دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

عنوان:

تمرین اول هوش محاسباتی: الگوریتم‌های ژنتیک

Genetic Algorithms

نگارش

دانیال شفیعی

مهدی مهدیه

امیررضا نجفی

استاد راهنما

دکتر کارشناس

اسفند ۱۴۰۳

فهرست مطالب

۰ مقدمه ۲

۱ مبانی و مفاهیم الگوریتم ژنتیک ۲

۲ درک و حل مسائل با الگوریتم ژنتیک ۱۱

۳ پیاده‌سازی، ارزیابی و تجزیه و تحلیل الگوریتم ژنتیک ۱۵

۰ مقدمه

هدف از این تمرین آشنایی بیشتر با الگوریتم‌های ژنتیک و استفاده‌ی بیشتر از آن‌ها در کاربردهای عملی است.

۱ مبانی و مفاهیم الگوریتم ژنتیک

۱. **تعریف الگوریتم تکاملی:** بر اساس مقاله مذکور در تمرین درس، الگوریتم‌های تکاملی (Evolution Strategies) یک کلاس از الگوریتم‌های بهینه‌سازی جعبه سیاه هستند که رویه‌های جستجوی اکتشافی الهام‌گرفته از تکامل طبیعی هستند. در هر تکرار (نسل) یک جمعیت از بردارهای پارامتر (ژنوتیپ) آشفته (جهش) می‌شوند و مقدار تابع هدف (Fitness function) آن‌ها ارزیابی می‌شود. سپس بردارهای پارامتر با بالاترین امتیاز برای تشکیل جمعیت نسل بعدی ترکیب می‌شوند و این رویه تا زمانی که هدف به طور کامل بهینه شود، تکرار می‌شود.

بر اساس مقاله، مزایای اصلی الگوریتم‌های تکاملی (ES) نسبت به الگوریتم‌های یادگیری تقویتی (RL) در برخی مسائل شامل موارد زیر است:

۱. **موازی‌سازی بالا و کاهش هزینه محاسباتی:** الگوریتم‌های تکاملی به طور طبیعی قابلیت موازی‌سازی بالایی دارند. با استفاده از یک استراتژی ارتباطی مبتنی بر اعداد تصادفی مشترک، این الگوریتم‌ها توانسته‌اند تنها با ارسال مقادیر اسکالر میان کارگران موازی (واحدهای پردازشی مستقل که در یک سیستم توزیع‌شده یا موازی کار می‌کنند تا بخشی از محاسبات را انجام دهد.)، به‌سرعت به بیش از هزار کارگر موازی مقیاس‌پذیر شود. این امر باعث می‌شود که ES در مقایسه با روش‌های یادگیری تقویتی مانند Policy Gradients که نیاز به ارسال گرادیان‌های کامل دارند، نیاز به پهنای باند بسیار کمتری داشته باشد. مقیاس‌پذیری بالا باعث می‌شود که ES بتواند مسائل پیچیده؛ مانند راه‌رفتن انسان‌نمای سه‌بعدی را در مدت‌زمان کوتاه ۱۰ دقیقه ای حل کند. به طور خلاصه :

- روش‌های ES به دلیل عدم نیاز به پس انتشار گرادیان (Backpropagation) به‌راحتی در سیستم‌های توزیع‌شده اجرا می‌شوند.
- ارتباطات بین پردازنده‌ها در ES بسیار کم‌حجم‌تر از RL است، زیرا فقط یک مقدار اسکالر ارسال می‌شود، درحالی‌که RL نیاز به ارسال گرادیان‌های کامل دارد.
- این الگوریتم‌ها امکان اجرای هم‌زمان روی هزاران پردازنده را فراهم می‌کنند که در برخی آزمایش‌ها باعث کاهش زمان یادگیری از ساعت‌ها به چند دقیقه شده است.

۲. **عدم نیاز به تابع ارزش (Value Function) و کاهش پیچیدگی مدل:** این الگوریتم‌ها نیازی به تخمین تابع ارزش ندارند. درحالی‌که بسیاری از الگوریتم‌های RL مانند Q-learning و Policy Gradients به تخمین تابع ارزش وابسته هستند. این امر باعث می‌شود که ES از نظر محاسباتی سبک‌تر باشد و نیاز به حافظه کمتری داشته باشد. عدم نیاز به تخمین تابع ارزش باعث می‌شود که ES در مواجهه با پاداش‌های بسیار پراکنده یا تأخیر دار عملکرد بهتری داشته باشد. به طور خلاصه:

- بسیاری از روش‌های RL مانند یادگیری سیاست (Policy Gradient) برای بهبود عملکرد نیاز به تخمین تابع ارزش دارند که محاسبات را پیچیده و وابسته به هایپرپارامترها می‌کند.
- ES مستقیماً پارامترهای سیاست را جستجو می‌کند و نیازی به تابع ارزش ندارد.

۳. **مقاومت در برابر افق‌های زمانی طولانی و پاداش‌های تأخیری:** این الگوریتم‌ها به دلیل اینکه تنها از بازده کل یک اپیزود استفاده می‌کند، نسبت به تأخیر در پاداش‌ها و افق‌های زمانی طولانی مقاوم است. این در حالی است که روش‌های RL مانند Policy Gradients ممکن است به دلیل نیاز به تخمین تابع ارزش و تأثیرات بلندمدت اقدامات، در مواجهه با افق‌های زمانی طولانی دچار مشکل شوند. به طور خلاصه:

- در RL، مقداردهی تنزیلی (Discounting) و تابع ارزش برای مدیریت پاداش‌های تأخیری استفاده می‌شود که در مسائل با افق‌های زمانی بلند (مانند تصمیم‌گیری استراتژیک) ممکن است منجر به یادگیری نامناسب شود.
- ES مستقیماً بر بهینه‌سازی کل اپیزود تمرکز دارد و نیازی به کاهش‌دادن پاداش ندارد، بنابراین در مسائل با وابستگی‌های بلندمدت کارآمدتر است.

۴. **کاوش بهتر و متنوع‌تر نسبت به روش‌های مبتنی بر گرادیان:** روش‌های مبتنی بر گرادیان مانند RL تمایل دارند که در مینیمم‌های محلی گیر بیفتند و در بسیاری از موارد به بهینه سراسری نرسند. ES با تغییر پارامترهای سیاست به‌صورت تصادفی، رفتارهای جدید و غیرمنتظره را تولید می‌کند که در برخی آزمایش‌ها، منجر به یادگیری انواع راه‌حل‌های جدید (مانند سبک‌های مختلف راه‌رفتن در شبیه‌سازی ربات‌ها) شده است.

۵. **عدم وابستگی به نرخ فریم (Frame-Skip) و فرکانس اقدامات:** در یادگیری تقویتی، نرخ فریم و نرخ انجام اقدام نقش مهمی در موفقیت یادگیری دارد. ES نسبت به فرکانس اقدامات (Action Frequency) بی‌تفاوت است، درحالی‌که در روش‌های RL ، تنظیم فرکانس اقدامات (Frame-skip) یک پارامتر حیاتی است که می‌تواند به طور قابل‌توجهی بر عملکرد الگوریتم تأثیر بگذارد.

