گزارش کد پیاده سازی ژنتیک :

در ابتدا با class graph شروع میکنیم. این کلاس برای ساخت گراف ماست. و اطلاعات مربوط به گرافمان را در آن ذخیره می کنیم. که این اطلاعات راس ما و راس هایی که به آن ها یال دارند هست.

```
class Graph:
    def __init__(self, v):
        self.v = v
        self.graph = defaultdict(list)

    def addEdge(self, s, d):
        self.graph[s].append(d)

    def countEdge(self):
        b = 0
        for i in range (self.v):
            a = len(self.graph[i])+1
            b = b + a
        return b

    def parent(self, s, d):  # check if s eating d
        for v in self.graph[s]:
            if (v == d):
                return True
        return False
```

در تابع addEdge دو ورودی داریم که یک راس و دومی راسی است که یال بین آن ها وجود دارد و کار این تابع اضافه کردن یال برای راس s هست.

تابع بعدی تابع countEdge یال های گراف را می‌شماریم. ازین تابع در تابع هدفامن استفاده خواهیم کرد.

و تابع parent ورودی مشابه تابع بالا میگیرد اما این تابع برای چک کردن این است که آیا رابطه ی شکارچی بودن راس اول و شکار بودن راس دوم برقرار است ویا خیر.

```
lines = []
with open("input.txt") as f:
    lines = f.readlines()
# INITIAL DATA & PARAMETERS
n = int(lines[0])
lines.pop(0)
graph = Graph(n)
for l in lines:
    e = l.split(' ')
    graph.addEdge(int(e[0]), int(e[1]))
```

از این بخش برای خواندن از روی فایل از این کد استفاده میکنیم و هر خط آن را در یک لیست که تعریف کردیم میریزیم. و خط اول فایل را دور میریزیم و از خط بعدی شروع میکنیم.

حال که تعداد راس ها را داریم یک گراف با کلاس graph میسازیم و برای دادن اطلاعات گراف هر خط را میخوانیم و با توجه به جداسازی عنصر اول را به عنوان راس و عنصر دوم را به عنوان یالی که خارج میشه میذاریم.

```
population_size = 100
max_pop_size = 200
crossover_coeff = 0.8
mutation_coeff = 0.4
max_iteration = 500
num_crossover = round(population_size*crossover_coeff)
num_mutation = round(population_size*mutation_coeff)
total = population_size+num_crossover+num_mutation
population = []
object_values = []
best_objective = 0
best_chromosome = np.zeros(n)
```

در این قسمت متغیر هامون رو تعریف میکنیم :

population_size هست که جمعیت اولیه را به ما میدهد.

Max_pop_size بیشترین تعداد ممکن جمعیت هست

Crossover_coef ضریبی از جمعیت که cross over بر روی ان ها انجام میشه.

Mutation_coed ضریبی از جمعیت که جهش داده میشوند.

Max_iteration بیشترین تعداد ممکن برای تکرار حلقه ماست.

Num_Crossover تعدادی از جمعیت که cross over بر روی ان ها انجام میشه.

Num_Mutation تعدادی از جمعیت که جهش داده میشوند.

Population مجموعه ای از اعضاست.

Object_valuesمقدار تابع هدف هر عضو است.

Best_objectives بیشترین مقدار تابع هدفمان است.

Best_chromosome بهترین و بهینه ترین پاسخ است.

```python
def objective(sol):
    e = graph.countEdge()
    for i in range(n):
        a = sol[i]
        for j in range(i,n):
            b = sol[j]
            if(graph.parent(b,a) and i!=j):
                e -= 1
    return e
```

