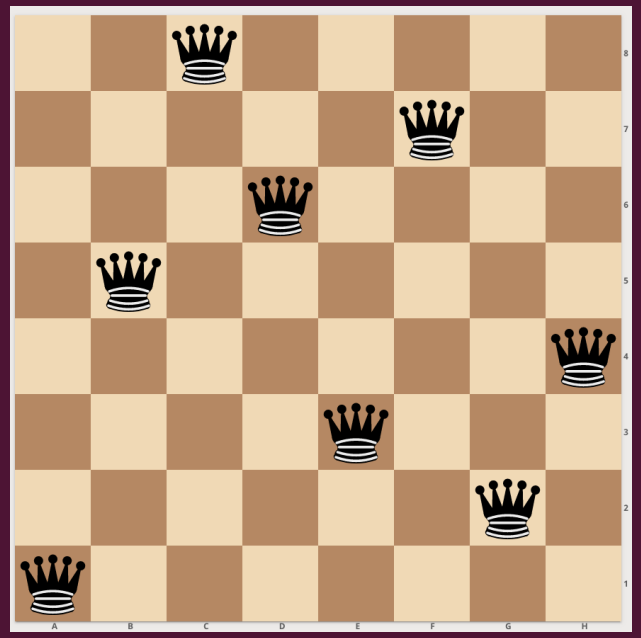
تمارش شماره دوم

تهیه کنندگان:
حسین دماوندی
امیرحسین وطنی باف

زیرنظر استاد محترم:
دکتر امیرفرید امینیان مدرس

حل مسئله 8 وزیر

به کمک الگوریتم های تکاملی



مقدمات و تعریف توابع جانبی

```
•[699]: import random
        import numpy as np
```

install requirements:

```
pip install numpy
```

برای اجرای این برنامه نیاز به نصب کتابخانه numpy خواهید داشت.

همچنین پیشنهاد می‌شود سورس کد برنامه را در محیط Jupyter Notebook یا Google Colab اجرا فرمایید

مقدمات و تعریف توابع جانبی

تعریف متغیر سراسری sx برای سایز مسئله

```
[2]: sx = 1000
```

کدگذاری

CODING

مرحله اول در هر الگوریتم تکاملی کدگذاری یا همان coding می‌باشد.
در این مرحله می‌بایست تعیین کنیم که در هر کروموزوم چه اطلاعاتی باید ذخیره شود.

در مسئله 8 وزیر هر وزیر در ستون خودش اجازه بالا و پایین رفتن دارد و با جابجایی به اندازه کافی وزیر در هر ستون به جواب خواهیم رسید.
می‌بایست فقط تهدید های افقی و مورب را چک کنیم زیرا تهدید وزیر در ستون ها را نداریم.
از یک آرایه 8 تایی که در هر خانه اعداد 0 تا 7 بصورت باینری نگهداری می‌شوند (شماره ردیفی که وزیر ها قرار دارند با شروع از صفر) استفاده می‌کنیم.

در حل این مسئله ما تصمیم گرفتیم کلاس Queens را ایجاد کنیم و هر حالت را یک شی در نظر بگیریم. (در اسلاید بعدی قابل مشاهده است)

CODING

کدگذاری

```
[3]: class Queens:

         def __init__(self, rows):
             self.rows = rows
             self.h = fitness_value(rows)
             self.rowsbin = to_binary(rows)
```

همانطور که قابل مشاهده است در این بلاک کلاس Queens با اشیاء مختلفی تعریف شده.

row شامل یک آرایه 8 خانه ای هست که جایگاه هر وزیر را مشخص می‌کند.

برای راحتی کار ما در همین قسمت، مرحله ارزیابی یا همان evaluation را نیز انجام میدهیم و برای هر موجود محاسبه انتفاع نیز با استفاده از تابع fitness_value (که در اسلاید های آینده قابل مشاهده است) انجام میپذیرد.

باتوجه به اینکه در الگوریتم های تکاملی غالبا رشته باینری است تابعی برای تبدیل به باینری با عنوان to_binary در نظر گرفته شده.