۶. **عدم نیاز به تخفیف زمانی (Temporal Discounting):** ES نیازی به اعمال تخفیف زمانی روی پاداش‌ها ندارد، درحالی‌که بسیاری از روش‌های RL برای کاهش واریانس گرادیان‌ها، از تخفیف زمانی استفاده می‌کنند. این امر می‌تواند باعث ایجاد مشکل در تخمین گرادیان‌ها شود، به‌ویژه زمانی که اقدامات تأثیرات بلندمدت دارند.

۷. **عدم نیاز به محاسبه گرادیان‌ها:** نیازی به محاسبه گرادیان‌ها از طریق backpropagation ندارد، این امر باعث کاهش حجم محاسبات و حافظه مورد نیاز می‌شود. همچنین، این ویژگی باعث می‌شود که ES در مواجهه با شبکه‌های عصبی با گرادیان‌های منفجرشده (Exploding Gradients) مقاوم‌تر باشد.

۸. **امکان استفاده از شبکه‌های عصبی غیرقابل‌تفکیک و سخت‌افزارهای کم‌دقت:** در RL، به دلیل نیاز به محاسبه گرادیان، معماری شبکه عصبی باید مشتق‌پذیر باشد. اما در ES، می‌توان از معماری‌هایی که غیرقابل‌تفکیک هستند (مانند توجه سخت یا شبکه‌های باینری) نیز استفاده کرد. این روش همچنین برای سخت‌افزارهای محاسباتی کم‌دقت مانند TPU مناسب است.

۲. الگوریتم ژنتیک (GA) یک روش بهینه‌سازی و جستجو است که بر اساس اصول تکامل طبیعی داروین، مانند انتخاب طبیعی، جهش (Mutation) و ترکیب (Crossover)، عمل می‌کند. این الگوریتم از یک جمعیت از راه‌حل‌های ممکن (کروموزوم‌ها) شروع کرده و با اعمال رفتارهای تکاملی، نسل‌های جدیدی ایجاد می‌کند تا در نهایت به یک جواب بهینه یا نزدیک بهینه برسد.

مراحل الگوریتم ژنتیک:

۱. **ایجاد جمعیت اولیه:** مجموعه‌ای از کروموزوم‌های تصادفی ایجاد می‌شود.

۲. **محاسبه تابع برازندگی (Fitness Function):** هر کروموزوم ارزیابی می‌شود تا میزان مناسب‌بودن آن برای حل مسئله مشخص شود.

۳. **انتخاب (Selection):** کروموزوم‌هایی که مقدار برازندگی بهتری دارند، شانس بیشتری برای تولید نسل بعدی به‌عنوان والد خواهند داشت.

۴. **ترکیب (Crossover):** دو کروموزوم والد با یکدیگر ترکیب شده و فرزندان جدیدی تولید می‌کنند.

۵. **جهش (Mutation):** به‌صورت تصادفی تغییراتی در برخی کروموزوم‌ها اعمال می‌شود تا تنوع حفظ شود.

۶. **تکرار فرایند:** این فرایند تا رسیدن به یک معیار توقف، مانند رسیدن به یک جواب بهینه یا پایان‌یافتن تعداد نسل‌ها ادامه می‌یابد (به‌عنوان شرط حلقه while به‌عنوان‌مثال در پیاده‌سازی).

ویژگی	الگوریتم ژنتیک (GA)	برنامه‌نویسی تکاملی (EP)	استراتژی‌های تکاملی (ES)
نمایش راه‌حل‌ها	کروموزوم‌ها معمولاً رشته‌های باینری، اعداد یا کدگذاری خاص دارند.	هر فرد به‌عنوان یک استراتژی مستقل نمایش داده می‌شود.	بردارهای عددی با پارامترهای تصادفی نمایش داده می‌شوند.
عملگرهای تکاملی	ترکیب (Crossover) و جهش (Mutation)	تمرکز بیشتر بر جهش، ترکیب کمتر مورد استفاده قرار می‌گیرد.	بیشتر از جهش استفاده شده و ترکیب محدود است.
کاربردها	مسائل عمومی مانند بهینه‌سازی، یادگیری ماشین، مهندسی و...	مسائل مرتبط با یادگیری و مدل‌سازی هوش مصنوعی	مسائل عددی، بهینه‌سازی پارامترها و کنترل تطبیقی
نحوه انتخاب والدین	انتخاب بر اساس تابع برازندگی	انتخاب تصادفی با احتمال بیشتر برای افراد بهتر	انتخاب بر اساس مقایسه مستقیم عملکرد فردی
نحوه ایجاد نسل جدید	ترکیب و جهش کروموزوم‌ها برای تولید فرزندان جدید	فقط جهش روی هر فرد اعمال می‌شود	استفاده از بردارهای تصادفی برای تغییر مقدار پارامترها

جدول ۱: تفاوت الگوریتم ژنتیک (GA) با الگوریتم برنامه‌نویسی تکاملی (EP) و استراتژی‌های تکاملی (ES)

۳. (آ) در مورد رشته‌های بیتی (Bitstring)، استفاده از عملگرهای الگوریتم به شکل زیر است:

۱. **ترکیب (Crossover):** عملیات ترکیب برای تولید فرزندان جدید از والدین انجام می‌شود. رایج‌ترین روش‌های ترکیب در رشته‌های بیتی عبارت‌اند از:

- **ترکیب تک‌نقطه‌ای (Single-Point Crossover):** یک نقطه تصادفی در رشته والدین انتخاب شده و بخش‌های قبل و بعد از آن بین دو والد جابه‌جا می‌شوند.
  مثال:
  والد اول: ۱۱۰۰ | ۱۰۱
  والد دوم: ۰۰۱۱ | ۰۱۱
  فرزندان: ۱۱۰۰ | ۰۱۱ و ۰۰۱۱ | ۱۰۱
- **ترکیب دونقطه‌ای (Two-Point Crossover):** دونقطه تصادفی انتخاب شده و بخش میانی رشته بین دو والد جابه‌جا می‌شود.
  مثال:
  والد اول: ۰۱ | ۱۱۱۰ | ۱۰
  والد دوم: ۱۰ | ۰۰۰۱ | ۰۱
  فرزندان: ۰۱ | ۰۰۰۱ | ۱۰ و ۱۰ | ۱۱۱۰ | ۰۱
- **ترکیب یکنواخت (Uniform Crossover):** هر بیت از هر والد با احتمال ٪۵۰ برای فرزند انتخاب می‌شود.
  مثال:
  والد اول: ۱۰۱۱۰۱
  والد دوم: ۰۱۰۰۱۰
  فرزند: ۱۱۰۰۱۰

۲. **جهش (Mutation):** عملیات جهش معمولاً برای حفظ تنوع در جمعیت و جلوگیری از همگرایی زودرس انجام می‌شود. متداول‌ترین روش برای جهش در رشته‌های بیتی، معکوس کردن بیت‌ها (Bit Flip Mutation) است که با یک احتمال مشخص (معمولاً مقدار کوچکی مانند 0.01 یا 0.05)، برخی از بیت‌های رشته به مقدار مخالف خود تغییر می‌کنند. مثال:
رشته اصلی:۱۰۱۱۱۰
پس از جهش: ۱۰۰۰۱۰ (دو بیت سوم و چهارم از سمت راست تغییر کرده اند.)

(ب) زمانی که از جای‌گشت‌ها (Permutations) یا نمایش‌های غیر باینری در الگوریتم ژنتیک (GA) استفاده می‌شود، عملیات ترکیب (Crossover) و جهش (Mutation) باید به‌گونه‌ای اصلاح شوند که ویژگی‌های جای‌گشتی را حفظ کنند.

