گزارش پروژه تمرین سری دوم بخش دوم

اشکان ودادی گرگری

9713032

چکیده مطالب:

کد نوشته شده چهار بخش دارد و دو بخش نیز مربوط به گزارش است که هر بخش را به طور کامل توضیح خواهیم داد.

1. کلاس گراف
2. گرفتن فایل متنی از input.txt
3. اجرای الگوریتم ژنتیک
    a. مقداردهی اولیه
    b. تابع هدف
    c. حلقه اصلی
    d. الگوریتم cross over
    e. الگوریتم mutation
4. نمودار همگرایی
5. چند نمونه خروجی
6. مقایسه با الگوریتم SA

بخش اول – کلاس گراف:

- زنجیره ارتباطاتی که در txt به ما میدهند با کمک کلاس گراف به صورت یک لیست مجاورت نگهداری می‌کنیم.
- در این بخش یک کلاس طراحی کردیم که بتوانیم اطلاعاتمون رو به صورت گراف دخیره کنیم. برای این ذخیره سازی از لیست مجاورت استفاده کردیم.
- خود گراف را درgraph نگهداری می‌کند که یک دیکشنری از لیست است.
- این کلاس مجموعه راس‌های گراف را در V ذخیره میکند.
- مقدار یال‌های گراف را به صورت e نگه داری میکنیم و در هر مرحله اضافه کردن هر یال یکی بهش اضافه می‌شود.

```
class Graph:
    def __init__(self,v):
        self.v = v
        self.e = 0
        self.graph = defaultdict(list)
```

- متد addEdge() برای این است که به مجموعه رئوسمون یال اضافه کنیم. علاوه بر اضافه شدن یال به تعداد یال‌های گراف نیز یکی اضافه می‌کند.
- متد printGraph() برای این است که تابع گرافمان را چاپ کنیم.

- متد isReachable(a,b) برای این است چک می‌کند که ایا از a به b مسیر است یا نه؟ (آیا a شکارچی b است). این متد با کمک BFS پیامش میکند و مسیر را پیدا می‌کند.
- متدهای DFS و DFSUtil نیز متدهای مربوط به DFS گراف است که البته در الگوریتم استفاده نشده است.

```
class Graph:
    def __init__(self,v):
        self.v = v
        self.e = 0
        self.graph = defaultdict(list)

    def addEdge(self, s, d):
        self.graph[s].append(d)
        self.e = self.e + 1

    def isReachable(self,src,dest):
        visited =[False]*(self.v+1)

        queue=[]
        queue.append(src)

        visited[src] = True
        while queue:
            n = queue.pop(0)
            if n == dest:
                    return True
            for i in self.graph[n]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
        return False

    def printGraph(self):
        for i in range(self.v):
            print("src=",i)
            for j in self.graph[i]:
                print(j)

    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for j in self.graph[v]:
            if j not in visited:
                self.DFSUtil(j, visited)
```

```
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
```

بخش دوم – گرفتن فایل متنی از input.txt:

- در این بخش فایل ورودی را میخوانیم.
- File_address همان آدرس فایل متنی ما است
- Lines خطوط فایل را در آن‌ها قرار می‌دهیم. که خط اول فایل همان تعداد راس‌های گرافمان است. و سایر خطوط را نیز با addEdge به گراف اضافه میکنیم.

```
# GET INPUT FUNCTION and COVERT TO GRAPH
file_address = "C:/Users/Ashkan/Desktop/Term 8/هوش/تمرین ها/تمرین 2 - بخش اول/SA/input.txt"

lines = []
with open(file_address) as f:
    lines = f.readlines() # har khato mirize to y khune array

V = int(lines[0])
graph = Graph(V+1)
count = 0
for line in lines:
    if(count!=0):
        edge = line.split(' ')
        graph.addEdge(int(edge[0]),int(edge[1]))
    count += 1
```

بخش سوم – اجرای الگوریتم ژنتیک:

