دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش تمرین عملی اول

مبانی هوش محاسباتی

نگارش

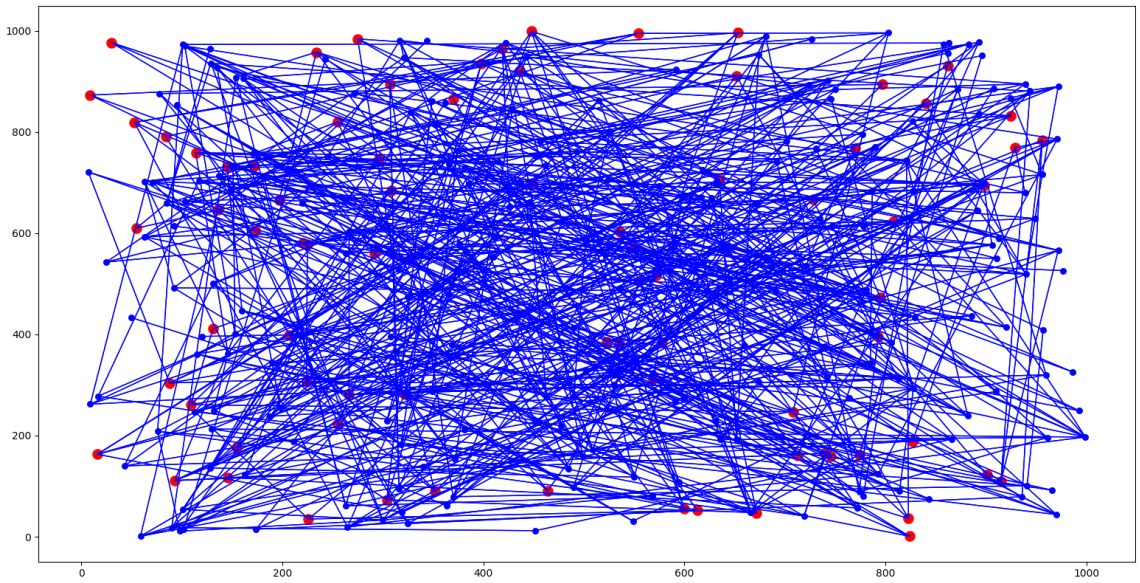
آرش حاجی صفی - ۹۶۳۱۰۱۹

آذر ۱۳۹۹

گزارش سوال اول:

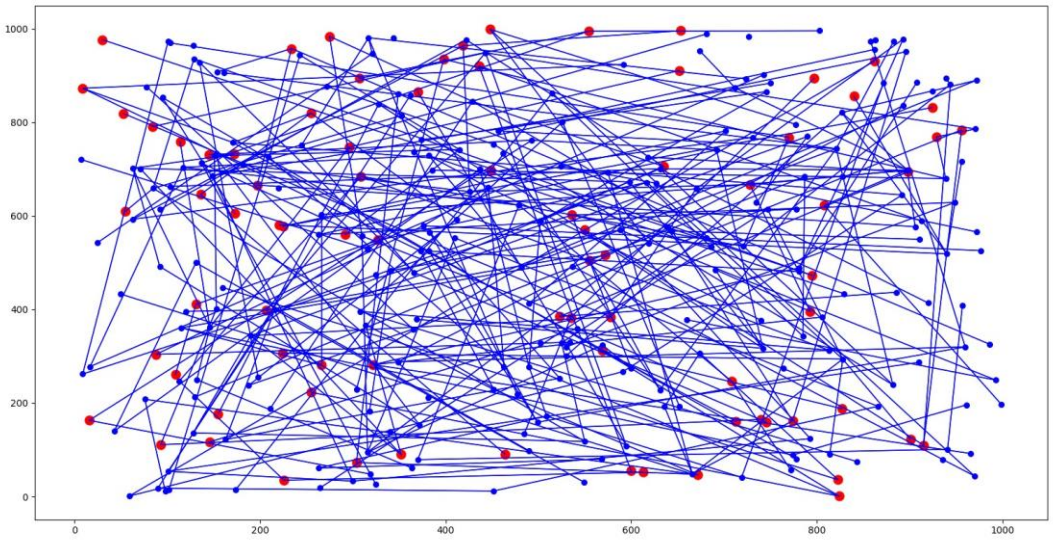
برای اجرا، باید فایل SteinerTree_Problem.py را اجرا نمود. در فایل ga.py توابع مورد نیاز الگوریتم ژنتیک پیاده‌سازی شده و شرایط مسئله (در استراکت problem) و پارامترهای قابل تنظیم (مثل احتمالات جهش و جمعیت والدین و غیره در استراکت params) از SteinerTree_Problem به ga پاس داده می‌شود و با اجرای تابع run(problem, params)، الگوریتم اجرا می‌شود.