۱. **عملگر ترکیب (Crossover) برای جای‌گشت‌ها:** در جای‌گشت‌ها، هر مقدار باید یک‌بار و فقط در یک موقعیت ظاهر شود. در نتیجه، روش‌های ترکیب رشته‌های بیتی که منجر به مقادیر تکراری یا حذف‌شده می‌شوند، نامناسب‌اند. روش‌های ترکیب مناسب عبارت‌اند از:

- **ترکیب جزئی همپوشان (Partially Mapped Crossover - PMX):** دونقطه تصادفی در والدین انتخاب می‌شوند. بخش بین این دونقطه بین والدین جابه‌جا می‌شود. باقی‌مانده مقادیر با استفاده از یک نگاشت تنظیم می‌شود تا جای‌گشت معتبر بماند. مثال:
  والد ۱: ۱ ۲ ۳ | ۴ ۵ ۶ | ۷ ۸ ۹

والد ۲: ۷ ۸ ۹ | ۴ ۵ ۶ | ۱ ۲ ۳
پس از ترکیب: ۳ ۲ ۱ | ۴ ۵ ۶ | ۹ ۸ ۷ (نگاشت رعایت شده است)

- **ترکیب ترتیب یافته (Order Crossover - OX):** یک بخش تصادفی از والد اول انتخاب و در فرزند کپی می‌شود. مقادیر باقی‌مانده از والد دوم به همان ترتیب قرار می‌گیرند. مثال:
والد ۱: ۳ ۲ ۱ | ۶ ۵ ۴ | ۹ ۸ ۷
والد ۲: ۷ ۸ ۹ | ۴ ۵ ۶ | ۱ ۲ ۳
فرزند: – – – | ۶ ۵ ۴ | – – – (بخش انتخابی از والد ۱)
پس از تکمیل با ترتیب والد ۲: ۷ ۸ ۹ | ۶ ۵ ۴ | ۱ ۲ ۳
- **ترکیب چرخه‌ای (Cycle Crossover - CX):** مقادیر در چرخه‌هایی بین والدین حفظ می‌شوند تا ترتیب کلی جای‌گشت رعایت شود. مثال:
والد ۱: ۱ ۲ ۳ ۴ ۵ ۶
والد ۲: ۴ ۱ ۲ ۶ ۵ ۳
فرزند: ۱ ۲ ۳ – ۵ – (چرخه اول از والد ۱) سپس، بقیه مقادیر از والد ۲ کپی می‌شوند: ۱ ۲ ۳ ۶ ۵ ۴

۲. **عملگر جهش (Mutation) برای جای‌گشت‌ها:** در جای‌گشت‌ها، نباید تکرار یا حذف اعداد رخ دهد. پس جهش به‌صورت تغییر ترتیب مقادیر موجود انجام می‌شود. روش‌های رایج عبارت‌اند از:

- **جهش جابه‌جایی (Swap Mutation):** دو موقعیت تصادفی انتخاب شده و مقدارشان جابه‌جا می‌شود.
مثال:
قبل از جهش: ۱ ۲ ۳ ۴ ۵ ۶ (جابجایی موقعیت‌های ۲ و ۵)
بعد از جهش: ۱ ۵ ۳ ۴ ۲ ۶
- **جهش معکوس (Reverse Mutation):** یک بازه تصادفی انتخاب شده و مقادیر آن معکوس می‌شوند. مثال:
قبل از جهش: ۱ ۲ ۳ | ۴ ۵ ۶ | ۷ ۸ ۹ (معکوس کردن بازه وسط)
بعد از جهش: ۱ ۲ ۳ | ۶ ۵ ۴ | ۷ ۸ ۹
- **جهش درج (Insertion Mutation):** یک مقدار از موقعیت تصادفی حذف شده و در موقعیت دیگری درج می‌شود. مثال:
قبل از جهش: ۱ ۲ ۳ ۴ ۵ ۶ (انتقال عدد ۳ بعد از ۵)
بعد از جهش: ۱ ۲ ۴ ۵ ۳ ۶

۴. در هنگام استفاده از الگوریتم ژنتیک و ساخت جمعیت اولیه اندازه کروموزوم و تعداد ژن‌های آن ثابت و وابسته به پارامترهای مسئله است. در پایان الگوریتم نیز اندازه کروموزوم و تعداد ژن آن باید با شروع کار الگوریتم برابر و ثابت مانده باشد. در اجرای یک الگوریتم ژنتیک (GA) روی رشته‌های بیتی، چندین خاصیت تغییرناپذیری (Invariance Properties) وجود دارند که باید حفظ شوند. این ویژگی‌ها تضمین می‌کنند که اولاً الگوریتم به طور مؤثر جستجو کرده است و ثانیاً عملکرد پایدار و قابل‌پیش‌بینی خواهد داشت. در ادامه، این خصوصیات و تأثیر آن‌ها بر کارایی الگوریتم بررسی می‌شود:

- **تغییرناپذیری تحت جای‌گشت بیت‌ها (Bit Permutation Invariance):** اگر یک مسئله با جای‌گشت موقعیت‌های بیت‌ها تغییری نکند، پس اجرای الگوریتم ژنتیک نیز نباید وابسته به ترتیب بیت‌ها باشد.

به‌عنوان‌مثال: اگر دو رشته بیتی ۱۱۰۰ و ۰۰۱۱ دارای مقدار برازش (یا مقدار تناسب) یکسانی باشند،

الگوریتم نباید به ترتیب بیت‌ها حساس باشد، بلکه باید فقط الگوی کلی ژن‌ها را مدنظر قرار دهد. این ویژگی کمک می‌کند تا الگوریتم روی تمام فضای جستجو به‌طور یکنواخت عمل کند و وابسته به نحوه‌ی کدگذاری مسئله نباشد. اگر این خاصیت حفظ نشود، ممکن است الگوریتم به برخی ساختارهای خاص در رشته‌های بیتی تمایل پیدا کند و جستجو مناسب نباشد.

- **تغییرناپذیری تحت مقداردهی اولیه تصادفی (Initial Population Invariance):** نتیجه‌ی الگوریتم نباید به توزیع خاصی از مقداردهی اولیه وابسته باشد، بلکه باید همواره به یک جواب بهینه یا نزدیک بهینه همگرا شود.

  به عنوان مثال: دو اجرای الگوریتم ژنتیک با مقداردهی اولیه‌ی تصادفی مختلف نباید نتایج کاملاً متفاوتی داشته باشند، بلکه باید به نقاط بهینه مشابهی همگرا شوند. با این اتفاق قابلیت تکرارپذیری (Reproducibility) افزایش می‌یابد. اگر این خاصیت رعایت نشود، الگوریتم ممکن است به پاسخ‌های مختلف و ناسازگار در اجراهای مختلف برسد.

- **تغییرناپذیری تحت تغییر مقیاس تابع برازش (Fitness Scale Invariance):** عملکرد الگوریتم نباید به تحولات خطی در تابع برازش حساس باشد.

  به‌عنوان‌مثال: اگر تابع برازش f(x) را به صورت $g(x) = af(x) + b$ که در آن$0 < a$ و b یک مقدار ثابت است) تغییر دهیم، نباید رفتار الگوریتم عوض شود. اگر این ویژگی حفظ نشود، مقادیر برازش بزرگ یا کوچک می‌توانند روی نحوه‌ی انتخاب و عملکرد اپراتورهای ژنتیکی تأثیر منفی بگذارند. معمولاً نرمال‌سازی تابع برازش می‌تواند این مشکل را برطرف کند.

