دانشکده مهندسی

گروه مهندسی کامپیوتر

گرایش هوش مصنوعی

الگویتم JSSP

استاد:

دکتر مجتبی روحانی

دانشجو:

علی گلی

آذر ماه

سال ۱۴۰۲

۱ الگوریتم JSSP(Job-shop scheduling)

۱.۱ تعریف مسئله

این مسئله شامل n کار یا job می باشد که هرکدام می‌تواند شامل m عملیات باشد. حال k ماشین قصد اجرای این عملیات ها را دارند. نکته حائز اهمیت مقدار زمان اجرای هر عملیات و حفظ ترتیب اجرای آن ها در هر کار می‌باشد.

مسئله ذکر شده در فایل؛ به هر عملیات یک ماشین نیز نسبت داده شده است که تنها همان ماشین باید اجرا کننده این عملیات باشد.

در تعریف مسئله ژن ها همان ترتیب اجرای این operation ها هستند. یعنی جایگشتی از operation ها (به شرط صحیح نگاه داشتن شرط ترتیب اجرای آنها در job) هستند. یعنی در Job_1 نباید $operation_2$ قبل از $operation_1$ باشد.

در ابتدا باید بتوانیم جمعیت اولیه‌ای را تولید کنیم که باید این جمعیت اولیه شرط ترتیب ذکر شده را حفظ کند. این کار را با ساختار داده Queue یا صف می‌توانیم انجام دهیم.

سپس باید راه حلی برای تعریف function evaluation داشته باشیم که در این مسئله جدول گانت مد نظر است.

عملیات های جستجو می‌تواند شامل crossover و mutation باشد.

همچنین عملیات selection نیز باید اجرا گردد که می‌تواند ژن ها؛ با کارایی پایین را حذف کند یا همانند چرخ رولت عمل کند. همچنین می‌توانیم از ترکیبی از این دو استفاده کنیم.

تعداد افراد ۲۰ نفر در نظر گرفته شده است و تعداد نسل ها نیز ۵۰۰ نسل می‌باشد. تعداد Job برابر ۱۰، تعداد operation هر Job برابر ۱۰ و تعداد ماشین ها نیز ۱۰ عدد می‌باشد.

۱.۲ تولید جمعیت اولیه

هر ژن شامل اعدادی از ۰ تا ۹۹ ،یعنی ۱۰ کار و ۱۰ عملیات است؛ که نوعی آدرس‌دهی به یک عملیات خاص است.

برای تولید جمعیت اولیه باید از صف ها استفاده کنیم. برای هر job یک صف تعریف کنیم. در نتیجه در مسئله فعلی ما ۱۰ صف خواهیم داشت. حال به صورت تصادفی شروع به استخراج از صف می‌کنیم که باعث می‌شود خاصیت ترتیبی بودن operation ها در job حفظ گردد.

همچنین برای بررسی امکان پذیر بودن یک ژن یا همان شرط ترتیب باید عملیات خاصی را ترتیب دهیم.

۱.۳ تابع function evaluation

تابع function evaluation باید بر اساس جدول گانت طراحی شود. هر ماشین باید یک آرایه شروع و یک آرایه پایان داشته باشد. همچنین هر Jon نیز باید مشخص باشد آخرین operation انجام شده آن در چه زمانی به انتها رسیده است. با این دو ماتریس و یک آرایه می‌توان جدول گانت را طراحی کرد.

۱.۴ عملیات crossover

عملیات crossover در این نوع مسئله کاربردی نخواهد بود. دلیل آن نیز این است که ترتیب در ژن ها خیلی مهم است و نمی‌خواهیم ژنی تکرار شود یا از آن حذف شود. در ساده ترین نوع crossover که one-point می‌باشد، اطلاعات با احتمال زیادی دچار تکرار می‌شوند و در نتیجه تعدادی نیز حذف خواهند شد. حتی در صورت وجود همه ژن ها؛ ژن ها فرزند تولید شده به احتمال زیادی از شرط ترتیب پیروی نخواهند کرد. در نتیجه ما باید در یک حلقه نزدیک به بی نهایت منتظر crossover صحیح باشیم.

من با نوع خاصی از الگوریتم ژنتیک این موضوع را بررسی کردم و در نتیجه کد crossover رو از کد حذف کردم.