قبل از اجرای الگوریتم نتیجه به صورت زیر می‌باشد:



که مجموع هزینه یال‌ها، ۳۴۰۹۵۵ می‌باشد.

با اجرای الگوریتم با تنظیمات فعلی در فایل SteinerTree_Problem، درخت زیر حاصل می‌شود:



که مجموع هزینه یال‌ها در این درخت، ۱۲۹۹۵۴ می‌باشد.

نحوه بازنمایی مسئله:

نحوه بازنمایی مسئله به این صورت می‌باشد که هر کروموزوم، به تعداد یال‌های موجود در مسئله ورودی، ژن خواهد داشت که هر ژن مقدار صفر و یا یک دارد (یک آرایه edges_bit_arr برای هر کروموزوم در نظر گرفته شده که به تعداد یال‌های فایل ورودی دارای المنت است و این مورد با آن پیاده‌سازی شده است)؛ صفر بودن ژن iام به معنی این است که یال iام حذف شده و ۱ بودن آن به معنی این است که یال مورد نظر حذف نشده است.

لازم به ذکر است که برای نگهداری هر کروموزوم از ساختمان داده ypstruct در python استفاده شده که هر کروموزوم دارای فیلدهای cost، edges (که آرایه‌ای از دو راس هر یال و هزینه آن یال است) و edges_bit_arr که قبلتر گفتیم می‌باشد و از edges_bit_arr برای ترکیب و جهش استفاده می‌شود و باقی موارد از روی آن قابل بدست آمدن است.

تعداد جمعیت والدین و تعداد فرزندان و تغییرات آن:

جمعیت والدین به صورت پارامتر npop در فایل SteinerTree_Problem.py مشخص شده و یک پارامتر pc هم در این فایل در نظر گرفته شده که تعداد فرزندان (nc) مساوی با pc برابر npop است ($nc = pc \times npop$).

در حالت فعلی تعداد والدین را برابر 20 و pc را برابر 2 گرفته ام که تعداد فرزندان برابر 40 می‌شود، در این حالت زمان بسیار زیادی برای اجرای الگوریتم مورد نیاز است که نتیجه اجرای آنرا در بالاتر گفتیم. هرچه تعداد والدین و یا تعداد فرزندان بیشتر شود، زمان اجرای الگوریتم بسیار طولانی تر می‌شود اما همگرایی سریعتر اتفاق می‌افتد (در iteration های کمتری به همگرایی می‌رسیم).

علت این است که در این پیاده‌سازی از الگوریتم $\mu + \lambda$ استفاده می‌کنیم و هرچه تعداد والدین یا فرزندان بیشتر باشد، تعداد گزینه بیشتری برای انتخاب داریم و هربار بهترین گزینه‌ها انتخاب می‌شوند و باعث می‌شود جستجوی عمومی صورت بگیرد و همگرایی دیرتر اتفاق بیفتد.

وقتی تعداد جمعیت کمتر می‌شود، همگرایی زودتر اتفاق می‌افتد که علت آن این است که خیلی طول می‌کشد تا جهش یا ترکیب به صورت رخ دهد که یک جستجوی عمومی صورت بگیرد و در اکسترمم محلی طولانی تر باقی می‌مانیم.

نحوه انتخاب والدین:

انتخاب والدین با الگوریتم Roulette Wheel صورت می‌گیرد؛ به این صورت که در ابتدا هزینه (cost) تمامی والدین محاسبه می‌شود که برابر با همان مجموع هزینه یال‌ها می‌باشد (البته در خصوص همبند بودن درخت حاصل هم یک چک صورت می‌گیرد و اگر همبند نبود و یا مثلاً بین همه‌ی ترمینال‌ها مسیر نبود، جریمه اضافه شود). سپس برای هر والد یک احتمال انتخاب که برابر با $\frac{1}{e^{\beta \times cost}}$ در نظر گرفته می‌شود که β یک پارامتری است که ما مقدارش را تعیین می‌کنیم. سپس الگوریتم Roulette Wheel با این احتمالات برای هر والد اجرا می‌شود و ۲ والد انتخاب می‌شوند.

