به نام خداوند بخشنده مهربان

سیستم‌های بی‌درنگ

دکتر صفری

مبینا حیدری - نیکا قادری
بهار ۱۴۰۴

تاریخ گزارش: ۱۲ تیر ۱۴۰۴ | **زمانبندی با استفاده از الگوریتم $Cuckoo$** | فاز دوم

فهرست مطالب

۱ مقدمه

امروزه با افزایش پیچیدگی سامانه‌های محاسباتی، استفاده از پردازنده‌های چندهسته‌ای ($multi-core\ processors$) به امری رایج بدل شده است. این پردازنده‌ها امکان اجرای موازی چندین وظیفه را فراهم می‌کنند که این امر به‌ویژه در سامانه‌های بی‌درنگ ($real-time\ systems$) از اهمیت بالایی برخوردار است. در این سامانه‌ها، صحت عملکرد نه تنها به درستی نتایج محاسباتی، بلکه به زمان ارائه این نتایج نیز وابسته است و هرگونه تأخیر در اجرای وظایف می‌تواند منجر به نتایج نامطلوب یا حتی فاجعه‌بار گردد.

یکی از چالش‌های اساسی در بهره‌برداری مؤثر از پردازنده‌های چندهسته‌ای، زمان‌بندی ($scheduling$) بهینه وظایف است. مسئله اصلی، که در دسته مسائل بهینه‌سازی $NP-hard$ قرار می‌گیرد، تخصیص مجموعه‌ای از وظایف به هسته‌های پردازشی به گونه‌ای است که اهداف عملکردی سیستم بهینه شوند. این اهداف در این پژوهش شامل کمینه‌سازی زمان تکمیل کل وظایف یا ($makespan$)، متعادل‌سازی بار ($load\ balancing$) میان هسته‌ها، و مهم‌تر از همه، بیشینه‌سازی کیفیت خدمات ($Quality\ of\ Service-QoS$) سامانه با رعایت موعدهای زمانی ($deadline$) وظایف است.

این پروژه به بررسی و مقایسه عملکرد دو الگوریتم فراابتکاری ($Metaheuristic$) قدرتمند برای حل این مسئله زمان‌بندی می‌پردازد: الگوریتم ژنتیک ($Genetic\ Algorithm-GA$) و الگوریتم جستجوی فاخته ($Cuckoo\ Search-CS$). هر دو الگوریتم با الهام از طبیعت، به دنبال یافتن راه‌حل‌های بهینه در فضاهای جستجوی پیچیده هستند. الگوریتم ژنتیک این کار را با شبیه‌سازی فرآیندهای تکامل داروینی مانند انتخاب، ترکیب و جهش انجام می‌دهد، در حالی که الگوریتم جستجوی فاخته از رفتار انگلی تخم‌گذاری فاخته و مکانیزم جستجوی پویای «پروازهای لِوی» ($Levy\ Flights$) بهره می‌برد.

برای ارائه یک مقایسه علمی و دقیق، ابتدا مجموعه‌ای از وظایف بی‌درنگ متناوب با استفاده از الگوریتم $UUNIFAST$ تولید می‌شوند. این وظایف دارای مشخصاتی نظیر زمان اجرا، دوره تناوب و موعد زمانی هستند. سپس، هر دو الگوریتم GA و CS برای تخصیص این وظایف به هسته‌ها در سناریوهای مختلف (با ۸، ۱۶ و ۳۲ هسته و با سطوح مختلف بار کاری) به کار گرفته می‌شوند. یک نوآوری کلیدی در این پژوهش، روش ارزیابی راه‌حل‌هاست: به جای استفاده از یک تابع برازندگی تقریبی، کیفیت هر تخصیص با اجرای یک شبیه‌سازی کامل زمان‌بندی EDF (اولین موعد زمانی نزدیک‌تر) بر روی هر هسته سنجیده می‌شود. این شبیه‌سازی به ما اجازه می‌دهد تا زمان اتمام واقعی هر کار را محاسبه کرده و کیفیت خدمات را بر اساس یک تابع مطلوبیت خطی و واقع‌گرایانه ارزیابی کنیم.

گزارش حاضر به تشریح کامل مبانی نظری هر دو الگوریتم، جزئیات پیاده‌سازی آن‌ها و محیط شبیه‌سازی دقیق طراحی‌شده می‌پردازد. در نهایت، نتایج به‌دست‌آمده از اجرای هر دو الگوریتم تحت پیکربندی‌های مختلف به صورت جامع ارائه شده و با یکدیگر مقایسه می‌شوند تا نقاط قوت و ضعف هر یک مشخص گردد و بتوانیم به این پرسش پاسخ دهیم که کدام استراتژی جستجو برای حل مسئله پیچیده زمان‌بندی وظایف بی‌درنگ بر روی پردازنده‌های چندهسته‌ای مناسب‌تر است.

مخزن گیتهاب این پروژه در آدرس `https://github.com/NikaGhaderi/Cuckoo-Search-Task-Scheduling` قابل مشاهده است.

۲ مبانی نظری الگوریتم‌های بهینه‌سازی

در این بخش، به تشریح مبانی نظری دو الگوریتم ژنتیک $(Genetic\ Algorithm)$ و الگوریتم جستجوی فاخته $(Cuckoo\ Search\ Algorithm)$، که در این پژوهش برای حل مسئله زمان‌بندی وظایف مورد استفاده قرار گرفته‌اند، می‌پردازیم. سپس، این دو الگوریتم از جنبه‌های مختلف با یکدیگر مقایسه خواهند شد.

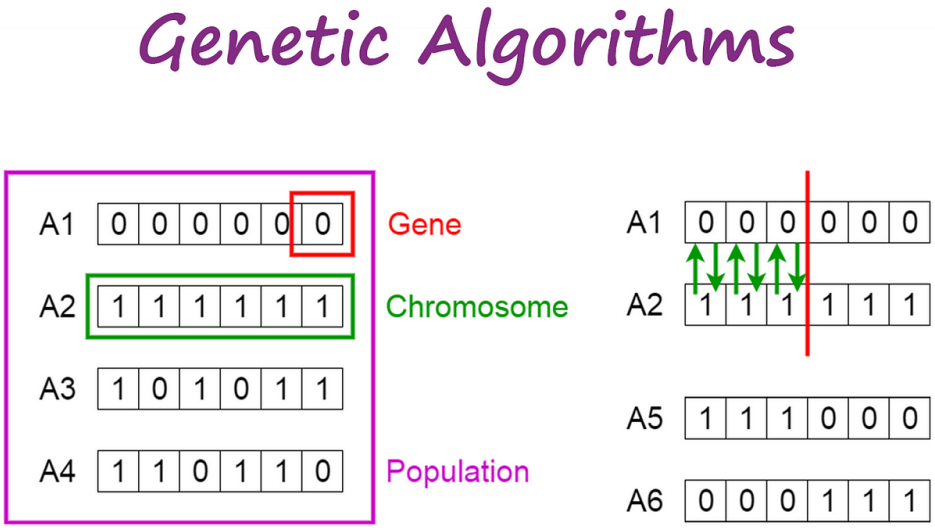
۱.۲ الگوریتم ژنتیک $(Genetic\ Algorithm\ -\ GA)$

الگوریتم ژنتیک یک روش جستجو و بهینه‌سازی مبتنی بر جمعیت است که از اصول تکامل طبیعی و ژنتیک داروینی الهام گرفته شده است. این الگوریتم در دهه ۱۹۷۰ توسط جان هالند $(John\ Holland)$ توسعه یافت و به دلیل کارایی بالا در یافتن پاسخ‌های بهینه یا نزدیک به بهینه برای مسائل پیچیده، به یکی از پرکاربردترین الگوریتم‌های فراابتکاری تبدیل شده است. ایده اصلی (GA) شبیه‌سازی فرآیند "بقای اصلح" $(survival\ of\ the\ fittest)$ در میان جمعیتی از راه‌حل‌های کاندید است.

۱.۱.۲ مفاهیم و اجزای کلیدی

در الگوریتم ژنتیک، هر راه‌حل کاندید برای مسئله، یک «فرد» $(individual)$ یا «کروموزوم» $(chromosome)$ نامیده می‌شود که از مجموعه‌ای از «ژن‌ها» $(genes)$ تشکیل شده است. در مسئله زمان‌بندی ما، یک کروموزوم، آرایه‌ای است که نحوه تخصیص هر وظیفه به یک هسته خاص را نمایش می‌دهد و هر ژن، شماره هسته‌ای است که یک وظیفه مشخص به آن اختصاص یافته است. جمعیت $(population)$ نیز مجموعه‌ای از این کروموزوم‌هاست. عملکرد الگوریتم ژنتیک بر پایه سه عملگر اصلی استوار است:

- **انتخاب** $(Selection)$: در این مرحله، کروموزوم‌های برتر جمعیت فعلی برای تولید نسل بعدی انتخاب می‌شوند. برتری هر کروموزوم با استفاده از یک «تابع برازندگی» $(fitness\ function)$ سنجیده می‌شود. کروموزوم‌هایی با برازندگی بالاتر، شانس بیشتری برای انتخاب شدن به عنوان «والد» $(parent)$ دارند. روش‌های مختلفی برای انتخاب وجود دارد، مانند روش «چرخ رولت» $(Roulette\ Wheel)$ که در آن احتمال انتخاب هر فرد متناسب با برازندگی آن است.
- **ترکیب** $(Crossover)$: این عملگر، فرآیند تولید مثل در طبیعت را شبیه‌سازی می‌کند. دو کروموزوم والد انتخاب شده، اطلاعات ژنتیکی خود را با یکدیگر مبادله کرده و یک یا دو «فرزند» $(offspring)$ جدید تولید می‌کنند. هدف از این کار، ترکیب ویژگی‌های خوب والدین و ایجاد راه‌حل‌های بالقوه بهتر است. روش‌های متداولی مانند «ترکیب تک‌نقطه‌ای» $(single-point\ crossover)$ یا «چندنقطه‌ای» وجود دارد.
- **جهش** $(Mutation)$: این عملگر به صورت تصادفی، مقدار یک یا چند ژن را در یک کروموزوم تغییر می‌دهد. جهش برای حفظ تنوع ژنتیکی در جمعیت و جلوگیری از همگرایی زودهنگام الگوریتم به یک بهینه محلی $(local\ optimum)$ ضروری است. نرخ جهش $(mutation\ rate)$ پارامتری است که احتمال وقوع این پدیده را کنترل می‌کند.



شکل ۱: مفاهیم کلی الگوریتم ژنتیک: ژن، کروموزوم، جمعیت و عمل ترکیب کروموزوم‌ها

۲.۱.۲ روند اجرای الگوریتم

فرآیند کلی اجرای الگوریتم ژنتیک را می‌توان در مراحل زیر خلاصه کرد:

۱. **ایجاد جمعیت اولیه:** مجموعه‌ای از راه‌حل‌های کاندید (کروموزوم‌ها) به صورت تصادفی ایجاد می‌شود.

۲. **ارزیابی برازندگی:** برازندگی هر کروموزوم در جمعیت با استفاده از تابع برازندگی محاسبه می‌شود.

۳. **تکرار نسل‌ها:** حلقه اصلی الگوریتم تا رسیدن به شرط توقف (مانند تعداد مشخصی از نسل‌ها) تکرار می‌شود:

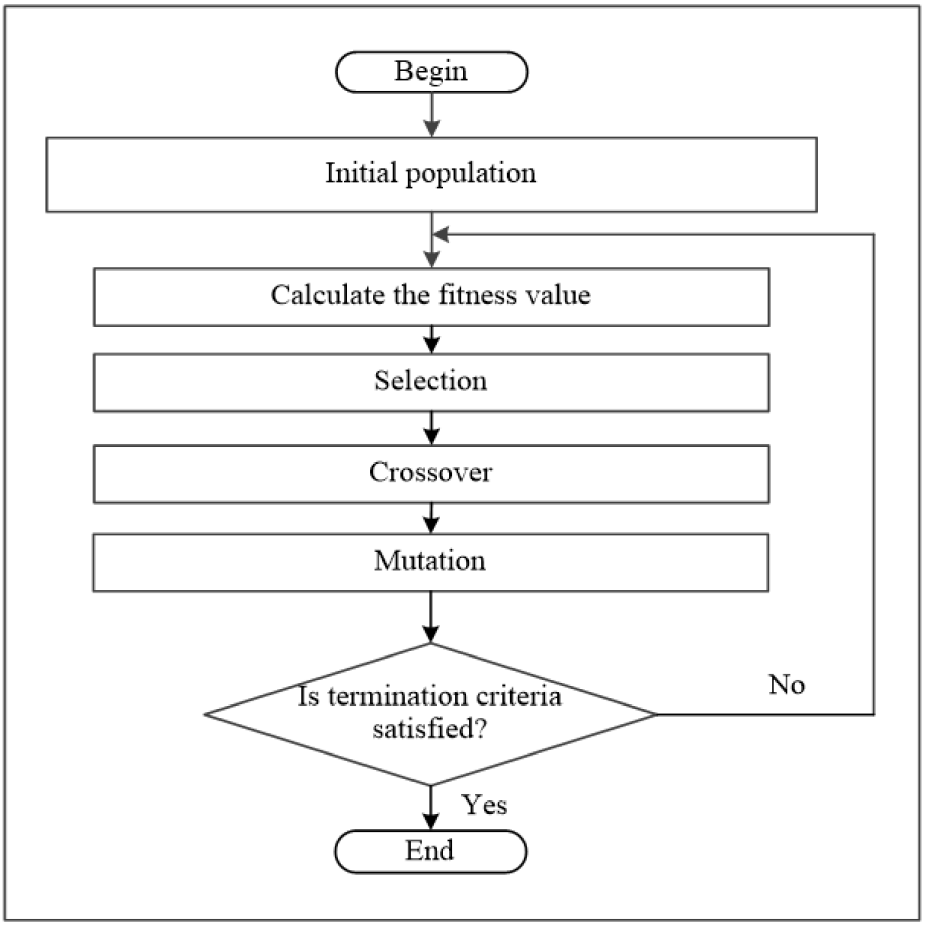
(آ) والدین بر اساس برازندگی خود از جمعیت فعلی انتخاب می‌شوند.

