- **نحوه تولید جمعیت اولیه:**

0	1	2	3	...	31	32	33
V_1	V_2	V_3	V_4	...	V_{31}	V_{32}	V_{33}

همانطور که در کروموزم بالا دیده میشود نحوه تولید جمعیت براساس یک آرایه 34 تایی میباشد که خانه **i**-ام نشان دهنده راس **i**-ام میباشد و اعداد قرار گرفته در خانه **i**-ام نشان دهنده راسی است که راس **i**-ام به آن وصل میباشد. از طریق این کروموزم میتوان خوشه ها را پیدا کرد.

این کروموزم را از طریق تابع choromosome_safty() میسازیم به طوری که مقادیر قرار گرفته در هر خانه معتبر باشد. یعنی حتما یالی بین آن راس ها باشد. **جمعیت اولیه را 100 انتخاب میکنیم.**

.

- **تابع fitness :**

در تابع def fitness(choromosome) یک تغییر مهم صورت گرفته است. برای آنکه حجم محاسباتی کم شود در هر بار

```
def choromosome_safty():
    choromosome=[]
    #population is 100
    for i in range(100):
        temp_gene=[]
        for j in range(number_of_node):
            nonzeroind_index=np.nonzero(adjaceny_matrix[j,:])
            conncted_node_j=random.choice(nonzeroind_index[0])
            temp_gene.append(conncted_node_j)

        choromosome.append(temp_gene)
    return choromosome
```

اجرای این تابع فقط 50 تا از بهترین جمعیت ها نگه داری میشود.

```
the_best_fetnes=[]
for i in range(len(fetnes)):
    the_best_fetnes.append((fetnes[i],choromosome[i],cluster[i]))
the_best_fetnes.sort(reverse=True,key=lambda x:x[0])
the_best_fetnes=the_best_fetnes[0:50]

fetnes=[]
choromosome=[]
cluster=[]
for i in range(len(the_best_fetnes)):
    fetnes.append(the_best_fetnes[i][0])
    choromosome.append(the_best_fetnes[i][1])
    cluster.append(the_best_fetnes[i][2])

return fetnes,cluster,choromosome;
```

- **Crossover** :

Index	0	1	2	...	31	32	33
Parent1	V_1	V_2	V_3	...	V_{31}	V_{32}	V_{33}
Parent2	V_1	V_1	V_1	...	V_1	V_1	V_1
vector	0	0	1	...	1	1	0
child	V_1	V_2	V_1	...	V	V_1	V_{33}

برای این مرحله یک بردار باینری با ابعاد 34×1 میسازیم. اگر و عملیات crossover را مطابق زیر انجام میدهیم.

یعنی اگر خانه **i**-ام بردار ما صفر باشد برای فرزند آن ژن را از Parent1 میگریم و در غیر این صورت از Parent2.
در این مرحله از هر جفت Parent دو فرزند تولید میشود. برای انتخاب والد نیز از **roulette selection** استفاده میکنیم به طوری که در هر مرحله **50** درصد از بهترین جمعیت انتخاب میشوند.