1. **مقداردهی اولیه:**

- مقدارهای اولیه شروع الگوریتم را در این قسمت مقداردهی می‌کنیم.
- N تعداد حیوانات (راس‌های گراف) است. در الگوریتم نیز اندازه آرایه جواب قرار است باشد.
- Population_size اندازه آغازی جمعیت است.
- Max_pop_size اندازه حداکثری جامعه است.
- Crossover_coeff ضریبی از جمعیت که روی آن‌ها عملیات cross_over انجام می‌شود.
- Mutation_coeff ضریبی از جمعیت که روی آن‌ها عملیات جهش انجام می‌شود.
- Max_iteration حداکثر تعداد دفعاتی که حلقه تکرار می‌شود.

```
# INITIAL
n = V
population_size = 200
max_pop_size = 600
crossover_coeff = 0.7
mutation_coeff = 0.04
max_iteration = 500
```

- Num_crossover تعداد اعضایی از جمعیت که crossover انجام میشود.
- Num_motation تعداد اعضایی از جمعیت که جهش روی آن انجام می‌شود.

```
num_crossover = round(population_size * crossover_coeff)
num_mutation = round(population_size * mutation_coeff)
total = population_size + num_crossover + num_mutation
```

- Population اعضای جمعیت را نگه میدارد.
- Object_values مقدار تابع هدف هر عضو جامعه را نگه میدارد.
- Best_objectives مقدار ماکسمم تابع هدف در کل جمعیت است.
- Best_chromosome خود عضوی که تایع هدف بهینه برای آن است.

```
population = []
object_values = []
best_objectives = 0
best_chromosome = np.zeros(n)
```

- حلقه تولید k جمعیت اولیه به این صورت است که تا زمانی که population به اندازه جمعیت اولیه بشود چینش تصادفی تولید می‌کند و به population اضافه می‌کند و همچنین مقدار بهینش را به object_values اضافه می‌کند.

```
# INITAIL K Population
while len(population) < population_size:
    sequence = [i for i in range(1,n+1)]
    solution = random.sample(sequence, n)
    population.append(solution.copy())
    object_values.append(objective(solution))
```

2. تابع هدف:

- تابع هدف مسئله به این صورت است که تعداد ترتیب‌های نادرست را می‌شماریم. روش این کار نیز به این صورت است که لیستی از ترتیب هر گره داریم و چک میکنیم ایا مسیر وجود دارد یا نه؟ اگر مسیر وجود داشت ولی جاشون برعکس بود یک واحد به q اضافه میکند. اما در نهایت q را از مقدار یال‌ها کم می‌کنیم. در نهایت هدف ما این است به تعداد یال‌های گراف برسیم.

```
def objective(sol):
```

```
    q = 0
    for i in range(n):
        for j in range(i,n):
            if(graph.isReachable(sol[j],sol[i]) and i!=j):
                q = q + 1
    return graph.e-q
```

3. حلقه اصلی:

- بدنه اصلی الگوریتم بعد مقدار دهی اولی اینجا انجام می‌شود.
- مشابه کدی که در کلاس کارگاه زده شد.
- تا زمانی که تعداد iteration ها از تعداد کل max_iteration کمتر باشد حلقه انجام می‌شود.
- بخش cross over و mutation را در ادامه به طور دقیق توضیح میدهم.
- در اخر هر جهش و cross over ، بهترین عضو جمعیت را بر اساس مقدار تابع هدف آن پیدا می‌کنیم و در best_objective ذخیره میکنیم و اندیس آن را در best_arg و خودش را در best_chromosome ذخیره می‌کنیم.
- همچنین اگر در یک مرحله از تولید مثل و جهش تعداد اعضای حاصل از ماکسیموم جمعیت جامعه بیشتر باشد ابتدا تمام اعضا را بر اساس مقدار تابع هدف مرتب میکنیم و به تعداد max_pop_size نگه میداریم و باقی را دور می‌ریزیم.
- Best_objective_plot برای رسم نمودار استفاده شده است که یک لیست است و مقدار ماکسیموم تابع هدف کل جمعیت در هر iteration را ذخیره می‌کند.

```
# MAIN LOOP
iteration = 0
best_objective_plot = []
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)

    # CROSS OVER
    for i in range(0, num_crossover, 2):
       . . .

    # MUTATION
    for i in range(num_mutation):
       . . .
```