(ب) عملگر ترکیب بر روی والدین منتخب اعمال شده و فرزندان جدید تولید می‌شوند.

(ج) عملگر جهش با احتمالی اندک بر روی فرزندان اعمال می‌شود.

(د) جمعیت جدید با جایگزین کردن کروموزوم‌های قدیمی با فرزندان جدید (و معمولاً با حفظ تعدادی از بهترین‌های نسل قبل یا «نخبگان» ($elites$)) تشکیل می‌شود.

۴. **بازگرداندن نتیجه:** پس از پایان حلقه، بهترین کروموزوم یافت‌شده در طول تمام نسل‌ها به عنوان راه‌حل نهایی مسئله بازگردانده می‌شود.



شکل ۲: فلوچارت عملکرد الگوریتم ژنتیک

۲.۲ الگوریتم جستجوی فاخته $(Cuckoo\ Search\ -\ CS)$

الگوریتم جستجوی فاخته که در سال ۲۰۰۹ توسط شین‌ـشه یانگ $(Xin-She\ Yang)$ و سُواش دب $(Suash\ Deb)$ معرفی شد، یک الگوریتم بهینه‌سازی الهام‌گرفته از طبیعت است که به دلیل سادگی، تعداد پارامترهای کم و کارایی بالا، محبوبیت زیادی کسب کرده است. این الگوریتم رفتار انگلی برخی از گونه‌های فاخته را شبیه‌سازی می‌کند که برای بقای نسل خود، تخم‌هایشان را در لانه پرندگان دیگر می‌گذارند.

شکل ۳: تخم‌های کوکو در لانه‌ی پرنده‌ی میزبان

۱.۲.۲ ایده اصلی: یک استراتژی هوشمندانه برای جستجو

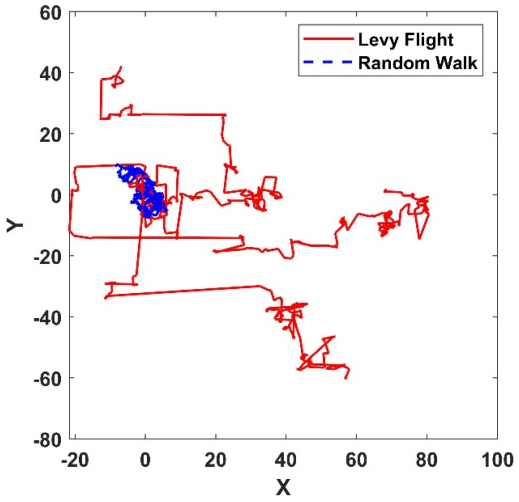
برای درک بهتر الگوریتم، یک مثال ساده را در نظر بگیرید. فرض کنید تعدادی لانه پرنده در یک منطقه وسیع وجود دارد.

- **هر لانه یک راه‌حل است:** در مسئله ما، هر «لانه» $(nest)$ یک راه‌حل کامل برای تخصیص وظایف به هسته‌هاست.
- **هدف، یافتن بهترین لانه است:** ما به دنبال یافتن لانه‌ای هستیم که بهترین کیفیت را داشته باشد (در مسئله ما، بالاترین کیفیت خدمات یا QoS را نتیجه دهد).

الگوریتم جستجوی فاخته از دو استراتژی اصلی برای یافتن این لانه بهینه استفاده می‌کند:

استراتژی اول: جستجوی هوشمند با پرواز لِوی (تولید فاخته جدید) یک فاخته به صورت تصادفی از یک لانه پرواز کرده تا یک مکان جدید و بهتر برای تخم‌گذاری پیدا کند. این پرواز، یک پرواز عادی نیست، بلکه یک «پرواز لِوی» $(Levy\ Flight)$ است.

- **پرواز لِوی چیست؟** پرواز لوی یک نوع راه رفتن تصادفی خاص است که از رفتار حیوانات در جستجوی غذا الهام گرفته شده. تصور کنید در یک دشت بزرگ به دنبال چیزی می‌گردید. بیشتر اوقات، گام‌های کوتاه و نزدیک به هم برمی‌دارید تا محیط اطراف خود را با دقت جستجو کنید (این معادل جستجوی محلی یا بهره‌برداری $(exploitation)$ است). اما هر از گاهی، یک پرش بسیار بلند و ناگهانی به یک نقطه کاملاً دوردست انجام می‌دهید تا یک ناحیه جدید را بررسی کنید (این معادل جستجوی سراسری یا کاوش $(exploration)$ است).
- **چرا این روش قدرتمند است؟** این ترکیب هوشمندانه از گام‌های کوتاه و پرش‌های بلند، به الگوریتم اجازه می‌دهد هم به خوبی در اطراف راه‌حل‌های خوب فعلی جستجو کند و هم از گیر افتادن در بهینه‌های محلی فرار کرده و کل فضای مسئله را کاوش نماید.



شکل ۴: نمونه‌ای از مسیر جستجوی دو بعدی با استفاده از پروازهای لِوی در مقایسه با گام‌های تصادفی استاندارد. پرش‌های بلند مشخصه اصلی پرواز لوی است.

پس از این پرواز، فاخته تخم خود (یک راه‌حل جدید) را در لانه جدید می‌گذارد. سپس این راه‌حل جدید با یک راه‌حل دیگر که به صورت تصادفی از جمعیت انتخاب شده، مقایسه می‌شود و اگر بهتر بود، جایگزین آن می‌گردد.

استراتژی دوم: رها کردن لانه‌های بد در طبیعت، این احتمال وجود دارد که پرنده میزبان، تخم فاخته را شناسایی کرده و آن را دور بیندازد یا کل لانه را رها کند.

- در الگوریتم، ما این رفتار را با شناسایی بدترین لانه‌ها (راه‌حل‌هایی با کمترین برازندگی) شبیه‌سازی می‌کنیم.
- با یک احتمال مشخص p_a، این راه‌حل‌های ضعیف حذف شده و با راه‌حل‌های کاملاً جدید و تصادفی جایگزین می‌شوند. این کار باعث تزریق تنوع به جمعیت شده و از همگرایی زودهنگام جلوگیری می‌کند.

۲.۲.۲ روند اجرای الگوریتم

با توجه به استراتژی‌های بالا، مراحل اجرای الگوریتم جستجوی فاخته به شرح زیر است:

۱. **ایجاد جمعیت اولیه:** مجموعه‌ای از n لانه (راه‌حل‌های اولیه) به صورت تصادفی ایجاد می‌شود.

۲. **ارزیابی برازندگی:** کیفیت هر راه‌حل محاسبه می‌شود.

۳. **تکرار نسل‌ها:** حلقه اصلی تا رسیدن به شرط توقف تکرار می‌شود:

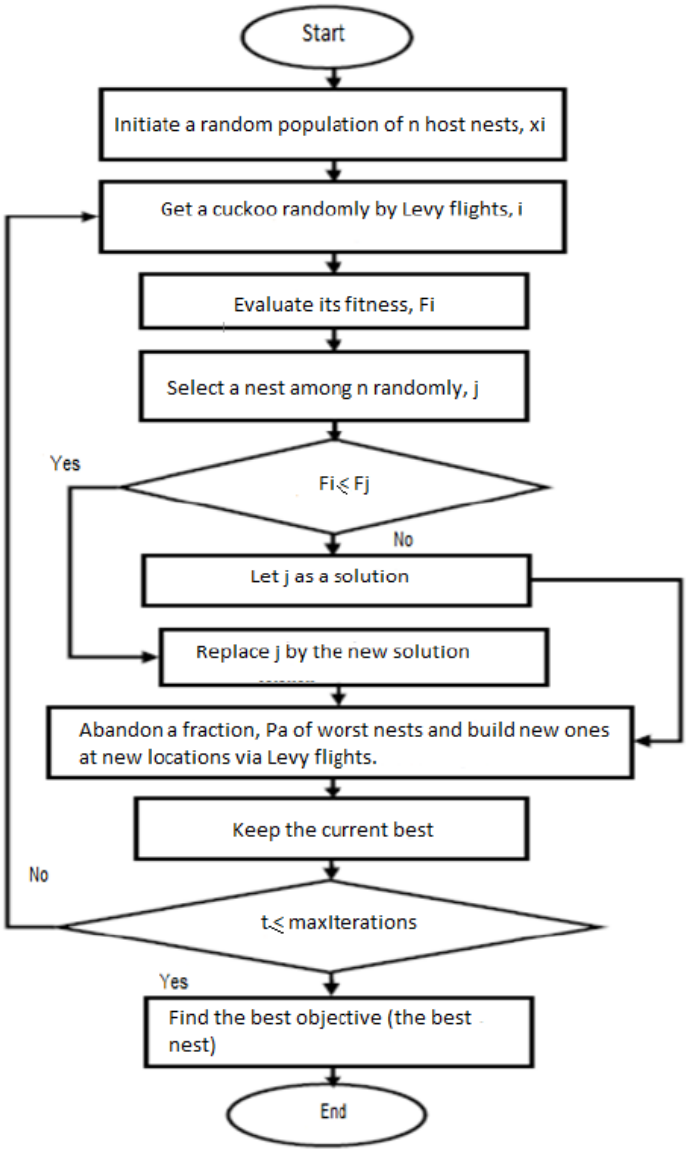
(آ) یک راه‌حل جدید با استفاده از پرواز لِوی تولید می‌شود (استراتژی اول).

(ب) این راه‌حل جدید با یک راه‌حل تصادفی دیگر از جمعیت مقایسه شده و در صورت بهتر بودن، جایگزین آن می‌شود.

(ج) بدترین بخش جمعیت (به نسبت p_a) با راه‌حل‌های کاملاً جدید و تصادفی جایگزین می‌شوند (استراتژی دوم).

(د) بهترین راه‌حل یافت‌شده تا به این لحظه، همواره حفظ می‌شود.

۴. **بازگرداندن نتیجه:** پس از پایان حلقه، بهترین لانه (راه‌حل) یافت‌شده بازگردانده می‌شود.



شکل ۵: فلوچارت الگوریتم جستجوی فاخته

قبل از اینکه به مقایسه این دو الگوریتم بپردازیم، در بخش زیر پرواز لوی را دقیق‌تر بررسی کرده و توضیح می‌دهیم که چگونه از آن در این پروژه استفاده کردیم.

۳.۲.۲ بررسی پرواز لِوی: از تئوری تا پیاده‌سازی

همانطور که اشاره شد، جزء حیاتی و متمایزکننده الگوریتم جستجوی فاخته، استفاده از پروازهای لِوی برای تولید راه‌حل‌های جدید است. به دلیل اهمیت این مکانیزم، در این بخش به تشریح دقیق‌تر مبانی ریاضی و نحوه انطباق آن با مسئله گسسته زمان‌بندی می‌پردازیم.

۱. مفهوم پرواز لِوی پرواز لِوی یک مدل ریاضی برای یک نوع گام تصادفی $(random\ walk)$ است که مشخصه اصلی آن، ترکیبی از گام‌های کوتاه متعدد و پرش‌های بلند و نادر است. این الگو در طبیعت در رفتار جستجوی حیوانات شکارچی به وفور دیده می‌شود. وقتی یک حیوان به دنبال غذا می‌گردد، بیشتر اوقات در یک محدوده کوچک به صورت فشرده جستجو می‌کند (گام‌های کوتاه). اما اگر در آن محدوده غذایی پیدا نکند، یک حرکت ناگهانی و بلند به یک منطقه کاملاً جدید انجام می‌دهد تا شانس خود را در آنجا امتحان کند.
این استراتژی دو مزیت کلیدی دارد که آن را برای بهینه‌سازی بسیار قدرتمند می‌سازد:

- **بهره‌برداری** $(Exploitation)$: گام‌های کوتاه و متوالی به الگوریتم اجازه می‌دهند تا به دقت اطراف یک راه‌حل خوب فعلی را جستجو کرده و آن را بهبود بخشد.
- **کاوش** $(Exploration)$: پرش‌های بلند و گاه‌به‌گاه، تضمین می‌کنند که الگوریتم در یک بهینه محلی $(local\ optimum)$ گیر نمی‌افتد و قادر است کل فضای جستجو را برای یافتن بهینه سراسری $(global\ optimum)$ کاوش کند.

این تعادل پویا بین کاوش و بهره‌برداری، که به صورت ذاتی در پرواز لوی وجود دارد، آن را از گام‌های تصادفی ساده (مانند جهش در الگوریتم ژنتیک) متمایز می‌کند.