۱.۵ عملیات mutation

عمیات mutation دیگر عملگر جستجو است. برای این عملگر سه نوع متفاوت Swap, Scramble, Inverse را تعریف کرده‌ایم. می‌توانیم ترکیبی از آنها استفاده کنیم یا تنها از یکی از آنها استفاده کنیم. در نتایج نشان خواهم داد برای این مسئله پر سرعت ترین mutation مربوط به نوع Swap است.

۱.۶ عملیات selection

من در ابتدا با عملیات چرخ رولت الگوریتم selection را پیاده سازی نمودم. اما روند حرکت خیلی تصادفی بود. روند selection را به حذف ژن های بد تغییر دادم که بسیار در ۲۰۰ نسل ابتدا در سرعت و دقت مفید بود. اما بعد از آن نسل ها رشدی نداشتند و تمامی ژن ها در یک نقطه دچار locality می‌شدند. در نتیجه تصمیم گرفتم تا قبل نسل ۳۰۰ از روند ساده selection استفاده کنم و بعد از آن به صورت یکی در میان روند selection را جابجا کنم تا چرخ رولت به ژن ها بدتر اجازه زنده ماندن بدهد و روند جدیدی را برای من ایجاد کند.

۲ کد JSSP

۲.۱ ابتداییات

ابتدا جدول ارائه شده در مسئله فرستاده شده را در قالب یک ماتریس، با تعداد ردیف برابر تعداد job و تعداد ستون برابر با تعداد operation و یک tuple با مقدار اولیه آیدی ماشین و مقدار ثانویه زمان مورد نیاز برای اجرای هر operation.

```
main_data = [
    [(0, 29), (1, 78), (2,  9), (3, 36), (4, 49), (5, 11), (6, 62), (7, 56), (8, 44), (9, 21)],
    [(0, 43), (2, 90), (4, 75), (9, 11), (3, 69), (1, 28), (6, 46), (5, 46), (7, 72), (8, 30)],
    [(1, 91), (0, 85), (3, 39), (2, 74), (8, 90), (5, 10), (7, 12), (6, 89), (9, 45), (4, 33)],
    [(1, 81), (2, 95), (0, 71), (4, 99), (6,  9), (8, 52), (7, 85), (3, 98), (9, 22), (5, 43)],
    [(2, 14), (0,  6), (1, 22), (5, 61), (3, 26), (4, 69), (8, 21), (7, 49), (9, 72), (6, 53)],
    [(2, 84), (1,  2), (5, 52), (3, 95), (8, 48), (9, 72), (0, 47), (6, 65), (4,  6), (7, 25)],
    [(1, 46), (0, 37), (3, 61), (2, 13), (6, 32), (5, 21), (9, 32), (8, 89), (7, 30), (4, 55)],
    [(2, 31), (0, 86), (1, 46), (5, 74), (4, 32), (6, 88), (8, 19), (9, 48), (7, 36), (3, 79)],
    [(0, 76), (1, 69), (3, 76), (5, 51), (2, 85), (9, 11), (6, 40), (7, 89), (4, 26), (8, 74)],
    [(1, 85), (0, 13), (2, 61), (6,  7), (8, 64), (9, 76), (5, 47), (3, 52), (4, 90), (7, 45)]
]
```

```
import numpy as np
import random
from queue import Queue
import threading
import matplotlib.pyplot as plt

main_data = np.array(main_data)
```

اضافه کردن کتابخوانه های مورد نیاز و تبدیل دیتا مسئله به دیتا از جنس numpy.

۲.۲ تولید جمعیت اولیه

یک کلاس نوشته ایم که در آن دو تابع تعریف شده است. یکی ژن معتبر تولید می‌کند و دیگری می‌تواند اعتبار یک ژن را به لحاظ ترتیب و جایگشتی بودن آن تشخیص دهد. تولید ژن با استفاده ایجاد صف برای هر job استفاده از هر صف به صورت رندوم است بدین شکل، ترتیب این جمعیت در ژن ها حفظ خواهد شد.