- **تغییرناپذیری تحت بازآرایی کروموزوم‌ها (Chromosome Encoding Invariance):** نتایج الگوریتم نباید به نوع کدگذاری ژن‌ها وابسته باشد.

  به‌عنوان‌مثال: اگر برای حل یک مسئله می‌توان از دو نوع کدگذاری باینری ۰۰۰۱ و ۱۱۱۰ (مثلاً با تعریف‌های متفاوت از بیت‌ها) استفاده کرد، عملکرد الگوریتم نباید وابسته به این انتخاب باشد. همچنین بهینه‌سازی نباید محدود به نوع خاصی از نمایش کروموزومی باشد. با این کار افزایش انعطاف‌پذیری الگوریتم ژنتیک برای حل مسائل مختلف رخ میدهد.

- **تغییرناپذیری تحت تغییرات جمعیت (Population Size Invariance:)** عملکرد کلی الگوریتم باید با تغییر اندازه‌ی جمعیت تغییر محسوسی نداشته باشد (تا حدی معقول).

  به‌عنوان‌مثال: اگر یک الگوریتم با اندازه‌ی جمعیت ۵۰ و یک اجرای دیگر با اندازه‌ی ۱۰۰ انجام شود، نتایج نباید کاملاً متفاوت باشند، بلکه باید هر دو به یک جواب بهینه مشابه همگرا شوند. پایداری الگوریتم در برابر تنظیمات مختلف و کاهش حساسیت الگوریتم به هایپرپارامترهای تنظیمی از این جهت است.

- **تغییرناپذیری نسبت به معکوس کردن بیت‌ها (Bit Flip Invariance):** معکوس کردن مقدار بیت‌ها (تبدیل ۰ به ۱ و برعکس) تأثیری بر عملکرد الگوریتم نباید داشته باشد. اگر این خاصیت حفظ نشود، الگوریتم ممکن است در جهت‌های نامناسب جستجو کند و زمان بیشتری برای همگرایی نیاز داشته باشد.

- **تغییرناپذیری نسبت به تغییر مقیاس (Scale Invariance):** تغییر مقیاس در تابع برازش (Fitness Function) نباید تأثیری بر رفتار الگوریتم داشته باشد.

  به‌عنوان‌مثال: اگر تابع برازش را در یک مسئله بهینه‌سازی در مقیاس‌های مختلف (مثلاً ضرب در یک ثابت) تغییر دهیم، الگوریتم باید همچنان به سمت جواب بهینه همگرا شود.

- **تغییرناپذیری نسبت به چرخش (Rotation Invariance):** چرخش در فضای جستجو (مانند تغییر جهت‌های جستجو) نباید تأثیری بر عملکرد الگوریتم داشته باشد. اگر این خاصیت حفظ نشود، الگوریتم ممکن است در جهت‌های خاصی از فضای جستجو گیر کند و نتواند به طور کامل فضای جستجو را بررسی کند.
- **تغییرناپذیری نسبت به انتقال (Translation Invariance):** انتقال در فضای جستجو (مانند جابه‌جایی نقطه شروع) تأثیری بر عملکرد الگوریتم نباید داشته باشد. حفظ این خاصیت باعث می‌شود الگوریتم به نقطه شروع اولیه حساس نباشد و عملکرد پایدارتری داشته باشد.

  به‌عنوان‌مثال: در مسئله‌ای مانند بهینه‌سازی تابع Sphere، انتقال نقطه شروع نباید بر نتیجه نهایی تأثیر بگذارد.
- **تغییرناپذیری نسبت به تغییرات کوچک در جمعیت (Population Perturbation Invariance):** تغییرات کوچک در جمعیت اولیه (مانند جهش‌های کوچک) نباید تأثیری بر رفتار کلی الگوریتم نداشته باشد. اگر این خاصیت حفظ نشود، الگوریتم ممکن است به دلیل تغییرات کوچک در جمعیت اولیه، به سمت جواب‌های زیر بهینه همگرا شود.

  به‌عنوان‌مثال: در مسئله‌ای مانند مسئله کوله‌پشتی، تغییرات کوچک در جمعیت اولیه نباید منجر به نتایج کاملاً متفاوت شود.
- **تغییرناپذیری نسبت به انتخاب عملگرها (Operator Invariance):** انتخاب عملگرهای مختلف (مانند جهش، ترکیب، و انتخاب) نباید تأثیری بر رفتار کلی الگوریتم داشته باشد. حفظ این خاصیت باعث می‌شود الگوریتم به انتخاب عملگرها حساس نباشد و انعطاف‌پذیری بیشتری داشته باشد.

  به‌عنوان‌مثال: در مسئله‌ای مانند فروشنده دوره‌گرد، استفاده از عملگرهای مختلف نباید منجر به نتایج کاملاً متفاوت شود.

۵. (آ) برای مقایسه زمان مورد انتظار الگوریتم ژنتیک و جستجوی تصادفی در یافتن جواب بهینه برای رشته‌های بیتی به طول n داریم:

۱. **جستجوی تصادفی (با توزیع یکنواخت):**
- هر رشته به‌صورت تصادفی و مستقل از موارد قبلی انتخاب می‌شود.
- چون فضای جستجو شامل 2^n رشته ممکن است، احتمال یافتن جواب بهینه در یک مرحله برابراست با $\frac{1}{2^n}$ خواهد بود.
- تعداد مراحل مورد انتظار برای یافتن جواب بهینه در جستجوی تصادفی برابر است با $\mathrm{O}(2^n)$ زیرا این یک فرآیند برنولی است که در آن موفقیت (یافتن جواب بهینه) با احتمال $\frac{1}{2^n}$ رخ می‌دهد.

۲. **الگوریتم ژنتیک:**
- طبق فرض مسئله، این روش به طور متوسط در $\mathrm{O}(3^n)$ مرحله جواب بهینه را پیدا می‌کند.

مقایسه:
- جستجوی تصادفی $\mathrm{O}(2^n)$
- الگوریتم ژنتیک $\mathrm{O}(3^n)$

ازآنجایی‌که $2^n < 3^n$ برای همه $n > 0$ ، الگوریتم ژنتیک موردنظر از جستجوی تصادفی کندتر است و انتظار می‌رود که زمان بیشتری برای یافتن جواب بهینه صرف کند. این نشان می‌دهد که این الگوریتم ژنتیک کارایی بهتری نسبت به جستجوی تصادفی ندارد و حتی ممکن است در عمل ضعیف‌تر عمل کند.

در الگوریتم‌های ژنتیک کارآمد، انتظار داریم که روش‌های انتخاب، جهش و تقاطع باعث بهبود نرخ همگرایی شوند و زمان مورد انتظار را کمتر از جستجوی تصادفی نگه دارند، اما در اینجا عملکرد بدتری مشاهده می‌شود که احتمالاً به دلیل طراحی نامناسب اپراتورهای ژنتیکی است.

