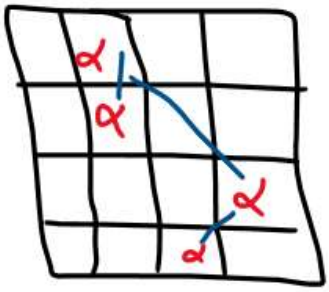
گزارش پیاده سازی مسئله n وزیر با الگوریتم ژنتیک

غزل پوراسفندیار بروجنی 9820453

تعریف کروموزم ها :

مثلا در شطرنج 4*4 کروموزم 4321 یعنی "سطر اول ،چهارمین خانه" و "سطر دوم ،سومین خانه" و "سطر سوم ،دومین خانه" و "سطر چهارم ،اولین خانه" دارای وزیر است. با این تعریف احتمال تهدید سطری را به صفر رساندیم و مطمئن هستیم که وزیر ها به صورت سطری یکدیگر را تهدید نمی کنند.



آخر هر کروموزم یک عدد به عنوان رنک (معیار مقایسه) آن کروموزم اضافه می کنیم. (در ابتدا همه صفر هستند و جلوتر مقدار دهی خواهند شد و بر این اساس مقایسه می شوند.)

معیار مقایسه را تعداد برخورد های پیش آمده قرار می دهیم. (برای مثال در شکل مقابل، ۳ برخورد (خطوط آبی) داریم.)

تابع تولید جمعیت اولیه به صورت رندوم :

```
3   def initialize_population(population_size, N):
4       population = []
5       for i in range(population_size):
6           child = []
7           for j in range(N):
8               child.append(rand.randint(1,N))
9           child.append(0)      #add rank of each genes at the end of that
10          population.append(child)
11      return population
```

تابع crossover که فرزندان دو والد را تشکیل می دهد :

خروجی این تابع فقط لیست فرزندان است و در قسمت main برنامه باید این فرزندان به جمعیت اولیه اضافه شوند تا جمعیت نهایی شکل گیرد.

منطق تشکیل فرزندان در این تابع این است که هر فرزند نیمی از ژنهای پدر و نیمی از ژنهای مادر را به ارث می برد و در نتیجه از هر زوج والد می توان ۲ فرزند تولید کرد.

```
13  def crossover(population):      #crossover 2 parent and make 2 children (logic  : half from each parent)
14      children = []
15      for i in range(0, len(population), 2):
16          child1 = population[i][:N//2] + population[i+1][N//2:N] + [0]
17          child2 = population[i+1][:N//2] + population[i][N//2:N] + [0]
18          children.append(child1)
19          children.append(child2)
20      return children
```

تابع mutation :

در این تابع جهش رندوم روی یکی از ژنهای کروموزوم های منتخب (درصد مشخصی از کل فرزندان) انجام می شود.

```
def mutation(children, rate, N):
    #choose rate percent of children and do mutation on them
    chosen_size = int(len(children)*rate)
    children_index = []
    chosen_index = []
    for i in range(len(children)):    #children_index = [0, 1, 2, ... , len(children)-1]
        children_index.append(i)
    for i in range(chosen_size):      #choose some of the children_index as chosen_index (sample make them not repeated)
        chosen_index = rand.sample(children_index, chosen_size)
    for i in range(chosen_size):
        chosen_chromosome = rand.randint(0, N-1)
        new_value = rand.randint(1, N)
        children[chosen_index[i]][chosen_chromosome] = new_value
    return children
```

تابع fitness :

در این تابع معیار ارزیابی کروموزم ها محاسبه می شود که در اینجا تعداد برخوردهاست (در تعریف کروموزم ها به آن اشاره کردیم.)

در نحوه تعریف کروموزم ها مطمئن شدیم که هیچ دو وزیری به صورت سطری برخورد ندارند.

پس کافیست مجموع برخورد های ستونی (ارقام تکراری در عدد کروموزم) و برخورد های قطری (وقتی تفاضل سطر ها با تفاضل ستون ها برابر شود) را شمرده و ذخیره کنیم.

در نهایت جمعیت را بر حسب این معیار مرتب سازی می کنیم. (به دنبال کروموزم هایی با برخورد ۰ هستیم.)

```
def fitness(population, N):
    #calculating conflicts
    for i in range(len(population)):
        conflict = 0
        for j in range(N):
            for k in range(j+1,N):
                if(population[i][j] == population[i][k]):    #check column
                    conflict += 1
                if(abs(j - k) == abs(population[i][j] - population[i][k])):    #check diagonal
                    conflict += 1
        population[i][N] = conflict
    #sort chromosomes based on conflicts
    for i in range(len(population)):
        minimum = i
        for j in range(i, len(population)):
            if(population[j][N] < population[minimum][N]):
                minimum = j
        population[i] , population[minimum] = population[minimum] , population[i]

    return population
```

تابع یافتن چینش های نهایی :

```python
def answers(population, N):
    answers = []
    for i in range(len(population)):
        if(population[i][N] == 0):
            if(population[i] not in answers):
                answers.append(population[i])
    print("The algorithm find", len(answers), "answers:")
    for i in range(len(answers)):
        print(answers[i])
    return len(answers)
```

تابع GA_performance که در آن توابع بالا صدا زده می شود تا مساله حل شود:

```python
def GA_performance(N, population_size, rate):
    start_time = time.time()
    population = initialize_population(population_size, N)
    # print("primary population:")
    # print(population)

    children = crossover(population)
    # print("population after crossover:")
    population += children
    # print(population)
    # print("children after crossover:")
    # print(children)

    children = mutation(children, rate, N)
    # print("children after mutation:")
    # print(children)

    population += children
    population = fitness(population, N)
    # print("population after fitness")
    # print(population)
    num_of_answers = answers(population, N)

    exe_time = time.time() - start_time
    performance = num_of_answers / exe_time
    print("The execution time is:", exe_time)
    print("performance:", performance)
    print("_____")
    return performance
```

معیار performance را حاصل تقسیم تعداد جواب ها به زمان کل قرار دادیم تا بتوان آنالیز کرد.