۲. فرمول ریاضی و پیاده‌سازی طول گام در پرواز لوی از یک توزیع احتمالاتی به نام «توزیع پایدار لِوی» $(Levy\ stable\ distribution)$ پیروی می‌کند. تولید مستقیم اعداد تصادفی از این توزیع پیچیده است. بنابراین، در عمل از الگوریتم‌های کارآمدی مانند «الگوریتم مانتنیا» $(Mantegna's\ algorithm)$ برای شبیه‌سازی آن استفاده می‌شود. فرمول گام لِوی (s) در این الگوریتم به صورت زیر است:

$$s = \frac{u}{|v|^{1/\beta}}$$

در این فرمول:

- $beta$ (بتا) پارامتر اصلی و توان توزیع لوی است که معمولاً مقداری بین ۱ و ۲ دارد (در پیاده‌سازی ما ۱/۵). این پارامتر شکل توزیع و طول پرش‌ها را کنترل می‌کند.
- u و v دو متغیر تصادفی هستند که از یک توزیع نرمال (گوسی) نمونه‌برداری می‌شوند:

$$u \sim N(0, \sigma_u^2) \quad , \quad v \sim N(0, \sigma_v^2)$$

- انحراف معیار v، یعنی $sigma_v$، معمولاً برابر با ۱ در نظر گرفته می‌شود. اما انحراف معیار u، یعنی $sigma_u$، باید بر اساس پارامتر $beta$ و با استفاده از تابع گاما $(Gamma)$ محاسبه شود تا اطمینان حاصل شود که توزیع نهایی صحیح است:

$$\sigma_u = \left(\frac{\Gamma(1+\beta) \cdot \sin(\pi\beta/2)}{\Gamma((1+\beta)/2) \cdot \beta \cdot 2^{(\beta-1)/2}} \right)^{1/\beta}$$

درک جزئیات استخراج این فرمول برای $sigma_u$ ضروری نیست. نکته کلیدی این است که این فرمول یک روش استاندارد برای تولید گام‌های تصادفی است که ویژگی‌های آماری پرواز لوی را به درستی شبیه‌سازی می‌کند.
کد زیر، پیاده‌سازی دقیق این فرمول‌ها در متد $\_levy\_flight\_step$ در پروژه ما را نشان می‌دهد:

```
def _levy_flight_step(self):
    # Calculate sigma_u based on beta and the Gamma function
    sigma_u = (gamma(1 + self.beta) * np.sin(np.pi * self.beta / 2) /
               (gamma((1 + self.beta) / 2) * self.beta * 2  ((self.beta
               - 1) / 2)))  (1 / self.beta)

    # Sample u and v from Normal distributions
    u = np.random.normal(0, sigma_u, 1)
    v = np.random.normal(0, 1, 1) # sigma_v is 1

    # Calculate the final Levy step
    step = u / (np.abs(v)  (1 / self.beta))
    return step
```

۳. انطباق پرواز لِوی با مسئله زمان‌بندی (گسسته‌سازی) یک چالش مهم باقی می‌ماند: گام لوی (s) یک عدد پیوسته ($continuous$) است، اما مسئله ما گسسته ($discrete$) است. یک وظیفه می‌تواند به هسته شماره ۱ یا ۲ تخصیص یابد، اما نمی‌تواند مثلا به هسته شماره ۱/۷۳ تخصیص یابد. بنابراین، ما نمی‌توانیم مستقیماً مقدار گام را به یک راه‌حل (که آرایه‌ای از اعداد صحیح است) اضافه کنیم.
راه‌حل ما، استفاده از اندازه یا بزرگی گام لوی برای کنترل میزان تغییر در یک راه‌حل است. به عبارت دیگر، به جای اینکه بپرسیم «چقدر حرکت کنیم؟»، می‌پرسیم «چند ژن را تغییر دهیم؟».
روند تولید یک راه‌حل جدید (new_nest) از یک راه‌حل موجود در کد به این صورت است:

۱. ابتدا یک گام لوی ($step$) محاسبه می‌شود.

۲. سپس یک «اندازه گام» ($step\_size$) کلی با ضرب این گام در یک ضریب کوچک و تفاضل دو راه‌حل تصادفی محاسبه می‌شود. این کار به مقیاس‌بندی حرکت کمک می‌کند.

۳. مرحله کلیدی انطباق: بزرگی (نُرم) بردار $step\_size$ را محاسبه کرده و آن را به یک عدد صحیح به نام $n\_changes$ تبدیل می‌کنیم. این عدد مشخص می‌کند که چند وظیفه باید در راه‌حل فعلی به صورت تصادفی تغییر کنند.

۴. در نهایت، به تعداد $n\_changes$ وظیفه را به صورت تصادفی انتخاب کرده و هسته تخصیص‌یافته به آن‌ها را با یک شماره هسته تصادفی جدید جایگزین می‌کنیم.

کد زیر این منطق را نشان می‌دهد:

```
# Calculate the step size
step = self._levy_flight_step()
step_size = 0.01 * step * (best_nest - nests[random.randint(0,
self.n_nests-1)])

# --- ADAPTATION STEP ---
# Use the magnitude of the step to determine HOW MANY tasks to change
n_changes = min(int(np.linalg.norm(step_size)) + 1, self.n_tasks)

# Apply the discrete change
indices_to_change = random.sample(range(self.n_tasks), n_changes)
for idx in indices_to_change:
    new_nest[idx] = random.randint(0, self.num_cores - 1)
```

با این روش، ما مفهوم پرواز لوی را به مسئله گسسته خود ترجمه کرده‌ایم. یک گام لوی کوتاه (که اغلب رخ می‌دهد) منجر به یک $n\_changes$ کوچک (مثلاً ۱ یا ۲) می‌شود و یک تغییر جزئی و محلی در راه‌حل ایجاد می‌کند. یک پرش لوی بلند (که به ندرت رخ می‌دهد) منجر به یک $n\_changes$ بزرگ شده و یک تغییر عمده و سراسری در راه‌حل ایجاد می‌کند و به این ترتیب، تعادل بین کاوش و بهره‌برداری حفظ می‌شود.

۳.۲ تحلیل مقایسه‌ای الگوریتم ژنتیک و جستجوی فاخته

اگرچه هر دو الگوریتم (GA) و (CS) در دسته الگوریتم‌های فراابتکاری مبتنی بر جمعیت قرار می‌گیرند و هدفشان یافتن راه‌حل بهینه است، اما تفاوت‌ها و شباهت‌های بنیادینی در فلسفه و نحوه عملکرد آن‌ها وجود دارد.

۱.۳.۲ وجوه اشتراک ($Similarities$)

- **مبتنی بر جمعیت:** هر دو الگوریتم به جای کار با یک راه‌حل واحد، با مجموعه‌ای از راه‌حل‌های کاندید به صورت موازی کار می‌کنند.
- **الهام از طبیعت:** هر دو از پدیده‌های طبیعی الهام گرفته‌اند؛ (GA) از تکامل داروینی و (CS) از رفتار انگلی پرندگان.
- **استفاده از تابع برازندگی:** هر دو برای ارزیابی کیفیت راه‌حل‌های کاندید از یک تابع هدف یا برازندگی استفاده می‌کنند.
- **ماهیت تصادفی:** هر دو الگوریتم شامل مولفه‌های تصادفی ($stochastic$) هستند که به آن‌ها در کاوش فضای جستجو کمک می‌کند.
- **کاربرد گسترده:** هر دو برای حل مسائل بهینه‌سازی پیچیده و از نوع ($NP-hard$) مانند مسئله زمان‌بندی، بسیار مناسب هستند.

۲.۳.۲ وجوه تمایز ($Differences$)

تفاوت‌های کلیدی بین این دو الگوریتم، که بر عملکرد آن‌ها تأثیر مستقیم دارد، در ادامه تشریح می‌شود.

- **نحوه تولید راه‌حل‌های جدید:** این اصلی‌ترین تفاوت میان دو الگوریتم است.
  - **GA:** برای تولید نسل جدید، به شدت به عملگرهای ترکیب ($crossover$) و جهش ($mutation$) وابسته است. ترکیب، اطلاعات را میان دو والد مبادله می‌کند، در حالی که جهش، تغییرات کوچک و تصادفی ایجاد می‌نماید.
  - **CS:** از پروازهای لِوی برای تولید راه‌حل‌های جدید استفاده می‌کند. این مکانیزم به طور ذاتی هم کاوش سراسری (از طریق پرش‌های بلند) و هم بهره‌برداری محلی (از طریق گام‌های کوتاه) را در یک فرآیند واحد ترکیب می‌کند.

- **استراتژی انتخاب و جایگزینی:**
  - **GA:** معمولاً یک نسل کاملاً جدید در هر تکرار ایجاد می‌شود. والدین بر اساس برازندگی انتخاب می‌شوند و فرزندان جایگزین بخش بزرگی از جمعیت قبلی (یا تمام آن، به جز نخبگان) می‌شوند.
  - **CS:** استراتژی ساده‌تری دارد. یک راه‌حل جدید با یک راه‌حل موجود که به صورت تصادفی انتخاب شده مقایسه می‌شود و تنها در صورت بهتر بودن، جایگزین می‌گردد. علاوه بر این، یک استراتژی حذف مستقیم برای بدترین راه‌حل‌ها (با احتمال p_a) وجود دارد.
- **تعداد پارامترهای کنترلی:**
  - **GA:** به تنظیم دقیق چندین پارامتر حساس است: اندازه جمعیت (pop_size)، نرخ ترکیب $(crossover\_rate)$، نرخ جهش $(mutation\_rate)$ و اندازه نخبگان $(elite\_size)$. یافتن ترکیب بهینه این پارامترها خود یک چالش است.
  - **CS:** تعداد پارامترهای کمتری دارد. پارامترهای اصلی آن شامل اندازه جمعیت (تعداد لانه‌ها، n) و احتمال حذف لانه‌های بد (p_a) است. این سادگی، پیاده‌سازی و تنظیم (CS) را آسان‌تر می‌کند.
- **تعادل بین کاوش و بهره‌برداری** $(Exploration\ vs.\ Exploitation)$**:**
  - **GA:** این تعادل را از طریق دو عملگر مجزا (جهش برای کاوش و ترکیب برای بهره‌برداری) مدیریت می‌کند. تنظیم نرخ این دو عملگر برای دستیابی به تعادل مناسب، حیاتی است.
  - **CS:** به لطف پروازهای لِوی، این تعادل به شکل پویاتری مدیریت می‌شود. مطالعات نشان داده‌اند که پروازهای لِوی می‌توانند به طور موثرتری فضای جستجو را در مقایسه با گام‌های تصادفی ساده (مانند جهش در (GA)) کاوش کنند، که این امر اغلب منجر به همگرایی سریع‌تر به سمت بهینه سراسری می‌شود.

به طور خلاصه، در حالی که الگوریتم ژنتیک یک روش کلاسیک، قدرتمند و اثبات‌شده است، الگوریتم جستجوی فاخته به عنوان یک روش جدیدتر، با پارامترهای کمتر و مکانیزم جستجوی پویاتر از طریق پروازهای لِوی، در بسیاری از کاربردها نتایج رقابتی و حتی بهتری را نشان داده است. انتخاب بین این دو الگوریتم برای مسئله زمان‌بندی بی‌درنگ، به ویژگی‌های خاص مسئله و معیارهای عملکردی مورد نظر بستگی دارد که در بخش‌های بعدی این پروژه به صورت عملی مورد ارزیابی قرار خواهد گرفت.

۳ توضیح کد

در این بخش، به تشریح دقیق توابع و کلاس‌های کلیدی پیاده‌سازی‌شده در این پروژه می‌پردازیم.

۱.۳ تابع تولید وظایف ($generate\_tasks$)

اساس هر شبیه‌سازی در این پژوهش، تولید مجموعه‌ای از وظایف بی‌درنگ مصنوعی است که بتوان الگوریتم‌های زمان‌بندی را بر روی آن‌ها آزمود. تابع $generate\_tasks$ این مسئولیت را بر عهده دارد. هدف اصلی این تابع، ایجاد مجموعه‌ای از وظایف با مشخصات واقع‌گرایانه است که مجموع بهره‌وری آن‌ها با مقدار هدف تعیین‌شده برای یک سناریوی خاص برابر باشد. برای این منظور، از الگوریتم شناخته‌شده $UUNIFAST$ استفاده می‌شود که تضمین می‌کند مجموع بهره‌وری به صورت عادلانه و بدون سوگیری بین وظایف تقسیم شود.

روند کار این تابع در دو مرحله اصلی انجام می‌شود. در مرحله اول، که بخش کلیدی الگوریتم $UUNIFAST$ است، بهره‌وری کل خواسته شده ($total\_utilization$) به صورت تکرارشونده بین تعداد مشخصی از وظایف (num_tasks) تقسیم می‌گردد. این فرآیند با استفاده از تولید اعداد تصادفی که به توان کسری خاصی می‌رسند، تضمین می‌کند که بردار بهره‌وری حاصل، توزیع یکنواختی داشته باشد. در هر مرحله از حلقه، بخشی از بهره‌وری باقیمانده به یک وظیفه تخصیص داده شده و مقدار باقیمانده برای تخصیص به وظایف بعدی به‌روزرسانی می‌شود.