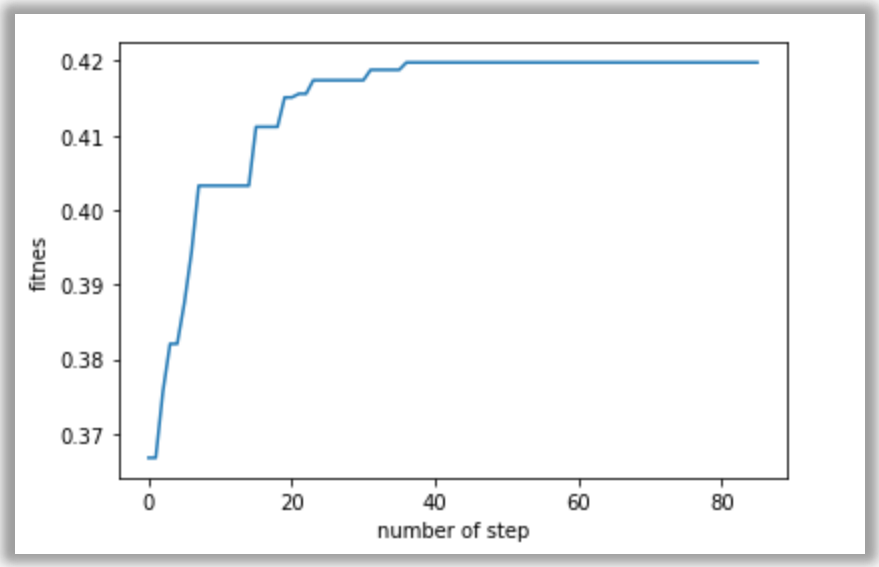
- **:Mutation**

در این مرحله ابتدا پارامتر **Pm=0.6** را تعریف میکنیم. و سپس عددی تصادفی بین **0** تا **1** به طور رندوم تولید میکنیم. اگر عدد تولید شده بزرگتر از 0.6 بود آنگاه جهش را انجام میدهیم. برای جهش ابتدا عدد طبیعی بین **0** و **5** تولید میکنیم که نشان دهنده تعداد ژن ها برای جهش میباشد. سپس بعد از مشخص شدن تعداد ژن ها برای جهش ایندکس ژن ها را به صورت تصادفی از کروموزم ها های تولید شده انتخاب میکنیم.

```python
def mutation(child,choromosome):
    pc=random.uniform(0, 1)
    if(pc>0.6):
        for i in range(len(child)):
            number_of_mutation=random.randint(0,5)
            indexs_mutation=random.sample(range(0,34),number_of_mutation)
            for j in indexs_mutation:
                nonzeroind_index=np.nonzero(adjaceny_matrix[j,:])
                conncted_node_j=random.choice(nonzeroind_index[0])
                child[i][j]=conncted_node_j
    child.extend(choromosome)
    return child
```

- **شرط توقف:**

شرط توقف را اینطور در نظر میگیریم که اگر تابع برازندگی بعد از **50** تکرار تغییری نکرد الگوریتم ما متوقف شود.



همانطور که در شکل بالا دیده میشود روند تغییرات تابع fetnes از تکرار چهلم به بعد ثابت میشود.

- **نتیجه:**

در نهایت بعد از چند تکرار به مقادیر زیر میرسیم:

➢ **توجه: در خوشه بندی و همینطور در گراف راس ها از صفرنام گذاری شده اند.**

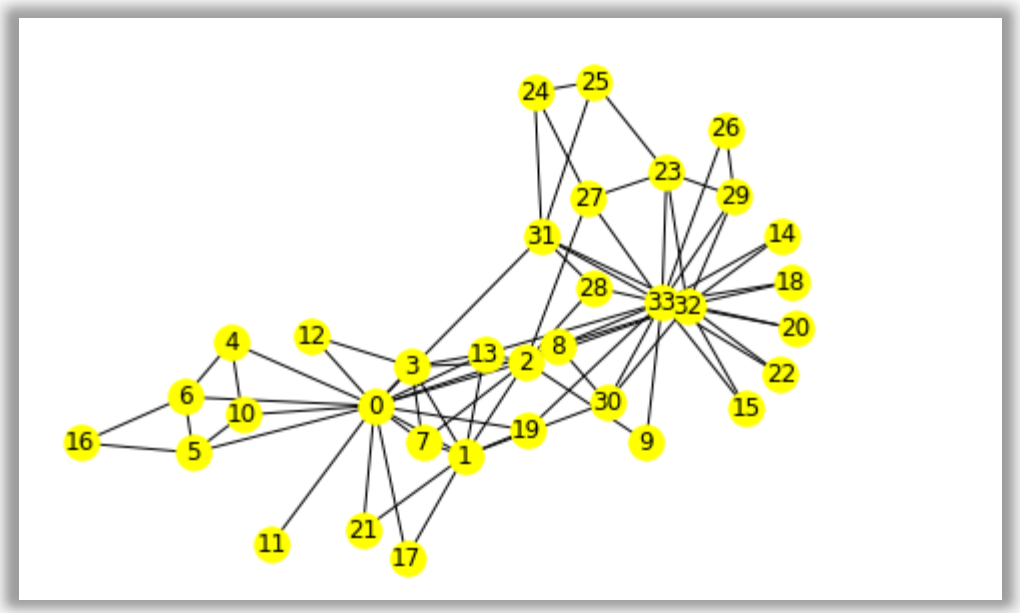
```
i=np.argmax(fetnes)
cluster=cluster[i]
print('maximum fetnes Q is:', max(fetnes))
```

```
maximum fetnes Q is: 0.41978961209730387
```

```
cluster
```

```
[[31, 25, 28, 23, 27, 24],
 [29, 26, 32, 14, 15, 18, 20, 22, 33, 9, 30, 8],
 [7, 0, 11, 12, 17, 3, 2, 13, 1, 19, 21],
 [5, 6, 16, 4, 10]]
```

- **گراف قبل از خوشه بندی:**



- **گراف بعد از خوشه بندی:**