تمرین دوم درس مبانی هوش محاسباتی

مبانی الگوریتم های تکاملی

آرش حریرپوش
9731505

آرش حریرپوش
9731505

❖ بخش اول – مباحث تئوری و مسائل تشریحی

1. دو اصل مهم تکامل را توضیح دهید و چرا اگر هر یک از آن ها نباشند، تکامل صورت نمی گیرد.

بر مبنای نظریه تکامل چارلز داروین که تکامل مبتنی بر دو اصل تغییرات (modification) در نژاد فعلی و انتخاب طبیعی (natural selection) می باشد.
تغییرات در هر نژاد تغییرات در گونه های رایج مشترک را که به مرور زمان منجر به توسعه گونه های جدید شده است توضیح می دهد. نیروی محرکه این نظریه همان انتخاب طبیعی می باشد. ارگانیسم هایی که از نظر ژنتیکی مناسب تر از بقیه هستند با فرکانس بیشتری به نسل های بعدی منتقل می شوند. این اتفاق باعث ایجاد تغییر در گونه ها در طول زمان می شود.
به دلیل اینکه منابع در طبیعت محدود هستند، ارگانیسم هایی که با صفات وراثتی که به نفع بقا و تولید مجدد هستند، تمایل به قرزندان بیشتری نسبت به همتایان خود خواهند داشت، و باعث می شوند صفات آن ها در طی نسل ها افزایش یابد.
انتخاب طبیعی باعث می شود که جامعه با گذر زمان با محیط خود سازگار یا مناسب شوند. انتخاب طبیعی به محیط و یک وراثت موجود در یک گروه نیاز دارد.
و به این دلیل برای تکامل به این دو اصل نیاز است که از طریق تغییر در ژن های موجود ژن های جدید ایجاد می شود و از طریق انتخاب نیز ژن هایی که با محیط تطابق بیشتری دارند باقی می مانند.
(منابع: https://wps.prenhall.com/wps/media/objects/2688/2752944/PDF/16_A01.pdf و https://www.khanacademy.org/science/biology/her/evolution-and-natural-selection/a/darwin-evolution-natural-selection)

2. تفاوت نسلی (نسل به نسل) و حالت پایدار در چیست.

حالت پایدار یک نسخه ساده تر از نسلی است و در آن دو تا از والدین (پدر و مادر) انتخاب می شوند و با پیوند آن ها دو فرزند جهش یافته بدست می آید و به جامعه اضافه می شود، در حالی که در نسلی بخش بزرگی از جامعه انتخاب و پیوند داده می شود (معمولا نیمی از افراد)، بچه های حاصل شده جهش یافته اند و به جامعه اضافه می شوند، بنابراین جایگزین افراد قبلی می شوند.
به همین دلیل، حالت پایدار هنگام استفاده در ، برای مثال multiobjective GA یا Michigan-style LCS، بسیار ساده تر است و انتخاب ارجح می باشد. و نسلی در Pitts-style و IRL LCSs استفاده می شود. به هر حال، از نظر محاسبات، هر دو نسخه کم و بیش معادل یک دیگر می باشند، اما حالت پایدار برای همگرایی به نسل بیشتری نیاز دارد، اما هزینه محاسبات آن (برای مثال هزینه هر دور iteration) بسیار پایین تر از نسلی می باشد.
تفاوت اصلی این دو در جایی است که فرزند اضافه می شود؛ در حالت پایدار تنها یک جامعه وجود دارد که فرزند در آن اضافه شود، بنابراین باید از الگوریتم جایگزینی قبل از

آرش حریرپوش
9731505

اینکه فرزندان را اضافه کرد استفاده کرد.
در نسلی از جامعه های میانی یا موقتی استفاده می شود. در دوره نسل t، با یک مقدار ثابت از افراد جامعه قبلی (نخبه گرایی (elitism)) مقدار دهی اولیه می شود و فرزندان در آنجا تا زمانی که اندازه جامعه به مقدار ماکزیمم برسد اضافه می شوند. در آن زمان این جامعه موقت برابر جامعه جدید (نسل t+ 1) می باشد، یک جامعه موقت جدید ساخته می شود و فرآیند در آن دوباره شروع می شود.
(منبع:
https://www.researchgate.net/post/Whats_the_difference_between_the_stea
dy_state_genetic_algorithm_and_the_generational_genetic_algorithm)

3. مزایا و معایب در روش $(\mu + \lambda)$ و (μ, λ) را نسبت به یک دیگر توضیح دهید.

انتخاب $\mu + \lambda$ یک استراتژی انتخاب (حفظ) نخبه است و در این روش تنوع کاهش می یابد به همین دلیل امکان ماندن در بهینه محلی وجود دارد و همچنین این روش باعث افزایش شایستگی و سرعت همگرایی می شود، و انتخابμ, λ قابلیت فراموشی دارد، و باعث افزایش تنوع می شود و همین فراموشی باعث می شود تا بتواند از نقاط بهینه محلی فرار کند.
روش μ, λ تنها بر اساس مجموعه بچه ها بازماندگان را انتخاب می کند، و بهترین فرزند μ را برای نسل بعد انتخاب می کند.
روش $\mu + \lambda$ بر اساس مجموعه والدین و بچه ها بازماندگان را انتخاب می کند، و بهترین فرزند μ را برای نسل بعد انتخاب می کند.
انتخاب μ, λ، در ترک بهینه محلی بهتر عمل می کند، همچنین در دنبال کردن بهینه متحرک موفق تر است و در هنگام استفاده از استراتژی + یک σ نامناسب در یک فرد (x, σ) احتمال دارد به علت شایستگي خوب x آن مدت زمان زیادی در جامعه باقي بماند.

4. مقدار پارامتر های a و b را طوري تعیین کنید که فشار انتخاب برابر یک شود. (در این رابطه موجودات براساس شایستگی مرتب شده اند (به صورت صعودی) و i برابر رتبه هر موجود می باشد. بدین معنا که موجود با رتبه 1 کمترین شایستگی و موجود با رتبه n (اندازه جمعیت) بیشترین شایستگی را دارد.)

$$P_i = a + b \times \exp(i)$$

در روابط زیر N_1 (برابر n است) برابر تعداد افرادی است که می خواهیم از بین آن ها انتخاب کنیم، و N_2 برابر تعداد موجوداتی است که انتخاب کرده ایم.
رابطه اول برابر است با:

آرش حریرپوش
9731505

$$\sum_{i=1}^{N_1} \mathrm{P_i} = 1 \Rightarrow \sum_{i=1}^{N_1} \mathrm{a} + \mathrm{b} \times \exp(\mathrm{i}) = 1$$

$$\Rightarrow a \times \sum_{i=1}^{N_1} 1 + b \times \sum_{i=1}^{N_1} \exp(i) = 1$$

$$\Rightarrow a \times N_1 + b \times \frac{1 - e^{N_1}}{1 - e} = 1 \Rightarrow a = \frac{1 - b \times \frac{1 - e^{N_1}}{1 - e}}{N_1}$$

و رابطه دوم برابر است با:

$$SP = N_2 \times P_b = N_s \times (a + b \times \exp(N_1)) \Rightarrow a + b \times \exp(N_1) = \frac{SP}{N_2}$$

همچنین با توجه به صورت مساله مقدار فشار انتخاب (SP) را در فرمول بالا برابر یک قرار می دهیم.

با جایگذاری مقدار a بدست آمده در رابطه اول در رابطه دوم داریم:

$$a + b \times \exp(N_1) = \frac{SP}{N_2} \Rightarrow \frac{1 - b \times \frac{1 - e^{N_1}}{1 - e}}{N_1} + b \times e^{N_1} = \frac{1}{N_2}$$

$$\Rightarrow 1 - b \times \frac{1 - e^{N_1}}{1 - e} + N_1 \times b \times e^{N_1} = \frac{1}{N_2} \Rightarrow 1 - \frac{1}{N_2}$$

$$= b \times \left(\frac{1 - e^{N_1}}{1 - e} - N_1 \times e^{N_1} \right)$$

$$\Rightarrow b = \frac{1 - \frac{1}{N_2}}{\left(\frac{1 - e^{N_1}}{1 - e} - N_1 \times e^{N_1} \right)}$$

سپس با جایگذاری مقدار b در رابطه 1 داریم:

$$a = \frac{1 - b \times \frac{1 - e^{N_1}}{1 - e}}{N_1}$$

$$\Rightarrow a = \frac{1 - \left(\frac{1 - \frac{1}{N_2}}{\left(\frac{1 - e^{N_1}}{1 - e} - N_1 \times e^{N_1} \right)} \right) \times \frac{1 - e^{N_1}}{1 - e}}{N_1}$$

بنابراین مقادیر a و b برابر است با:

آرش حریرپوش
9731505

$$a = \frac{1 - \left(\frac{1 - \frac{1}{N_2}}{\left(\frac{1 - e^{N_1}}{1 - e} - N_1 \times e^{N_1} \right)} \right) \times \frac{1 - e^{N_1}}{1 - e}}{N_1}$$

$$b = \frac{1 - \frac{1}{N_2}}{\left(\frac{1 - e^{N_1}}{1 - e} - N_1 \times e^{N_1} \right)}$$

5. چرا روش $(\mu + \lambda)$ وقتی از روش خود تطبیقی استفاده می شود مناسب تر است.