پس از آنکه میزان بهره‌وری ($utilization$) برای هر وظیفه مشخص شد، مرحله دوم یعنی تعیین سایر مشخصات هر وظیفه آغاز می‌شود. برای هر وظیفه، ابتدا یک دوره تناوب ($period$) به صورت تصادفی از میان مجموعه‌ای از مقادیر متداول (مانند ۱۰، ۲۰، ۴۰، ۵۰، ۱۰۰ و ۲۰۰ واحد زمانی) انتخاب می‌گردد. سپس، با استفاده از رابطه بنیادین $C_i = U_i \times T_i$، زمان اجرای ($execution\ time$) وظیفه محاسبه می‌شود. یک نکته مهم در پیاده‌سازی ما این است که زمان اجرا نمی‌تواند کمتر از یک واحد زمانی باشد؛ بنابراین، اگر مقدار محاسبه‌شده کمتر از ۱ باشد، به ۱ گرد می‌شود. در نهایت، موعد زمانی ($deadline$) هر وظیفه برابر با دوره تناوب آن در نظر گرفته می‌شود (که به آن موعد زمانی ضمنی یا $implicit\ deadline$ می‌گویند) و بهره‌وری نهایی وظیفه بر اساس زمان اجرای واقعی و دوره تناوب آن مجدداً محاسبه می‌شود تا از سازگاری کامل داده‌ها اطمینان حاصل شود. خروجی نهایی این تابع، لیستی از دیکشنری‌های پایتون است که هر دیکشنری، یک وظیفه را با تمام مشخصات کلیدی آن (شامل شناسه، زمان اجرا، دوره تناوب، موعد زمانی و بهره‌وری) نمایش می‌دهد.

```
def generate_tasks(num_tasks, total_utilization):
    utilizations = []
    remaining_util = total_utilization
    if remaining_util <= 0: return []

    for i in range(1, num_tasks):
        if num_tasks - i <= 0: continue
        next_util = remaining_util * random.random() ** (1 / (num_tasks - i))
        utilizations.append(remaining_util - next_util)
        remaining_util = next_util
    if remaining_util > 0: utilizations.append(remaining_util)

    tasks = []
    for i, util in enumerate(utilizations):
        period = random.choice([10, 20, 40, 50, 100, 200])
        execution = util * period
        if execution < 1: execution = 1
        tasks.append({
            'id': i, 'execution': execution, 'period': period,
            'deadline': period, 'utilization': execution / period if
            period > 0 else 0
        })
    return tasks
```

۲.۳ کلاس زمان‌بند ژنتیک ($GeneticScheduler$)

این کلاس، تمامی منطق و عملیات مربوط به الگوریتم ژنتیک را برای مسئله زمان‌بندی وظایف بر روی هسته‌های پردازنده پیاده می‌کند. این الگوریتم سعی دارد با الهام از فرآیندهای تکاملی طبیعی، یک تخصیص بهینه از وظایف به هسته‌ها را پیدا کند به طوری که بهره‌وری هسته‌ها متوازن شده و سربار اضافی به حداقل برسد.

۱.۲.۳ متد سازنده ($\_\_init\_\_$)

متد سازنده ($constructor$) کلاس، مسئول مقداردهی اولیه به پارامترها و ویژگی‌های اصلی الگوریتم ژنتیک است.

```
class GeneticScheduler:
    def __init__(self, tasks, num_cores, pop_size=50, elite=0.2,
    mutation_rate=0.1, generations=100):
        self.tasks = tasks
        self.num_cores = num_cores
        self.pop_size = pop_size
        self.elite = int(elite * pop_size)
        self.mutation_rate = mutation_rate
        self.generations = generations
```

توضیحات خط به خط:

- $class\ GeneticScheduler$: تعریف کلاس با نام $GeneticScheduler$.
- $def\_\_init\_\_(self, tasks, num\_cores, pop\_size = ۵۰, elite = ۰/۲, mutation\_rate = ۰/۱, generations = ۱۰۰)$:
  تعریف متد سازنده $\_\_init\_\_$.
  - $self$: ارجاع به نمونه فعلی کلاس.
  - $tasks$: لیستی از دیکشنری‌ها که هر دیکشنری اطلاعات یک وظیفه را در خود دارد.
  - num_cores: تعداد هسته‌های پردازنده موجود.
  - $pop\_size = ۵۰$: اندازه جمعیت اولیه کروموزوم‌ها، با مقدار پیش‌فرض ۵۰.
  - $elite = ۰/۲$: درصد نخبگان جمعیت که مستقیماً به نسل بعد منتقل می‌شوند، با مقدار پیش‌فرض ۰/۲ (یعنی ۲۰ درصد).
  - $mutation\_rate = ۰/۱$: نرخ جهش برای هر ژن در کروموزوم، با مقدار پیش‌فرض ۰/۱ (یعنی ۱۰ درصد).
  - $generations = ۱۰۰$: تعداد نسل‌هایی که الگوریتم اجرا خواهد شد، با مقدار پیش‌فرض ۱۰۰.
- $self.tasks = tasks$: لیست وظایف ورودی در متغیر نمونه $self.tasks$ ذخیره می‌شود.
- $self.num\_cores = num\_cores$: تعداد هسته‌ها در متغیر نمونه $self.num\_cores$ ذخیره می‌شود.
- $self.pop\_size = pop\_size$: اندازه جمعیت در متغیر نمونه $self.pop\_size$ ذخیره می‌شود.
- $self.elite = int(elite * pop\_size)$: تعداد دقیق نخبگان با ضرب درصد نخبگان در اندازه جمعیت و تبدیل به عدد صحیح (int) محاسبه و در $self.elite$ ذخیره می‌شود.
- $self.mutation\_rate = mutation\_rate$: نرخ جهش در متغیر نمونه $self.mutation\_rate$ ذخیره می‌شود.
- $self.generations = generations$: تعداد نسل‌ها در متغیر نمونه $self.generations$ ذخیره می‌شود.

۲.۲.۳ متد ایجاد جمعیت اولیه ($initialize\_population$)

این متد مسئول ایجاد جمعیت اولیه از راه‌حل‌های تصادفی (کروموزوم‌ها) است.

```
    def initialize_population(self):
        return [np.random.randint(0, self.num_cores, len(self.tasks))
                for _ in range(self.pop_size)]
```

توضیحات:

- یک لیست ($list\ comprehension$) برای ساخت جمعیت استفاده می‌شود. این لیست شامل $self.pop\_size$ تعداد کروموزوم است.
- هر کروموزوم با استفاده از $np.random.randint(۰, self.num\_cores, len(self.tasks))$ ساخته می‌شود:
  - این دستور یک آرایه ($array$) از اعداد صحیح تصادفی تولید می‌کند.
  - ۰: حد پایین برای اعداد تصادفی (اندیس اولین هسته).
  - $self.num\_cores$: حد بالا برای اعداد تصادفی (تعداد کل هسته‌ها).
  - $len(self.tasks)$: طول آرایه، که برابر با تعداد وظایف است. هر عنصر آرایه نشان می‌دهد که وظیفه متناظر با آن اندیس، به کدام هسته تخصیص داده شده است.

۳.۲.۳ تابع محاسبه برازندگی (متد $\_fitness$)

در فاز اول، تابع برازندگی بر اساس یک فرمول تقریبی و سریع عمل می‌کرد که سعی داشت با جریمه کردن سربار هسته‌ها و عدم توازن بار، الگوریتم را به سمت راه‌حل‌های بهتر هدایت کند. اگرچه این روش سریع بود، اما یک معیار غیرمستقیم برای سنجش هدف اصلی ما، یعنی بیشینه‌سازی کیفیت خدمات (QoS) سیستم، محسوب می‌شد.

در فاز دوم، این رویکرد به طور کامل متحول شده است. اکنون، برازندگی یک راه‌حل (یک کروموزوم) به صورت مستقیم برابر با میانگین کیفیت خدمات کل سیستم در نظر گرفته می‌شود. برای محاسبه این مقدار، به ازای هر راه‌حل کاندید، یک شبیه‌سازی کامل زمان‌بندی EDF بر روی تمام هسته‌ها اجرا شده و QoS نهایی سیستم محاسبه می‌گردد. این روش، اگرچه از نظر محاسباتی بسیار سنگین‌تر است، اما دقیق‌ترین معیار ممکن را برای سنجش کیفیت یک تخصیص ارائه می‌دهد، زیرا مستقیماً همان چیزی را اندازه‌گیری می‌کند که ما به دنبال بهینه‌سازی آن هستیم.

به دلیل هزینه محاسباتی بالای این ارزیابی، یک مکانیزم حافظه پنهان یا «کش» ($cache$) در متد $\_fitness$ پیاده‌سازی شده است. این متد به عنوان یک پوشش ($wrapper$) عمل می‌کند. قبل از انجام محاسبه کامل، ابتدا بررسی می‌کند که آیا برازندگی این راه‌حل خاص قبلاً محاسبه و ذخیره شده است یا خیر. اگر پاسخ مثبت باشد، مقدار ذخیره‌شده بلافاصله بازگردانده می‌شود و از اجرای شبیه‌سازی تکراری و زمان‌بر جلوگیری می‌گردد. در غیر این صورت، تابع اصلی ارزیابی ($get\_solution\_fitness$) فراخوانی شده، نتیجه آن در کش ذخیره و سپس برگردانده می‌شود. این بهینه‌سازی برای عملکرد الگوریتم در نسل‌های مختلف، که احتمال مواجهه با راه‌حل‌های تکراری وجود دارد، حیاتی است.

```
def _fitness(self, solution):
    sol_tuple = tuple(solution)
    if sol_tuple in self.fitness_cache:
        return self.fitness_cache[sol_tuple]
    fitness_val = get_solution_fitness(solution, self.tasks,
    self.num_cores)
    self.fitness_cache[sol_tuple] = fitness_val
    return fitness_val
```

۴.۲.۳ متد انتخاب والدین ($select\_parents$)

این متد والدین را برای تولید نسل بعدی از میان جمعیت فعلی بر اساس برازندگی آن‌ها انتخاب می‌کند (روش چرخ رولت).

```
def select_parents(self, population, fitnesses):
    total_fitness = sum(fitnesses)
    probs = [f / total_fitness for f in fitnesses]
    parents_indices = np.random.choice(
        range(len(population)),
        size=len(population) - self.elite,
        p=probs
    )
    return [population[i] for i in parents_indices]
```

توضیحات:

- $def\ select\_parents(self,\ population,\ fitnesses)$: تعریف متد که جمعیت فعلی و لیست برازندگی‌های متناظر را به عنوان ورودی می‌گیرد.
- $total\_fitness = sum(fitnesses)$: مجموع کل برازندگی تمام کروموزوم‌های جمعیت محاسبه می‌شود.
- $probs = [f/total\_fitness\ for\ f\ in\ fitnesses]$: احتمال انتخاب هر کروموزوم به عنوان والد محاسبه می‌شود. این احتمال متناسب با برازندگی نسبی آن کروموزوم به کل برازندگی جمعیت است.
- $parents\_indices = np.random.choice(...)$: اندیس‌های والدین با استفاده از تابع $np.random.choice$ انتخاب می‌شوند.
  - $range(len(population))$: مجموعه‌ای از اندیس‌های ممکن (از ۰ تا تعداد کروموزوم‌ها منهای ۱).
  - $size = len(population) - self.elite$: تعداد والدینی که باید انتخاب شوند، برابر است با اندازه جمعیت منهای تعداد نخبگان (زیرا نخبگان مستقیماً به نسل بعد می‌روند).
  - $p = probs$: انتخاب‌ها بر اساس احتمالات محاسبه‌شده در $probs$ انجام می‌شود (انتخاب با جایگزینی).
- $return[population[i]\ for\ i\ in\ parents\_indices]$: لیست کروموزوم‌های والد انتخاب شده برگردانده می‌شود.

۵.۲.۳ متد ترکیب ($crossover$)

این متد عملگر ترکیب (تولید فرزند از والدین) را پیاده‌سازی می‌کند. در اینجا از ترکیب تک نقطه‌ای استفاده شده است.

```
def crossover(self, parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2
```

توضیحات:

- $def\ crossover(self,\ parent۱,\ parent۲)$: تعریف متد که دو کروموزوم والد را به عنوان ورودی می‌گیرد.
- $point = random.randint(۱,\ len(parent۱) - ۱)$: یک نقطه برش تصادفی در طول کروموزوم والد انتخاب می‌شود. این نقطه بین ژن اول و آخر است تا اطمینان حاصل شود که هر دو والد در تولید فرزند مشارکت دارند.
- $child۱ = np.concatenate((parent۱[:\ point], parent۲[point\ :]))$: فرزند اول با ترکیب بخش اول از $parent۱$ (تا قبل از نقطه برش) و بخش دوم از $parent۲$ (از نقطه برش تا انتها) ایجاد می‌شود. تابع $np.concatenate$ برای اتصال این دو بخش استفاده می‌شود.
- $child۲ = np.concatenate((parent۲[:\ point], parent۱[point\ :]))$: فرزند دوم با ترکیب بخش اول از $parent۲$ و بخش دوم از $parent۱$ ایجاد می‌شود.
- $return child۱, child۲$: دو فرزند تولید شده برگردانده می‌شوند.

۶.۲.۳ متد تکامل ($evolve$)

این متد اصلی، فرآیند الگوریتم ژنتیک را برای چندین نسل اجرا می‌کند تا به یک راه‌حل بهینه یا نزدیک به بهینه دست یابد.

```
def evolve(self):
    population = self.initialize_population()

    for _ in range(self.generations):
        fitnesses = [self.fitness(chromo) for chromo in
        population]

        elite_indices = np.argsort(fitnesses)[-self.elite:]
        new_population = [population[i] for i in elite_indices]

        parents = self.select_parents(population, fitnesses)
        random.shuffle(parents)

        for i in range(0, len(parents), 2):
            if i + 1 < len(parents):
                child1, child2 = self.crossover(parents[i],
                parents[i + 1])
                new_population += [self.mutate(child1),
                self.mutate(child2)]

        population = new_population

    fitnesses = [self.fitness(chromo) for chromo in population]
    return population[np.argmax(fitnesses)]
```

توضیحات:

- $def\ evolve(self)$: تعریف متد اصلی اجرای الگوریتم.
- $population = self.initialize\_population()$: جمعیت اولیه با فراخوانی متد $self.initialize\_population()$ ایجاد می‌شود.
- $for\ \_\ in\ range(self.generations)$: حلقه اصلی الگوریتم که به تعداد $self.generations$ نسل تکرار می‌شود.
  - $fitnesses = [self.fitness(chromo)\ for\ chromo\ in\ population]$: برازندگی برای تمام کروموزوم‌های جمعیت فعلی با استفاده از متد $self.fitness()$ محاسبه می‌شود.
  - $elite\_indices = np.argsort(fitnesses)[-self.elite\ :]$: اندیس‌های کروموزوم‌های نخبه (با بالاترین برازندگی) شناسایی می‌شوند. $np.argsort(fitnesses)$ اندیس‌ها را بر اساس برازندگی مرتب‌شده برمی‌گرداند و $[-self.elite\ :]$ بخش آخر آن (نخبگان) را انتخاب می‌کند.
  - $new\_population = [population[i]\ for\ i\ in\ elite\_indices]$: کروموزوم‌های نخبه مستقیماً به جمعیت نسل بعدی ($new\_population$) اضافه می‌شوند (عملگر نخبه‌گرایی یا $(elitism)$).
  - $parents = self.select\_parents(population,\ fitnesses)$:
    والدین از جمعیت فعلی با استفاده از متد $self.select\_parents()$ انتخاب می‌شوند.
  - $random.shuffle(parents)$: لیست والدین به صورت تصادفی برهم زده می‌شود تا جفت‌گیری برای ترکیب به صورت تصادفی انجام شود.
  - $for\ i\ in\ range(۰,\ len(parents),\ ۲)$: حلقه‌ای برای انتخاب جفت والدین و تولید فرزندان. حلقه با گام ۲ حرکت می‌کند.

