مقدمه

هدف از برنامه نوشته شده پیدا کردن مقدار ماکزیمم تابع Q با استفاده از الگوریتم بهینه سازی فاخته است. الگوریتم فاخته مانند بقیه الگوریتم‌ها نیاز به یک ساختمان مشخص برای نشان دادن پاسخ‌ها(کروموزن در الگوریتم‌های ژنتیک) نیاز دارد. در اینجا این پاسخ‌ها به صورت آرایه‌ای به طول تعداد راس‌های گراف اصلی در نظر گرفته شده است که اگر عنصر iام مقدار j را داشته باشد یعنی این دور راس در یک گروه قرار دارند. بنابرین به نوعی از گراف پیچیده قبلی، درختی به عنوان درخت روابط اصلی بیرون می‌کشیم. عملگرهای مهمی که اینجا وجود دارند، محاسبه Q، تخم گذاری فاخته‌ها و مهاجرت فاخته‌ها به سمت قسمت‌های بهتر است.

کد

برای بخش‌های ذکر شده قبلی توضیحاتی درباره جزئیات پیاده سازی ارائه می‌دهیم.

تابع Q

همانگونه که از تعریف مشخص است، نمی‌توان به صورت مستقیم از روی آرایه‌ی کوروموزوم، تشخصی داد که آیا دو راس در یک گروه هستند یا خیر. برای همین ابتدا در ابتدای هر بار محاسبه این تابع، با استفاده از الگوریتم جستجوی اول عمق، DFS، گروه‌ها را به دست آورده و مقدار Q را محاسبه می‌کنیم. تمام این عملیات‌ها در Problem.py انجام شده‌اند و کد نسبتا واضحی دارند.

تولید تخم‌های اولیه تصادفی

ابتدا به سادگی می‌توان ثابت کرد که اگر درجه راسی صفر باشد آنگاه آزادیم که آن را در هر گروه دلخواهی قرار دهیم. بنابرین در صورتی که درجه راسی صفر باشد به راس ۰ وصل شده است در غیر این صورت به یکی از همسایه‌های خود که به صورت تصادفی انتخاب شده است. پیاده سازی این تابع در فایل RadomWalk قابل مشاهده است.

```
function randomChromosome()
  let x be a new chromosome
  for i in 0 ... n do
    if degree of i in graph == 0 do
      x.genomes[i] = 0
    else
      x.genomes[i] = select randomly from ajdacency list of i
    end
  end
  return x
end
```

تخم گذاری هر کدام از فاخته‌ها

در ابتدا تابعی داریم که به ازای یک فاخته مشخص، در فاصله خاصی از آن تخمی جدید می‌سازد. در این‌جا باید توجه کرد که در مقاله اصلی از نرم‌۲ به عنوان فاصله طبیعی بین دو جواب استفاده شده است در حالی که در این مثال به دلیل ترتیب ناپذیر بودن رئوس نمی‌توان این کار را انجام داد. برای مثال دو کرورموزون زیر را در نظر بگیرید.

$$X = \begin{pmatrix} ۰ & ۱ & ۲ & ۳ \\ ۱ & ۰ & ۱ & ۱ \end{pmatrix} \qquad X = \begin{pmatrix} ۰ & ۱ & ۲ & ۳ \\ ۲ & ۰ & ۳ & ۱ \end{pmatrix}$$

به سادگی می‌توان دید که هر دو کروموزون بالا یک افراز از راس‌ها را نشان می‌دهند و معادل هم هستند. در حالی که می‌توان دید نرم‌۲ آنها عددی مخالف صفر است. به همین دلیل در اینجا فاصله دو کروموزون به صورت زوج‌های i و jی تعریف شده که در یکی از کروموزون‌ها در یک گروه واقع شده‌اند و در دیگری در گروه متفاوتی واقع‌اند. با این حساب برای تولید کردن تخم‌هایی در فاصله L می‌توان $\sqrt{L}$ تا از کوروموزون‌ها را تغییر داد. یا به عبارت دیگر

```
function ChromosomeAtL(c, l)
  let x be a copy of c
  for _ in 0 ... sqrt(l) do
    i = select a random vertex from graph
    x.genomes[i] = randomly select from i`s neighbours
  end
```

```
  return c
end
```

می‌توان به سادگی دید که جایگزین کردن L و $\sqrt{L}$ تفاوتی در اصل الگوریتم ایجاد نمی‌کند و حتی با تغییر پارامتر آلفا(بعدا توضیح داده می‌شود) می‌توان به طور کامل اثر این تغییر را خنثی کرد.
تابع GenerateWithELR از فایل RandomWalk برای پیاده‌سازی توضیحات بالا استفاده شده است.