کدگذاری

to_binary(rows)

[296]:
```
def to_binary(rows):
    rows_bin = ''
    for i in range(0,len(rows)) :
        r_bin = "{0:b}".format(int(rows[i]))
        while len(r_bin) < 3 :
            r_bin = '0' + r_bin

        rows_bin = rows_bin + r_bin

    return rows_bin
```

این تابع برای تبدیل به باینری مورد استفاده قرار می‌گیرد و هر کدام از مقادیر row را در 3 بیت ذخیره می‌کند و یک رشته 24 تایی باینری بر می‌گرداند. (برای درک بیشتر می‌توانید به مثال زیر توجه فرمایید)

[297]:
```
a=[4,0,7,3,1,6,2,5]
to_binary(a)
```

[297]:
```
'100000111011001110010101'
```

کدگذاری

to_decimal(rows_bin)

[298]:
```
def to_decimal(rows_bin) :

    rows = []
    for i in range(0,len(rows_bin),3):
        temp = rows_bin[i:i+3]
        temp = int(temp, 2)
        rows.append(temp)

    return rows
```

با توجه به نیاز تابعی برای تبدیل به اعداد دهدهی درنظر گرفته شده است (عملکرد این تابع در مثال زیر واضح تر است)

[299]:
```
a='100000111011001110010101'
to_decimal(a)
```

[299]:
```
[4, 0, 7, 3, 1, 6, 2, 5]
```

ارزیابی

fitness_value(rows)

[294]:
```
def fitness_value(rows):
    value = 0
    for i in range(0,len(rows)) :
        for j in range(i+1,len(rows)) :
            if (rows[i]==rows[j]) or (abs(rows[i]-rows[j]) == abs(i-j)) :
                value += 1

    return value
```

تعداد زوج وزیر هایی هم دیگر را می‌زنند توسط این تابع برگردانده می‌شود.(چک کردن ردیف ها و چک کردن اینکه آیا بصورت مورب یکدیگر را میزنند یا خیر)

پر واضح است که هرچه تعداد وزیر هایی که یکدیگر را می‌زنند کمتر باشد آن حالت از ارزش بالاتری برخوردار است.

مراحل یک الگوریتم تکاملی

- تولید جمعیت اولیه
- محاسبه برازندگی (ارزیابی جمعیت ورودی)
- انتخاب برای تولید مثل
- باز ترکیب والدین (تولید مثل)
- جهش فرزندان تولید شده
- محاسبه برازندگی (ارزیابی جمعیت فرزندان)
- انتخاب برای جایگزینی
- بررسی شرط توقف

1 - تولید جمعیت اولیه

first_population(size)

```
[300]: def first_population(size):
           population = []
           for i in range(size):
               row = []
               for j in range(0,8) :
                   row.append(random.randint(0 , 7))

               population.append(Queens(row))

           return population
```

در مرحله اول یعنی تولید جمعیت اولیه از **روش تصادفی** (تولید مقادیر تصادفی در بازه مجاز 0 تا 7 برای هر ژن)

این تابع در ازای تعداد size دریافتی جمعیت تصادفی ایجاد می‌کند. (مثال زیر عملکرد تابع را بطور واضح نشان میدهد)

```
[301]: a=first_population(2)
       print(a[0].rows,"@",a[1].rows)
```

```
[1, 3, 4, 5, 5, 5, 2, 3] @ [1, 6, 4, 7, 5, 0, 1, 5]
```

2 - تابع برازش

fitness_value(x, y)

همانطور که قبلا در بخش ارزیابی معرفی شد این اقدام با استفاده از تابع fitness_value(rows) انجام می‌پذیرد

[294]:
```
def fitness_value(rows):
    value = 0
    for i in range(0,len(rows)) :
        for j in range(i+1,len(rows)) :
            if (rows[i]==rows[j]) or (abs(rows[i]-rows[j]) == abs(i-j)) :
                value += 1

    return value
```

3 - عملگر انتخاب

select_parents(population,size)

روش نسبی - fitness بیشتر ؛ شانس انتخاب بیشتر

```
[302]: def select_parents(population,size):

           All_Parent = []
           sort_population = []
           weights = []
           sort_population = sorted(population, key=lambda p: p.h, reverse=True)

           for i in range(len(sort_population)):
               weights.append(i+1)

           All_Parent = random.choices(sort_population, weights = weights , k = size)

           return All_Parent
```