* $if\ i+1 < len(parents)$: اطمینان از وجود والد دوم برای تشکیل زوج.
* $child1,\ child2 = self.crossover(parents[i], parents[i+1])$: دو فرزند از جفت والدین $parents[i]$ و $parents[i+1]$ با استفاده از متد $self.crossover()$ تولید می‌شوند.
* $new\_population += [self.mutate(child1), self.mutate(child2)]$: هر دو فرزند تولید شده ابتدا با متد $self.mutate()$ جهش داده شده و سپس به جمعیت جدید ($new\_population$) اضافه می‌شوند.

– $population = new\_population$: جمعیت جدید ایجاد شده، جایگزین جمعیت فعلی برای نسل بعدی می‌شود.

- $fitnesses = [self.fitness(chromo)\ for\ chromo\ in\ population]$: پس از اتمام تمام نسل‌ها، برازندگی کروموزوم‌های جمعیت نهایی محاسبه می‌شود.
- $return\ population[np.argmax(fitnesses)]$: بهترین کروموزوم (با بالاترین برازندگی) از جمعیت نهایی انتخاب شده و به عنوان نتیجه الگوریتم برگردانده می‌شود. $np.argmax(fitnesses)$ اندیس کروموزومی که بیشترین برازندگی را دارد، برمی‌گرداند.

۷.۲.۳ متد جهش ($mutate$)

این متد عملگر جهش را بر روی یک کروموزوم اعمال می‌کند تا تنوع ژنتیکی در جمعیت حفظ شود و از همگرایی زودرس جلوگیری گردد.

```
def mutate(self, chromosome):
    for i in range(len(chromosome)):
        if random.random() < self.mutation_rate:
            chromosome[i] = random.randint(0, self.num_cores - 1)
    return chromosome
```

توضیحات:

- $def\ mutate(self,\ chromosome)$: تعریف متد که یک کروموزوم را به عنوان ورودی می‌گیرد.
- $for\ i\ in\ range(len(chromosome))$: حلقه‌ای که بر روی تمام ژن‌های کروموزوم (تخصیص هر وظیفه) پیمایش می‌کند.
- $if\ random.random() < self.mutation\_rate$: برای هر ژن، یک عدد تصادفی در بازه $[0, 1)$ تولید می‌شود. اگر این عدد کوچکتر از نرخ جهش ($self.mutation\_rate$) باشد، آنگاه ژن جهش پیدا می‌کند.
- $chromosome[i] = random.randint(0,\ self.num\_cores - 1)$: در صورت وقوع جهش، مقدار ژن iـام (یعنی هسته تخصیص‌یافته به وظیفه iـام) با یک شماره هسته تصادفی جدید (بین ۰ و $self.num\_cores - 1$) جایگزین می‌شود.
- $return\ chromosome$: کروموزوم جهش‌یافته (یا اصلی، اگر جهشی رخ نداده باشد) برگردانده می‌شود.

۳.۳ شبیه‌ساز زمان‌بندی EDF (تابع $edf\_schedule\_on\_core$)

یکی از بنیادی‌ترین تغییرات در فاز دوم پروژه، حرکت از یک ارزیابی تقریبی به سمت یک شبیه‌سازی دقیق از اجرای وظایف است.

تابع $edf\_schedule\_on\_core$ قلب این شبیه‌سازی است. این تابع، مجموعه‌ای از وظایف را که توسط الگوریتم‌های فراابتکاری به یک هسته خاص تخصیص داده شده‌اند، دریافت کرده و اجرای آن‌ها را در طول یک هایپرپریود با استفاده از الگوریتم زمان‌بندی «اولین موعد زمانی نزدیک‌تر» ($Earliest\ Deadline\ First - EDF$) شبیه‌سازی می‌کند. هدف نهایی این تابع، محاسبه زمان اتمام واقعی هر «کار» (job) از هر وظیفه است تا بتوانیم کیفیت خدمات را به صورت دقیق ارزیابی کنیم.

روند کار این تابع به این صورت است که ابتدا تمام «کارهای» متناظر با هر وظیفه که در طول یک هایپرپریود آزاد می‌شوند، تولید می‌گردند. هر کار دارای زمان ورود، زمان اجرای مورد نیاز و یک موعد زمانی مطلق است. سپس، یک حلقه شبیه‌سازی بر اساس یک ساعت سراسری ($time$) اجرا می‌شود. در هر واحد زمانی، تمام کارهایی که زمان ورودشان فرا رسیده، به یک «صف آماده» ($ready\ queue$) اضافه می‌شوند. این صف، یک صف اولویت‌دار است که همواره کارها را بر اساس نزدیک‌ترین موعد زمانی مرتب نگه می‌دارد. در هر لحظه، اگر پردازنده آزاد باشد، کاری که در ابتدای صف آماده قرار دارد (یعنی کاری با نزدیک‌ترین موعد زمانی) برای اجرا انتخاب می‌شود. این فرآیند تا زمانی که تمام کارها تکمیل شوند، ادامه می‌یابد. در طول اجرا، زمان دقیق اتمام هر کار ثبت می‌شود. خروجی این تابع، یک دیکشنری حاوی جزئیات زمان اتمام و موعد زمانی تمام کارهای اجرا شده و همچنین «زمان اتمام کل» ($makespan$) برای آن هسته خاص است.

```
def edf_schedule_on_core(tasks_on_core, hyperperiod):
    if not tasks_on_core: return {}, 0
    if sum(t['utilization'] for t in tasks_on_core) > 1: return {},
    float('inf')

    time = 0
    ready_queue, job_arrivals = [], []
    for task in tasks_on_core:
        if task['period'] > 0:
            for i in range(hyperperiod // task['period']):
                arrival_time = i * task['period']
                job_arrivals.append((arrival_time,
```

```
                task['execution'], arrival_time + task['deadline'],
                task['id']))
    job_arrivals.sort()

    job_results = {task['id']: {'finish_times': [], 'deadlines': []}
    for task in tasks_on_core}
    current_job = None

    while time < hyperperiod * 2:
        while job_arrivals and job_arrivals[0][0] <= time:
            arrival, exec_time, deadline, task_id =
            job_arrivals.pop(0)
            heapq.heappush(ready_queue, (deadline, exec_time,
            task_id))

        if current_job is None and ready_queue:
            deadline, exec_time, task_id = heapq.heappop(ready_queue)
            current_job = {'deadline': deadline, 'remaining_exec':
            exec_time, 'task_id': task_id}

        if current_job:
            current_job['remaining_exec'] -= 1
            if current_job['remaining_exec'] <= 0:
                task_id = current_job['task_id']
                job_results[task_id]['finish_times'].append(time + 1)
                job_results[task_id]
                ['deadlines'].append(current_job['deadline'])
                current_job = None

        time += 1
        if not current_job and not ready_queue and not job_arrivals:
        break

    latest_finish = max((ft for res in job_results.values() for ft
    in res['finish_times']), default=0)
    return job_results, latest_finish
```

۴.۳ تابع محاسبه کیفیت خدمات ($calculate\_qos$)

پس از آنکه زمان اتمام واقعی هر کار توسط شبیه‌ساز EDF مشخص شد، نیاز به یک معیار دقیق برای سنجش کیفیت عملکرد آن کار داریم. تابع $calculate\_qos$ این مسئولیت را بر عهده دارد. این تابع، تعریف ساده‌انگارانه قبلی از کیفیت خدمات (که صرفاً بر اساس بهره‌وری بود) را با یک «تابع مطلوبیت» ($utility\ function$) خطی و واقع‌گرایانه‌تر جایگزین می‌کند. این نوع تابع در سیستم‌های بی‌درنگ نرم ($soft\ real-time$) بسیار متداول است، جایی که از دست دادن یک موعد زمانی به معنای شکست کامل سیستم نیست، بلکه ارزش نتیجه با گذشت زمان کاهش می‌یابد.

منطق پیاده‌سازی‌شده در این تابع بسیار سرراست است. این تابع زمان اتمام ($finish\_time$) و موعد زمانی ($deadline$) یک کار را به عنوان ورودی دریافت کرده و بر اساس سه قانون، یک مقدار عددی بین ۰ تا ۱۰۰ را به عنوان کیفیت خدمات آن کار برمی‌گرداند:

۱. اگر کار قبل از موعد زمانی خود یا دقیقاً در همان لحظه به پایان برسد ($finish\_time \leq deadline$)، به عملکرد ایده‌آل دست یافته‌ایم و کیفیت خدمات برابر با ۱۰۰ خواهد بود.

۲. اگر زمان اتمام کار به اندازه‌ی دو برابر موعد زمانی آن یا بیشتر به طول انجامد ($finish\_time \geq 2\ times\ deadline$)، فرض بر این است که نتیجه کار دیگر هیچ ارزشی نداشته و کیفیت خدمات آن برابر با ۰ است.

۳. اگر زمان اتمام کار بین موعد زمانی و دو برابر موعد زمانی آن باشد، کیفیت خدمات به صورت خطی از ۱۰۰ به ۰ کاهش می‌یابد. این کاهش خطی تضمین می‌کند که هرچه یک کار به موعد زمانی خود نزدیک‌تر تمام شود، امتیاز بالاتری دریافت کند.

این تعریف دقیق از QoS، به ما اجازه می‌دهد تا راه‌حل‌های مختلف را به صورت بسیار معنادارتری با یکدیگر مقایسه کنیم.

```
def calculate_qos(finish_time, deadline):
    if finish_time <= deadline: return 100
    if deadline <= 0: return 0
    if finish_time >= 2 * deadline: return 0
    return 100 * (1 - (finish_time - deadline) / deadline)
```

۵.۳ تابع جامع محاسبه متریک‌ها (*get_full_metrics_for_solution*)

اکنون که ابزارهای لازم برای شبیه‌سازی یک هسته (*edf_schedule_on_core*) و ارزیابی کیفیت خدمات یک کار (*calculate_qos*) را در اختیار داریم، به تابعی نیاز داریم که این قطعات را به یکدیگر متصل کرده و یک ارزیابی جامع از یک راه‌حل کامل (یعنی یک تخصیص کامل وظایف به تمام هسته‌ها) ارائه دهد. تابع *get_full_metrics_for_solution* دقیقاً همین نقش را ایفا می‌کند. این تابع، نقطه اتصال دنیای الگوریتم‌های فراابتکاری (که راه‌حل تولید می‌کنند) و دنیای شبیه‌سازی دقیق (که آن راه‌حل‌ها را ارزیابی می‌کند) است.

روند کار این تابع کاملاً نظام‌مند است. به عنوان ورودی، یک راه‌حل کاندید (*solution*)، لیست کامل وظایف و تعداد هسته‌ها را دریافت می‌کند. اولین گام، سازماندهی وظایف بر اساس راه‌حل ورودی است؛ یعنی مشخص می‌شود که کدام مجموعه از وظایف به هر یک از هسته‌ها تخصیص داده شده‌اند. سپس، هایپرپریود کل سیستم محاسبه می‌شود تا افق زمانی شبیه‌سازی مشخص گردد.

قلب تپنده این تابع، یک حلقه است که بر روی هر یک از هسته‌های پردازنده پیمایش می‌کند. در داخل این حلقه، برای هر هسته، تابع شبیه‌ساز *edf_schedule_on_core* فراخوانی می‌شود تا اجرای وظایف تخصیص‌یافته به آن هسته را شبیه‌سازی کند. پس از اتمام شبیه‌سازی برای یک هسته، نتایج دقیق (شامل زمان اتمام هر کار) به همراه زمان اتمام کل (*makespan*) برای آن هسته بازگردانده می‌شود. این تابع، بیشترین مقدار *makespan* را در میان تمام هسته‌ها به عنوان *makespan* کل سیستم ثبت می‌کند.