نحوه انتخاب بازماندگان:

به طور خلاصه، μ تا کروموزوم با بیشترین شایستگی در بین فرزندان و والدین انتخاب می‌شوند.

الگوریتم انتخاب را به صورت μ + λ پیاده‌سازی نموده‌ام؛ انتخاب فرزندان به این صورت است که بعد از تولید همه فرزندان، به لیست والدین اضافه می‌شوند، این لیست به صورت صعودی بر حسب cost هر کروموزوم مرتب می‌شود (یعنی کروموزوم با cost کمتر که شایستگی بیشتری دارد، در ابتدای لیست قرار می‌گیرد)، سپس μ تا کروموزوم اول لیست انتخاب می‌شوند:

```
233         # merge, sort, and select
234         pop += popc # Lambda + mu selection
235         pop = sorted(pop, key=lambda p: p.cost)
236         pop = pop[0:npop] # top npop population
237
```

نحوه ترکیب کروموزوم‌ها با همدیگر و مقدار احتمال ترکیب و تاثیر آن در سرعت همگرایی:

ترکیب کروموزوم‌ها با هم به صورت Uniform Crossover پیاده‌سازی شده‌است؛ به این صورت که به ازای هر ژن که یک بیت است، یک عدد تصادفی بین ۰ و ۱ تولید می‌شود، اگر این عدد بزرگتر از احتمال crossover که در حال حاضر 0.5 در نظر گرفته شده‌است بود، ژن مورد نظر دو والد در فرزندها با هم جابجا می‌شود، اما اگر کمتر بود، این ژن والدها در فرزند بدون تغییر باقی می‌ماند.

این احتمال 0.5 هرچه کمتر می‌شود، سرعت همگرایی بالاتر می‌رود، علت این است که تغییرات فرزندان بسیار اندک می‌شود و بدون تغییر خاصی فرزندان به مرحله بعدی منتقل می‌شوند و درهمان اکسترمم محلی باقی می‌مانیم.

به طور معادل، تا یک حدی که این احتمال را زیاد می‌کنیم، سرعت همگرایی کمتر می‌شود، چون فرزندان تغییرات زیادی را خواهند داشت و جستجوی عمومی صورت می‌گیرد.

```
46  def uniform_crossover(A, B, P):
47      for i in range(len(P)):
48          if P[i] < 0.5:
49              temp = A[i]
50              A[i] = B[i]
51              B[i] = temp
52      return A, B
```

نحوه جهش کروموزوم‌ها و مقدار احتمال جهش و تاثیر آن در سرعت همگرایی:

جهش هر کروموزوم به این صورت است که یک احتمال mu در نظر گرفته شده‌است و به ازای هر ژن (که بیت است) برای کروموزوم، یک عدد تصادفی بین صفر و یک تولید می‌شود؛ اگر این عدد کوچکتر از mu بود، بیت مورد نظر را معکوس می‌کنیم (اگر صفر بود آنرا یک می‌کنیم و اگر یک بود آنرا صفر می‌کنیم)

```
54  # mu is the probability of mutation for each bit
55  def mutation(A, mu):
56      res = [i for i in A]
57      flag = np.random.rand(len(A)) <= mu
58      for i in range(len(flag)):
59          if flag[i]:
60              res[i] = 1 if A[i] == 0 else 0
61      return res
```

در حال حاضر این مقدار احتمال جهش یا همان mu، برابر با 0.2 در نظر گرفته شده‌است.

با افزایش mu سرعت همگرایی کاهش می‌یابد (چون فرزندان متفاوتی تولید می‌شوند و جستجوی عمومی صورت می‌گیرد) و با کاهش آن، سرعت همگرایی افزایش می‌یابد چون فرزند خیلی متفاوتی تولید نمی‌شود.

شرط خاتمه

شرط خاتمه به این صورت است که یک پارامتر maxit در نظر گرفته‌ام که حداکثر تعداد iteration (انتخاب والدین، بازترکیبی و تولید فرزندان، جهش، و انتخاب بازماندگان) را مشخص می‌کند. هر موقع به تعداد maxit این چرخه تکرار شود، از حلقه خارج می‌شویم و الگوریتم پایان می‌پذیرد.