ابتدا جمعیت را بر اساس انتفاع مرتب کرده و برای هر کدام یک وزن (شانس) با توجه به جایگاهش داده می‌شود؛ و با توجه به وزن هر موجود شانس بیشتر برای انتخاب دارد

3 - عملگر انتخاب select_parents(population,size)

روش مسابقه ای

```
[303]: def select_parents_computation(population,size):

           All_Parent = []
           sort_population = []

           for i in range(0,size) :
               temp_population = population.copy()
               random_population = []
               for j in range(0,len(population)//5) :
                   index = random.randint(0,len(population)-1-j)
                   random_population.append(temp_population[index])
                   temp_population.pop(index)

               sort_population = sorted(random_population, key=lambda p: p.h, reverse=False)
               All_Parent.append(sort_population[0])


           return All_Parent
```

ابتدا انتخاب تصادفی از بین موجودات داریم و سپس بر اساس انتفاع آن ها را مرتب کرده و بهترین را انتخاب می‌کنیم

4 - عملگر های الگوریتم ژنتیک

cross_over(parents)

تابع اصلی تولید مثل در تصویر زیر آمده است که به دو روش one point و uniform (تابع های اسلاید بعدی) فرزندان جدید را برمی‌گرداند

[304]:
```
def cross_over(parents) :
    childeren = []
    for i in range(0,len(parents)-1) :

#         rowbin_child1 = cross_one_point(parents[i].rowsbin, parents[i+1].rowsbin)
        rowbin_child1 = cross_uniform(parents[i].rowsbin, parents[i+1].rowsbin)
        rowbin_child2 = rowbin_child1[::-1]

        row_child1 = to_decimal(rowbin_child1)
        row_child2 = to_decimal(rowbin_child2)

        childeren.append(Queens(row_child1))
        childeren.append(Queens(row_child2))

        i += 1

    return childeren
```

4 – عملگر های الگوریتم ژنتیک

select_parents(population,size)
cross_one_point_z(parent1,parent2)

روش اول: ترکیب یک نقطه ای (One point)

[305]:
```
def cross_one_point(parent1, parent2) :
    index = random.randint(1,23)
    child = ''

    for i in range(0,len(parent1)) :
        if i < index :
            child = child + parent1[i]
        else :
            child = child + parent2[i]

    return child
```

[306]:
```
cross_one_point('100000111011001110010101','101100111011001010010001')
```

مثال

[306]:
```
'100000111011001010010001'
```

4 – عملگر های الگوریتم ژنتیک

select_parents(population,size)
cross_one_point_z(parent1,parent2)

روش دوم: ترکیب یکنواخت (Uniform)

[307]:
```
def cross_uniform(parent1, parent2) :
    child = ''

    for i in range(0,len(parent1)) :
        if random.choice(['1', '2']) == '1' :
            child = child + parent1[i]
        else :
            child = child + parent2[i]

    return child
```

5 – عملگر های الگوریتم ژنتیک

mutate(childeren)

جهش (mutation) – جهش ژنتیکی

یکی از کروموزوم هارا بصورت تصادفی انتخاب کرده و یکی از ژن های آن را (اینبار هم با احتمال 50 درصد) تغییر می‌دهیم

[308]:
```
def mutate(childeren):
    mutatant = childeren

    for i in range(0,len(mutatant)) :
        index = random.randint(0,len(mutatant[i].rowsbin)-1)
        if random.random() < 0.2:
            b = random.choice(['0', '1'])
            mutatant[i].rowsbin = mutatant[i].rowsbin[:index] + b + mutatant[i].rowsbin[index+1:]

    return mutatant
```

6 – جایگزینی نسل

replacement(population, parents, childeren)

```
[309]: def replacement(population, parents, childeren) :
           population_copy = population.copy()
           parents_copy = parents.copy()
           childeren_copy = childeren.copy()

           newlistChild = []
           newlistParent = []
           newlistChild = sorted(childeren_copy, key=lambda ch: ch.h, reverse=False)      #STEP 6
           newlistParent = sorted(parents_copy, key=lambda p: p.h, reverse=True)

           for i in range(0,int((len(parents_copy)*30/100)+0.5)) :
               newlistParent[i] = newlistChild[i]

           newpopulation = newlistParent
       #     sort_pop = sorted(population_copy, key=lambda p: p.h, reverse=True)

           for i in range(0,len(newpopulation)) :
               index = random.randint(0,len(population)-1)
               population[index] = newpopulation[i]

           return population
```