(ب) برخی عوامل مؤثر بر عملکرد الگوریتم ژنتیک در عمل:

۱. **انتخاب پارامترهای الگوریتم:**

- **اندازه جمعیت (Population Size):** جمعیت بزرگ‌تر ممکن است به تنوع بیشتر و پوشش بهتر فضای جستجو منجر شود، اما هزینه محاسباتی بیشتری دارد.
- **نرخ جهش (Mutation Rate):** نرخ جهش بالا می‌تواند به اکتشاف بیشتر فضای جستجو کمک کند، اما ممکن است باعث ازدست‌رفتن جواب‌های خوب شود. نرخ جهش پایین ممکن است باعث گیرکردن الگوریتم در بهینه‌های محلی شود.
- **نرخ جهش (Mutation Rate):** نرخ جهش بالا می‌تواند به اکتشاف بیشتر فضای جستجو کمک کند، اما ممکن است باعث ازدست‌رفتن جواب‌های خوب شود. نرخ جهش پایین ممکن است باعث گیرکردن الگوریتم در بهینه‌های محلی شود.
- **نرخ ترکیب (Crossover Rate):** نرخ ترکیب بالا می‌تواند به‌اشتراک‌گذاری اطلاعات بین افراد جمعیت کمک کند، اما ممکن است باعث ازدست‌رفتن تنوع شود.
- **تعداد نسل‌ها (Number of Generations):** تعداد نسل‌های بیشتر ممکن است به یافتن جواب بهینه نزدیک‌تر کمک کند، اما زمان اجرا را افزایش می‌دهد.

۲. **:طراحی عملگرهای الگوریتم**

- **عملگر ترکیب (Crossover Operator):** استفاده درست و بجا از این عملگر می‌تواند به بهبود همگرایی الگوریتم کمک کند.
- **عملگر جهش (Mutation Operator):** عملگر جهش باید به‌گونه‌ای طراحی شود که تعادل مناسبی بین اکتشاف (Exploration) و بهره‌برداری (Exploitation) ایجاد کند.
- **عملگر انتخاب (Selection Operator):** روش‌های انتخاب مانند انتخاب چرخ رولت، انتخاب رقابتی یا انتخاب بر اساس رتبه می‌توانند بر سرعت همگرایی و تنوع جمعیت تأثیر بگذارند.

۳. **بازنمایی مسئله (Problem Representation):**

- **کدگذاری جواب‌ها (Encoding):** روش کدگذاری جواب‌ها (مثلاً رشته‌های بیتی، درخت‌ها، یا بردارهای واقعی) باید به‌گونه‌ای باشد که فضای جستجو را به طور مؤثر پوشش دهد.
- **تناسب بازنمایی با عملگرها:** عملگرهای ترکیب و جهش باید با روش کدگذاری جواب‌ها سازگار باشند تا بتوانند جواب‌های معتبر تولید کنند.

۴. **تابع برازش (Fitness Function):**

- **طراحی تابع برازش:** : تابع برازش باید به‌گونه‌ای طراحی شود که بتواند به طور دقیق کیفیت جواب‌ها را ارزیابی کند.
- **مقیاس تابع برازش:** اگر مقیاس تابع برازش نامناسب باشد (مثلاً تفاوت‌های بسیار کوچک یا بسیار بزرگ بین جواب‌ها)، ممکن است الگوریتم در بهینه‌های محلی گیر کند.

۵. **مشخصات مسئله (Problem Characteristics):**

- **فضای جستجو (Search Space):** اگر فضای جستجو بسیار بزرگ یا پیچیده باشد، الگوریتم ممکن است به زمان بیشتری برای یافتن جواب بهینه نیاز داشته باشد.
- **تعداد بهینه‌های محلی (Local Optimal):** اگر مسئله دارای تعداد زیادی بهینه محلی باشد، الگوریتم ممکن است در یکی از آنها گیر کند.
- **همبستگی بین متغیرها (Variable Correlation):** اگر متغیرهای مسئله به‌شدت به‌هم‌وابسته باشند، الگوریتم ممکن است در یافتن جواب بهینه با مشکل مواجه شود.

۶. **:پیاده‌سازی و محاسبات**

- **کارایی پیاده‌سازی:** پیاده‌سازی بهینه الگوریتم می‌تواند زمان اجرا را کاهش دهد.
- **موازی‌سازی (Parallelization):** استفاده از روش‌های موازی‌سازی می‌تواند سرعت اجرای الگوریتم را افزایش دهد.
- **دقت محاسباتی:** دقت محاسباتی می‌تواند بر نتایج الگوریتم تأثیر بگذارد، به‌ویژه در مسائل با اعداد حقیقی.

۷. **شرایط اولیه (Initial Conditions):**

- **مقداردهی اولیه جمعیت (Initial Population):** اگر جمعیت اولیه بهینه‌های محلی را پوشش ندهد، الگوریتم ممکن است در یافتن جواب بهینه با مشکل مواجه شود.
- **تصادفی بودن (Randomness):** الگوریتم‌های ژنتیک به دلیل ماهیت تصادفی خود ممکن است در اجراهای مختلف نتایج متفاوتی تولید کنند.

۸. **تعادل بین اکتشاف و بهره‌برداری (Exploration vs. Exploitation):**

- **اکتشاف (Exploration):** جستجو در مناطق جدید فضای جستجو برای یافتن جواب‌های بهتر.
- **بهره‌برداری (Exploitation):** بهبود جواب‌های موجود در مناطق شناخته‌شده.
- اگر الگوریتم بیش از حد بر اکتشاف تمرکز کند، ممکن است زمان زیادی صرف جستجو در مناطق بی‌کیفیت شود .
- اگر الگوریتم بیش از حد بر بهره‌برداری تمرکز کند، ممکن است در بهینه‌های محلی گیر کند.

۹. **شرایط توقف (Stopping Criteria):**

- **تعداد نسل‌ها:** اگر تعداد نسل‌ها کم باشد، الگوریتم ممکن است به جواب بهینه نرسد.
- **معیار همگرایی (Convergence Criterion):** اگر معیار همگرایی به‌درستی تنظیم نشود، الگوریتم ممکن است زودتر از موعد متوقف شود یا زمان زیادی را هدر دهد.

۱۰. **نویز و عدم قطعیت (Noise and Uncertainty):**

- اگر تابع برازش نویزی باشد یا داده‌ها دارای عدم قطعیت باشند، الگوریتم ممکن است در ارزیابی جواب‌ها با مشکل مواجه شود.

۲ درک و حل مسائل با الگوریتم ژنتیک

۱. (آ) اگر هیچ گره‌ای نباید دو بار دیده شود، یک کروموزوم باید باید یک دور بین همه‌ی گره‌ها باشد که این دور شامل طی ترتیب طی کردن ۱۰ گره یا معادل طی کردن ۱۰ یال است. پس کروموزوم ما شامل ۱۰ ژن است.

(ب) اینکه بین کدام شهرها ارتباط وجود داشته باشد پیش‌فرض‌های مسئله است اما به طور کلی می‌توان گفت اگر گراف کامل و بدون طوقه باشد، از هر گره‌ای به همه‌ی گره‌های دیگر یال وجود دارد. ما در نظر گرفتیم این یال‌ها جهت‌دار است پس اگر از یک گره به گره‌ی دیگر رفت برگشت نیازی نیست. با این اوصاف تعداد کل ژن‌های ممکن $n \times \frac{n-1}{2}$ یال می‌شود. که اینجا $n = 10$ است پس $10 \times \frac{9}{2} = 45$ ژن وجود دارد.

