون مسیرشان تغییر خواهد یافت.

حالا برای job های زیر می خواهیم scheduling را انجام دهیم:

```
jobTimesMach = [[(1, 3), (2, 3), (3, 2)],
                [(1, 2), (3, 1), (2, 4)],
                [(2, 4), (3, 3)]]

Nodelist = CreateNodelist(jobTimesMach)
for node in Nodelist:
  print(node)
print("")
system =  MaxMin(Nodelist)
system.run()
```

مجموعه نودها برای تسک‌های بالا به شکل زیر خواهد بود:

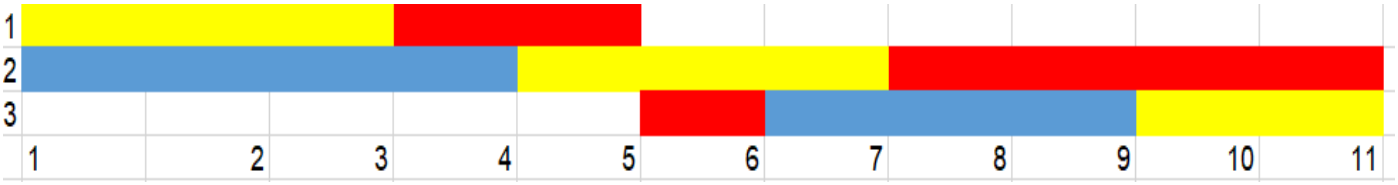
```
name: 1, machine: 0, time: 0 , preds: [] , succ: [2, 5, 8]
name: 2, machine: 1, time: 3 , preds: [1] , succ: [3]
name: 3, machine: 2, time: 3 , preds: [2] , succ: [4]
name: 4, machine: 3, time: 2 , preds: [3] , succ: [10]
name: 5, machine: 1, time: 2 , preds: [1] , succ: [6]
name: 6, machine: 3, time: 1 , preds: [5] , succ: [7]
name: 7, machine: 2, time: 4 , preds: [6] , succ: [10]
name: 8, machine: 2, time: 4 , preds: [1] , succ: [9]
name: 9, machine: 3, time: 3 , preds: [8] , succ: [10]
name: 10, machine: 0, time: 0 , preds: [4, 7, 9] , succ: []
```

نتیجه اجرا الگوریتم به شکل زیر خواهد بود:

```
running iteration: 0, best result_permutation is: 12
running iteration: 1, best result_permutation is: 11
running iteration: 2, best result_permutation is: 12
running iteration: 3, best result_permutation is: 11
running iteration: 4, best result_permutation is: 11
running iteration: 5, best result_permutation is: 11
running iteration: 6, best result_permutation is: 11
running iteration: 7, best result_permutation is: 11
running iteration: 8, best result_permutation is: 12
running iteration: 9, best result_permutation is: 11
running iteration: 10, best result_permutation is: 12
running iteration: 11, best result_permutation is: 12
running iteration: 12, best result_permutation is: 11
running iteration: 13, best result_permutation is: 11
running iteration: 14, best result_permutation is: 11
running iteration: 15, best result_permutation is: 12
running iteration: 16, best result_permutation is: 11
running iteration: 17, best result_permutation is: 11
running iteration: 18, best result_permutation is: 11
running iteration: 19, best result_permutation is: 11
running iteration: 20, best result_permutation is: 11
running iteration: 21, best result_permutation is: 11
running iteration: 22, best result_permutation is: 11
running iteration: 23, best result_permutation is: 11
running iteration: 24, best result_permutation is: 11
running iteration: 25, best result_permutation is: 11
running iteration: 26, best result_permutation is: 11
running iteration: 27, best result_permutation is: 12
running iteration: 28, best result_permutation is: 11
running iteration: 29, best result_permutation is: 11
result_permutation history:
[12, 11, 12, 11, 11, 11, 11, 11, 12, 11, 12, 12, 11, 11, 11, 12, 11, 11,
11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 11, 11]
best path: 11
                          1 -> 2 -> 8 -> 5 -> 6 -> 9 -> 3 -> 7 -> 4 -> 10
```

Best_result_permutation میزان زمانی که طول خواهد کشید تا در بهترین حالت در هر iteration کارها تمام شوند را نشان می‌دهد که بهترین عدد ۱۱ است. بهترین مسیر هم نشان داده شده‌است که ترتیب نودهای مربوط به مسیر را نشان می‌دهد.

تصویر زیر نیز با توجه به مسیری که در نتیجه گفته شد برنامه ریزی تسک‌ها را نشان می‌دهد که بهینه است:



رنگ زرد نشان دهنده جاب اول، رنگ قرمز نشان دهنده جاب دوم و رنگ آبی نشان دهنده جاب سوم است.