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

    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)
    best_objective_plot.append(best_objective)
    if (best_objective == graph.e):
        break

    iteration = iteration + 1

print(best_chromosome)
print(best_objective)
```

- در انتها نیز بهترین عضو و مقدار تابع هدفش را چاپ می‌کنیم.

4. الگوریتم Cross Over:

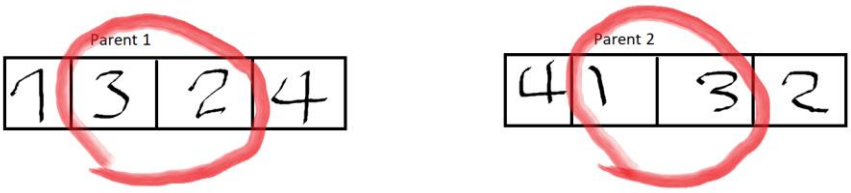
- دو بخش دارد:
  - انتخاب اعضای برگزیده برای کراس اور
  - اجرای عمل کراس اور
- برای انتخاب اعضای برگزیده کراس اور با توجه به تعداد کراس اور مجاز هر مرحله دوتا دوتا تصادفی جدا میکنیم و به تابع cross_over میدهیم. خروجی تابع cross_over دوتا children خواهد بود.

```
# CROSS OVER
    for i in range(0, num_crossover, 2):
        p1 = 0
        temp = np.random.rand()
```

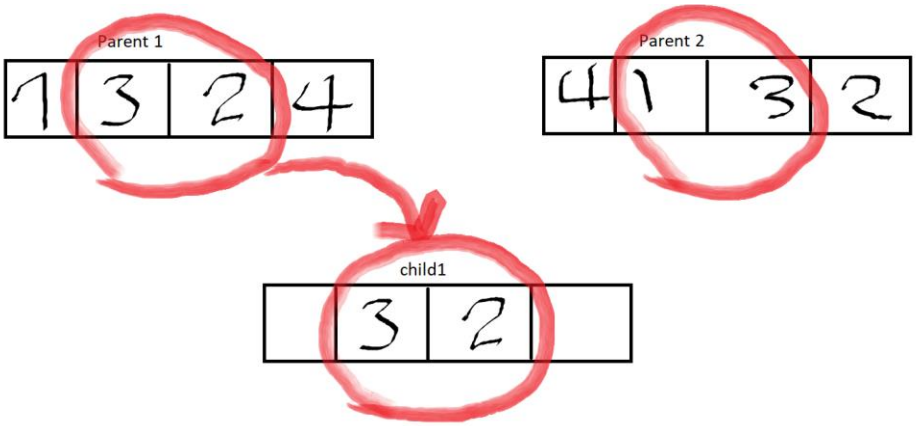
```
        while cumulative_pr[p1] < temp:
            p1 = p1 + 1
        p2 = p1
        while p1 == p2:
            temp = np.random.rand()
            p = 0
            while cumulative_pr[p] < temp:
                p = p + 1
            p2 = p
        parent1 = population[p1]
        parent2 = population[p2]

        children = cross_over(parent1, parent2)

        child1 = children[0]
        child2 = children[1]

        population.append(child1)
        object_values.append(objective(child1))