۲. (آ) ژن‌ها را به تابع fitness می‌بریم:

$$\text{fit}(x_1) = 6 + 5 - 4 - 1 + 3 + 5 - 3 - 2 = 9$$

$$\text{fit}(x_2) = 8 + 7 - 1 - 2 + 6 + 6 - 0 - 1 = 23$$

$$\text{fit}(x_3) = 2 + 3 - 9 - 2 + 1 + 2 - 8 - 5 = -16$$

$$\text{fit}(x_4) = 4 + 1 - 8 - 5 + 2 + 0 - 9 - 4 = -19$$

به ترتیب x_2، x_1، x_3 و x_4 برازنده هستند.

(ب) عملیات ترکیب

- ترکیب نقطه‌ای: در این روش به دو فرزند جدید می‌رسیم.

$$x_{21} = 8712|3532$$

$$x_{21} = 6541|6601$$

- ترکیب دو نقطه‌ای: با استفاده از این روش به دو فرزند جدید می‌رسیم. ما فرض می‌کنیم منظور از نقاط b و f یعنی بعد از این نقاط ترکیب اتفاق می‌افتد

$$x_{131} = 65|9212|35$$

$$x_{313} = 23|4135|85$$

- ترکیب یکنواخت: برای انجام این ترکیب نیازمند به یک ماسک هستیم. این ماسک یک ژن تصادفی با مقادیر دودویی است که نشانگر این است که آن ژن را از کروموزوم اول بگیریم یا دوم. که انتخاب اول یا دوم هم احتمال است. ما با استفاده از

برنامهٔ ۱: تولید ماسک تصادفی

```
import random
mask = ''.join(random.choice('01') for _ in range(8))
print(mask)
```

یک رشته‌ی تصادفی از ۰ و ۱ تولید می‌کنیم. ما فرض می‌کنیم ۰ معادل رشته‌ی اول و ۱ معادل رشته‌ی سوم باشد.

$$\text{mask} = 01001010$$

$$x_{13} = 8|3|12|1|6|8|1$$

$$x_{31} = 2|7|92|6|2|0|5$$

(ج) برازش فرزندان: با استفاده از تکه کد زیر برازندگی هر فرزند را محاسبه می‌کنیم:

برنامهٔ ۲: محاسبه‌ی برازندگی

```
chromosome = input()
a, b, c, d, e, f, g, h = [int(char) for char in chromosome]
fitness = a + b - c - d + e + f - g - h
print(fitness)
```

$$\text{fit}(x_{21}) = 87123532 = 15$$

$$\text{fit}(x_{21}) = 65416601 = 17$$

$$\text{fit}(x_{131}) = 65921235 = -5$$

$$\text{fit}(x_{313}) = 23413585 = -5$$

$$\text{fit}(x_{23}) = 83121681 = 6$$

$$\text{fit}(x_{32}) = 27926205 = 1$$

تعبیر بهتر شدن و بدتر شدن تعبیر نا دقیقی است. ما دو شاخص را برای بهتر شدن و بدتر شدن در نظر می‌گیریم.

۱. بالاترین برازندگی: در والدها بالاترین برازندگی ۲۳ بود که به ۱۷ کاهش یافت یعنی بدتر شده.

۲. میانگین برازندگی: در شرایط قبلی برازندگی معادل $-0.75 = \frac{-3}{4} = \frac{9+23-16-19}{4}$ می‌شود و در فرزندان $\frac{15+17-5-5+6+1}{6} = \frac{29}{6} \approx 4.83$ می‌شود که رشد قابل توجهی است.

(د) برای بیشینه کردن برازندگی، ژن‌های a، b، e و f باید مقدار ۹ داشته باشند و c، d، g و h باید مقدار ۰ را داشته باشند. برازندگی بهینه برابر $4 \times 9 - 0 = 36$ می‌شود.

(ه) ما سعی کردیم بهترین ترکیب را بسیازیم و آن $x_{\text{optimal}} = 87116601$ خواهد بود که برازندگی آن ۲۴ خواهد شد. پس نمی‌توان بدون جهش به نقطه‌ی بهینه رسید و حداقل ۱۲ تا فاصله با نقطه‌ی برازندگی وجود خواهد داشت.

۳. kjhghghghghg

(آ) مقدار برازندگی به ازای هر x:

$$\text{fit}(x_1) = 1 - 4 + 7 = 4$$

$$\text{fit}(x_2) = 8 - 16 + 7 = -1$$

$$\text{fit}(x_3) = 27 - 36 + 7 = -2$$

$$\text{fit}(x_4) = 64 - 64 + 7 = 7$$

(ب) بله. می‌توانیم با اضافه کردن $\forall c : c \geq 2$ همه‌ی مقدارها را نامنفی کنیم. مثلا اگر $c = 3$ در نظر بگیریم رابطه‌ی برازندگی $\text{fit}(x) = x^3 - 4x^2 + 10$ خواهد شد.

(ج) به هر برازندگی مقدار ثابت ۲ اضافه می‌شود پس

$$\text{TotalFitness} = (4+3) \times 2 + (-1+3) \times 3 + (-2+3) \times 3 + (7+3) \times 2$$

$$= 14 + 6 + 3 + 20 = 43$$

(د) مقدار برازندگی نسبی برای هر نمونه‌ی x به صورت زیر خواهد شد:

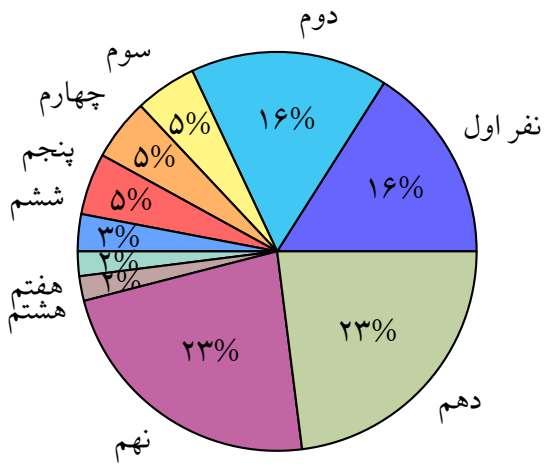
$$P(x=1) = \frac{7}{43} = 0.1628$$

$$P(x=2) = \frac{2}{43} = 0.0465$$

$$P(x=3) = \frac{1}{43} = 0.0233$$

$$P(x=4) = \frac{10}{43} = 0.2326$$

می‌توانیم آن را به صورت یک گردونه هم نشان دهیم



شکل ۱: گردونه‌ی شانس برای این نمونه از جمعیت

(ه) مزیت تابع جدید این است که به ازای هر مقدار x، تابع برازندگی همواره نامنفی است. برای محاسبه‌ی $g(x)$ تمام

مقدادیر بدست آمده در بخش آ را به توان ۲ می‌رسانیم.

$$\text{fit}(x_1) = 4^2 = 16$$

$$\text{fit}(x_2) = (-1)^2 = 1$$

$$\text{fit}(x_3) = (-2)^2 = 4$$

$$\text{fit}(x_4) = 7^2 = 49$$

(و)
- **فشار انتخاب**: فشار انتخاب یعنی درجه‌ی اینکه افراد برازنده‌تر چقدر شانس زنده ماندن دارند. برعکس اضافه کردن مقدار ثابت، در «به توان رساندن» فشار انتخاب زیاد می‌شود. البته این تا حدودی بستگی به روش انتخاب هم دارد. مثلا اگر از الگوریتم انتخاب رتبه پایه [۱] استفاده کنیم دیگر این مسئله جدی نیست.
- **همگرایی**: می‌توان گفت با افزایش فشار انتخاب، همگرایی سریع‌تر می‌شود اما خطر گیر کردن در یک نقطه‌ی بهینه‌ی محلی وجود دارد.
- **تنوع:** افزایش فشار انتخاب باعث کاهش تنوع و همگرایی سریع به یک قله‌ی محلی خاص می‌شود که ممکن است بهترین نباشد. با افزایش فشار انتخاب تنوع در انتخاب گونه‌ها را از دست می‌دهیم.