تابع objective تابع هدف ما است. در این تابع تعداد کل یال هارو بدست اورده و با چک کردن نا بجایی ها از این یال ها یکی کم میکنیم و در اخر هم تعداد یال ها را برمیگرداند.

```python
def recombination(parent1, parent2):
    l = np.random.randint(n)
    r = np.random.randint(n)
    while(l==r):
        l = np.random.randint(n)
    if(l>r):
        temp = l
        l = r
        r = temp

    child1 = np.zeros(n, dtype=int)
    child2 = child1.copy()
    contain1 = child1.copy()
    contain2 = child1.copy()

    for i in range(l, r+1):
        child1[i] = parent1[i]
        contain1[parent1[i]-1] = 1

    for i in range(l, r+1):
        child2[i] = parent2[i]
        contain2[parent2[i]-1] = 1

    index_child = 0
    index_parent = 0
    while index_child < n and index_parent < n:
        if index_child >= l and index_child <= r:
            index_child = index_child + 1
            continue
```

تابع recombination . در این تابع به دنبال ساخت فرزند های جدید هستیم. پس ابتدا دو parent به ورودی میدهیم و خروجی دو children به ما برمیگرداند.

به این شکل عمل میکنیم که ابتدا ۲ نقطه تصادفی پیدا میکنیم که اگر این ۲ برابر بودند دوباره انتخاب میکنیم و اگر l از r بزرگ تر بود این دو را جابجا میکنیم.

حال برای ساخت child1 این چنین شروع میکنیم:

درایه های l تا r را در خانه های متناظر child1 از درایه های parent1 جایگذاری میکنیم. و بقیه را صفر نگهداری میکنیم.

```
    while index_parent < n and contain1[parent2[index_parent]-1]:
        index_parent = index_parent + 1
    if index_parent == n:
        break
    child1[index_child] = parent2[index_parent]
    index_child = index_child + 1
    index_parent = index_parent + 1

index_child = 0
index_parent = 0
while index_child < n and index_parent < n:
    if index_child >= l and index_child <= r:
        index_child = index_child + 1
        continue
    while index_parent < n and contain2[parent1[index_parent]-1]:
        index_parent = index_parent + 1
    if index_parent == n:
        break
    child2[index_child] = parent1[index_parent]
    index_child = index_child + 1
    index_parent = index_parent + 1

return (child1, child2)
```

حلقه تا زمانی که index_child و index_parent کوچکتر از n هستند اجرا میشود. با کمک این شمارنده ها قرار است خانه‌های صفر child را پر کنیم.

در شرط اول بررسی میکنیم اگر index_child بین دو نقطه l و r است قطعا صفر نیست و شمارنده حرکت کند تا ازین بازه خارج شود.

در حلقه میانی علاوه بر چک کردن شمارنده index_parent که قرار است روی لیست contain1 با کمک parent2 حرکت کند. به این صورت که از ایندکس شماره صفر parent2 عدد داخلش را میگیرد و مقدارش را در contain1 چک میکند که صفر است یا یک.

اگر یک بود به این معنی است این عدد در child1 است و نمیتوان اضافه کرد و اینقدز ادامه میدهد (index_parent را زیاد میکند) تا به اولین صفر برسد.

وقتی صفر بود یعنی آن عدد استفاده نشده است

البته قبل صفر شدن نیز یک شرط گذاشتم که چک میکند index_parent با n مساوی شده باشد یعنی ته لیست رسیدیم و کار تمام است اما اگر نرسیده باشیم سه خط اخر اجرا میشود

که همان قرار دادن مقدار parent2 در خانه‌های صفر child1 است به صورت غیر تکراری و بعدش هردو شمارنده را جلو میبرد.

به طور مشابه همین روند برای child2 انجام میدهد.

```
while len(population) < population_size:
    sequence = [i for i in range(1, n + 1)]
    temp = random.sample(sequence, n)
    population.append(temp)
    object_values.append(objective(temp))
```

در این حلقه شروع میکنیم به اندازه ای که برای جمعیت تعریف کردیم جمعیت (اعضا به صورت تصادفی انتخاب شده میشوند) میسازیم و آن را در متغیر population میریزیم و همچنین مقادیر تابع هدف های این اعضا را هم در متغیر object_values ذخیره میکنیم.

```
iteration = 0
pl = []
while iteration < max_iteration:
    summation = sum(object_values)
    pr = []
    cumulative_pr = []
    for i in range(population_size):
        pr.append(object_values[i] / summation)
    cumulative_pr.append(pr[0])
    for i in range(1, population_size - 1):
        temp = cumulative_pr[i - 1] + pr[i]
        cumulative_pr.append(temp)
    cumulative_pr.append(1)
```

این حلقه، حلقه اصلی برنامه ماست.