برای مشخص کردن اعتبار یک ژن، باید بررسی گردد $operation_{i+1}$ هر job بعد از $operation_i$ قرار گیرد. که این کار را تابع is_gene() انجام می‌دهد.

```python
class GeneCreator:
    jobs = 10
    operations = 10
    def gene_create(self, seed=1):
        random.seed(seed)
        queues = []
        gene = []
        for j in range(self.jobs):
            q = Queue()
            for o in range(self.operations):q.put(j*self.operations+o)
            queues.append(q)
        _ = 0
        while _ < self.jobs*self.operations:
            nucleotide = random.randint(0, self.jobs-1)
            if not queues[nucleotide].empty():
                gene.append(queues[nucleotide].get())
                _ += 1
        return gene
    def is_gene(self, gene):
        for index in gene:
            index = int(index)
            index_befor = [i for i in range(int(index/10)*10, index)]
            result = [element for element in gene if element in index_befor]
            if not result == index_befor: return False
        return True
```

۲.۳ جدول گانت و function evaluation

برای تولید این جدول نیاز به دو ماتریس و یک آرایه داریم. ماتریس اول هر ردیف را برای هر ماشین نگاه داری می‌کند و مشخص می‌کند و زمان شروع کار n را در ردیف ماشین kام می‌نویسد. ماتریس دوم هر ردیف را برای هر ماشین نگاه داری می‌کند و مشخص می‌کند زمان پایان کار n را در ردیف ماشین kام می‌نویسد. یک آرایه نیز باید تعریف کنیم که برای هر کار یک سلول نگاه داری کند که در آن زمان انتها هرکار در آن نوشته شود. به طور مثال operation اول از job اول اگر ۲۰ واحد زمان به طول انجامد، سلول اول این آرایه برابر ۲۰ خواهد بود تا عملیات بعدی job اول از واحد زمانی ۲۰ به بعد شروع شود.

در نهایت تابع span مقدار بیشینه ماتریس دوم را انتخاب می‌کند و برمی‌گرداند که نشان می‌دهد آخرین کار در چه زمانی به انتها رسیده است.

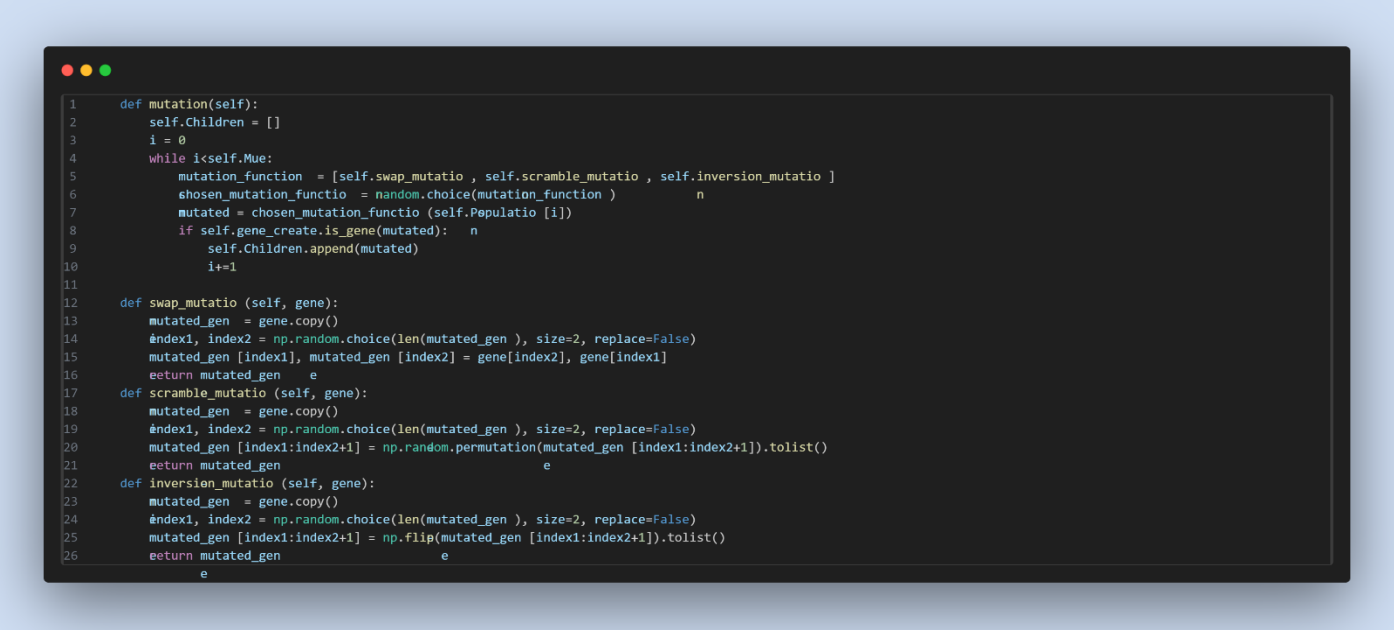
```python
class GanttChar :
    gene = []
    machines = 10
    jobs = 10
    operation  = 10
    data = main_data.reshape((100,2))

    def __init__(self, gene) -> None:
        self.gene = gene
        self.mch_st = [[0] for _ in range(self.machines)]
        self.mch_fn = [[] for _ in range(self.machines)]
        self.job_next_star  = [0 for _ in range(self.jobs)]
            t
    def machin_id(self, index):
        return self.data[index][0]
    def machine_tim (self, index):return self.data[index][1]
        e
    def add_to_machin (self, index):
        id = self.machin_id(index)
        time = self.machine_tim (index)
                    e
        if self.mch_st[id][-1] < self.job_next_star [int(index/self.jobs)]:
            self.mch_st[id][-1] = selftjob_next_star [int(index/self.jobs)]
                                t
        finish_time = time+self.mch_st[id][-1]
        self.mch_fn[id].append(finish_time)
        self.mch_st[id].append(finish_time)

        self.job_next_star [int(index/self.jobs)] = finish_time
            t
    def calculate(self):
        for nucleotid  in self.gene:
            self.add_to_machin (nucleotid )
                e               e
    def span(self):
        return np.max(np.array(self.mch_fn))
```