[۱] rank-based selection

۳ پیاده‌سازی، ارزیابی و تجزیه و تحلیل الگوریتم ژنتیک جهت انتخاب بهترین ویژگی برای مسئله‌ی واقعی دسته‌بندی مشتریان

۱. پیش پردازش داده‌ها:

(آ) حذف دادههای پرت:

برای پر کردن دادههای پرت از روش IQR method استفاده میکنیم این روش به این صورت است که IQR را برابر با Q3 - Q1 قرار میدهیم(چارک اول : Q1، چارک سوم: Q3) سپس دادههای کوچکتر از Q1 - 1.5*IQR و بزرگتر از Q3 + 1.5IQR را حذف میکنیم.

برنامهٔ ۳: حذف دادههای پرت

```
df = pd.read_csv('Customer Classification dataset/Train.csv')

numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()

numeric_cols.remove('ID')

for col in numeric_cols:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]
```

(ب) رمزگذاری ویژگیهای دسته ای (categorical):

مهم ترین پارامتر در رمزگذاری (encoding) داده های دسته ای این است که ببینیم که این داده ها داده های کیفی ترتیبی (Ordinal Qualitative) هستند یا کیفی اسمی (Nominal Qualitative) اگر داده ها کیفی ترتیبی باشند نیاز داریم که داده ها را به گونه ای پیش پردازش بکنیم که این ترتیب همچنان حفظ شود و در صورتی که داده ها اسمی باشند نیازی به این کار نداریم و میتوانیم برای هر دسته یک ستون درست کنیم که با مقادیر درست (True) و غلط (False) مشخص کنیم که به این دسته قرار دارد یا خیر.

در داده هایی که در سوال به ما داده شده بود ستون های Gender، Ever_Married، Graduated، Profession و Var_1 ستونهایی بودند که داده های انها به صورت کیفی اسمی بود و برای همین برای رمز گذاری انها از get_dummies استفاده کردیم.

برنامهٔ ۴: رمزگذاری با get_dummies

```
columns_to_encode = ['Gender', 'Ever_Married', 'Graduated', 'Profession',
    'Var_1']

features = pd.get_dummies(
    features,
    columns=columns_to_encode,
```

```
    prefix=columns_to_encode,
    drop_first=True
)
features.head()
```

و ستون Spending_Score را که دارای داده های ترتیبی به نام های Low، Average و High بود را به صورت دیگری رمزگذاری کردیم تا مدل توانایی درک این که این ۳ مقدار دارای ترتیب مشخصی هستند را متوجه شود. اینکار را به این صورت انجام دادیم که این ۳ ستون را به ترتیب از کوچک به بزرگ از ۰ تا ۲ مقداردهی کردیم.

برنامهٔ ۵: رمزگذاری برای ستون spending_score

```
score_mapping = {
    'Low': 0,
    'Average': 1,
    'High': 2
}
features['Spending_Score'] = features['Spending_Score'].map(score_mapping)
features.head()
```

(ج) پر کردن داده های خالی (Nan) :

برای پرکردن داده های خالی از KNNImputer در کتابخانه sikitlearn استفاده کردیم به این صورت که n_neighbors که یکی از پارامترهای این تابع است را ۸ انتخاب کردیم و این به این معناست که هر ردیفی که مقدار خالی داشته باشد میاید و ۸ ردیف نزدیک به ان را پیدا میکند و سپس از مقادیر ستون مورد نظر(یعنی ستونی که مقدار خالی در ان قرار دارد.) در ان ردیفها میانگین گرفته و انرا به عنوان مقدار جدید سلول خالی قرار میدهد.

برنامهٔ ۶: پر کردن سلول‌های خالی

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=8)
imputed_X = imputer.fit_transform(features)
features = pd.DataFrame(imputed_X, columns=features.columns)
```

۱. پیاده سازی الگوریتم ژنتیک:

الگوریتم ژنتیک یک روش جستجو و بهینه‌سازی مبتنی بر تکامل طبیعی است که در آن مفاهیمی از ژنتیک مانند انتخاب طبیعی، ترکیب و جهش برای یافتن بهترین جواب به کار گرفته می‌شوند. این الگوریتم در مسائل مختلف از جمله بهینه‌سازی، یادگیری ماشین و مسائل ترکیبیاتی مورد استفاده قرار می‌گیرد.

مراحل الگوریتم ژنتیک:

(آ) مقداردهی اولیه (Initialization) :

مجموعه‌ای از کروموزوم‌ها (یا راه‌حل‌های ممکن) به‌صورت تصادفی ایجاد می‌شود.

برنامهٔ ۷: مقداردهی اولیه

```
def initialize(count, column_count):
    zero_array = np.zeros((1, column_count), dtype=int)
    my_set = set()

    while len(my_set) != count:
        random_numbers = random.sample(range(0, 21), column_count)
        copied_zero_array = zero_array.copy()
        for i in range(column_count):
            copied_zero_array[0][i] = random_numbers[i]
        my_set.add(tuple(copied_zero_array[0]))
    my_list = list(my_set)
    return my_set
```

(ب) ارزیابی برازندگی (Fitness Evaluation):

هر کروموزوم بر اساس یک تابع برازندگی ارزیابی می‌شود تا میزان تطابق آن با هدف مشخص شود. در کدی که ما زدیم تابع برازندگی، accuracy_score مدل است.

(ج) انتخاب (Selection):

کروموزوم‌های بهتر شانس بیشتری برای انتخاب شدن و انتقال به نسل بعدی دارند. روش‌های مختلفی برای این کار وجود دارد، از جمله:

چرخ رولت (Roulette Wheel Selection)

انتخاب بر اساس رتبه (Rank-Based Selection)

انتخاب تورنمنت (Tournament Selection)

که ما هر ۳تای انها را پیاده سازی کردیم.

برنامهٔ ۸: roulette-wheel-selection

```
def roulette_wheel_selection(initialize_gen, fitness_list, repeatable,
    count):
    indices = list(range(len(fitness_list)))
    if repeatable:
        #repeatable
        selected_index = random.choices(indices, weights=fitness_list,
    k=count)
    else:
        #non-repeatable
        selected_index = np.random.choice(indices, size=count,
    replace=False, p=fitness_list/sum(fitness_list))
    selected_ones = list()
    initialize_gen = list(initialize_gen)
    for i in selected_index:
        selected_ones.append(initialize_gen[i])
    return selected_ones
```

برنامهٔ ۹: rank-based-selection