مهاجرت فاخته‌ها

در مهاجرت یک فاخته به سمت فاخته دیگر، مانند قبل می‌توان با استفاده از شبه‌ـنرم تعریف شده فاصله دو فاخته را تخمین زد. بعد از تخمین فاصله آن‌ها با تابع احتمال یکنواخت کسری از آن را انتخاب می‌کنیم و فاخته را به سمت بهترین فاخته موجود حرکت می‌دهیم. برای حرکت دادن فقط بعضی از ژن‌های کروموزون فاخته هدف را در فاخته اولیه جایگذاری می‌کنیم. همانطور که از مقاله اصلی توضیح داده شده است به مقدار نویز، انحراف، در حرکت دادن فاخته‌ها وجود دارد که برای اعمال آن، در برخی مراحل به صورت تصادفی بجای ژن‌های فاخته هدف، از یکی از همسایگان راس به صورت تصادفی استفاده می‌کنیم.

```
function getDiff(c1, c2)
  diff = 0
  for i in 0 .. n do
    for j in 0 .. n do
      if c1[i] == c1[j] && c2[i] != c[j] ||  c2[i] == c2[j] && c1[i] != c1[j] do
        diff  += 1
      end
    end
  end
  return sqrt(diff)
end

function Immigrate(x,y)
  xc, yc = communities of x and y repectively
  numberOfGenomesToChange = getDiff(xc, yc) as int

  for _ in 0 ... numberOfGenomesToChange do
    j = randomly selected vertex
    if (a random uniform number) < deviation ratio then
      x.genomes[j] = randomly selected vetex from adjcency list of j
    else
      x.genomes[j] = y.genomes[j]
    end
  end
  reuturn x
end
```

این توضیحات در فایل RandomWalk و تحت تابع Immigrate پیاده سازی شده‌اند.

الگوریتم فاخته

الگوریتم به صورت ساده شده در هر تکرار شامل مراحل زیر است

۱. تخم گذاری فاخته‌ها به این صورت که هر فاخته بین ۵ تا ۲۰ تخم در فاصله مشخصی از خود می‌گذارد. این فاصله مشخص از روی ابعاد مسئله و نسبت تخم‌های فاخته به کل تخم‌ها و همچنین پارامتر آلفا مشخص می‌شود. با افزایش آلفا می‌توان شعاع تخم گذاری فاخته‌ها را افزایش داد. برای مثال در قسمت قبل دیدیم که می‌توان در پیاده سازی تابع تخم گذاری از L یا $\sqrt{L}$ استفاده کنیم. در اینجا می‌توان دید که با تغییر آلفا می‌توان اثر پیاده‌سازی مختلف آن تابع را خنثی کرد. البته باید توجه داشت با توجه به غیر خطی بودن تابع جذر، اثر به طور کلی خنثی نمی‌شود و به هر حال در پیاده‌سازی با $\sqrt{L}$ شعاع تخم گذاری فاخته‌ها از واریانس کمتری برخوردار است.

۲. پیدا کردن بهترین تخم‌ها و از بین بردن تخم‌هایی که به اندازه کافی خوب نیستن.

۳. مهاجرت فاخته‌ها به سمت بهترین جواب مسئله.

این مراحل به صورت کامل در تابع cuckooSearch در فایل CuckooOptimization پیاده شده اند.

اجرای کد

ورودی‌ها

اسکریپ Console.py انتظار دارد گراف مسئله از طریق فایل graph.txt در کنار فایل اسکریپ فراهم شده باشد. خط اول ورودی تعداد راس‌ها و خط‌های بعدی هر کدام یک یال را باید معرفی کنند.

خروجی

جدا از لاگ‌های برنامه، در آخر بهترین کروموزون، اجتماعات استخراج شده در خروجی چاپ خواهند شد. همچنین گراف رنگ‌آمیزی شده با استفاده از تجمع‌های مختلف با فرمت dot در فایل graph.dot نوشته می‌شود که با ابزارهایی مثل graphviz می‌توان آن را به نمایش گذاشت.
در آخر نیز روند پیشرفت الگوریتم و بهترین Q در هر مرحله به صورت نمودار نمایش داده می‌شود.