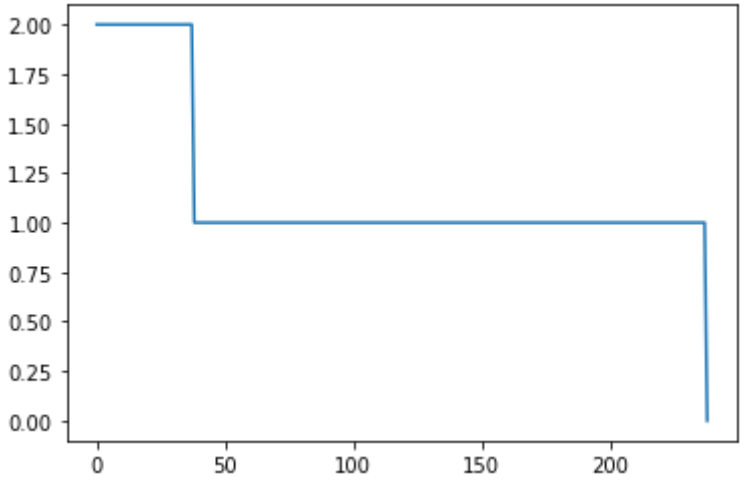
توسط این تابع یکی از بین والدین و یکسری را انتخاب میکنیم و به نسل بعدی انتقال می‌دهیم (نسبت 30 – 70). بخش بزرگی از والدین و بخش کمتری از فرزندان به دلیل جلوگیری از همگرا شدن سریع (در الگوریتم های تکاملی میبایست سرعت همگرایی کم باشد زیرا ممکن از موجودی که میتواند بهترین نقطه را پیدا کند از بین برود)

main()