در مرحله بعد، با استفاده از نتایج شبیه‌سازی و فراخوانی تابع *calculate_qos* برای هر کار اجرا شده، میانگین کیفیت خدمات برای هر وظیفه محاسبه می‌شود. در نهایت، مهم‌ترین معیار خروجی، یعنی کیفیت خدمات کل سیستم (*system_qos*)، از طریق میانگین‌گیری از کیفیت خدمات تمام وظایف در کل سیستم به دست می‌آید. این مقدار دقیقاً همان عددی است که به عنوان مقدار برازندگی به الگوریتم‌های ژنتیک و فاخته بازگردانده می‌شود تا آن‌ها را در مسیر یافتن راه‌حل‌های بهتر هدایت کند. خروجی نهایی این تابع، یک دیکشنری جامع شامل تمام متریک‌های کلیدی عملکرد است.

```
def get_full_metrics_for_solution(solution, tasks, num_cores):
    if len(solution) == 0: return {}
    core_assignments = {i: [] for i in range(num_cores)}
    for task_idx, core_idx in enumerate(solution):
        core_assignments[core_idx].append(tasks[task_idx])

    if not tasks: return {}
    periods = [t['period'] for t in tasks if t['period'] > 0]
    if not periods: return {}
    hyperperiod = reduce(lambda a, b: a * b // gcd(a, b) if a>0 and
    b>0 else a or b, periods, 1)

    per_task_qos_list, all_core_utils = [], [sum(t['utilization']
    for t in core_assignments[i]) for i in range(num_cores)]
    total_makespan = 0

    for core_id in range(num_cores):
        tasks_on_core = core_assignments[core_id]
        job_results, core_makespan =
        edf_schedule_on_core(tasks_on_core, hyperperiod)
        total_makespan = max(total_makespan, core_makespan)
        for task in tasks_on_core:
            if task['id'] in job_results and job_results[task['id']]
            ['finish_times']:
                qos_values = [calculate_qos(ft, dl) for ft, dl in
                zip(job_results[task['id']]['finish_times'],
                job_results[task['id']]['deadlines'])]
                avg_qos = np.mean(qos_values) if qos_values else 0
                per_task_qos_list.append({'id': task['id'], 'qos':
                avg_qos})
            else:
                per_task_qos_list.append({'id': task['id'], 'qos':
                0})

    per_task_qos_list.sort(key=lambda x: x['id'])
    system_qos_val = np.mean([item['qos'] for item in
    per_task_qos_list]) if per_task_qos_list else 0

    return {'core_utils': all_core_utils, 'makespan':
    total_makespan, 'per_task_qos': per_task_qos_list, 'system_qos':
    system_qos_val}
```

۶.۳ تابع اجرای شبیه‌سازی و مقایسه الگوریتم‌ها ($run\_simulation$)

این تابع، هماهنگ‌کننده کل فرآیند شبیه‌سازی در فاز دوم پروژه است. وظیفه این تابع، دیگر تنها اجرای یک الگوریتم برای یک سناریو نیست، بلکه اجرای یک آزمایش کامل و جامع برای مقایسه عملکرد دو الگوریتم ژنتیک و جستجوی فاخته تحت شرایط مختلف و به شیوه‌ای علمی و قابل اعتماد است. این تابع، تمام مراحل از تولید وظیفه تا اجرای الگوریتم‌ها و جمع‌آوری نتایج را مدیریت می‌کند.
روند اجرای این تابع به صورت یک ساختار تودرتو طراحی شده است. در لایه بیرونی، یک حلقه بر روی لیستی از «پیکربندی‌ها» ($configurations$) اجرا می‌شود. هر پیکربندی، یک سناریوی خاص را با تعداد مشخصی از هسته‌ها و میزان بهره‌وری هدف برای هر هسته تعریف می‌کند. این کار به ما اجازه می‌دهد تا عملکرد الگوریتم‌ها را در مقیاس‌ها و بارهای کاری متفاوت بسنجیم.

در لایه درونی، برای هر پیکربندی، یک حلقه دیگر به تعداد مشخصی (مثلاً ۳ بار) تکرار می‌شود. این تکرار برای افزایش اعتبار آماری نتایج ضروری است. در هر تکرار، یک «مجموعه وظیفه» ($task\ set$) کاملاً جدید و منحصربه‌فرد با استفاده از تابع $generate\_tasks$ تولید می‌شود. سپس، هر دو الگوریتم ژنتیک و جستجوی فاخته بر روی همین مجموعه وظیفه یکسان اجرا می‌شوند. این نکته بسیار حائز اهمیت است، زیرا تضمین می‌کند که مقایسه بین دو الگوریتم در هر اجرا کاملاً عادلانه باشد.

پس از آنکه هر الگوریتم بهترین راه‌حل خود را برای تخصیص وظایف پیدا کرد، تابع $get\_full\_metrics\_for\_solution$ فراخوانی می‌شود تا متریک‌های عملکردی دقیق (شامل QoS، $makespan$ و...) برای آن راه‌حل محاسبه شود. در نهایت، تمام نتایج خام هر اجرا (شامل لیست وظایف و متریک‌های محاسبه‌شده برای هر دو الگوریتم) در یک دیکشنری جامع با ساختاری منظم ذخیره می‌گردد. این دیکشنری خروجی، که حاوی داده‌های کامل برای تمام پیکربندی‌ها و تمام اجراهاست، به عنوان ورودی به توابع بصری‌سازی ارسال می‌شود تا نمودارهای مقایسه‌ای و تحلیلی پروژه تولید شوند.

```
def run_simulation(num_runs_per_config=3):
    configurations = [(8, 0.5), (8, 0.75), (8, 1.0),
                      (16, 0.5), (16, 0.75), (16, 1.0),
                      (32, 0.5), (32, 0.75), (32, 1.0)]
    results = {}
    for cores, util_per_core in configurations:
        print(f"Running Config: {cores} Cores, {util_per_core}
        Util/Core...")
        config_results = {'GA': [], 'CS': []}
        for i in range(num_runs_per_config):
            tasks = generate_tasks(num_tasks=int(4 * cores),
            total_utilization=cores * util_per_core)
            if not tasks: continue

            # Run Genetic Algorithm
            ga_scheduler = GeneticScheduler(tasks, cores)
            ga_assignment = ga_scheduler.evolve()
            if len(ga_assignment) > 0:
                config_results['GA'].append({
                    'tasks': tasks,
                    'metrics':
                    get_full_metrics_for_solution(ga_assignment,
                    tasks, cores)
                })

            # Run Cuckoo Search Algorithm on the same task set
            cs_scheduler = CuckooScheduler(tasks, cores)
            cs_assignment = cs_scheduler.run()
            if len(cs_assignment) > 0:
                config_results['CS'].append({
                    'tasks': tasks,
                    'metrics':
                    get_full_metrics_for_solution(cs_assignment,
                    tasks, cores)
                })

        results[(cores, util_per_core)] = config_results
    return results
```

۷.۳ کلاس زمان‌بند فاخته ($CuckooScheduler$)

این کلاس، پیاده‌سازی کامل الگوریتم جستجوی فاخته را برای مسئله تخصیص وظایف به هسته‌ها در خود جای داده است. همانطور که در بخش مبانی نظری تشریح شد، این الگوریتم با الهام از رفتار انگلی فاخته و با استفاده از مکانیزم قدرتمند پرواز لِوی، به دنبال یافتن راه‌حل بهینه می‌گردد. در ادامه، متدهای کلیدی این کلاس به تفصیل شرح داده می‌شوند.

۱.۷.۳ متد سازنده ($\_init\_$)

این متد، پارامترهای اصلی و تنظیمات الگوریتم جستجوی فاخته را مقداردهی اولیه می‌کند. این پارامترها رفتار و کارایی الگوریتم را در طول فرآیند جستجو کنترل می‌نمایند.

```
class CuckooScheduler:
    def __init__(self, tasks, num_cores, n_nests=25, pa=0.25,
    beta=1.5, generations=30):
        self.tasks = tasks; self.num_cores = num_cores; self.n_nests
        = n_nests
        self.pa = pa; self.beta = beta; self.generations =
        generations
        self.n_tasks = len(tasks); self.fitness_cache = {}
```

پارامترهای ورودی این متد عبارتند از:

- $tasks$ و num_cores: لیست وظایف و تعداد هسته‌ها.
- n_nests: تعداد لانه‌ها یا اندازه جمعیت راه‌حل‌ها.
- pa: احتمال کشف یا رها کردن بدترین لانه‌ها در هر نسل.
- $beta$: توان توزیع لِوی که شکل پروازها را کنترل می‌کند.
- $generations$: تعداد کل تکرارها یا نسل‌هایی که الگوریتم اجرا می‌شود.

علاوه بر این، یک حافظه پنهان ($fitness\_cache$) نیز برای ذخیره نتایج تابع برازندگی و جلوگیری از محاسبات تکراری ایجاد می‌شود.

۲.۷.۳ متد برازندگی ($\_fitness$)

این متد دقیقاً مشابه متد همنام خود در کلاس $GeneticScheduler$ عمل می‌کند. وظیفه آن، ارزیابی کیفیت یک لانه (یک راه‌حل) است. به دلیل هزینه محاسباتی بالای تابع $get\_solution\_fitness$ (که مبتنی بر شبیه‌سازی EDF است)، این متد ابتدا حافظه پنهان را بررسی می‌کند. اگر برازندگی راه‌حل ورودی قبلاً محاسبه شده باشد، مقدار آن مستقیماً از کش خوانده می‌شود؛ در غیر این صورت، محاسبه کامل انجام شده و نتیجه قبل از بازگردانده شدن، در کش ذخیره می‌گردد.

```
def _fitness(self, nest):
    nest_tuple = tuple(nest)
    if nest_tuple in self.fitness_cache: return
    self.fitness_cache[nest_tuple]
    fitness_val = get_solution_fitness(nest, self.tasks,
    self.num_cores)
    self.fitness_cache[nest_tuple] = fitness_val
    return fitness_val
```

۳.۷.۳ متد گام پرواز لِوی ($\_levy\_flight\_step$)

این متد، بخش ریاضیاتی و هسته اصلی تولید حرکات در الگوریتم فاخته است. همانطور که در بخش مبانی نظری به تفصیل توضیح داده شد، این متد با استفاده از الگوریتم مانتنیا، یک عدد تصادفی را تولید می‌کند که از توزیع پایدار لِوی پیروی می‌کند. این عدد که معرف اندازه گام است، دارای این ویژگی کلیدی است که اغلب کوچک بوده (برای جستجوی محلی) و گاهی بسیار بزرگ است (برای کاوش سراسری).

```
def _levy_flight_step(self):
    sigma_u = (gamma(1 + self.beta) * np.sin(np.pi * self.beta / 2)
    /
              (gamma((1 + self.beta) / 2) * self.beta * 2
              ((self.beta - 1) / 2)))  (1 / self.beta)
    u = np.random.normal(0, sigma_u, 1); v = np.random.normal(0, 1, 1)
    return u / (np.abs(v)  (1 / self.beta))
```

۴.۷.۳ متد اصلی اجرای الگوریتم (run)

این متد، حلقه اصلی و منطق کامل الگوریتم جستجوی فاخته را پیاده‌سازی می‌کند. فرآیند اجرای آن در هر نسل شامل دو استراتژی اصلی است: تولید راه‌حل جدید از طریق پرواز لِوی و جایگزینی بدترین راه‌حل‌ها.

```
def run(self):
    if not self.tasks or self.num_cores == 0: return []
    nests = [np.random.randint(0, self.num_cores, self.n_tasks) for
    _ in range(self.n_nests)]
    fitnesses = np.array([self._fitness(nest) for nest in nests])
    best_nest = nests[np.argmax(fitnesses)]; best_fitness =
    np.max(fitnesses)

    for _ in range(self.generations):
        # Generate a new cuckoo via Levy flights (Strategy 1)
        step = self._levy_flight_step()
        step_size = 0.01 * step * (best_nest -
        nests[random.randint(0, self.n_nests-1)])
        new_nest = nests[random.randint(0, self.n_nests-1)].copy()
        n_changes = min(int(np.linalg.norm(step_size)) + 1,
        self.n_tasks)
        indices_to_change = random.sample(range(self.n_tasks),
        n_changes)
        for idx in indices_to_change: new_nest[idx] =
        random.randint(0, self.num_cores - 1)

        f_new = self._fitness(new_nest)
        j = random.randint(0, self.n_nests - 1)
        if f_new > fitnesses[j]: nests[j], fitnesses[j] = new_nest,
        f_new
        if f_new > best_fitness: best_fitness, best_nest = f_new,
        new_nest

        # Abandon a fraction of worst nests (Strategy 2)
        n_abandon = int(self.pa * self.n_nests)
        if n_abandon > 0:
            sorted_indices = np.argsort(fitnesses)
            for k in range(n_abandon):
                idx_to_abandon = sorted_indices[k]
                nests[idx_to_abandon] = np.random.randint(0,
                self.num_cores, self.n_tasks)
                fitnesses[idx_to_abandon] =
                self._fitness(nests[idx_to_abandon])

    return nests[np.argmax(fitnesses)]
```

در ابتدا، جمعیت اولیه‌ای از لانه‌ها (راه‌حل‌ها) به صورت تصادفی ایجاد و ارزیابی می‌شود و بهترین راه‌حل ثبت می‌گردد. سپس در حلقه اصلی، ابتدا یک راه‌حل جدید با استفاده از پرواز لِوی تولید می‌شود. همانطور که پیشتر توضیح داده شد، اندازه گام لِوی به تعداد تغییرات در یک راه‌حل نگاشت می‌شود. پس از ارزیابی، این راه‌حل جدید با یک راه‌حل تصادفی دیگر از جمعیت مقایسه و در صورت بهتر بودن، جایگزین آن می‌شود. در گام بعدی، با احتمال pa، بدترین بخش جمعیت شناسایی شده و با راه‌حل‌های کاملاً جدید و تصادفی جایگزین می‌گردند. این فرآیند برای تعداد مشخصی از نسل‌ها تکرار شده و در نهایت، بهترین راه‌حل یافت‌شده در کل فرآیند به عنوان خروجی بازگردانده می‌شود.