یک متغیر iteration تعریف میکنیم که شرط ما کمتر بودن این متغیر از حداکثر مقداری که در بالا تعریف کردیم هست.

در ابتدای این حلقه جمع تمامی object_values ها و ۲ لیست داریم.

در این حلقه از جهش و تابع recombination استفاده میکنیم.

```
for i in range(0, num_crossover, 2):
    p1 = 0
    temp = np.random.rand()
    while cumulative_pr[p1] < temp:
        p1 = p1 + 1
    p2 = p1
    while p1 == p2:
        temp = np.random.rand()
        p = 0
        while cumulative_pr[p] < temp:
            p = p + 1
        p2 = p
    parent1 = population[p1]
    parent2 = population[p2]
    children = recombination(parent1, parent2)
    child1 = children[0]
    child2 = children[1]
    population.append(child1)
    object_values.append(objective(child1))
    population.append(child2)
    object_values.append(objective(child2))
```

دراین حلقه for ما همان تابع recombination را فراخوانی میکنیم و با استفاده از آن جمعیت جدید میسازیم و خود این child هارا در ارایه ای که جمعیت را نگهداری میکردیم اضافه کرده و همچنین مقدار هدف های آن ها را

این حلقه برای جهش است. به این صورت که ۲ عدد تصادفی انتخاب میکنیم و قرار هست درایه های خانه ی این ۲ عدد را با هم جا بجا کنیم. حال انتخاب این خانه که این عمل را رو آن اجرا کنیم هم به صورت تصادفی است. حال این مدل جدید را هم در هارا در ارایه ای که جمعیت را نگهداری میکردیم اضافه کرده و همچنین مقدار هدف آن را.

```
for i in range(num_mutation):
    temp = np.random.randint(num_crossover)
    temp = temp + population_size
    mutated = population[temp]
    temp1 = np.random.randint(n)
    temp2 = np.random.randint(n)
    while ((graph.parent(mutated[temp1], mutated[temp2]) and (temp1 < temp2))
        temp1 = np.random.randint(n)
        temp2 = np.random.randint(n)
    temp_data = mutated[temp1]
    mutated[temp1] = mutated[temp2]
    mutated[temp2] = temp_data
    population.append(mutated)
    object_values.append(objective(mutated))
```

```
best_objective = max(object_values)
best_arg = np.argmax(object_values)
best_chromosome = population[best_arg]
```

در این بخش بیشترین که بهترین مقدار تابع هدف است را در متغیرbest_objective و پاسخ متناطر با آن مقدار هدف را در best_chromosome میریزیم که این همان بهترین پاسخ ما است در آن مرحله.

در این بخش ممکن هست تعداد جمعیت ما از حداکثری که تعریف کرده ایم بیشتر شود پس میایم آن را sort میکنیم تا بهترین های آن را نگهداریم و بقیه را دیگر نیاز نداریم. در انتهای حلقه هم iteration را یکی افزایش میدهیم تا در مرحله بعد در شرط حلقه اصلی بررسی شود . همچنین مقدار هدف هارا هم در یک ارایه ی جدا میریزیم برای رسم نمودار و همچنین بررسی میکنیم که اگر best_objective با تعداد یال های ما برابر باشد به پاسخ رسیدیم.

```
if len(population) > max_pop_size:
    temp_population = []
    temp_objective = []
    args = np.argsort(object_values)
    for i in range(max_pop_size):
        temp = len(population) - 1 - i
        temp_population.append(population[args[temp]])
        temp_objective.append(object_values[args[temp]])

    population = temp_population
    object_values = temp_objective
    population_size = max_pop_size
# print(best_objective)
iteration = iteration + 1
# print('graph : ',graph.countEdge()-1)
pl.append(best_objective)
if(graph.countEdge() - 1 == best_objective):
    break
```