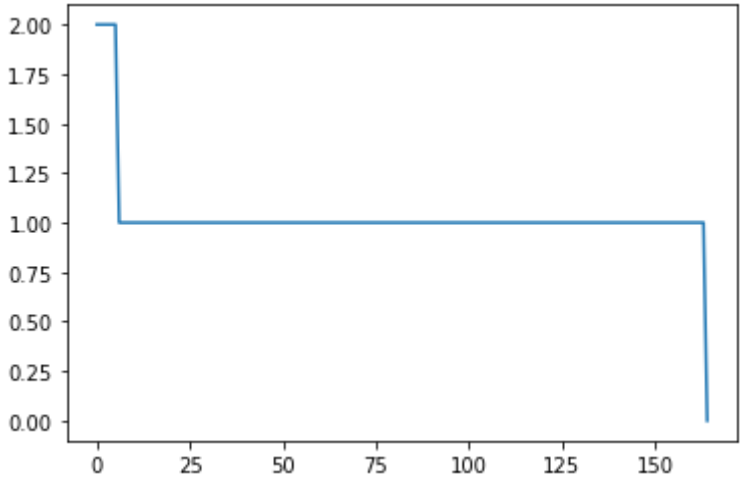
تابع اصلی

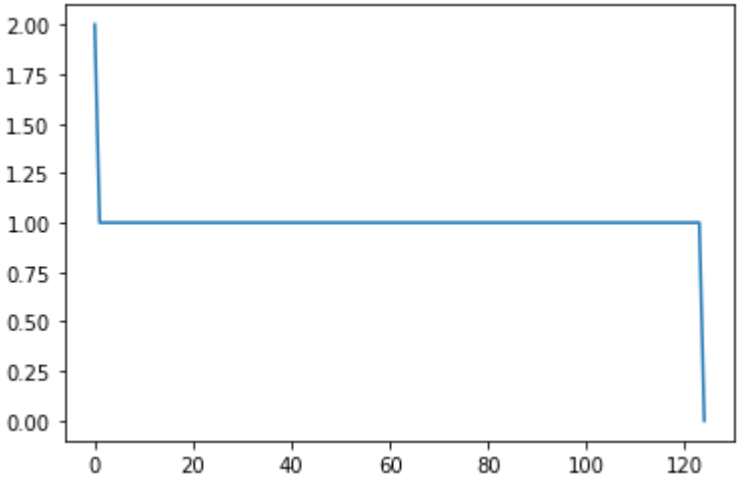
[146]:
```
def main():
    draw_fitness = []    #for draw result
    cnt=0                # for count number of repeat best in generations
    Best=100             # maximum of ackley
    gen=0                # for count number of generations
    target=0             # minimum of ackley
    firstpopulation = first_population(sx)   #STEP 1
    current_population = firstpopulation    #STEP 2
    while True :
        if gen > 1000 :
            break
        if cnt == 800 :
            break
        parents = select_parents(current_population,sx//2)   #STEP 3
        childeren = cross_over(parents)      #STEP 4
        childeren = mutate(childeren)        #STEP 5
        next_population = replacement(current_population, parents, childeren)  #STEP 6
        BestNew = GetBest(current_population)      #STEP 7

        if Best == BestNew :
            cnt += 1
        elif Best > BestNew :
            cnt=0
            Best = BestNew
        else :
            cnt = 0

        print('gen :', gen)
        print("Best (in gen ",gen,") :",Best)
        print("********************")
        draw_fitness.append(Best)
        if Best == target :
            break

        current_population = next_population
        gen += 1
    hist = draw_fitness
    plt.plot(hist)
```

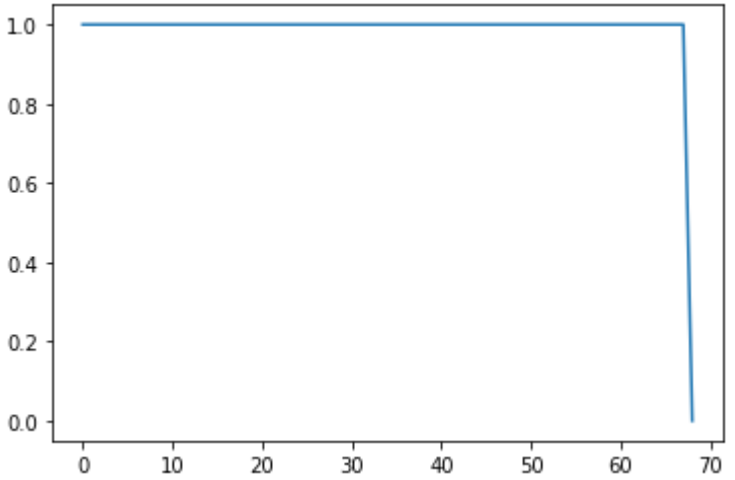
نمودار رسیدن به انتفاع ایده آل در چند اجرای متفاوت



همانطور که مشاهده می‌شود در نسل های متمادی مقدار برازش ما
به مقدار ایده آل 0 می‌رسد





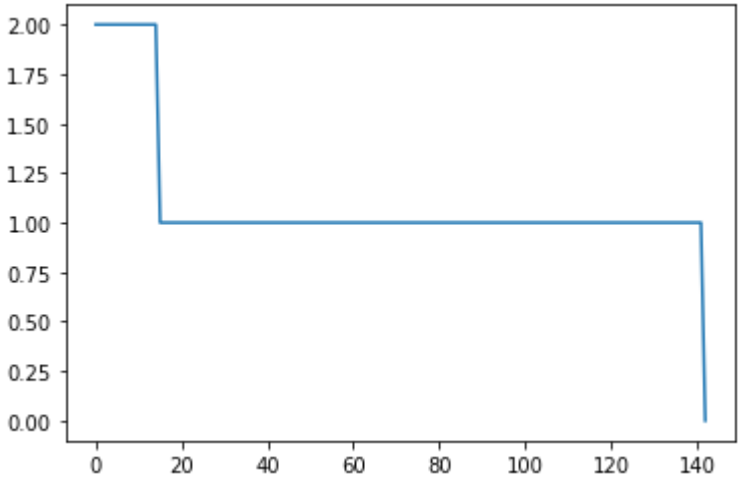


نتیجه گیری

تفاوت در پاسخ انتخاب پدر و مادر به روش های نسبی و مسابقه ای

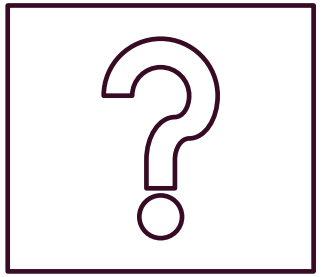
روش نسبی:

در این روش ما با توجه به انتفاع به هر موجود شانسی را انتصاب میدهیم (مدت زمان رسیدن به پاسخ معمولا بین 100 تا 300 نسل متغیر)



روش مسابقه ای:

خیلی کم پیش می‌آید تا اینکه در روش مسابقه ای به جواب برسیم به نظر میرسد که روش مناسبی برای این مسئله نمی‌باشد زیرا انتفاع و پویش به یک حد نیست و انتفاع بیشتر از پویش است



در مسئله 8 وزیر اگر سرعت همگرایی زیاد باشد و به سمت این برویم که زوج وزیر های کمتری هم دیگر را بزنند ممکن است حالت ایده آل از بین برود و مینیمم های دیگر به عنوان پاسخ در نظر گرفته شوند

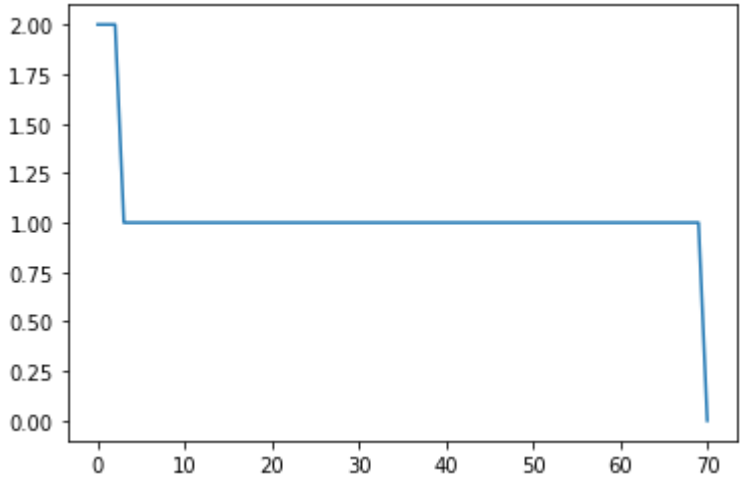
مستندات در فایل های پوشه select parents قابل مشاهده است

نتیجه گیری

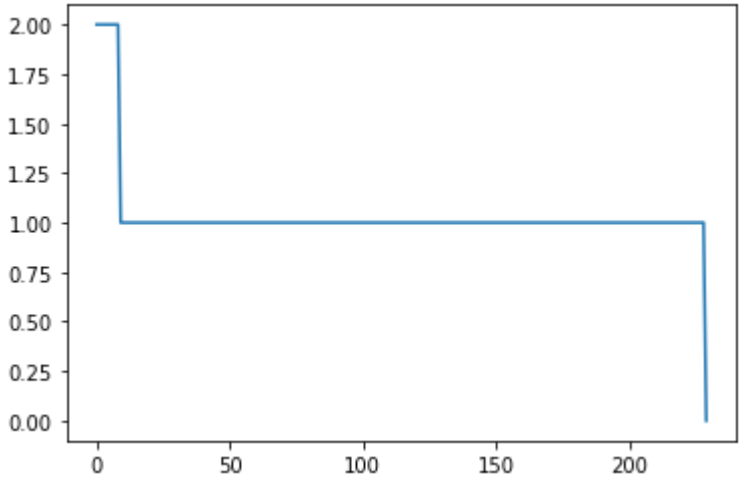
تفاوت در پاسخ تولید مثل به روش های وان پوینت و یونیفرم

زمان رسیدن به پاسخ در روش یونیفرم نسبت به وان پوینت بیشتر است. دلیل این اتفاق میتواند تنوع زیاد در فرزندانی که بوجود می‌آیند باشد و ممکنه فرزندانی بوجود آیند که از جواب بسیار دور تر باشند

روش وان پوینت:



روش یونیفرم:



مستندات در فایل های پوشه cross over قابل مشاهده است