به دلیل اینکه، این روش می تواند مانع خود تطبیقی نرخ جهش موثر در الگوریتم های تکاملی شود، چرا که افرادی که تا کنون شایستگی خوبی دارند اما پارامتر های استراتژی (مثل نرخ های جهش) ضعیفی دارند می توانند برای مدت زمان طولانی در جامعه باقی بمانند، و نرخ بهبود شایستگی جامعه را کاهش دهند.
همچنین در روش (μ, λ) ممکن است به دلیل اینکه نویز گوسی باعث بهتر شدن x شده باشد و در نتیجه ژن های قبلی را که خوب بوده اند فراموش می کنیم.

6. در روش استراتژی تکامل چرا ابتدا سیگما (σ) و سپس x را جهش می دهیم.

به دلیل اینکه برای موثر بودن جهش دو ارزیابی زیر را انجام می دهیم:
ارزیابی اولیه: در صورتی که $f(x')$ خوب باشد، بنابراین x' خوب است.
ارزیابی ثانویه: در صورتی که x' خوب باشد، بنابراین σ' خوب است.
به همین دلیل اگر ابتدا x را جهش دهیم، استدلال بالا برقرار نخواهد شد.
در اصل برای این که مشاهده کنیم σ مناسب است یا خیر باید پس از برداشتن گام بررسی کنیم که آیا تغییرات x خوب بوده یا خیر، بنابراین باید اول σ را جهش داد چرا که خوب و بد بودن x و σ هر دو از روی x مشخص می شود؛ حال اگر اول x را جهش بدهیم نسلی که داریم ارزیابی می کنیم با σ ای است که مقدار آن را قبلا تغییر داده ایم.

7. اصلی ترین تفاوت نظریه لامارک و داروین در چه مورد است؟ به نظر شما کدام یک از این دو نظریه در این مور د صحیح می باشد (با ذکر مثالی نظر خود ر ا اثبات کنید).

نظریه لامارک عنوان می کرد که افراد موجود در جامعه دارای مشخصات یکسانی هستند. و هر فرد می تواند در خود تغییرات به وجود آورد، بدین صورت که تغییرات در موجودات براساس مشخصات اکتسابی وراثتی منتقل می شوند که برای اینکه نیاز های آن بخش را تامین کنند ایجاد شده اند، برای مثال ابتدا گردن همه زرافه ها کوتاه بوده و سپس به دلیل نیاز به گردن بلند تر برای دست یابی به برگ درختان گردنشان بلند شده است . اما نظریه داروین عنوان می کند که افراد جامعه تغییر نمی کنند. و جامعه است که تغییرات را ایجاد می کند، بدین صورت که افرادی که با شرایط محیط تطابق بیشتری دارند به

صورت انتخاب طبیعی باقی می مانند و نه براساس نیاز های یک بخش و بقیه از بین می روند، برای مثال زرافه های با گردن کوتاه و بلند وجود داشته اند اما به مرور زرافه های با گردن کوتاه به دلیل اینکه نتوانستند به غذای کافی دست پیدا کنند از بین می روند. (در اصل هر دو معتقد بودند که تکامل باعث ایجاد تغییر در جوامع می شود اما در نحوه رسیدن به آن تکامل با یک دیگر تفاوت داشتند، لامارک معتقد بود که تکامل ناشی از فعالیت های بیولوژیکی است در حالی که داروین معتقد بود که تکامل بر اساس انتخاب طبیعی می باشد.)

نظریه داروین در این مورد درست می باشد، همانطور که خرس های قطبی از نژاد خرس های قهوه ای می باشند که برای اینکه بتوانند بهتر با محیط استتار کنند و شکار خود را انجام دهند با آن محیط سازگار شده اند.

(منابع: http://sciencenetlinks.com/student-teacher-sheets/lamarck-and-darwin-summary-theories/ و
https://image.slidesharecdn.com/malwareevolution-121104235151-phpapp01/95/the-evolution-theory-of-malware-and-our-thought-32-638.jpg?cb=1352073145 و
https://arctic.au.dk/news-and-events/news/show/artikel/from-brown-to-white-evolution-of-the-polar-bear/)

8. در مورد عمل انتخاب به سوالات زیر پاسخ دهید.

الف) در انتخاب براساس چرخ رولت اگر تعداد انتخاب (ns) کوچک باشد، چه مشکلی ممکن است پیش بیاید؟

ممکن است اعداد تصادفی که بین صفر و یک تولید می کنیم به صورت یکنواخت پخش نشده باشند، یعنی افراد باید در جامعه متناسب با این فرمول $ns \times p_i$ (که p_i برابر احتمال انتخاب آن فرد می باشد) انتخاب شوند اما در حالتی که تعداد انتخاب کم باشد ممکن است این تناسب برقرار نشود چراکه میزان برقراری این تناسب با تعداد انتخاب نسبت مستقیم دارد، برای مثال هنگامی که ما یک تاس را 1200 بار پرتاب می کنیم احتمال اینکه در $\frac{1}{6}$ این پرتاب ها عدد 1 ظاهر شده باشد بیشتر است نسبت به زمانی که ما همین تاس را 24 بار پرتاب کرده باشیم.

ب) برای انتخاب جمعیت بازماندگان در حالت $(\mu + \lambda)$ استفاده از روش SUS مناسب تر است یا Q-Tournament؟ علت را توضیح دهید.

Q-Tournament بهتر است، به این دلیل که در انتخاب بازماندگان در حالت $(\mu + \lambda)$ تنوع جمعیت کاهش می یابد، و اگر از روش SUS نیز استفاده کنیم باز هم تنوع کاهش می یابد و امکان ماندن در بهینه محلی وجود دارد، در صورتی که روش Q-Tournament تنوع را بالا می برد و باعث ایجاد تعادل می شود.

آرش حریرپوش
9731505

ج) شایستگی اعضای یک جمعیت فرضی 10 تایی ارائه شده است . در صورت استفاده از روش SUS برای انتخاب 5 عضو، در ادامه روند تکامل چه مشکلی پیش می آید؟ چرا؟

8, 2, 6, 250, 4, 4, 7, 1, 4, 9

ابتدا شایستگی های جامعه را به صورت زیر مرتب می کنیم:

250, 9, 8, 7, 6, 4, 4, 4, 2, 1

به دلیل اینکه شایستگی موجود اول با بقیه اختلاف بسیار زیادی دارد همگرایی زودرس و سکون رخ می دهد و بعد از چند نسل همه افراد جامعه از آن نوع می شوند. (به دلیل اختلاف زیاد شایستگی یکی از افراد با میانگین باعث پر شدن جمعیت والدین توسط آن فرد می شود و باعث همگرایی زودرس و سکون می شود)

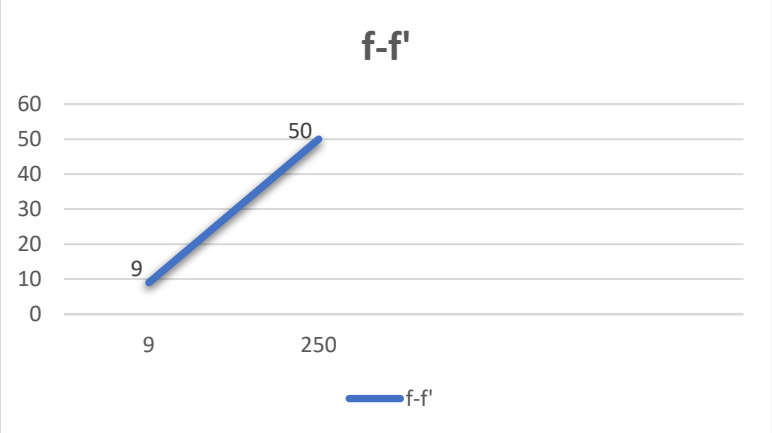
د) برای جمعیت مطرح شده در قسمت ج، با تعریف شایستگی مجازی و مقیاس سازی خطی، مشکل به وجود آمده را حل کنید. (تعیین β بر عهده خودتان است)

ابتدا دو تا از بزرگ ترین شایستگی ها را در شایستگی های بالا در نظر می گیریم، که برابر است با: 250 و 9 حال به کمک این دو شایستگی شایستگی مجازی را به گونه ای تعیین می کنیم تا فاصله این دو عدد با یک دیگر کم شود؛ بنابراین برای تعریف شایستگی مجازی به نام f_i' به صورت زیر عمل می کنیم:

ابتدا مقدار شایستگی 9 را در f_i' نیز برابر 9 می گذاریم.

سپس مقدار شایستگی 250 را برابر $\beta \times 250$ می گذاریم به صورتی فاصله این مقدار با 9 کم شود اما همچنان بزرگ تر از 9 باشد، مقدار β را در این مثال برابر 0.02 می گذاریم.

سپس نمودار شایستگی- شایستگی مجازی را به صورت زیر رسم می کنیم:



سپس معادله خط نمودار بالا را به صورت زیر محاسبه می کنیم:

$$f_i' - 9 = \frac{50-9}{250-9} \times (f_i - 9) \Rightarrow f_i' = \frac{41}{241} \times (f_i - 9) + 9 \Rightarrow f_i' \simeq 0.170 \times (f_i - 9) + 9$$

سپس، به کمک فرمول زیر شایستگی های مجازی شایستگی بالا را محاسبه می کنیم:

آرش حریرپوش
9731505

$$f_i' \simeq 0.170 \times (f_i - 9) + 9$$

بنابراین، شایستگی های مجازی این مثال برابر است با:

50, 9, 8.83, 8.66, 8.49, 8.15, 8.15, 8.15, 7.81, 7.64

بنابراین با قرار دادن شایستگی های مجازی بالا مشکلات بخش قبل حل می شود.