در حال حاضر چون با پارامترهای فعلی اجرای الگوریتم بسیار زمانبر است، maxit را برابر با ۵ گرفته ام تا الگوریتم سریعتر پایان پذیرد اما مقدار فعلی خیلی کم است؛ هرچه آنرا بالاتر ببریم جواب بهتری بدست می‌آید.

فایل steiner_out.txt که خروجی گفته شده در صورت سوال است، در فولدر Q1 موجود می‌باشد.

درخت رسم شده هم در فولدر Q1/Results وجود دارد و هم در ابتدای گزارش این سوال نمایش داده شد.

گزارش سوال دوم:

برای اجرا، باید فایل EggHolder_FindMin_Problem.py را اجرا نمود. در فایل ES.py توابع مورد نیاز الگوریتم ES پیاده‌سازی شده و شرایط مسئله (در استراکت problem) و پارامترهای قابل تنظیم (مثل احتمالات جهش و جمعیت والدین و غیره در استراکت params) از EggHolder_FindMin_Problem.py به ES پاس داده می‌شود و با اجرای تابع run(problem, params)، الگوریتم اجرا می‌شود.

نتیجه نهایی اجرای الگوریتم با پارامترهای فعلی به صورت زیر است:

```
Iteration: 29
Best    Chromosome: fitness = inf, f = -959.640700
Worst   Chromosome: fitness = 0.00050674, f = 1013.767162
Average Chromosome: fitness = inf, f = -959.640700


Final Result:
Resulted Global Minimum = -959.6407, x1 = 512.0110273866192, x2 = 404.4226528694866
```

که می‌بینیم مقدار minimum را 959.6407- بدست آورده که به ازای مقادیر x1 = 512.01102 و x2 = 404.42265 می‌باشد.

نحوه بازنمایی مسئله:

بازنمایی به صورت اعداد حقیقی صورت گرفته است؛ برای هر کروموزوم یک ژن به صورت بردار ۲تایی [x1, x2] و یک ژن برای نگهداری σ (نرخ یادگیری) به صورت sigma نگه می‌داریم؛ در اصل در محاسبات ۳ ژن به این صورت داریم: [x1, x2. Sigma]

```
# empty chromosome
empty_chromosome = structure()
empty_chromosome.f = None # f = EggHolder(x)
empty_chromosome.fitness = None
empty_chromosome.x = None # x = [x1, x2]
empty_chromosome.sigma = sigma # learning rate
```

تعداد جمعیت والدین و تعداد فرزندان و تغییرات آن:

جمعیت والدین به صورت پارامتر npop در فایل EggHolder_FindMin_Problem.py مشخص شده و یک پارامتر pc هم در این فایل در نظر گرفته شده که تعداد فرزندان (nc) مساوی با pc برابر npop است (nc = pc × npop).

در حالت فعلی تعداد والدین را برابر ۴۰ و pc را برابر ۵ گرفته ام که تعداد فرزندان برابر ۲۰۰ می‌شود، در این حالت به طور متوسط در iteration ۱۴ ام به جواب بهینه می‌رسیم. هرچه تعداد فرزندان را بیشتر و یا کمتر می‌کنیم، سرعت همگرایی بالاتر می‌رود و در اکتسرمم محلی گرفتار می‌شویم، به نظر می‌آید نسبت ۵ برابر بودن تعداد فرزندان بهترین حالت برای فرار از همگرایی زودرس

است. علت این است که اگر تعداد فرزندان کم باشد که در اکسترمم محلی باقی می‌مانیم، اگر هم زیاد باشد، به علت برداشتن گام‌های خیلی بد شایستگی ها زیاد خوب نیست و همان جمعیت‌های قبلی شایستگی بهتری را داشته اند.

با افزایش تعداد والدین، با حفظ نسبت ۵ برابر بودن فرزندان، همگرایی دیرتر اتفاق می‌افتد و با کاهش تعداد والدین به سرعت به همگرایی می‌رسیم.