```
def rank_based_selection(initialize_gen, fitness_list, repeatable, count):
    fitness_list = get_rank_array(list(fitness_list))

    indices = list(range(len(fitness_list)))
    if repeatable:
        #repeatable
        selected_index = random.choices(indices, weights=fitness_list, k=count)
    else:
        #non-repeatable
        selected_index = np.random.choice(indices, size=count, replace=False, p=fitness_list/sum(fitness_list))
    selected_ones = list()
    initialize_gen = list(initialize_gen)
    for i in selected_index:
        selected_ones.append(initialize_gen[i])
    return selected_ones
```

برنامهٔ ۱۰: tournament-selection

```
def tournament_selection(initial_gen, gen_fitnesss, replacement, count, sample=2):
    selected_population = []

    if replacement:
        for _ in range(count):
            tournament_indices = random.choices(range(len(initial_gen)), k=sample)
            winner_index = max(tournament_indices, key=lambda i: gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])
    else:
        not_used = list(range(len(initial_gen)))
        thrown = []

        while len(not_used) >= sample and len(selected_population) < count:
            tournament_indices = random.sample(not_used, sample)
            winner_index = max(tournament_indices, key=lambda i: gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])
            for idx in tournament_indices:
                not_used.remove(idx)
                if idx != winner_index:
                    thrown.append(idx)
```

```
        pool = thrown + not_used
        while len(selected_population) < count and pool:
            if len(pool) >= sample:
                tournament_indices = random.sample(pool, sample)
            else:
                tournament_indices = random.choices(pool, k=sample)
            winner_index = max(tournament_indices, key=lambda i:
    gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])

    return selected_population
```

(د) ترکیب (Crossover):

کروموزوم‌های انتخاب شده با یکدیگر ترکیب می‌شوند تا فرزندان جدید تولید شود. روش‌های متداول شامل:

ترکیب تک‌نقطه‌ای (Single-Point Crossover)

ترکیب دونقطه‌ای (Two-Point Crossover)

ترکیب یکنواخت (Uniform Crossover)

که ما هر ۳تای اینها را هم پیاده سازی کردیم.

برنامهٔ ۱۱: single-point-crossover

```
def single_point_crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            set1 = set()
            set2 = set()
            p1, p2 = parents[i], parents[i + 1]
            child_size = len(p1)
            cross_point = random.randint(1, len(p1) - 1)
            make1 = p1[:cross_point] + p2[cross_point:] + p1[cross_point:]
    + p2[:cross_point]
            make2 = p2[:cross_point] + p1[cross_point:] + p2[cross_point:]
    + p1[:cross_point]
            for j in make1:
                if len(set1) == child_size:
                    break
                set1.add(j)
            for j in make2:
                if len(set2) == child_size:
                    break
                set2.add(j)
            child1 = list(set1)
            child2 = list(set2)
```

```
            offspring.extend([child1, child2])
    return offspring
```

برنامهٔ ۱۲: two-point-crossover

```
def two_point_crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            set1 = set()
            set2 = set()
            p1, p2 = parents[i], parents[i + 1]
            child_size = len(p1)
            point1 = random.randint(1, len(p1) - 2)
            point2 = random.randint(point1 + 1, len(p1) - 1)
            make1 = p1[:point1] + p2[point1:point2] + p1[point2:] +
    p2[:point1] + p1[point1:point2] + p2[point2:]
            make2 = p2[:point1] + p1[point1:point2] + p2[point2:] +
    p1[:point1] + p2[point1:point2] + p1[point2:]
            for j in make1:
                if len(set1) == child_size:
                    break
                set1.add(j)
            for j in make2:
                if len(set2) == child_size:
                    break
                set2.add(j)
            child1 = list(set1)
            child2 = list(set2)
            offspring.extend([child1, child2])
    return offspring
```

برنامهٔ ۱۳: uniform-crossover

```
def uniform_crossover(parents, swap_prob=0.5):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            p1, p2 = parents[i], parents[i + 1]
            child_size = len(p1)
            make1 = []
            make2 = []
            for g1, g2 in zip(p1, p2):
                if random.random() < swap_prob:
                    make1.append(g2)
                    make2.append(g1)
```

```
                else:
                    make1.append(g1)
                    make2.append(g2)
            make1 += p1 + p2
            make2 += p1 + p2
            set1 = set()
            set2 = set()
            for gene in make1:
                if len(set1) == child_size:
                    break
                set1.add(gene)
            for gene in make2:
                if len(set2) == child_size:
                    break
                set2.add(gene)
            offspring.append(list(set1))
            offspring.append(list(set2))
    return offspring
```

(ه) جهش (Mutation):

برخی از کروموزوم‌ها دچار تغییرات جزئی می‌شوند تا از همگرایی زودرس جلوگیری شود و تنوع حفظ گردد.

برنامهٔ ۱۴: mutate

```
def mutate(individual, max_value):
    index = random.randint(0, len(individual) - 1)
    new_value = random.randint(1, max_value)
    while new_value in individual:
        new_value = random.randint(1, max_value)
    individual[index] = new_value
    return individual
```

(و) تکرار (Iteration):

همه‌ی مراحل الف تا ه را دوباره انجام میدهیم تا به یکی از دو شرط زیر برسیم:

رسیدن به حداکثر تعداد تکرار (iterations)

رسیدن به نقطه‌ی همگرایی(عدم بهبود برای مدت طولانی)

حالا با استفاده از الگوریتم‌هایی که در بالا توضیح داده و پیاده سازی کردیم. الگوریتم ژنتیک را پیاده‌سازی میکنیم.

برنامهٔ ۱۵: Algorithm Genetic

```
def GA(X, y, count, column_count, repeatable, select_count,
    selection_type='roulette_wheel_selection', crossover='single_point',
    max_generations=100, target_accuracy=0.95, max_stagnation=50):
     initial_gen = initialize(count, column_count)
     generation = 0
     best_fitness = 0
     stagnation_counter = 0  # Track generations without improvement
     max_fitness_list = list()
     while generation < max_generations and best_fitness < target_accuracy:
         fitness_list = list()

         for i in range(len(initial_gen)):
             fitness_list.append(get_fitness(initial_gen, X, i, y))

         current_max = max(fitness_list)
         previous_best = best_fitness
         best_fitness = max(best_fitness, current_max)

         if best_fitness == previous_best:
             stagnation_counter += 1
         else:
             stagnation_counter = 0

         max_fitness_list.append(current_max)

         if stagnation_counter >= max_stagnation:
             break

         if selection_type == 'roulette_wheel_selection':
             selected_ones = roulette_wheel_selection(initial_gen,
    fitness_list, repeatable, select_count)
         elif selection_type == 'rank_based_selection':
             selected_ones = rank_based_selection(initial_gen,
    fitness_list, repeatable, select_count)
         elif selection_type == 'tournament_selection':
             selected_ones = tournament_selection(initial_gen,
    fitness_list, repeatable, select_count)
         else:
             raise ValueError('Invalid selection')

         if crossover == 'single_point':
             offspring = single_point_crossover(selected_ones)
```

```
        elif crossover == 'two_point':
            offspring = two_point_crossover(selected_ones)
        elif crossover == 'uniform':
            offspring = uniform_crossover(selected_ones)
        else:
            raise ValueError('Invalid crossover')

        mutated_offspring = [mutate(list(child), column_count * 2) for child in offspring]
        initial_gen = mutated_offspring
        generation += 1

    plt.plot(max_fitness_list)
    plt.xlabel('Generation')
    plt.ylabel('Best Fitness')
    plt.title('Best Fitness per Generation')
    plt.show()

    return mutated_offspring
```