9. در مورد شرط خاتمه الگوریتم ها تکاملی به سوالات زیر پاسخ دهید.

الف) در تعیین شرط خاتمه تکامل، شرط تعداد ارزیابی های انجام شده چه مزیتی نسبت به شرط تعداد نسل ها(دور یا اپیاک) دارد؟

در روش تعداد ارزیابی ها می توانیم زمان الگوریتم را محاسبه کنیم چراکه پیچیدگی زمانی الگوریتم های تکاملی برابر تعداد ارزیابی ها می باشد، برای مقایسه الگوریتم ها تکاملی از این روش استفاده می کنیم چون در هر الگوریتم تکاملی تعداد فرزندان متفاوت است برای مقایسه باید تعداد ارزیابی ها را با یک دیگر برابر بگذاریم، تنها شرط خاتمه ای می باشد که تضمین می کند که از حلقه خارج می شود.

ب) همگرایی و عدم تنوع دو روش بر ای تعیین خاتمه الگوریتم تکاملی هستند. این دو روش چه تفاوت هایی با هم دارند و معیار وقوع هر کدام چیست؟

در روش همگرایی اگر مقدار متوسط شایستگی هر تکرار با تکرار قبلی اختلاف زیادی نداشته باشد، بدین معنی که شیب نمودار کارآیی بر حسب iteration، به صفر نزدیک شود، الگوریتم تکاملی خاتمه می یابد؛ در حالی که در روش عدم تنوع هنگامی که شایستگی ها به هم نزدیک و یا حتی برابر می شوند الگوریتم تکاملی خاتمه می یابد. (انحراف معیار ژنوتیپ ها را محاسبه می کنیم در صورتی که صفر شود الگوریتم تکاملی خاتمه می یابد)

در روش همگرایی در صورت عدم تنوع فنوتیپی و در روش عدم تنوع در صورت عدم تنوع ژنوتیپی الگوریتم خاتمه می یابد.

معیار وقوع روش همگرایی عدم اختلاف زیاد تکرار های متوالی با یک دیگر، و روش عدم تنوع صفر شدن انحراف معیار ژنوتیپی می باشد.

آرش حریرپوش
9731505

❖ بخش دوم – مسائل برنامه نویسی و پیاده سازی

- مسئله knapsack صفر و یک

ابتدا موارد زیر را در الگوریتم نوشته شده مشخص می کنیم:

بایزری (Binary)	نحوه بازنمایی مسئله
100	تعداد جمعیت والدین یا فرزندان
چرخ رولت (roulette wheel selection)	نحوه انتخاب والدین
انتخاب (μ+λ)	نحوه انتخاب بازماندگان
باز ترکیبی –دو –نقطه(2-point)	الگوریتم باز ترکیبی
جهش–بیتی (bit-wise)	الگوریتم جهش
تعداد نسل های تولید شده	شرط خاتمه

برای پیاده سازی این مسئله یک کلاس به نام Item نیاز داریم که مشخصات هر کالا را براساس آن ذخیره می کنیم، این کلاس را به صورت زیر پیاده سازی می کنیم:

```
class Item:
    def __init__(self, weight, value):
        self.weight = int(weight)
        self.value = int(value)

    def worthiness(self, item):
        return item.value/item.weight
```

همچنین یک کلاس دیگر به نام Fitness نیاز داریم که به کمک آن مقدار fitness (شایستگی) را برای اعضای جامعه مشخص کنیم، این کلاس به صورت زیر پیاده سازی می شود:

```
import numpy as np
class Fitness:
    def __init__(self, bag, Items, maxCap):
        self.bag = bag
        self.Items = Items
        self.maxCapacity = int(maxCap)
        self.fitness = 0

    # Calculate the fitness of individuals
    def bagFitness(self):
        values = [x.value for x in self.Items]
        weights = [x.weight for x in self.Items]
        # print(self.bag, values)
        totalVal = np.sum(self.bag * values)
        totalWeight = np.sum(self.bag * weights)
        if totalWeight <= self.maxCapacity:
            self.fitness = totalVal
        else:
            self.fitness = 0
        return self.fitness
```

سپس بقیه توابع مورد نیاز را پیاده سازی می کنیم، ابتدا به یک تابع نیاز داریم تا اطلاعات را از فایل بخواند، که این تابع را به صورت زیر پیاده سازی می کنیم:

```
import random as rd
from random import randint
import numpy as np, random, matplotlib.pyplot as plt

NumOfItems = None
weightOfKnapsack = None

# Read the data from the file
def read_data(filename):
    with open(filename) as f:
        Items = f.readlines()
        return Items
```

و پس از آن به کمک یک تابع دیگر که به صورت زیر پیاده سازی شده است، اطلاعات کالا ها را در یک آرایه که هر عضو آن از جنس کلاس Item می باشد ذخیره می کنیم:

```
# Return all of the items in the file in a suitable format
def all_items(items):
    all_item = []
    for i in items:
        items_info = i.rsplit()
        item = Item(items_info[1], items_info[0])
        all_item.append(item)
    return all_item
```

اکنون توابع مورد نیاز برای پیاده سازی الگوریتم تکاملی را پیاده سازی می کنیم، ابتدا به کمک تابع زیر جمعیت اولیه الگوریتم را تولید می کنیم، مقدار ژنوتیپ در هر یک از اعضای جامعه برابر آرایه ای از بیت های صفر یا یک به اندازه تعداد کالا های قابل انتخاب می باشد که یک بودن هر یک از این بیت ها را به منظور برداشتن آن کالا و صفر بودن آن را به منظور بر نداشتن آن کالا در نظر می گیریم؛ و اعضای اول جامعه ای که تولید می کنیم هر کدام به صورت تصادفی یکی از بیت های موجود در ژنوتیپشان 1 می باشد (یعنی اعضای اول هر کدام یک کالا به صورت تصادفی انتخاب کرده و برداشته اند):

```
# Create the first population
def initialPopulation(popSize, numOfItems):
    all_initial_population = []
    for i in range(popSize):
        initial_population = np.zeros((numOfItems, ))
        randNum = np.random.randint(0, NumOfItems-1, 1)
        initial_population[randNum] = 1
        initial_population = initial_population.astype(int)
        all_initial_population.append(initial_population)
    return np.array(all_initial_population)
```

حال که جمعیت اولیه را ساخته ایم، به کمک تابع زیر مقدار شایستگی (Fitness) را برای همه اعضای آن به کمک تابع زیر محاسبه می کنیم:

```
# Calculate Fitness of the population
def popFitness(population, items):
    fitnessResults = np.empty(len(population))
    for i in range(len(population)):
        fitnessResults[i] = Fitness(population[i],items,
weightOfKnapsack).bagFitness()
    return fitnessResults.astype(int)
```

که در این مسئله مقدار شایستگی برابر جمع کالا هایی است که هر فرد برداشته است، هر چه کالای گران قیمت تری برداشته باشد به شرط آن که وزن آن شرایط مسئله را رعایت کند شایستگی بیشتری دارد، و در صورتی که جمع وزن کالا های انتخابی بیش از مقدار آستانه کوله پشتی شود مقدار شایستگی آن فرد برابر صفر در نظر گرفته می شود؛ حال به کمک تابع زیر افراد شایسته را انتخاب می کنیم:

```
# Select the remaining individuals
def selection(fit, remainingSize):
    fitness = list(fit)
    length = len(fit) - remainingSize
    selectionResults = np.zeros((len(fit), ))
    for i in range(remainingSize):
        selectionResults[i] = fit[i]
        fitness[i] = -999999

    for i in range(length):
        max_fitness_idx = np.where(fit == np.max(fitness))
        selectionResults[i] = int(max_fitness_idx[0][0])
        fitness[max_fitness_idx[0][0]] = -999999
    return selectionResults
```

که در این مسئله نیز تعدادی افراد که به صورت هایپر پارامتر مشخص می شوند مستقیم به نسل بعد منتقل می شوند و سپس بقیه افراد بر حسب شایستگی شان بر اساس الگوریتم چرخ رولت انتخاب می شوند، در تابع بالا تنها به شایستگی افراد دسترسی داریم، بنابراین به کمک یک تابع دیگر والدین نسل بعدی را به کمک تابع قبلی انتخاب کرده ایم به صورت یک آرایه بر می گردانیم، که برابر است با:

```
# Selecting the parents of the next Generation
def matingPool(population, selectionResults):
    matingpool = []

    for I in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[int(index)])
    return matingpool
```

سپس تابع باز ترکیبی را به صورت زیر پیاده سازی می کنیم:

```
# Create a crossover function for two parents to create one child
def crossOver(parent1, parent2, items, crossover_rate):
    offspring = np.empty(len(parent2))
    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))
    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    fit_par1 = Fitness(parent1, items, weightOfKnapsack).bagFitness()
    fit_par2 = Fitness(parent2, items, weightOfKnapsack).bagFitness()
    if fit_par1 > fit_par2:
        stronger_parent = parent1
        weaker_parent = parent2

    else:
        stronger_parent = parent2
        weaker_parent = parent1

    x = rd.random()
    if x < crossover_rate:
        offspring[startGene:endGene] = stronger_parent[startGene:endGene]
        offspring[endGene:] = weaker_parent[endGene:]
        offspring[:startGene] = weaker_parent[:startGene]

    elif x == crossover_rate:
        offspring = stronger_parent

    else:
        offspring = weaker_parent

    return offspring
```