۲.۴ عملیات mutation

برای این عملیات از سه تابع mutation مختلف استفاده شده است که احتمال استفاده از هرکدام از آنها برابر می‌باشد. از هر والد یک فرزند جهش یافته تولید می‌شود، در صورتی که فرزند معتبر نباشد دوباره mutation انجام می‌شود.

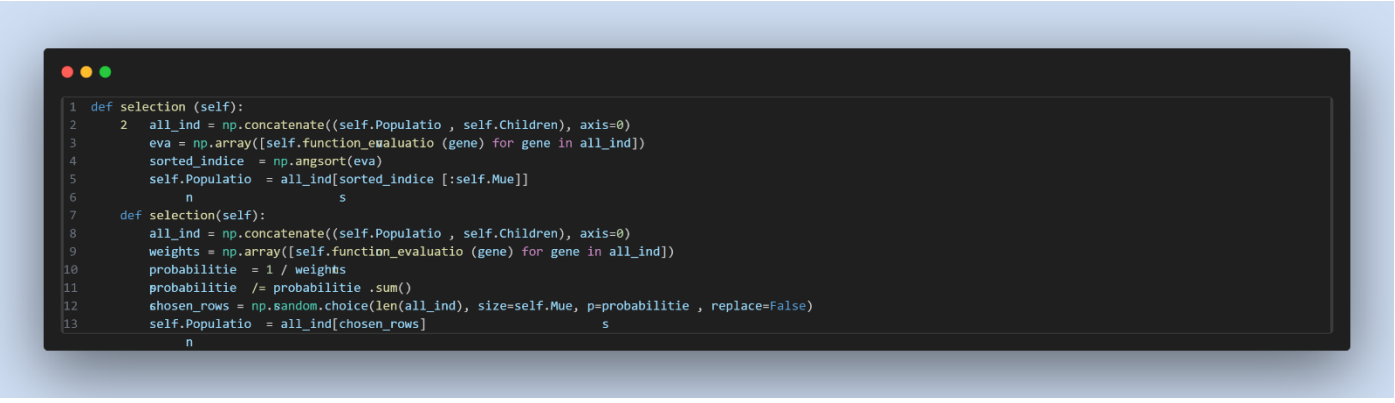


```
    def mutation(self):
        self.Children = []
        i = 0
        while i<self.Mue:
            mutation_function  = [self.swap_mutatio , self.scramble_mutatio , self.inversion_mutatio ]
            chosen_mutation_functio  = random.choice(mutation_function )           n
            mutated = chosen_mutation_functio (self.Populatio [i])
            if self.gene_create.is_gene(mutated):   n
                self.Children.append(mutated)
                i+=1

    def swap_mutatio (self, gene):
        mutated_gen  = gene.copy()
        index1, index2 = np.random.choice(len(mutated_gen ), size=2, replace=False)
        mutated_gen [index1], mutated_gen [index2] = gene[index2], gene[index1]
        return mutated_gen    e
    def scramble_mutatio (self, gene):
        mutated_gen  = gene.copy()
        index1, index2 = np.random.choice(len(mutated_gen ), size=2, replace=False)
        mutated_gen [index1:index2+1] = np.random.permutation(mutated_gen [index1:index2+1]).tolist()
        return mutated_gen                                  e
    def inversion_mutatio (self, gene):
        mutated_gen  = gene.copy()
        index1, index2 = np.random.choice(len(mutated_gen ), size=2, replace=False)
        mutated_gen [index1:index2+1] = np.flip(mutated_gen [index1:index2+1]).tolist()
        return mutated_gen                      e
              e
```