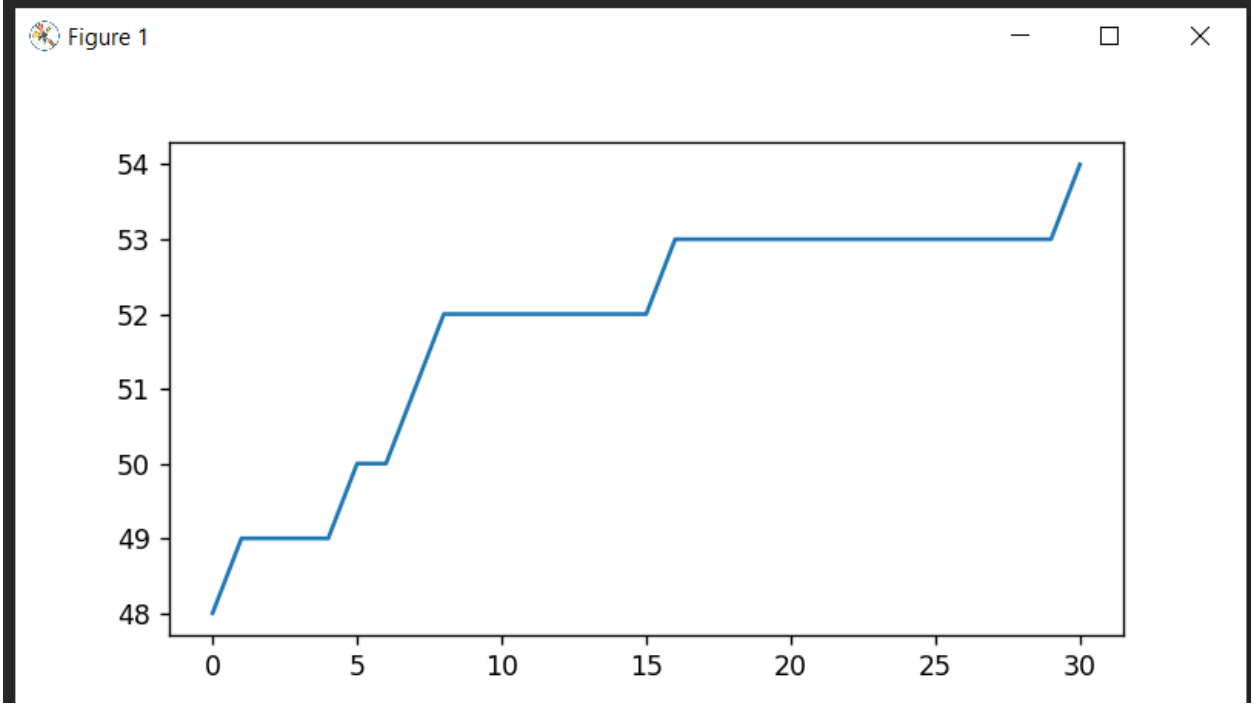
```
print(best_chromosome)
print(best_objective)
# print(pl)
plt.plot(pl)
plt.show()
```

در اخر هم بهترین جواب و مقدار تابع هدف را چاپ میکنیم.

و با استفاده از دو خط آخر نموار آن را رسم میکنیم.

یک نمونه از خروجی کد :

```
[20 17  3  4  2  1 10 19  8 16 15 11  7  6 18 12 13 14  9  5]
54
```



در انتها هم در مقایسه با الگوریتم sa به جوابی بهتری برای این مسئله میرسیم اما الگوریتم sa نسبت به ژنتیک ساده تر است.