نحوه انتخاب والدین:

چون الگوریتم ما ES است، انتخاب والدین به صورت کاملاً تصادفی صورت می‌گیرد؛ یعنی هربار یک جایگشت کاملاً تصادفی از جمعیت فعلی ایجاد می‌کنیم و ۲ کروموزوم اول را جایگشت را به عنوان والدین انتخاب می‌کنیم. به عبارت دیگر، هربار ۲ والد کاملاً تصادفی انتخاب می‌شوند:

```
# select parents
q = np.random.permutation(npop) # a random permutation from integers 0 to npop - 1
p1 = pop[q[0]] # random first parent
p2 = pop[q[1]] # random second parent
```

نحوه انتخاب بازماندگان:

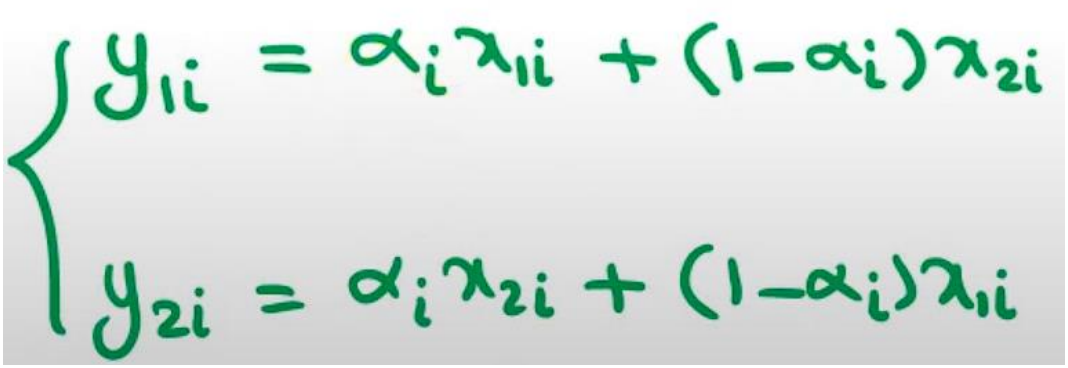
به طور خلاصه، μ تا کروموزوم با بیشترین شایستگی در بین فرزندان انتخاب می‌شوند.

الگوریتم انتخاب را به صورت μ , λ پیاده‌سازی نموده‌ام؛ انتخاب فرزندان به این صورت است که بعد از تولید همه فرزندان، لیست آنها به صورت نزولی بر حسب fitness هر کروموزوم مرتب می‌شود (یعنی کروموزوم با شایستگی بیشتر، در ابتدای لیست قرار می‌گیرد)، سپس μ تا کروموزوم اول لیست انتخاب می‌شوند:

```
119         # merge, sort, and select
120         pop = popc # Lambda, mu selection
121         pop = sorted(pop, key=lambda p: p.fitness, reverse=True)
122         pop = pop[0:npop] # top npop population
123
```

نحوه ترکیب کروموزوم‌ها با همدیگر و مقدار احتمال ترکیب و تاثیر آن در سرعت همگرایی:

چون بازنمایی به صورت عدد حقیقی است، ترکیب کروموزوم‌ها با هم با استفاده از رابطه زیر انجام می‌شود:



یک مقدار تصادفی α به صورت عددی تصادفی بین $\gamma, \gamma-$ انتخاب می‌کنیم (گاما یک پارامتر است که خودمان مشخص می‌کنیم و در حال حاضر مقدار آنرا 0.01 گرفته ام) و با این فرمول فرزندان را محاسبه می‌کنیم.

```
5  def crossover(p1, p2, gamma=0.1):
6      c1 = p1.deepcopy()
7      c2 = p2.deepcopy()
8      alpha = np.random.uniform(-gamma, 1+gamma, *c1.x.shape)
9      c1.x = alpha*p1.x + (1-alpha)*p2.x
10     c2.x = alpha*p2.x + (1-alpha)*p1.x
11     return c1, c2
12
```

با تست‌هایی که صورت گرفت مشخص شد که هرچه این مقدار گاما را کمتر یا بیشتر کنیم، هیچ تاثیر قابل توجهی روی سرعت همگرایی نمی‌گذارد و البته وقتی کمتر باشد سرعت همگرایی اندکی بیشتر است؛ علت این است که گام‌های تصادفی بد کمتر می‌شود و سریعتر به می‌نیمم محلی می‌رسیم.