که به کمک این تابع باز ترکیبی را به صورت بازترکیبی دو نقطه بدین صورت که در صورتی که عدد تصادفی از نرخ باز ترکیبی کمتر باشد فاصله بین نقاط انتخابی را برابر ژنوتیپ های والد شایسته تر و بقیه نقاط را برابر ژنوتیپ والد دیگر قرار می دهیم و به منظور ایجاد تنوع در نسل بعدی اگر عدد تصادفی برابر با نرخ باز ترکیبی بود والد قوی تر و در غیر این صورت والد ضعیف تر به صورت مستقیم به نسل بعد منتقل می شوند، سپس به کمک تابع زیر باز ترکیبی را برای همه اعضا والدینی که انتخاب کرده ایم انجام می دهیم:

```
# Create function to run crossover over all of population pair
def crossoverPopulation(matingpool, remainingSize, items, crossover_rate):
    children = []
    length = len(matingpool) - remainingSize

    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, remainingSize):
        children.append(matingpool[i])

    for i in range(0, length):
        i = np.random.randint(0, len(children))
        child = crossOver(pool[i], pool[len(matingpool) - i - 1], items,
crossover_rate)
        children.append(child)
    return np.array(children).astype(int)
```

حال پس از انجام باز ترکیبی به کمک تابع بالا، توسط تابع زیر اعضای جامعه را به صورت جهش بیتی جهش می دهیم بدین صورت که در نقاطی که به صورت تصادفی انتخاب می شوند مقدار آن ژن را تغییر می دهیم بدین صورت که اگر یک بود آن را صفر و اگر صفر بود آن را یک می کنیم:

```
# Create mutation over all of our population
def mutatePopulation(population, mutationRate):
    mutants = [x for x in population]
    for i in range(len(population)):
        random_value = rd.random()

        if random_value > mutationRate:
            continue
        int_random_value = randint(0,len(population[i])-1)
        if mutants[i][int_random_value] == 0 :
            mutants[i][int_random_value] = 1
        else :
            mutants[i][int_random_value] = 0
    return mutants
```

سپس به کمک تابع زیر با استفاده از توابع بالا نسل بعد را تولید می کنیم:

```
# Create the next generation by pulling all steps together
def nextGeneration(currentGen, remainingSize, mutationRate, crossover_rate,
items):
    popRanked = popFitness(currentGen, items)
    selectionResults = selection(popRanked, remainingSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = crossoverPopulation(matingpool, remainingSize, items,
crossover_rate)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
```

و پس از آن به کمک تابع زیر الگوریتم تکاملی را ایجاد می کنیم و سپس مقادیر مورد نیاز را چاپ و یا رسم می کنیم:

```
# Final step: create the genetic algorithm
def geneticAlgorithm(items, popSize, remainingSize, mutationRate,
crossover_rate, generations, plot=True):
    pop = initialPopulation(popSize, len(items))
    maxValHist = []
    sum_fit = []
    values = np.array(pop) * [x.value for x in items]
    weights = np.array(pop) * [x.weight for x in items]

    for f in popFitness(pop, items):
        sum_fit.append(np.sum(f))
    maxValHist.append(np.max(sum_fit))

    max_weight = 0
    for v in range(len(values)):
        if np.sum(values[v]) == maxValHist[0]:
            max_weight = np.sum(weights[v])
            break

    print("Initial Value: " + str(maxValHist[0])+ ", Initial Weight: " +
str(max_weight))
```

```
    for i in range(0, generations):
        pop = nextGeneration(pop, remainingSize, mutationRate, crossover_rate,
items)
        if plot:
            sum_fit = []
            for j in popFitness(pop, items):
                sum_fit.append(np.sum(j))
            maxValHist.append(np.max(sum_fit))

    fitness_last_gen = popFitness(pop, items)
    max_value = np.max(fitness_last_gen)
    values = np.array(pop) * [x.value for x in items]
    weights = np.array(pop) * [x.weight for x in items]
    max_weight = 0
    for v in range(len(values)):
        if np.sum(values[v]) == max_value:
            max_weight = np.sum(weights[v])
            break

    print("Max value is : " + str(max_value) + " and knapsack weight is : " +
str(max_weight))
    if plot:
        plt.plot(maxValHist, color='blue', linestyle='dotted')
        plt.ylabel('MaxValue')
        plt.xlabel('Generation')
        plt.show()
    return max_value
```

و در نهایت الگوریتم تکاملی را به صورت زیر اجرا می کنیم:

```
if __name__ == '__main__':
    readItems = read_data('knapsack_1.txt')
    info = readItems[0].rsplit()
    NumOfItems = int(info[0])
    weightOfKnapsack = int(info[1])
    item_list = all_items(readItems[1:])
    geneticAlgorithm(items=item_list, popSize=100, remainingSize=25,
mutationRate=0.4, crossover_rate=0.8, generations=1000)
```

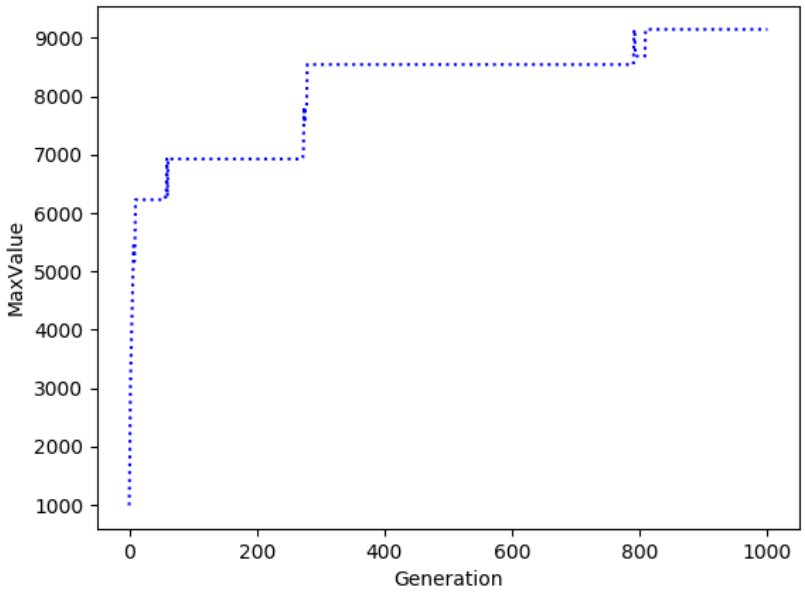
و در نتیجه خروجی آن در بهترین حالت برابر است با:

برای فایل knapsack_1.txt:

Initial Value: 992, Initial Weight: 298

Max value is : 9147 and knapsack weight is : 985

و همچنین نمودار شایستگی آن برابر است با:

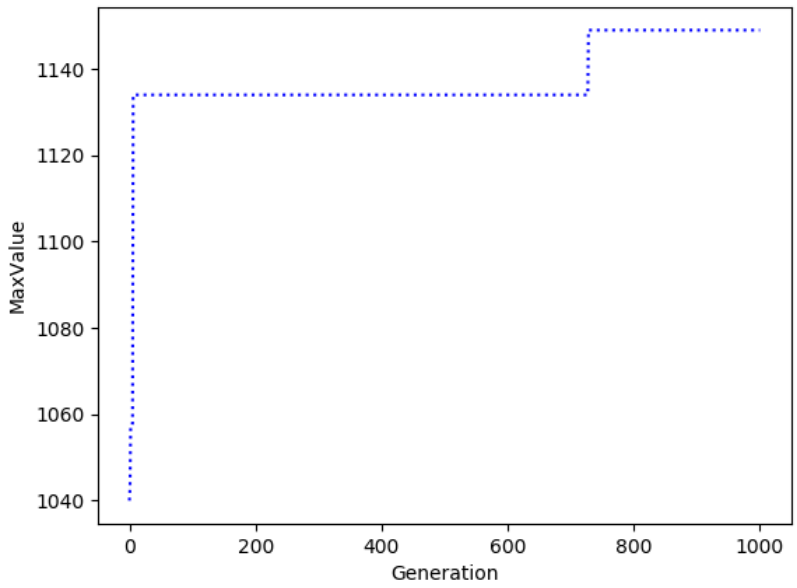


برای فایل knapsack_2.txt:

Initial Value: 1040, Initial Weight: 972

Max value is : 1149 and knapsack weight is : 970

و همچنین نمودار شایستگی آن برابر است با:

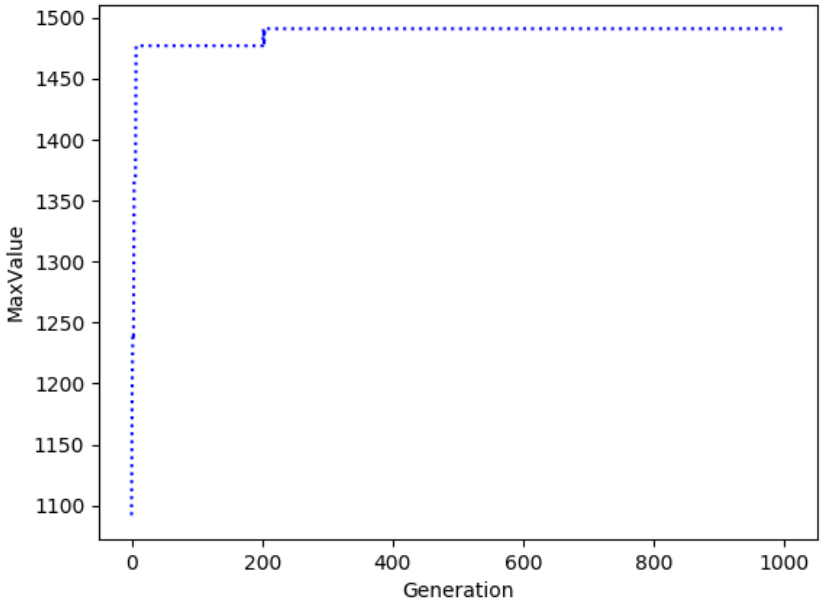


برای فایل knapsack_3.txt:

Initial Value: 1092, Initial Weight: 992

Max value is : 1491 and knapsack weight is : 991

و همچنین نمودار شایستگی آن برابر است با:



همانطور که از نتایج بالا مشخص است، به دلیل استفاده از روش چرخ رولت که مبتنی بر شایستگی می باشد در انتخاب بازماندگان سرعت همگرایی ها سریع می باشد اما مدت زمان ماندن در آن همگرایی ها نیز به همان اندازه زیاد می باشد، و مسئله دچار همگرایی های زودرس و پس از آن برای مدتی سکون می شود.

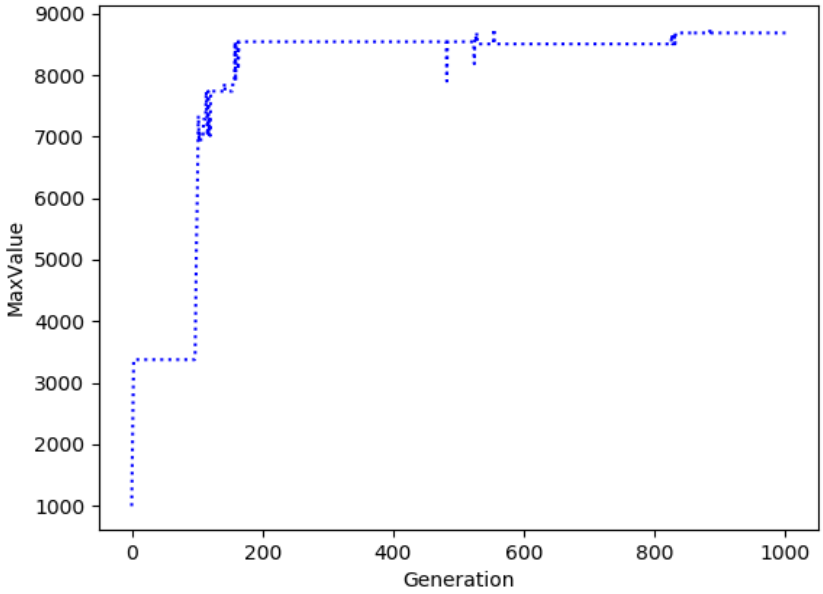
با افزایش مقدار نرخ جهش در مسئله بالا به 0.9 داریم:

برای فایل knapsack_1.txt:

Initial Value: 994, Initial Weight: 367

Max value is : 8687 and knapsack weight is : 908

و همچنین نمودار شایستگی آن برابر است با:

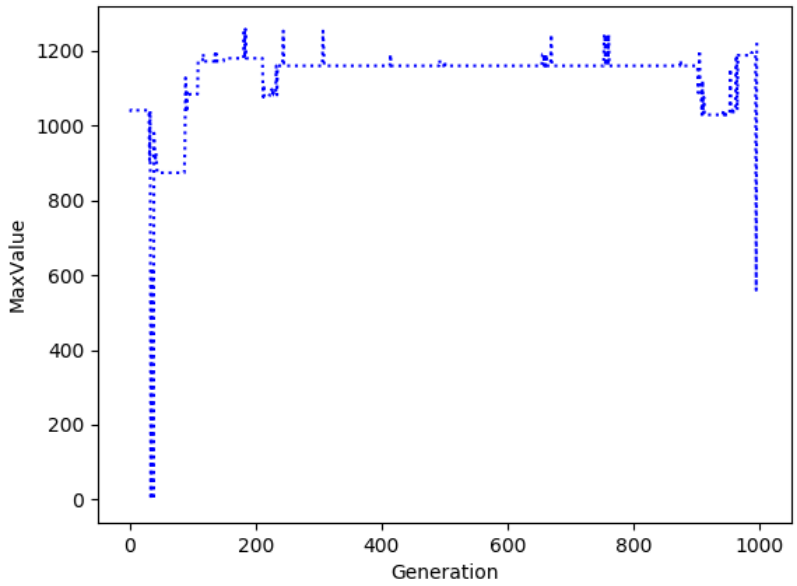


برای فایل knapsack_2.txt:

Initial Value: 1040, Initial Weight: 972

Max value is : 1226 and knapsack weight is : 946

و همچنین نمودار شایستگی آن برابر است با:



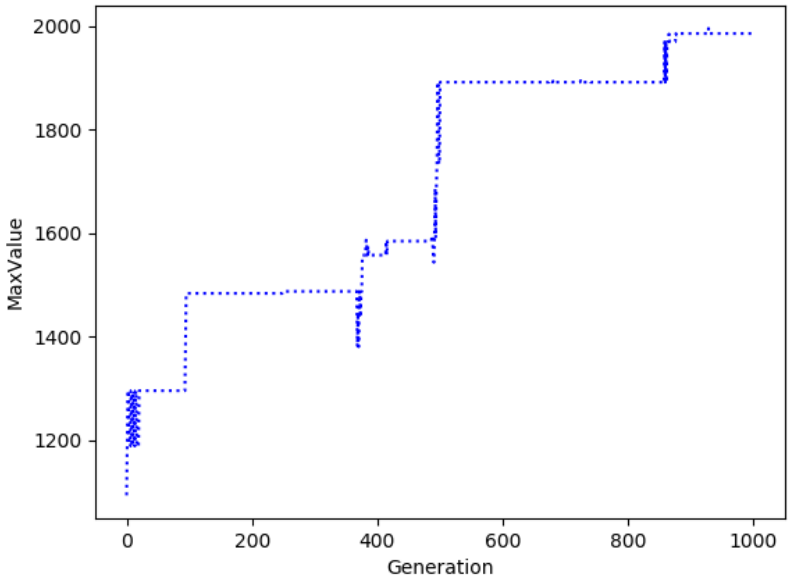
آرش حریرپوش
9731505

برای فایل knapsack_3.txt:

Initial Value: 1094, Initial Weight: 994

Max value is : 1986 and knapsack weight is : 986

و همچنین نمودار شایستگی آن برابر است با:



همانطور که مشاهده شد، در اغلب موارد با افزایش مقدار نرخ جهش شایستگی بهبود پیدا کرد و میزان نوسانات داده نیز بیشتر شده است که این نوسانات در این مسائل اغلب باعث بهبود شایستگی شده است.

حال مقدار نسل های تولید شده را 2000 و نرخ جهش را 0.9 قرار می دهیم:

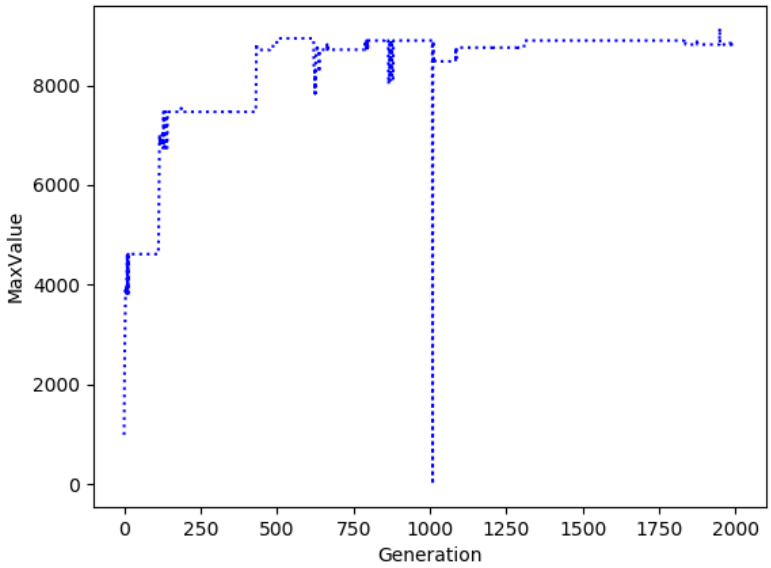
برای فایل knapsack_1.txt:

Initial Value: 994, Initial Weight: 367

Max value is : 8817 and knapsack weight is : 908

و همچنین نمودار شایستگی آن برابر است با:

آرش حریرپوش
9731505



همانطور که در این مثال مشاهده می شود، با افزایش تعداد نسل ها نیز همچنان شرایط قبل وجود دارد بدین صورت که مدتی دچار سکون و همگرایی زودرس می شود، که با تغییر مقادیر نرخ جهش می توان با ایجاد پراکندگی در جمعیت قدری از میزان تاثیر آن را کم کرد چراکه با انجام جستجو های بیشتر در فضا های بیشتر و جلوگیری از همگرایی باعث بهبود مسئله شده است و به نوعی اندکی تعادل ایجاد می کند، همچنین در این مسئله ما بیشتر نیاز داریم تا در هر فضای مسئله جستجو های بیشتری انجام دهیم تا حالات متفاوت را بررسی کنیم به همین دلیل افزایش نرخ جهش تقریبا در برخی موارد موثر عمل می کند؛ همچنین با افزایش تعداد افرادی که به صورت مستقیم به نسل بعد می روند نیز به دلیل اینکه تنوع را کاهش می دهیم مقدار شایستگی افراد جدید کاهش می یابد، چرا که در این مسئله افراد با ترکیب شدن با یک دیگر و ایجاد جهش کالا های دیگری را به کوله پشتی خود منتقل می کنند و اگر این تعداد کم شود در نتیجه میزان تنوع کالا های انتخاب شده در نسل های بعد کاهش می یابد؛ همچنین صفر شدن ناگهانی شایستگی نیز به دلیل است که در آن نسل هیچ فردی نتوانسته آستانه کوله پشتی را رعایت کند به همین دلیل شایستگی اش صفر شده که این اتفاق در نسل های بعد اصلاح می شود، همچنین به دلیل اینکه از روش انتخاب چرخ رولت استفاده کرده ایم باعث می شود تنوع نسل ها کاهش پیدا کند بنابراین با کم کردن تعداد افرادی که در هر نسل به صورت مستقیم به نسل بعد می روند می توانیم به نتایج بهتری برسیم چرا که با کم کردن تعداد این افراد تنوع نسل ها را افزایش می دهیم (نسبت با حالتی که تعداد این افراد زیاد است تنوع را افزایش داده ایم).

- مسئله فروشنده دوره گرد

ابتدا موارد زیر را در الگوریتم نوشته شده مشخص می کنیم:

اعداد حقیقی	نحوه بازنمایی مسئله
200	تعداد جمعیت والدین یا فرزندان
چرخ رولت (roulette wheel selection)	نحوه انتخاب والدین
انتخاب (μ+λ)	نحوه انتخاب بازماندگان
باز ترکیبی –مرتبه یک (Order 1)	الگوریتم باز ترکیبی
جهش–تعویض (swap)	الگوریتم جهش
تعداد نسل های تولید شده	شرط خاتمه

برای پیاده سازی این مسئله یک کلاس به نام City نیاز داریم که مشخصات شهر را براساس آن ذخیره می کنیم، همچنین فاصله بین دو شهر را نیز به کمک یک تابع در این کلاس محاسبه می کنیم، این کلاس را به صورت زیر پیاده سازی می کنیم:

```
import numpy as np
class City:
    def __init__(self,i, x, y):
        self.i = int(i)
        self.x = float(x)
        self.y = float(y)

    # Calculate the distance between two cities
    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

همچنین یک کلاس دیگر به نام Fitness نیاز داریم که به کمک آن مقدار fitness (شایستگی) را برای اعضای جامعه مشخص کنیم، این کلاس به صورت زیر پیاده سازی می شود:

```
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0.0
        self.fitness = 0.0

    # Calculate the distance for one individual
    def routeDistance(self):
        if self.distance == 0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
```

```
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    # Calculate fitness for one individual
    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness
```

سپس بقیه توابع مورد نیاز را پیاده سازی می کنیم، ابتدا به یک تابع نیاز داریم تا اطلاعات را از فایل بخواند، که این تابع را به صورت زیر پیاده سازی می کنیم:

```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt
import time

# Read the data from the file
def read_data(filename):
    with open(filename) as f:
        data = f.readlines()
        return data
```

و پس از آن به کمک یک تابع دیگر که به صورت زیر پیاده سازی شده است، اطلاعات کالا ها را در یک آرایه که هر عضو آن از جنس کلاس City می باشد ذخیره می کنیم:

```
# Return all of the cities in the file in a suitable format
def all_Citys(citys):
    all_city = []
    for c in citys:
        city_info = c.rsplit()
        city = City(i=city_info[0], x=city_info[1], y=city_info[2])
        all_city.append(city)
    return all_city
```

اکنون توابع مورد نیاز برای پیاده سازی الگوریتم تکاملی را پیاده سازی می کنیم، ابتدا به کمک تابع زیر جمعیت اولیه الگوریتم را تولید می کنیم، مقدار ژنوتیپ در هر یک از اعضای جامعه برابر آرایه ای از شهر های موجود در فایل داده شده به اندازه تعداد کالا های قابل انتخاب می باشد که از جنس کلاس City می باشند (ژنوتیپ هر فرد در جامعه برابر یک مسیر می باشد که از تمامی نقاط شهر را شامل می شود):

```
# Create one individual
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
```

سپس به کمک تابع بالا تمامی اعضای جامعه را تولید می کنیم:

```
# Create the first population
def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))

    return population
```

حال که جمعیت اولیه را ساخته ایم، به کمک تابع زیر مقدار شایستگی (Fitness) را برای همه اعضای آن به کمک تابع زیر محاسبه و سپس به ترتیب نزولی مرتب می کنیم:

```
# Rank individuals
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse
= True)
```

که در این مسئله مقدار شایستگی برابر معکوس مقدار مسیری است که طی می کند تا از کل شهر ها یک بار عبور کند در نتیجه هر چه از مسیر های کوتاه تری عبور کند شایستگی آن فرد بیشتر می شود؛ حال به کمک تابع زیر افراد شایسته را انتخاب می کنیم:

```
# Create a selection function that will be used to make the list of parent
routes
def selection(popRanked, remainingSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100 * df.cum_sum / df.Fitness.sum()

    for i in range(0, remainingSize):
        selectionResults.append(popRanked[i][0])

    for i in range(0, len(popRanked) - remainingSize):
        pick = 100 * random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i, 3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

که در این مسئله نیز تعدادی افراد که به صورت هایپر پارامتر مشخص می شوند مستقیم به نسل بعد منتقل می شوند و سپس بقیه افراد بر حسب شایستگی شان بر اساس الگوریتم چرخ رولت انتخاب می شوند، در تابع بالا تنها به شایستگی افراد دسترسی داریم، بنابراین به کمک یک تابع دیگر والدین نسل بعدی را به کمک تابع قبلی انتخاب کرده ایم به صورت یک آرایه بر می گردانیم، که برابر است با:

```
# Selecting the parents of the next Generation
def matingPool(population, selectionResults):
    matingpool = []

    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool
```

سپس تابع بازترکیبی را به صورت زیر پیاده سازی می کنیم:

```
# Create a crossover function for two parents to create one child (should be
changed)
def crossover(parent1, parent2, crossover_rate):
    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))
    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    fit_par1 = Fitness(parent1).routeFitness()
    fit_par2 = Fitness(parent2).routeFitness()
    if fit_par1 > fit_par2:
        stronger_parent = parent1
        weaker_parent = parent2
    else:
        stronger_parent = parent2
        weaker_parent = parent1

    x = random.random()
    if x > crossover_rate:
        child = stronger_parent

    else:
        child1 = []

        for i in range(startGene, endGene):
            child1.append(stronger_parent[i])

        child2 = [item for item in weaker_parent if item not in child1]
        child =child1 + child2
    return child
```

که به کمک این تابع باز ترکیبی را به صورت بازترکیبی مرتبه یک بدین صورت که در صورتی که عدد تصادفی از نرخ باز ترکیبی کمتر باشد والد قوی تر به صورت مستقیم به نسل بعد منتقل می شوند و در غیر این صورت فاصله بین نقاط انتخابی را برابر ژنوتیپ های والد شایسته تر و بقیه نقاط را برابر ژنوتیپ والد دیگر قرار می دهیم، سپس به کمک تابع زیر باز ترکیبی را برای همه اعضا والدینی که انتخاب کرده ایم انجام می دهیم:

```
# Create function to run crossover over all of population pair
def crossoverPopulation(matingpool, remainingSize, crossover_rate):
    children = []
    length = len(matingpool) - remainingSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, remainingSize):
```

```
        children.append(matingpool[i])

    while len(children) - remainingSize != length:
        par_index = np.random.randint(0, len(pool),2)
        if pool[par_index[0]] == pool[par_index[1]]:
            continue
        child = crossover(pool[par_index[0]], pool[par_index[1]],
crossover_rate)
        if child not in children:
            children.append(child)
    return children
```

حال پس از انجام باز ترکیبی به کمک تابع بالا، توسط تابع زیر یکی از اعضا جامعه را به صورت جهش تعویض جهش می دهیم بدین صورت که اگر مقدار تصادفی از نرخ جهش کمتر باشد ترتیب ژنوتیپ های مورد بررسی با یک دیگر جابجا می شود :

```
# Create mutation for a single route
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if (random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
```