مثالی از نمونه خروجی نهایی برای چینش ۴ وزیر:

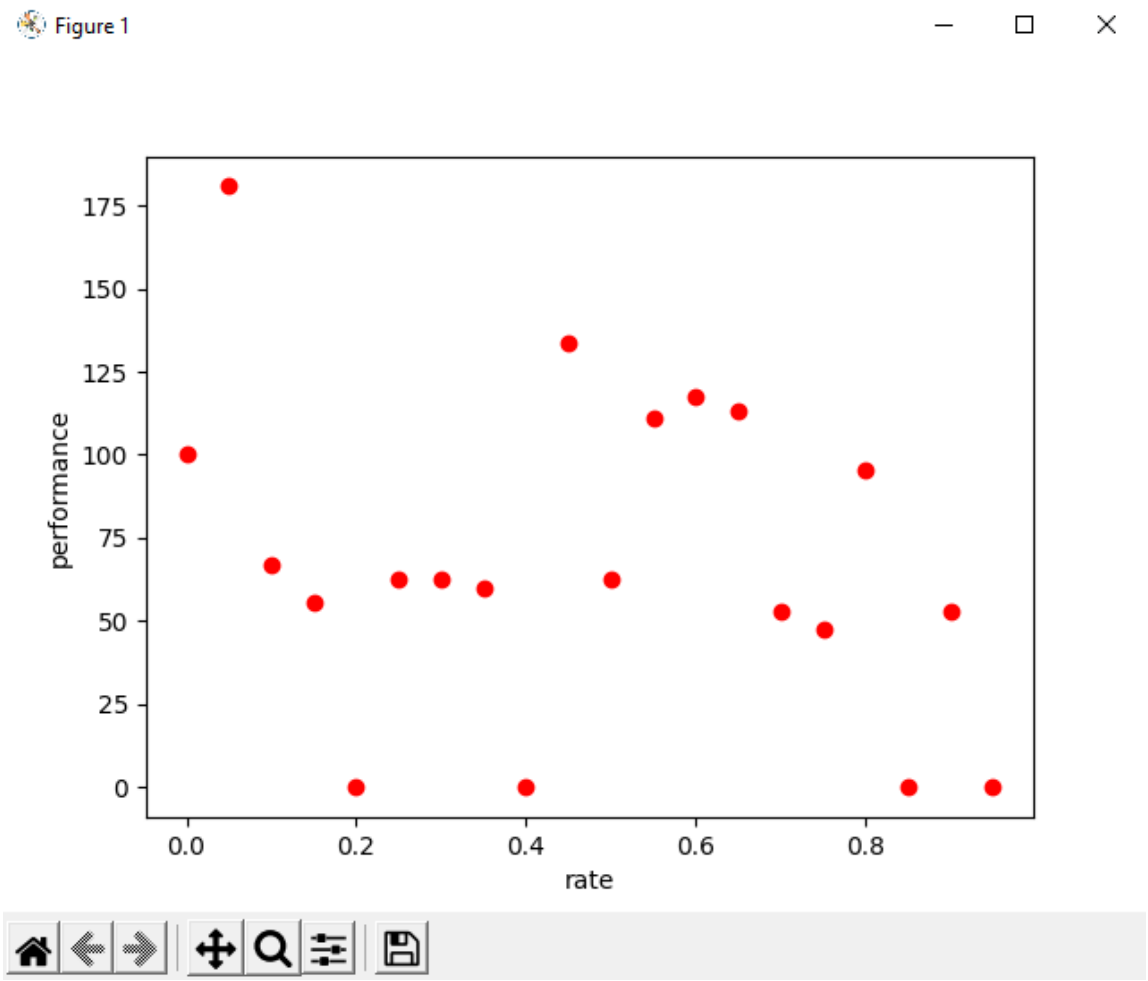
```
_____
The algorithm find 2 answers:
[2, 4, 1, 3, 0]
[3, 1, 4, 2, 0]
The execution time is: 0.019984960556030273
performance: 100.07525380862054
```

بخش آنالیز

تاثیر نرخ mutation :

```python
def mutation_rate_effect(N):
    population_size = 100   #fixed
    rate = 0
    for i in range(20):
        performance = GA_performance(N, population_size, rate)
        plt.plot(rate, performance, marker="o", color="red")
        rate += 0.05
    plt.xlabel('rate')
    plt.ylabel('performance')
    plt.show()
```

برنامه ۲۰ بار با نرخ های مختلف اجرا شده و نتیجه آن در نمودار زیر آورده شده است:



تاثیر تعداد جمعیت اولیه:

```python
def population_size_effect(N):
    rate = 0.2    #fixed
    population_size = 100
    for i in range(20):
        performance = GA_performance(N, population_size, rate)
        plt.plot(population_size, performance, marker="o", color="blue")
        population_size += 100
    plt.xlabel('population size')
    plt.ylabel('performance')
    plt.show()
```

برنامه ۲۰ بار با تعداد جمعیت های مختلف (ضمن محاسبات سنگین) اجرا شده و نتیجه آن در نمودار زیر آورده شده است :