        population.append(child2)
        object_values.append(objective(child2))
```

- تابع cross_over به عنوان ورودی دو parent میگیرد و به عنوان خروجی دو children میدهد.
- در ابتدا دو عدد تصادفی تولید میکنیم اگر دو عدد تصادفی یکسان بودند دوباره تولید میکنیم و به صورتی که temp1 کوچکتر از temp2 باشد.
- اما روش ترکیب ما به چه صورت است؟
    - فرض کنید n=4 باشد و دو parentبه صورت زیر باشد:

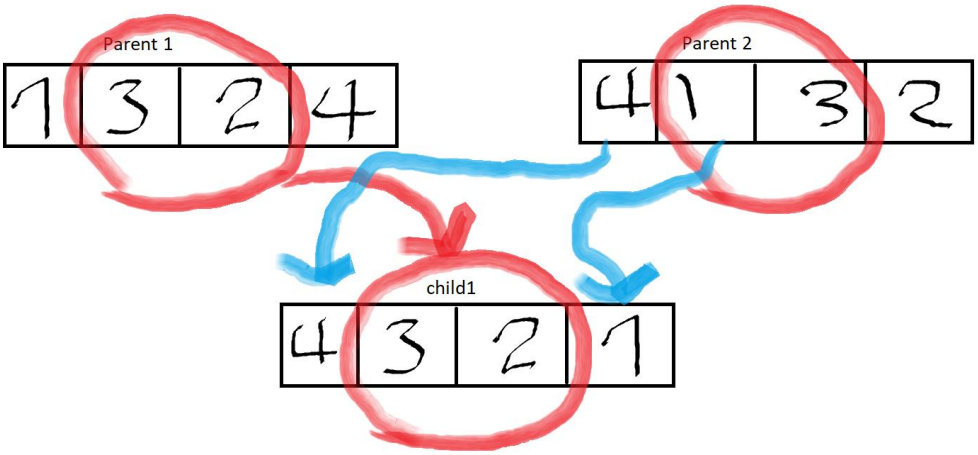
    - حال با کمک تولید تصادفی temp1 = 1 و temp2 = 2 بدست اوردیم.



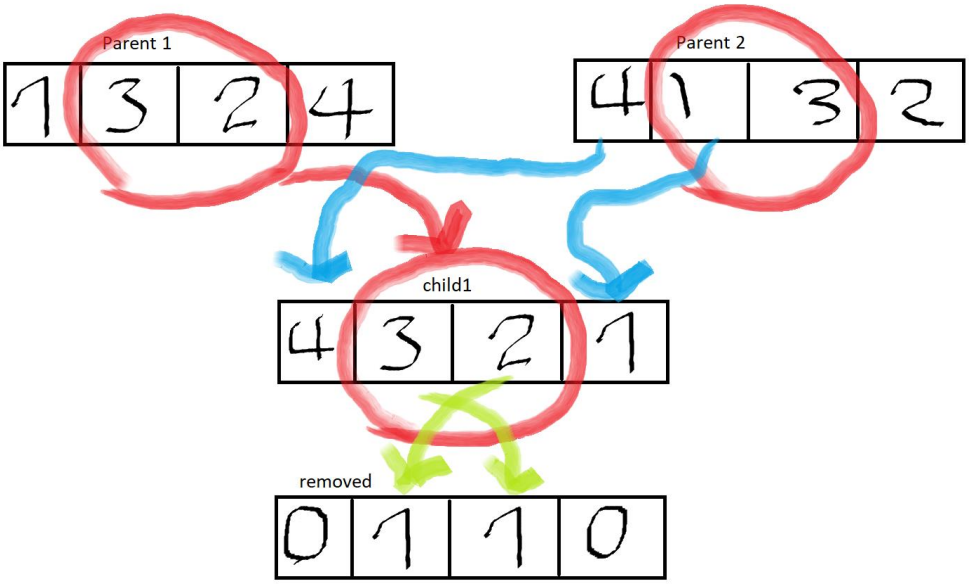
    - حال میخواهیم فرزند اول را بسازیم. از parent1، عنصرهای temp1 تا temp2 را نگه میداریم.



- و از parent2 به ترتیب آنهایی که استفاده نشده اند را قرار میدهیم.



- برای این کار یک آرایه با نام removed استفاده میکنیم و اندیس‌هایی که عددش در child1 استفاده شده است را 1 میگذاریم و باقی 0 میماند.



- به طور مشابه همین کار را برای child2 انجام میدهیم و آن را تولید میکنیم.