۸.۳ توابع بصری‌سازی نتایج

پس از اجرای کامل شبیه‌سازی و جمع‌آوری داده‌های عملکرد برای هر دو الگوریتم، مرحله نهایی، تبدیل این داده‌های خام به نمودارهای معنادار و قابل تحلیل است. برای این منظور، دو تابع مجزا برای بصری‌سازی طراحی شده است که هر کدام هدف مشخصی را دنبال می‌کنند.

۱.۸.۳ تابع بصری‌سازی جزئیات (*visualize_detailed_results*)

هدف این تابع، ارائه یک تصویر عمیق و چندوجهی از عملکرد یک الگوریتم خاص است. این تابع یک دیکشنری کامل از نتایج و نام الگوریتم (مثلاً $'GA'$ یا $'CS'$) را به عنوان ورودی دریافت کرده و یک فایل تصویری شامل شش نمودار و جدول مجزا تولید می‌کند. این شش مولفه، شامل نمودارهای کیفیت خدمات وظایف، میانگین کیفیت خدمات سیستم، میانگین *makespan*، توزیع بهره‌وری هسته‌ها، نمودار میله‌ای زمان‌بندی‌پذیری و جدول مشخصات وظایف نمونه می‌باشند.

در برنامه اصلی، این تابع دو بار فراخوانی می‌شود: یک بار برای الگوریتم ژنتیک (و فایل *phase_two_GA_details.png* را تولید می‌کند) و یک بار برای الگوریتم جستجوی فاخته (و فایل *phase_two_CS_details.png* را تولید می‌کند). این به ما اجازه می‌دهد تا رفتار هر الگوریتم را به صورت جداگانه و با جزئیات کامل تحلیل کنیم.

```
def visualize_detailed_results(results, algo_name, filename):
```

```
    fig, axs = plt.subplots(3, 2, figsize=(18, 24))
    fig.suptitle(f'Detailed Results for {algo_name}', fontsize=20,
    y=0.95)
    # ... (Implementation for plotting 6 detailed graphs) ...
    plt.savefig(filename, bbox_inches='tight', dpi=150)
    plt.close()
```

۲.۸.۳ تابع بصری‌سازی مقایسه‌ای ($visualize\_comparison\_results$)

درحالی‌که نمودارهای جزئی برای تحلیل عمیق رفتار هر الگوریتم مفید هستند، ما به یک نمودار اصلی برای مقایسه مستقیم عملکرد دو الگوریتم در کنار یکدیگر نیاز داریم. تابع $visualize\_comparison\_results$ این وظیفه را بر عهده دارد.

این تابع، داده‌های کلیدی عملکرد (مشخصاً، میانگین کیفیت خدمات کل سیستم) را برای هر دو الگوریتم ژنتیک و جستجوی فاخته استخراج کرده و آن‌ها را بر روی یک محور مختصات واحد رسم می‌کند. در این نمودار، عملکرد هر الگوریتم برای تعداد هسته‌های مختلف با یک رنگ مشخص و با سبک خط متفاوت (مثلاً خط ممتد برای GA و خط‌چین برای CS) نمایش داده می‌شود. این نمودار به ما اجازه می‌دهد تا با یک نگاه سریع، به سوال اصلی پروژه پاسخ دهیم: کدام الگوریتم تحت کدام شرایط (تعداد هسته‌ها و بار کاری مختلف) عملکرد بهتری از خود نشان می‌دهد؟ این نمودار، خروجی اصلی و نتیجه‌گیری نهایی فاز دوم پروژه را به تصویر می‌کشد.

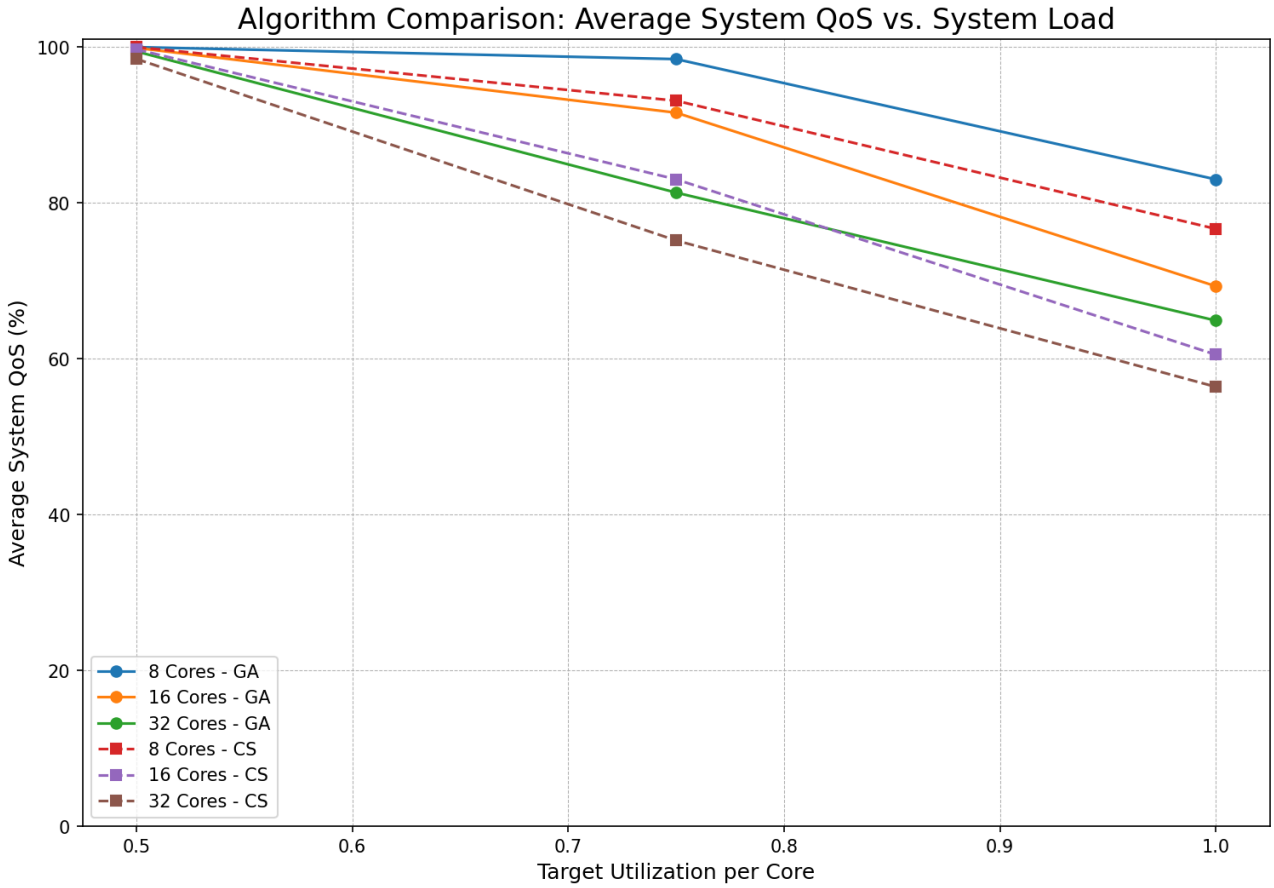
```
def visualize_comparison_results(results, filename):
    fig, ax = plt.subplots(1, 1, figsize=(12, 8))
    # ... (Implementation for plotting comparative graph) ...
    ax.set_title('Algorithm Comparison: Average System QoS vs.
    System Load', fontsize=16)
    ax.legend()
    plt.savefig(filename, bbox_inches='tight', dpi=150)
    plt.close()
```

۴ تحلیل و مقایسه نتایج فاز دوم

در این بخش، به تحلیل دقیق نتایج به‌دست‌آمده از شبیه‌سازی فاز دوم می‌پردازیم. هدف اصلی این فاز، مقایسه عملکرد دو الگوریتم بهینه‌سازی ژنتیک (GA) و جستجوی فاخته (CS) در حل مسئله زمان‌بندی وظایف بی‌درنگ بر روی پردازنده‌های چندهسته‌ای است. برخلاف فاز اول که ارزیابی بر اساس معیارهای تقریبی بود، در این فاز، کیفیت هر راه‌حل با استفاده از یک شبیه‌سازی دقیق زمان‌بندی EDF و یک تابع مطلوبیت خطی برای کیفیت خدمات (QoS) سنجیده شده است. این رویکرد، نتایجی بسیار واقع‌گرایانه‌تر و معنادارتر را در اختیار ما قرار می‌دهد. تحلیل ما بر اساس سه تصویر خروجی اصلی است: نمودار مقایسه‌ای کلی و دو نمودار جزئیات برای هر یک از الگوریتم‌ها.

۱.۴ تحلیل نمودار مقایسه‌ای اصلی ($Algorithm\ Comparison$)

اولین و مهم‌ترین نمودار، $figurename\ ref\{fig : comparison\}$، عملکرد دو الگوریتم را به صورت مستقیم در کنار یکدیگر نمایش می‌دهد. محور عمودی، میانگین کیفیت خدمات کل سیستم ($Average\ System\ QoS$) را نشان می‌دهد که معیار اصلی ما برای سنجش موفقیت یک الگوریتم است.



شکل ۶: نمودار مقایسه میانگین کیفیت خدمات سیستم بین الگوریتم ژنتیک (خطوط ممتد) و الگوریتم جستجوی فاخته (خطوط منقطع) تحت بارها و تعداد هسته‌های مختلف.

مشاهده کلیدی اول: برتری کلی الگوریتم ژنتیک با یک نگاه کلی به نمودار، واضح‌ترین نتیجه، عملکرد برتر الگوریتم ژنتیک (خطوط ممتد) نسبت به الگوریتم جستجوی فاخته (خطوط منقطع) در تمامی سناریوهای آزمایش‌شده است. برای هر تعداد هسته (۸، ۱۶ و ۳۲) و در هر سطح از بار کاری (بهره‌وری ۰/۵, ۰/۷۵, ۱/۰)، خطوط مربوط به GA همواره بالاتر از خطوط متناظر CS قرار دارند. این بدان معناست که GA به طور میانگین، توانسته است تخصیص‌های بهتری برای وظایف پیدا کند که منجر به از دست رفتن تعداد کمتری از موعدهای زمانی و در نتیجه، کیفیت خدمات بالاتری شده است.

مشاهده کلیدی دوم: تأثیر افزایش بار سیستم همانطور که انتظار می‌رود، با افزایش بار کاری سیستم (حرکت از چپ به راست بر روی محور افقی)، عملکرد هر دو الگوریتم کاهش می‌یابد. زمانی که بهره‌وری هدف برای هر هسته ۰/۵ است، هر دو الگوریتم عملکردی نزدیک به ایده‌آل (نزدیک به ۱۰۰٪) دارند، زیرا فضای جستجو برای یافتن یک راه‌حل معتبر بسیار بزرگ است. اما با افزایش بهره‌وری به ۰/۷۵ و سپس ۱/۰، پیدا کردن یک تخصیص که در آن هیچ هسته‌ای دچار سربار نشود، به شدت دشوارتر می‌شود. در این شرایط، افت عملکرد CS محسوس‌تر از GA است. برای مثال، در حالت ۳۲ هسته‌ای با بار ۱، GA به QoS حدود ۶۸٪ دست می‌یابد، در حالی که CS به سختی به QoS حدود ۵۶٪ می‌رسد. این نشان می‌دهد که GA در شرایط دشوار و تحت فشار، مقاوم‌تر عمل می‌کند.

مشاهده کلیدی سوم: مقیاس‌پذیری با افزایش تعداد هسته‌ها نکته جالب دیگر، نحوه واکنش دو الگوریتم به افزایش تعداد هسته‌ها (و به تبع آن، تعداد وظایف) است. در الگوریتم ژنتیک، خطوط مربوط به ۸، ۱۶ و ۳۲ هسته نسبتاً به یکدیگر نزدیک هستند و الگوی مشابهی را دنبال می‌کنند. این نشان‌دهنده مقیاس‌پذیری خوب ($scalability$) الگوریتم ژنتیک است. اما در الگوریتم جستجوی فاخته، با افزایش تعداد هسته‌ها، افت عملکرد بیشتر می‌شود. این ممکن است به این دلیل باشد که با بزرگ‌تر شدن فضای جستجو (تعداد بیشتر وظایف و هسته‌ها)، مکانیزم پرواز لِوی در CS برای همگرایی به سمت بهینه‌های باکیفیت، به نسل‌های بیشتری نیاز دارد و در تعداد نسل‌های محدود ما، عملکرد ضعیف‌تری از خود نشان می‌دهد.

۲.۴ تحلیل نمودارهای جزئی هر الگوریتم

برای درک عمیق‌تر دلایل این تفاوت عملکرد، به سراغ نمودارهای جزئی هر الگوریتم می‌رویم.

تحلیل توزیع بهره‌وری هسته‌ها مهم‌ترین نمودار برای درک تفاوت دو الگوریتم، هیستوگرام ($Overall\ Core\ Utilization\ Distribution$) است (نمودار وسط_راست در شکل ۷ و شکل ۸).

- **در الگوریتم ژنتیک:** این هیستوگرام به وضوح یک توزیع بسیار متمرکز و باریک را حول مقدار بهره‌وری ۰/۸ تا ۰/۹ نشان می‌دهد. تعداد بسیار کمی از هسته‌ها بهره‌وری کمتر از ۰/۵ یا بیشتر از ۱/۰ دارند. این نتیجه نشان می‌دهد که GA در متعادل‌سازی بار ($load\ balancing$) بسیار موفق عمل کرده است. عملگر ترکیب ($crossover$) به طور موثری وظایف را بین هسته‌ها توزیع می‌کند تا بهره‌وری آن‌ها به یکدیگر نزدیک شود.
- **در الگوریتم جستجوی فاخته:** هیستوگرام مربوط به CS پراکنده‌تر است. ما شاهد تعدادی بهره‌وری بسیار پایین (نزدیک به صفر) و همچنین تعداد بهره‌وری بالای ۱/۰ (حالت سربار) هستیم. این نشان می‌دهد که CS، با وجود قدرت کاوش بالای پروازهای لِوی، در تنظیم دقیق و متعادل‌سازی بار به اندازه GA موفق نبوده است. پرش‌های بلند لِوی ممکن است راه‌حل‌ها را به مناطق جدیدی از فضای جستجو پرتاب کنند، اما لزوماً منجر به توزیع یکنواخت بار نمی‌شوند.