نحوه جهش کروموزوم‌ها و مقدار احتمال جهش و تاثیر آن در سرعت همگرایی:

جهش هر کروموزوم با رابطه زیر پیاده‌سازی شده است:

$$\sigma' = \sigma \cdot \exp(\tau \cdot N(0,1))$$
$$x'_i = x_i + \sigma' \cdot N(0,1)$$

ابتدا نرخ یادگیری (sigma) را با رابطه فوق جهش داده‌ام برای هر عضو جمعیت، سپس مقدار بردار x آنهارا (یعنی هم مقدار x1 و هم x2) را به این صورت با جمع خودش با ضرب یک عدد تصادفی از توزیع نرمال در نرخ یادگیری جدید، جهش داده ام:

```
# mutation
for j in range(len(popc)):
    popc[j].sigma = popc[j].sigma * math.exp(popc[j].sigma * np.random.randn())
    popc[j].x = popc[j].x + popc[j].sigma * np.random.randn(*popc[j].x.shape)
```

شرط خاتمه

شرط خاتمه به این صورت است که یک پارامتر maxit در نظر گرفته‌ام که حداکثر تعداد iteration (انتخاب والدین، بازترکیبی و تولید فرزندان، جهش، و انتخاب بازماندگان) را مشخص می‌کند. هر موقع به تعداد maxit این چرخه تکرار شود، از حلقه خارج می‌شویم و الگوریتم پایان می‌پذیرد.

در حال حاضر چون با پارامترهای فعلی اجرای الگوریتم بسیار زمانبر است، maxit را برابر با ۳۰ گرفته ام که با پارامترهای فعلی به طور میانگین در گام ۱۴ می‌نیمم را بدست می‌آوریم. البته می‌شد یک شرط هم چک کرد که در حلقه هر موقع بهترین شایستگی به بی‌نهایت میل می‌کرد، حلقه را بشکنیم و دیگر ادامه ندهیم تا زمان ذخیره شود.

نتیجه نهایی در ابتدای گزارش این سوال نمایش داده شد که به صورت زیر بود:

```
Iteration: 29
Best    Chromosome: fitness = inf, f = -959.640700
Worst   Chromosome: fitness = 0.00050674, f = 1013.767162
Average Chromosome: fitness = inf, f = -959.640700


Final Result:
Resulted Global Minimum = -959.6407, x1 = 512.0110273866192, x2 = 404.4226528694866
```

گزارش سوال سوم:

برای اجرا، باید فایل Nurses_Schedule_Problem.py را اجرا نمود. در فایل GA.py توابع مورد نیاز الگوریتم ژنتیک پیاده‌سازی شده و شرایط مسئله (در استراکت problem) و پارامترهای قابل تنظیم (مثل احتمالات جهش و جمعیت والدین و غیره در استراکت params) از Nurses_Schedule_Problem به GA پاس داده می‌شود و با اجرای تابع run(problem, params)، الگوریتم اجرا می‌شود.

نتیجه حاصل از یکبار اجرای الگوریتم به صورت زیر است که تمام محدودیت‌ها را برآورده کرده‌است:

```
Iteration: 7
Best    Chromosome: fitness = 350.00000000
Worst   Chromosome: fitness = 65.00000000
Average Chromosome: fitness = 345.05000000



Output:
6,1 2,4 8
5,8 3,4 6
6,3,5 4,2,1,8 6
2,5 7,1 3
2,8 7,5 2
8,4 7,3 1
6,4 1,3 5
```

نحوه بازنمایی مسئله:

نحوه بازنمایی مسئله به این صورت می‌باشد که برای هر کروموزوم، یک لیست دو بعدی داریم که دارای ۷ خانه‌ی ۹ المنتی است. هر یک از این خانه ها زمان‌بندی یک روز را نشان می‌دهد که در لیست مربوط به هر روز، ۳ المنت اول مربوط به شیفت صبح، ۴ المنت بعدی مربوط به شیفت بعد از ظهر، و ۲ المنت آخر مربوط به شیفت شب هستند. یک نمونه کروموزوم در شکل زیر نشان داده شده است:

```
> 0: [3, 2, 0, 8, 2, 6, 3, 4, 0]
> 1: [2, 4, 1, 8, 1, 0, 0, 4, 0]
> 2: [1, 7, 0, 7, 5, 8, 0, 7, 0]
> 3: [7, 4, 0, 2, 8, 3, 4, 7, 8]
> 4: [6, 7, 1, 2, 8, 0, 0, 2, 0]
> 5: [5, 1, 0, 5, 3, 0, 0, 8, 7]
> 6: [3, 7, 4, 5, 2, 3, 0, 3, 6]
  len(): 7
```

تعداد جمعیت والدین و تعداد فرزندان و تغییرات آن:

جمعیت والدین به صورت پارامتر npop در فایل Nurses_Schedule_Problem.py مشخص شده و یک پارامتر pc هم در این فایل در نظر گرفته شده که تعداد فرزندان (nc) مساوی با pc برابر npop است (nc = pc × npop).