```
def cross_over(parent1, parent2):
    temp1 = np.random.randint(n)
    temp2 = np.random.randint(n)
    while (temp1==temp2):
        temp1 = np.random.randint(n)
        temp2 = np.random.randint(n)
    if temp2 < temp1:
        temp1,temp2 = temp2,temp1

    # CREATE CHILD 1
    child1 = np.zeros(n,dtype=int)
    removed = child1.copy()
    for i in range(temp1, temp2+1):
        child1[i] = parent1[i]
        removed[parent1[i]-1] = 1

    i = 0
    j = 0
    while i < n and j < n:
        if i >= temp1 and i <= temp2:
            i = i + 1
            continue
        while j < n and removed[parent2[j]-1]:
            j = j + 1
        if j == n:
            break

        child1[i] = parent2[j]
        i = i + 1
        j = j + 1


    #CREATE CHILD 2
    child2 = np.zeros(n,dtype=int)
    removed = child2.copy()
    for i in range(temp1, temp2 + 1):
        child2[i] = parent2[i]
        removed[parent2[i]-1] = 1

    i = 0
    j = 0
    while i < n and j < n:
        if i >= temp1 and i <= temp2:
            i = i + 1
            continue
```

```
        while j < n and removed[parent1[j]-1]:
            j = j+ 1
        if j == n:
            break

        child2[i] = parent1[j]
        i = i + 1
        j = j + 1

    return (child1, child2)
```

5. الگوریتم Mutation:

- الگوریتم جهش به این صورت است دو عدد تصادفی تولید میکنیم و در عضو برگزیده شده برای جهش جابه‌جایش میکنیم. البته یک شرط گذاشتم که همواره جهش به سمت بهتر شدن نیز برود و اگر عدد تصادفی تولید شده باعث بدتر شدن وضعیت میشد عوضش کند.

```
# MUTATION
    for i in range(num_mutation):
        temp = np.random.randint(num_crossover)
        temp = temp + population_size
        mutated = population[temp]

        # MUTATION -> shuffle 2 places
        temp = np.random.randint(n)
        temp2 = np.random.randint(n)
        while ((graph.isReachable(mutated[temp],mutated[temp2]) and temp<temp2)):
            temp = np.random.randint(n)
            temp2 = np.random.randint(n)

        mutated[temp] , mutated[temp2] = mutated[temp2] , mutated[temp]

        population.append(mutated)
        object_values.append(objective(mutated))
```

بخش چهارم – نمودار همگرایی:

- برای این بخش از کتاب خانه matplotlib استفاده کردم.
- برای اضافه کردن این کتابخانه ابتدا دو پکیج زیر را نصب کنید:

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

- سپس pyplot را اضافه میکنیم.