۲.۵ عملیات selection

برای این عملیات من دو نوع selection طراحی نموده‌ام. در ابتدا تا نسل ۳۰۰ از selection معمولی اولیه استفاده می‌کنم که کمک می‌کند تا با دقت و سرعت بسیاری به سمت نقطه بهینه حرکت کنیم. سپس بعد از نسل ۳۰۰ هر یک بار در میان از selection ثانویه استفاده می‌کنم تا از محلی شدن و یا جمع شدن در بهینه محلی جلوگیری کنم. نوع selection اولیه حذف ساده ژن های بد است و نوع selection ثانویه چرخ رولت می‌باشد.

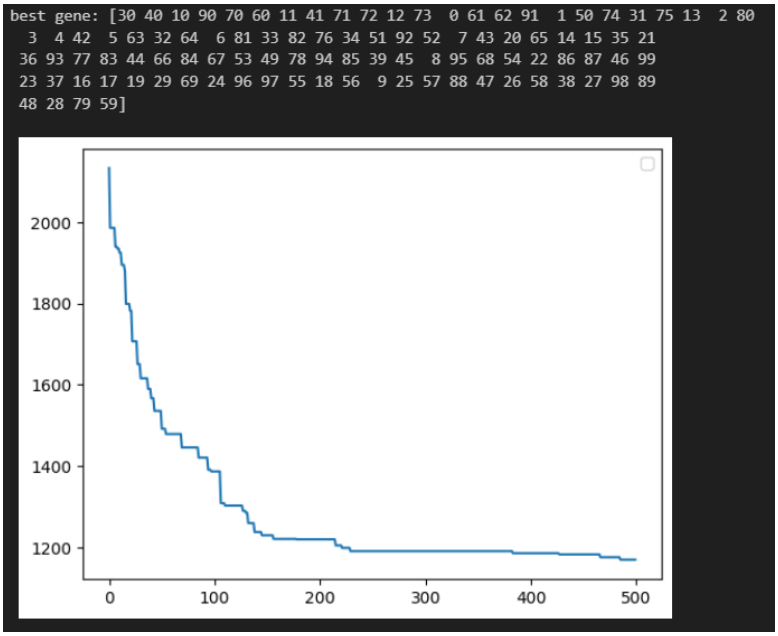


```
def selection (self):
    2   all_ind = np.concatenate((self.Populatio , self.Children), axis=0)
        eva = np.array([self.function_evaluatio (gene) for gene in all_ind])
        sorted_indice  = np.angsort(eva)
        self.Populatio  = all_ind[sorted_indice [:self.Mue]]
              n                    s
    def selection(self):
        all_ind = np.concatenate((self.Populatio , self.Children), axis=0)
        weights = np.array([self.function_evaluatio (gene) for gene in all_ind])
        probabilitie  = 1 / weights
        probabilitie  /= probabilitie .sum()
        chosen_rows = np.random.choice(len(all_ind), size=self.Mue, p=probabilitie , replace=False)
        self.Populatio  = all_ind[chosen_rows]                      s
            n
```

۳ نتایج

۳.۱ نتیجه اجرا ساده کد

ابتدا بهترین ژن نوشته شده است. محور x تعداد نسل هارا نشان می‌دهد. محور y مقدار span در هر نسل را نمایش می‌دهد. مقدار span این ژن ۱۱۷۰ می‌باشد.



۳.۲ نتیجه mutation از نوع swap

ابتدا بهترین ژن نوشته شده است. محور x تعداد نسل هارا نشان می‌دهد. محور y مقدار span در هر نسل را نمایش می‌دهد. مقدار span این ژن ۱۱۰۶ می باشد. همانطور که مشاهده می‌شود در این روش سرعت و دقت هر دو بیشتر است اما از نسل تقریبا ۱۰۰ به بعد ثابت شده و محلیت رخ می‌دهد.