و به کمک تابع زیر تمامی اعضای جامعه را جهش می دهیم:

```
# Create mutation over all of our population
def mutatePopulation(population, mutationRate):
    mutatedPop = []
    length = len(population)

    while len(mutatedPop) != length:
        ind = np.random.randint(0, length)
        mutatedInd = mutate(population[ind], mutationRate)

        if mutatedInd not in mutatedPop:
            mutatedPop.append(mutatedInd)

    return mutatedPop
```

سپس به کمک تابع زیر با استفاده از توابع بالا نسل بعد را تولید می کنیم:

```
# Create the next generation by pulling all steps together
def nextGeneration(currentGen, remainingSize, mutationRate, crossover_rate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, remainingSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = crossoverPopulation(matingpool, remainingSize, crossover_rate)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
```

و پس از آن به کمک تابع زیر الگوریتم تکاملی را ایجاد می کنیم و سپس مقادیر مورد نیاز را چاپ و یا رسم می کنیم:

```
# Final step: create the genetic algorithm
def geneticAlgorithm(population, popSize, remainingSize, mutationRate,
crossover_rate, generations, plot= True):
    pop = initialPopulation(popSize, population)
    fit = []
    dis = []
    if plot:
        rank = rankRoutes(pop)[0][1]
        fit.append(rank)
        dis.append(1/rank)

    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, remainingSize, mutationRate, crossover_rate)
        if plot:
            rank = rankRoutes(pop)[0][1]
            fit.append(rank)
            dis.append(1/rank)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    if plot:
        plt.plot(fit, color='blue', linestyle='dotted')
        plt.ylabel('Fit')
        plt.xlabel('Generation')
        plt.show()
        plt.plot(dis, color='blue', linestyle='dotted')
        plt.ylabel('Distance')
        plt.xlabel('Generation')
        plt.show()

    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute
```

و در نهایت الگوریتم تکاملی را به صورت زیر اجرا می کنیم:

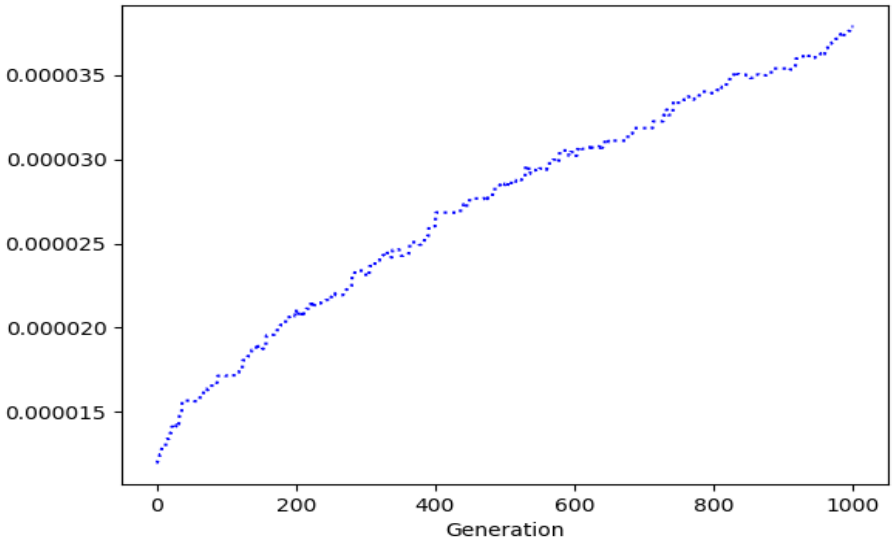
```
if __name__ == '__main__':

    start = time.time()

    cityList = all_Citys(read_data('tsp_data.txt'))
    bestRoute = geneticAlgorithm(population=cityList, popSize=200,
remainingSize=50,
                                 mutationRate=0.0001, crossover_rate=0.8,
generations=1000)
    end = time.time()
    print('elapsed time is: ', (end-start)/60.0, 'min')
    print(bestRoute)
```

و در نتیجه خروجی آن در بهترین حالت برابر است با:

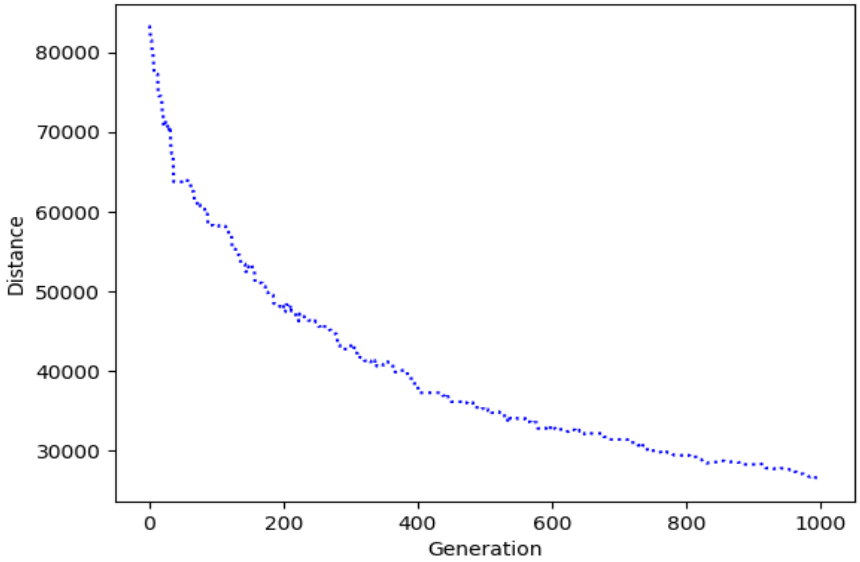
Initial distance: 83308.49704360946

Final distance: 26414.14024810142

و همچنین نمودار شایستگی آن برابر است با:



و نمودار کمترین فاصله در هر نسل آن برابر است با:

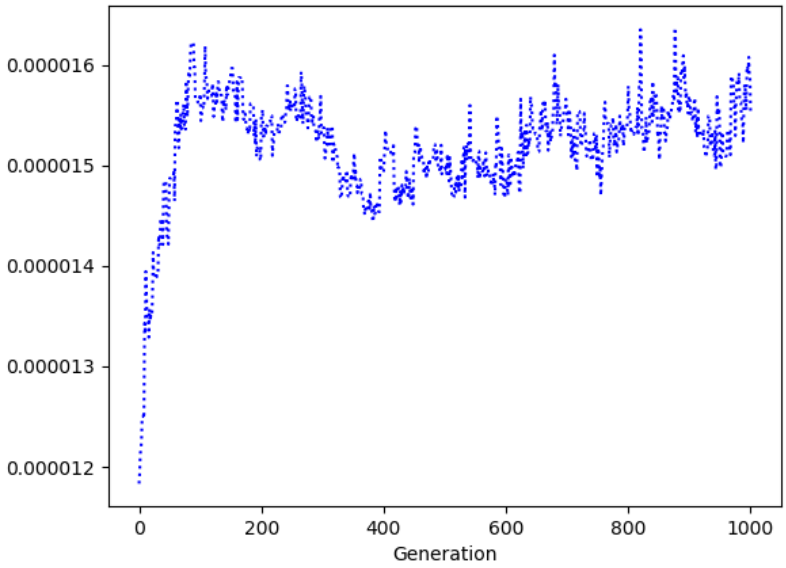


با افزایش نرخ جهش به مقدار 0.001 داریم:

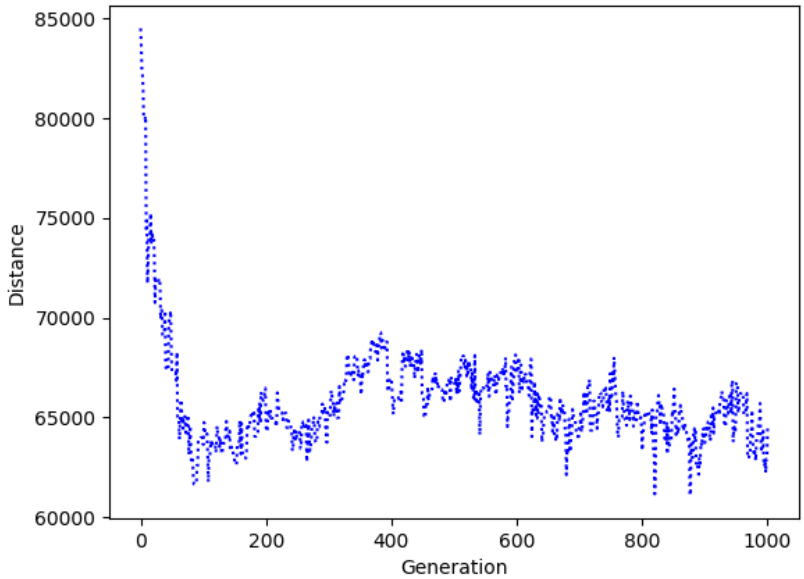
Initial distance: 84506.44408407834

Final distance: 64493.403854257755

و همچنین نمودار شایستگی آن برابر است با:



و نمودار کمترین فاصله در هر نسل آن برابر است با:

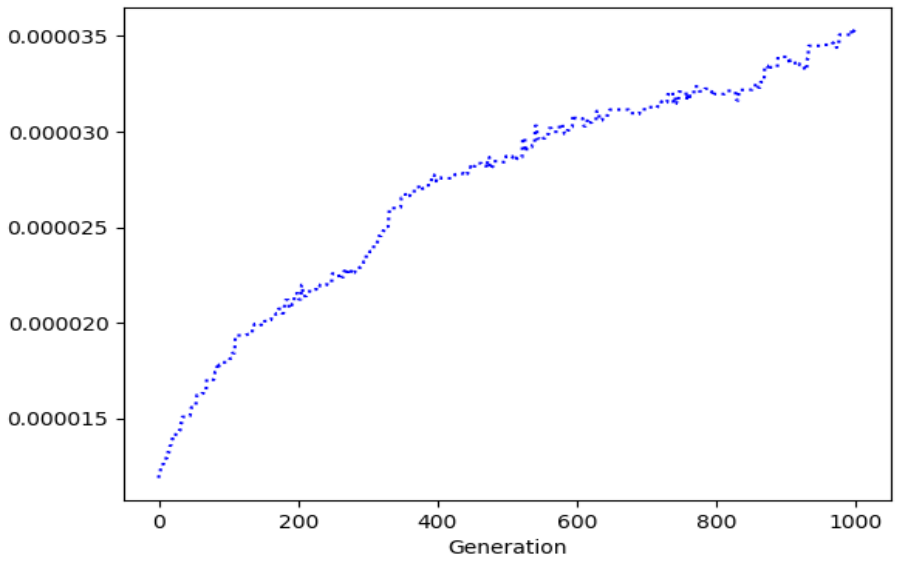


با تغییر نرخ جهش به مقدار 0.0001 و میزان افراد باقی مانده در هر نسل به 75 نفر داریم:

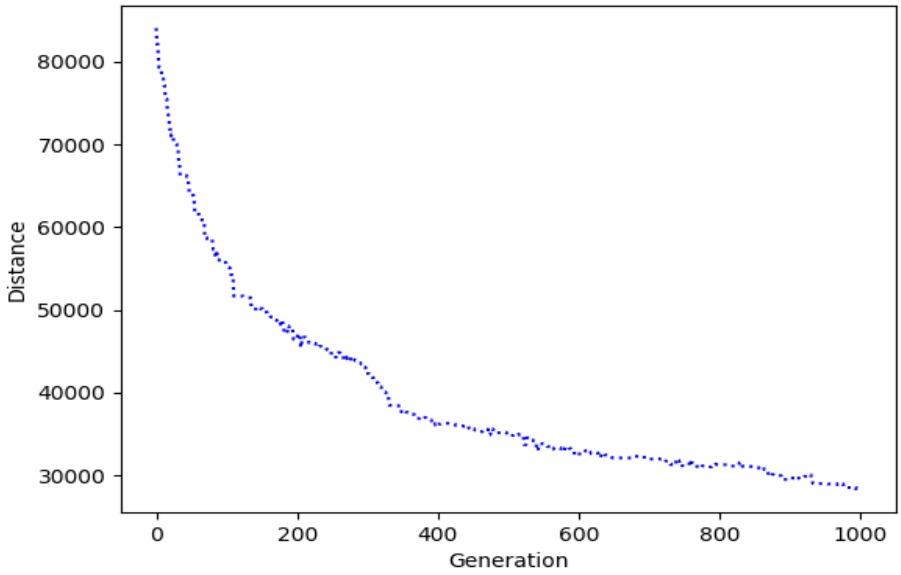
Initial distance: 84075.71618730489

Final distance: 28285.30423256685

و همچنین نمودار شایستگی آن برابر است با:



و نمودار کمترین فاصله در هر نسل آن برابر است با:

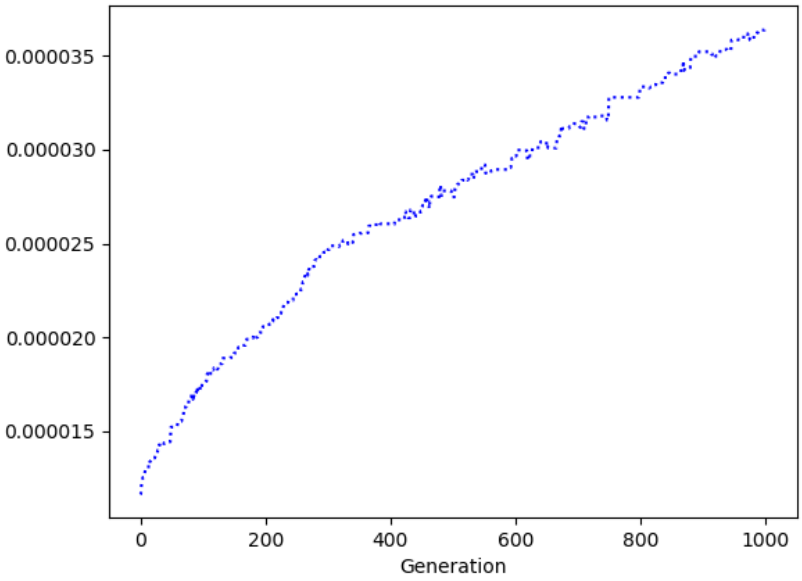


با تغییر نرخ جهش به مقدار 0.0001 و میزان افراد باقی مانده در هر نسل به 25 نفر داریم:

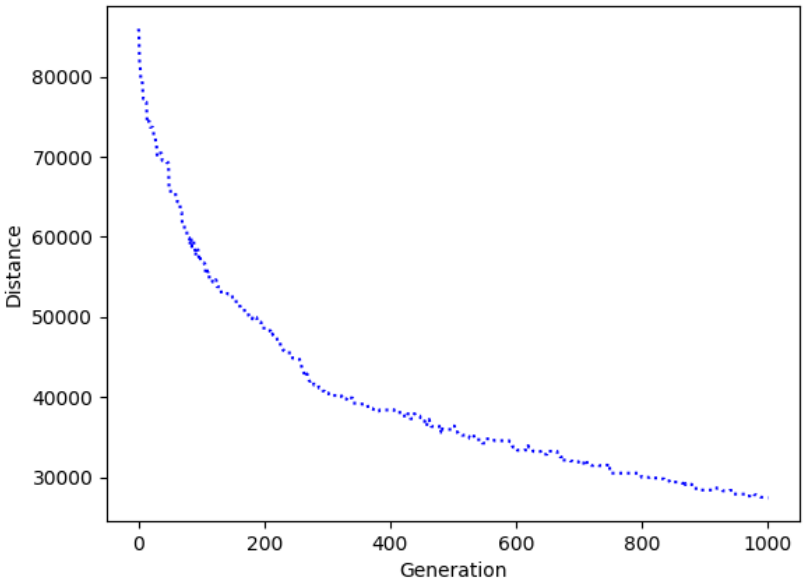
Initial distance: 83937.805004851

Final distance: 31751.64470174861

و همچنین نمودار شایستگی آن برابر است با:



و نمودار کمترین فاصله در هر نسل آن برابر است با:



همانطور که در نمونه های بالا مشاهده می کنید، در حالت اول که حالت بهینه می باشد به دلیل استفاده از روش چرخ رولت مسئله در حال همگرایی می باشد و در این مسئله مشکل سکون یا همگرایی زودرس ایجاد نشده است، اما احتمال دارد با افزایش نسل ها به دلیل کوچک بودن مقدار نرخ جهش در نقاط بهینه محلی به دام افتد، سپس با افزایش نرخ جهش مقدار بهینه مسئله به شدت افزایش پیدا کرده است، که این به این دلیل می باشد که در این مسئله با افزایش نرخ جهش میزان جستجو محلی افزایش پیدا کرده است و از همگرایی نیز کاسته شده است اگر چه این روش سرعت همگرایی کم می باشد اما با افزایش نسل ها ممکن است به نتایج بهتری برسد چرا که به دلیل بیشتر بودن نرخ جهش امکان به دام افتادن در نقاط بهینه محلی کمتر می شود، و همچنین با افزایش یا کاهش تعداد افرادی که وارد نسل بعدی می شوند در این تعداد نسل هر دو باعث افزایش نقطه بهینه محلی شده اند چرا که با افزایش تعداد افرادی که به نسل بعد می روند تنوع کاهش می یابد و همچنین اگر تعداد افرادی که به نسل بعد به صورت مستقیم منتقل می شوند را از یک مقداری بیشتر کاهش دهیم، به دلیل افزایش تنوع میزان همگرایی کاهش پیدا می کند؛ همچنین به دلیل اینکه از روش چرخ رولت برای انتخاب والدین انتخاب کرده ایم که خود باعث کاهش تنوع در نسل بعدی می شود کاهش تعداد افرادی که به صورت مستقیم به نسل بعد منتقل می شوند تا حدی می توانند بهتر باشد چرا که باعث می شود تنوع جامعه خیلی کم نشود.

(برای هر بار اجرای این الگوریتم برای 1000 نسل بین 10 تا 15 دقیقه زمان نیاز است.)

(کد های این دو به صورت دو پوشه مجزا در پوشه ESAlgorithm قرار داده شده است که پوشه Knapsack مربوط به مسئله اول و پوشه TravelingSalesmanProblem مربوط به مسئله دوم می باشد؛ همچنین فایل های jupyter notebook این مسائل نیز به نام های Knapsack_01.ipynb برای مسئله اول و TravelingSalesmanProblem(TSP).ipyn برای مسئله دوم قرار داده شده است.)

(منبع در ارتباط با کوچک بودن نرخ جهش:
https://www.researchgate.net/post/Why_is_the_mutation_rate_in_genetic_algorithms_very_small)