```
import matplotlib.pyplot as plt
```

- همانطور که در بدنه اصلی حلقه گفتیم، Best_objective_plot برای رسم نمودار استفاده شده است که یک لیست است و مقدار ماکسیموم تابع هدف کل جمعیت در هر iteration را ذخیره می‌کند.

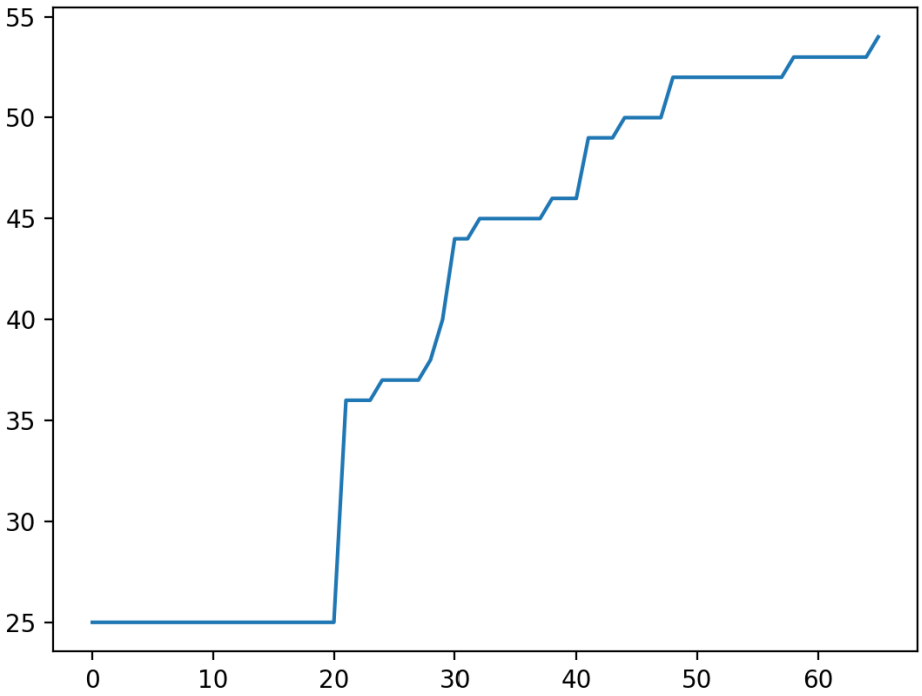
```
plt.plot(best_objective_plot)
plt.show()
```

- در نمودار محور x ها تعداد iteration ها است و محور y ها مقدار تابع هدف ماکسیموم است.

بخش پنجم – چند نمونه خروجی:

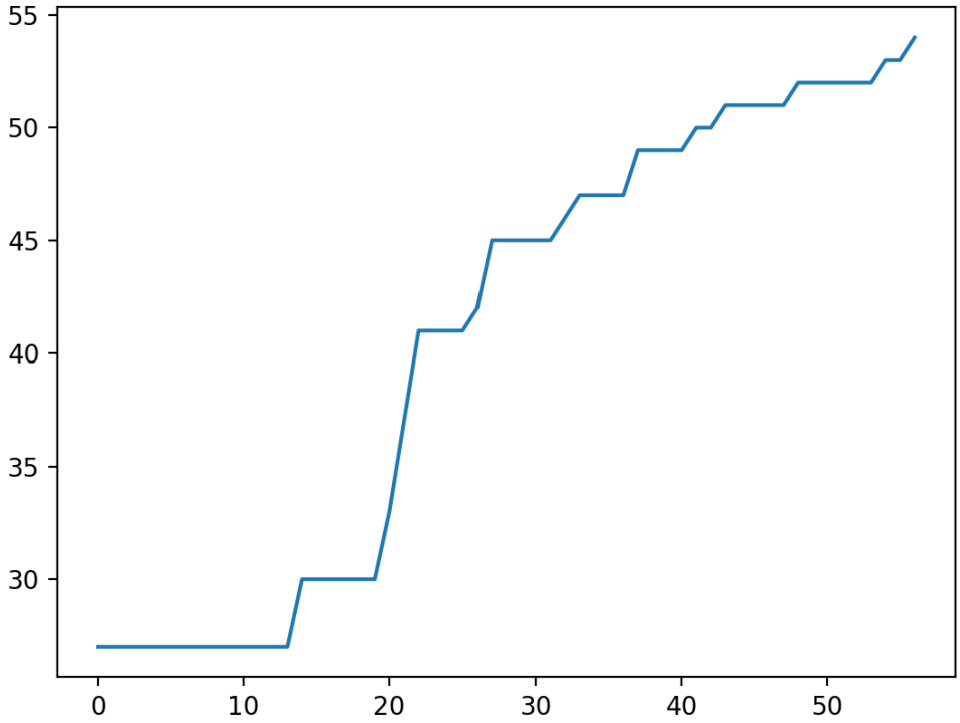
- خروجی 1:

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



- خروجی 2:

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



بخش ششم – مقایسه با الگوریتم SA:

- با توجه به الگوریتمی که من نوشتم الگوریتم SA در این مسئله خاص سریع‌تر است. شاید مقدار iterationهای الگوریتم ژنتیک در جمعیت‌های زیاد کمتر باشد ولی محاسباتش پیچیده‌تر و هزینه‌بر تر است. برای مسئله‌های پیچیده‌تر الگوریتم ژنتیک میتواند به شدت سریع تر باشد و به جواب برسد ولی الگوریتم SA ممکن است مسیر اشتباه بیافتد.
- مورد بعدی که به چشم من خورد خطای کمتر الگوریتم ژنتیک نسبت به SA است. الگوریتم ژنتیک در اجرا های مختلف همواره به جواب درست رسید. اما الگوریتم SA کمتر به جواب درست میرسید. (البته اندازه جمعیت در این مورد تاثیر گذار است).