این تفاوت در متعادل‌سازی بار، دلیل اصلی عملکرد بهتر GA است. وقتی بار به خوبی توزیع شود، احتمال سربار شدن هسته‌ها و از دست رفتن موعدهای زمانی کاهش یافته و در نتیجه، QoS کل سیستم افزایش می‌یابد.

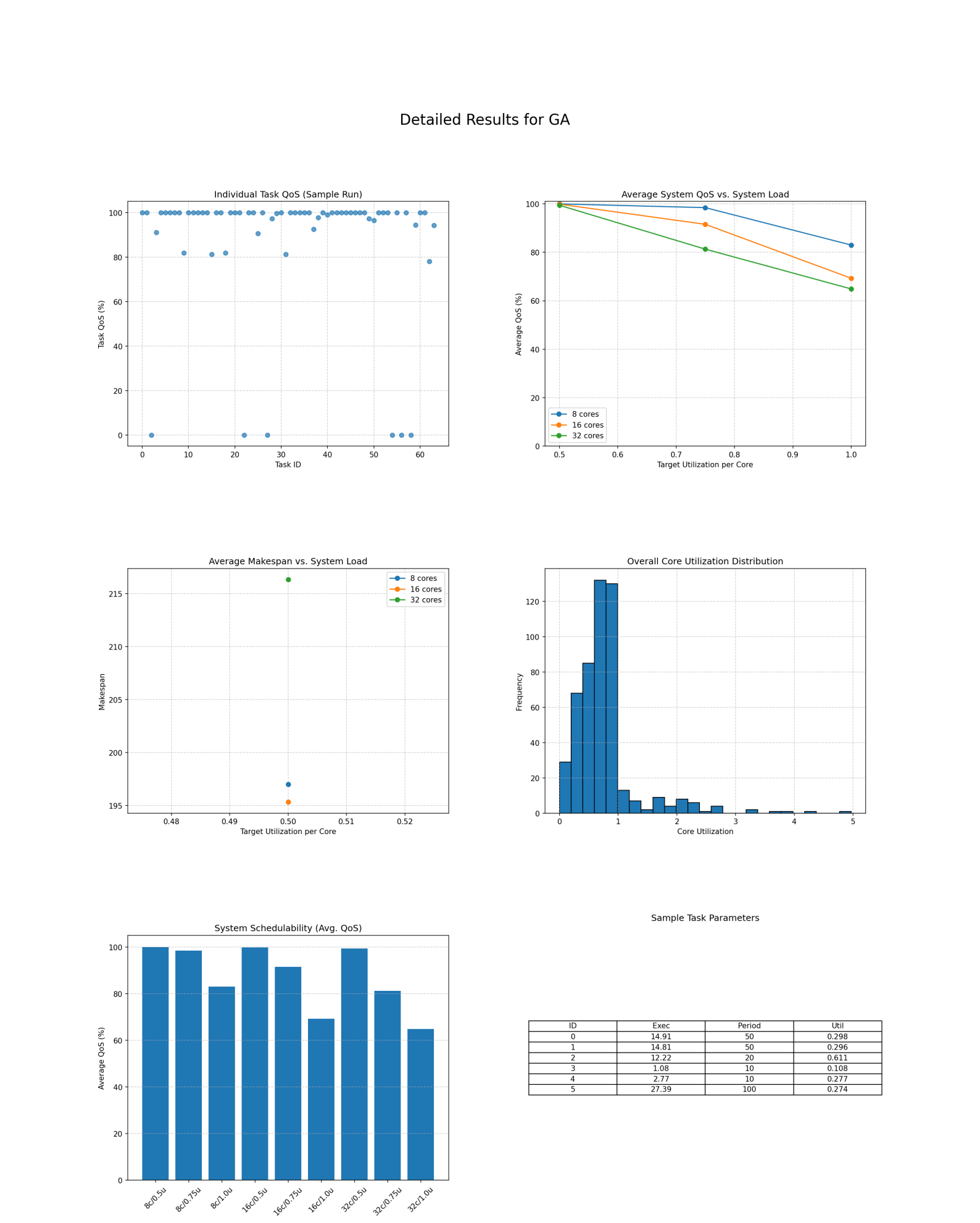
تحلیل سایر نمودارهای جزئی

- **کیفیت خدمات سیستم و زمان‌بندی‌پذیری:** نمودارهای $Average\ System\ QoS$ و $System\ Schedulability$ در هر دو تصویر، نتایج نمودار مقایسه‌ای اصلی را تأیید می‌کنند: افت تدریجی عملکرد با افزایش بار، و عملکرد کلی بهتر GA در تمام سناریوها.
- **Makespan:** نمودارهای $Average\ Makespan$ در هر دو تصویر روند کلی مورد انتظار یعنی افزایش $makespan$ با افزایش بار سیستم را تأیید می‌کنند.
- **کیفیت خدمات وظایف:** نمودار $Individual\ Task\ QoS$ برای یک اجرای نمونه، نشان می‌دهد که حتی در راه‌حل‌های بهینه، برخی وظایف ممکن است به QoS پایین‌تری دست یابند، به خصوص اگر در هسته‌های شلوغ قرار گرفته باشند. پراکندگی نقاط در این نمودار برای CS کمی بیشتر به نظر می‌رسد که با یافته‌های دیگر ما سازگار است.

۳.۴ نتیجه‌گیری نهایی تحلیل

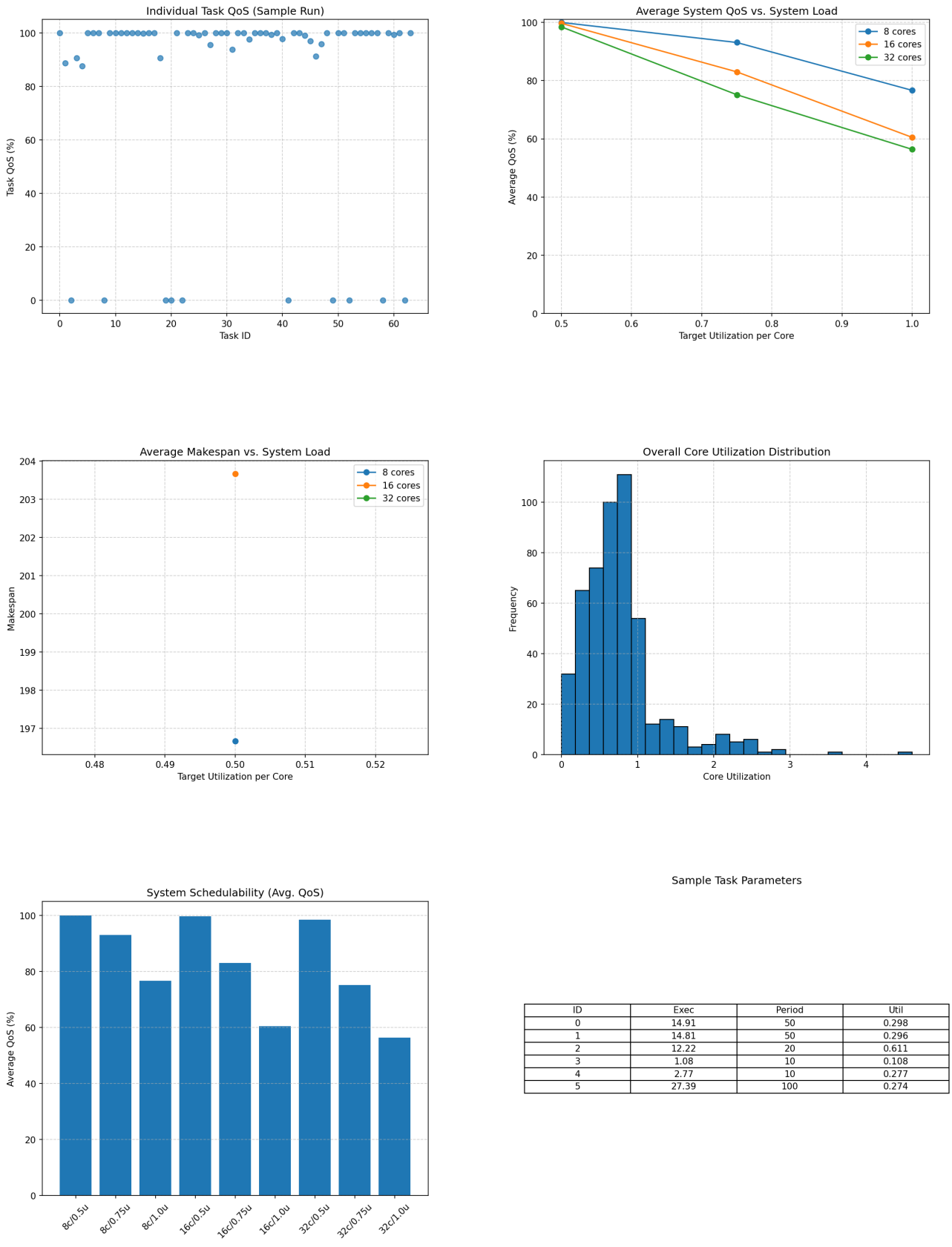
بر اساس تحلیل جامع نتایج، می‌توان نتیجه‌گیری کرد که برای مسئله زمان‌بندی وظایف بی‌درنگ بر روی پردازنده‌های چندهسته‌ای و تحت شرایط آزمایش‌شده در این پژوهش، الگوریتم ژنتیک عملکردی برتر و مقاوم‌تر نسبت به الگوریتم جستجوی فاخته از خود نشان می‌دهد.

دلیل اصلی این برتری، توانایی بالاتر الگوریتم ژنتیک در متعادل‌سازی بار میان هسته‌هاست. مکانیزم‌های انتخاب و ترکیب در GA به طور موثری راه‌حل‌هایی را ترویج می‌دهند که در آن‌ها وظایف به صورت یکنواخت توزیع شده‌اند. در مقابل، اگرچه الگوریتم جستجوی فاخته با مکانیزم پرواز لِوی خود یک کاوشگر قدرتمند است، اما به نظر می‌رسد این استراتژی به تنهایی برای دستیابی به توازن بار دقیق در این مسئله کافی نیست. این یافته نشان می‌دهد که برای مسائل تخصیص منابع مانند زمان‌بندی، صرفاً یافتن مناطق خوب در فضای جستجو کافی نیست، بلکه توانایی «تنظیم دقیق» راه‌حل‌ها برای برآورده کردن قیود چندگانه (مانند محدودیت بهره‌وری هر هسته) از اهمیت بالایی برخوردار است که الگوریتم ژنتیک در این زمینه موفق‌تر عمل کرده است.



ID	Exec	Period	Util
0	14.91	50	0.298
1	14.81	50	0.296
2	12.22	20	0.611
3	1.08	10	0.108
4	2.77	10	0.277
5	27.39	100	0.274

شکل ۷: نمودارهای تحلیل جزئی عملکرد برای الگوریتم ژنتیک (GA).



شکل ۸: نمودارهای تحلیل جزئی عملکرد برای الگوریتم جستجوی فاخته (CS).

مراجع

[1] X.-S. Yang and S. Deb, "Cuckoo search via Lévy flights," in *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pp. 210–214, IEEE, 2009.

[2] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992. (Originally published by University of Michigan Press, 1975).

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[4] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005. (This paper introduces the UUNIFAST algorithm).

[5] X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms, 2nd Edition*. Luniver Press, 2010.

[6] Ritika xRay Pixy, "Cuckoo Search Algorithm STEP-BY-STEP Explanation [1/4] xRay Pixy" *YouTube*, Jul. 29, 2020. [Online]. Available: `https://youtu.be/46we_zNBhKA?si=TxkkUMI8Wf4hBCXi`

Accessed on: Jul. 1, 2025.

[7] Ritika xRay Pixy, "Learn Cuckoo Search Algorithm Demo Example |Solved Step-by-Step|[2/4] xRay Pixy" *YouTube*, Aug. 4, 2020. [Online]. Available: `https://youtu.be/PuO0-udnCVY?si=SUy5zedOzLO3YY1s`

Accessed on: Jul. 1, 2025.

[8] Ritika xRay Pixy, "Numerical Example| Learn Cuckoo Search Algorithm Step-by-Step Explanation [3/4] xRay Pixy" *YouTube*, Aug. 8, 2020. [Online]. Available: `https://youtu.be/NP47NRpRLQo?si=TY-L-AaRvJVLXroW`

Accessed on: Jul. 1, 2025.

[9] Ritika xRay Pixy, "Levy Flight Numerical Example || Step-By-Step || xRay Pixy" *YouTube*, Apr. 18, 2024. [Online]. Available: `https://youtu.be/M-jnE9DNt-Q?si=N4gh5p4CHPnb6Utn`

Accessed on: Jul. 1, 2025.

[10] Gem WeBlog, "Optimization Techniques 2: Cuckoo Search algorithm | Metaheuristic algorithms" *YouTube*, Aug. 26, 2021. [Online]. Available: `https://youtu.be/H8xvxPk3vtI?si=XR4G6ewyDJbtmLWT`

Accessed on: Jul. 1, 2025.