در حالت فعلی تعداد والدین را برابر ۱۰۰ و pc را برابر ۱ گرفته ام که تعداد فرزندان هم برابر ۱۰۰ می‌شود، در این حالت به طور متوسط در iterate ۶ام به نتیجه می‌رسیم. هرچه تعداد جمعیت والدین را بیشتر می‌کنیم، سرعت همگرایی دیرتر رخ میدهد اما به ماکزیمم global می‌رسیم.

علت این است که در این پیاده‌سازی از الگوریتم μ + λ استفاده می‌کنیم و هرچه تعداد والدین یا فرزندان بیشتر باشد، تعداد گزینه بیشتری برای انتخاب داریم و هربار بهترین گزینه‌ها انتخاب می‌شوند و باعث می‌شود جستجوی عمومی صورت بگیرد و همگرایی دیرتر اتفاق بیفتد.

وقتی تعداد جمعیت کمتر می‌شود، همگرایی زودتر اتفاق می‌افتد که علت آن این است که خیلی طول می‌کشد تا جهش یا ترکیب به صورت رخ دهد که یک جستجوی عمومی صورت بگیرد و در اکسترمم محلی طولانی تر باقی می‌مانیم.

نحوه انتخاب والدین:

انتخاب والدین به صورت کاملاً تصادفی صورت می‌گیرد؛ یعنی هربار یک جایگشت کاملاً تصادفی از جمعیت فعلی ایجاد می‌کنیم و ۲ کروموزوم اول را جایگشت را به عنوان والدین انتخاب می‌کنیم. به عبارت دیگر، هربار ۲ والد کاملاً تصادفی انتخاب می‌شوند:

```
# select parents
q = np.random.permutation(npop) # a random permutation from integers 0 to npop - 1
p1 = pop[q[0]] # random first parent
p2 = pop[q[1]] # random second parent
```

نحوه انتخاب بازماندگان:

به طور خلاصه، μ تا کروموزوم با بیشترین شایستگی در بین فرزندان و والدین انتخاب می‌شوند.

الگوریتم انتخاب را به صورت μ + λ پیاده‌سازی نموده‌ام؛ انتخاب فرزندان به این صورت است که بعد از تولید همه فرزندان، به لیست والدین اضافه می‌شوند، این لیست به صورت نزولی بر حسب fitness هر کروموزوم مرتب می‌شود (یعنی کروموزوم با شایستگی بیشتر، در ابتدای لیست قرار می‌گیرد)، سپس μ تا کروموزوم اول لیست انتخاب می‌شوند:

```
# merge, sort, and select
pop += popc # Lambda, mu selection
pop = sorted(pop, key=lambda p: p.fitness, reverse=True)
pop = pop[0:npop] # top npop population
```

نحوه ترکیب کروموزوم‌ها با همدیگر و مقدار احتمال ترکیب و تاثیر آن در سرعت همگرایی:

ترکیب کروموزوم‌ها با هم به صورت Uniform Crossover پیاده‌سازی شده‌است؛ به این صورت که به ازای هر شیفت (و نه ژن!) که یک لیست است، یک عدد تصادفی بین ۰ و ۱ تولید می‌شود، اگر این عدد بزرگتر از احتمال crossover که در حال حاضر 0.5 در نظر گرفته شده‌است بود، شیفت مورد نظر دو والد در فرزندها با هم جابجا می‌شود، اما اگر کمتر بود، این ژن والدها در فرزند بدون تغییر باقی می‌ماند. یعنی مثلاً شیفت اول فرزند اول، یا شیفت اول والد اول خواهد بود و یا شیفت اول والد دوم، همینطور تا شیفت سوم.

این احتمال 0.5 هرچه کمتر می‌شود، سرعت همگرایی بالاتر می‌رود، علت این است که تغییرات فرزندان بسیار اندک می‌شود و بدون تغییر خاصی فرزندان به مرحله بعدی منتقل می‌شوند و درهمان اکسترمم محلی باقی می‌مانیم.

به طور معادل، تا یک حدی که این احتمال را زیاد می‌کنیم، سرعت همگرایی کمتر می‌شود، چون فرزندان تغییرات زیادی را خواهند داشت و جستجوی عمومی صورت می‌گیرد.

```
34  def uniform_crossover(A, B, P):
35      c1 = A.deepcopy()
36      c2 = B.deepcopy()
37      c1week = c1.schedule
38      c2week = c2.schedule
39      for i in range(len(c1week)):
40          temp = [0] * 9
41          if P[0] < 0.5:
42              temp[0:3] = c1week[i][0:3]
43              c1week[i][0:3] = c2week[i][0:3]
44              c2week[i][0:3] = temp[0:3]
45          if P[1] < 0.5:
46              temp[3:7] = c1week[i][3:7]
47              c1week[i][3:7] = c2week[i][3:7]
48              c2week[i][3:7] = temp[3:7]
49          if P[2] < 0.5:
50              temp[7:9] = c1week[i][7:9]
51              c1week[i][7:9] = c2week[i][7:9]
52              c2week[i][7:9] = temp[7:9]
53
54      c1.schedule = c1week
55      c2.schedule = c2week
56      return c1, c2
```

نحوه جهش کروموزوم‌ها و مقدار احتمال جهش و تاثیر آن در سرعت همگرایی:

جهش هر کروموزوم به این صورت است که یک احتمال mu در نظر گرفته شده‌است و به ازای هر روز هفته یک مقدار تصادفی ایجاد می‌شود (یعنی ۷ مقدار تصادفی)؛ اگر این مقدار تصادفی از mu کمتر باشد، در لیست زمانبندی آن روز جهش رخ می‌دهد. جهش در لیست زمان‌بندی آن روز به این صورت است که با توجه به اینکه گفته بودیم هر روز دارای ۹ تا مقدار است که نشان می‌دهد کدام پرستار در شیفت متناظر آن جایگاه مشغول به کار است (پرستار ۱ تا ۸) یا هیچکس مشغول به کار نیست (عدد ۰)، ۲تا مقدار از این ۹ مقدار با هم جابجا می‌شوند.

```
def mutate(p, mu):
    res = p.deepcopy()
    flag = np.random.rand(7) <= mu
    for i in range(len(flag)):
        if flag[i]:
            n1 = np.random.randint(0, 9)
            n2 = np.random.randint(0, 9)
            tmp = res.schedule[i][n1]
            res.schedule[i][n1] = res.schedule[i][n2]
            res.schedule[i][n2] = tmp
    return res
```

در حال حاضر این مقدار احتمال جهش یا همان mu، برابر با 0.4 در نظر گرفته شده‌است.

با افزایش mu سرعت همگرایی کاهش می‌یابد (چون فرزندان متفاوتی تولید می‌شوند و جستجوی عمومی صورت می‌گیرد) و با کاهش آن، سرعت همگرایی افزایش می‌یابد چون فرزند خیلی متفاوتی تولید نمی‌شود و در ماکزیمم محلی گرفتار می‌شویم.

شرط خاتمه

ابتدا لازم به ذکر است که تابع شایستگی را برای محاسبه شایستگی کروموزوم‌ها به این صورت در نظر گرفته ام که شامل 4 فاز می‌شود و هر فاز دارای یک امتیاز است. این تابع در هر فاز، یک نوع محدودیت گفته‌شده را در زمان‌بندی مربوط به آن کروموزوم بررسی می‌کند و به تناسب اینکه چقدر این زمان‌بندی این محدودیت را نقض می‌کند، امتیاز کسر می‌کند؛ هرچه بیشتر محدودیت مذکور نقض شده باشد امتیاز بیشتری کسر می‌شود. نهایتاً مجموع امتیازات باقی‌مانده برای هر کروموزوم به عنوان شایستگی آن کروموزوم در نظر گرفته می‌شود.

شرط خاتمه را به این صورت گرفته ام که هر وقت بهترین کروموزوم موجود در جمعیت حاصل، دارای حداکثر شایستگی ممکن که عدد ۳۵۰ است شد (یعنی هیچ محدودیتی را نقض نکرده که این امتیاز را گرفته)، الگوریتم پایان می‌یابد و این کروموزوم برگردانده

می‌شود. البته تعداد iteration هم از maxit اگر بیشتر شود (که در الگوریتمی که نوشته ام این مشکل هیچوقت پیش نمی‌آید و به طور متوسط در iteration ۶ام به جواب می‌رسیم)، حلقه پایان می‌یابد و بهترین کروموزوم برگردانده می‌شود.

فایل nurse_out.txt که خروجی گفته شده در صورت سوال است، در فولدر Q3 موجود می‌باشد.