الگوریتم ژنتیک برای تشخیص اجتماع

مقدمه

این قسمت پیاده سازی الگوریتم ژنتیک برای تشخیص اجتماع‌های گرافی دلخواه را تشریح می‌دهد. برای اینکار کروموزوم‌ها نشان دهنده یک تقسیم بندی برای گراف در نظر گرفته‌ایم و الگوریتم ژنتیک باید با تغییر این کروموزوم‌ها تابع بهینگی Q را بیشینه کند.

برای استفاده از الگوریتم‌های ژنتیک باید توابع زیر را برای مسئله تعریف کنیم:

1. جمیت اولیه
2. تابع برگزیدگی هر عضو
3. تابع جهش یک کروموزوم
4. ترکیب دو کروموزوم

با فرض تعریف درست توابع بالا، قسمت اصلی الگوریتم ژنتیک به صورت زیر خواهد بود(*genetic.py*):

```
def genetic(populationSize, maxGenerations, mutationProb, crossoverProb):
    population = # initial population
    for _ in range(maxGenerations):
        if rand() < crossoverProbability:
            for i in range(populationSize):
                j = random.randint(0, populationSize)
                child = # result of crossover over population[i] and  population[j]
                population = np.vstack((population, child)) # append to population

            for i in range(len(population)):
                if rand() < mutationProbability:
                    child = # result of mutation over population[i]
                    population = np.vstack((population, child))# append to population
        population = # sort population by fitness
        population = population[:populationSize]
    return population[0]
```

در قسمت‌های بعد هر کدام از توابع بالا را به طور کامل توضیح خواهم داد.

کروموزوم‌ها

کروموزم‌ها در تعریف‌های بالا نقش مهمی دارند برای همین ابتدا نوع نمایش جواب‌ها به صورت کروموزوم را نشان می‌دهیم. ژن‌های هر کروموزم با مفهمومی به نام یال موثر ارتباط دارند. در هر کروموزوم به تعداد یال‌های گراف اصلی ژن وجود دارد که مقدار باینری دارند. اگر مقدار یک ژن برابر با صفر باشد یعنی یال متناظر با آن در گراف موثر نیست. در غیر این صورت آن یال موثر است. مفهموم موثر بودن را در اینجا می‌توان اینگونه تفسیر کرد:

اگر یال (i, j) موثر باشد آنگاه دو راس i و j در یک تجمع (community) قرار دارند.

با این تعریف هر تجمع مولفه همبندی گرافی است که با یال‌های موثر ساخته می‌شود. از آنجایی که تجمع‌ها ارتباط تنگاتنگی با یال‌ها دارند این روش می‌تواند برای تشخیص تجمع مفید باشد. مثلا اگر دو قسمت از گراف اصلی متصل نباشند انتظار داریم که در دو تجمع مجزا باشند، در همین حین این روش تعریف کروموزوم تضمین می‌کند که در گروه‌های مجزا قرار بگیرند. یا در حالت‌هایی که دو راس به طور مستقیم به هم وصل نیستند، هر چه تعداد راس‌های میانی آن‌ها بیشتر باشد، احتمال قرار گرفتن آن‌ها در یک تجمع کاهش می‌یابد. در این نوع نمایش دادن کروموزوم نیز می‌توان دید که تعداد کروموزوم‌هایی که دو راس مورد نظر را در دو مولفه همبندی مختلف قرار می‌دهند افزایش می‌یابد.

برای اینکه بتوانیم از کروموزوم‌های تعریف شده تجمع‌ها را بیرون کشید باید مولفه‌های همبندی گرافی که از حذف کردن یال‌های ناموثر گراف اصلی به دست می‌آید را حساب کنیم. برای اینکه با استفاده از یال‌ها گراف را تشکیل داده (*contruct_tree* از فایل *chromosome.py*) و با استفاده از الگوریتم جستجوی اول عمق (BFS)، مولفه‌ها را استخراج می‌کنیم (*connected_componnets* از فایل *chromosome.py*). درباره نحوه پیاده‌سازی این دو الگوریتم با توجه به اینکه پیاده‌سازی رایجی دارند توضیحی داده نمی‌شود.

تابع برازندگی

تابع برازندگی یا Q در اینجا طبق تابع داده شده پیاده شده است. با توجه به اینکه نمایش کروموزوم‌ها مولفه‌های لازم برای محاسبه Q را شامل نمیشود از تابع‌های کمکی *components_map* و *connected_components* برای استخراج تجمع‌ها استفاده میکنیم و بعد از آن قطعه کد زیر مقدار Q را محاسبه می‌کند.

```
q = 0
for i in range(n):
    for j in range(n):
        if groups[i] == groups[j]:
            q += A[i][j] - len(tree[i]) * len(tree[j]) / (2 * m)
Q = q / (2 * m)
return Q
```

جمعیت اولیه

برای ساختن جمعیت اولیه به صورت تصادفی تعدادی از یال‌ها را به عنوان یال موثر در نظر میگیرم. در اینجا هم می‌توان دید که این روش با نوع تعریف کروموزوم‌ها سازگاری دارد. اگر یال‌ها را به صورت تصادفی موثر یا ناموثر در نظر بگیریم آن گاه امید ریاضی یال‌های موثر، نصف تعداد کل یال‌ها خواهد بود. در گراف‌های چگال برای مثال، در کروموزوم‌ها تعداد زیادی یال وجود دارد که باعث می‌شود اندازه تجمع‌ها بزرگتر باشد و تعداد آن‌ها کمتر، که همان چیزی است که از گراف چگال مد نظر داشتیم. برای گراف‌های خلوت نیز استدلالی مشابه صدق می‌کند و نشان می‌دهد که این روش انتخاب تصادفی در عین سادگی، جواب‌های اولیه نسبتا مناسبی دارد. برای مثال در گراف نمونه داده شده جمعیت اولیه تصادفی با اندازه ۳۰ به طور میانگین روی ۱۰۰ تست، تابع برازش مقدار ۰.۳۳ را در بهترین عضو جمعیت اولیه دارد که نسبت به جواب بهینه که برازندگی ۰.۴۳ دارد بسیار تخمین خوبی است.

برای پیاده سازی این الگوریتم از کد زیر استفاده شده است(*initial_population*):

```
def initial_population(m, popSize):
    return np.random.randint(0, 2, (popSize, m))
```

جهش هر کروموزوم

برای جهش دادن کروموزوم تعدا از ژن‌های آن را تغییر می‌دهیم، یعنی صفرها را یک و یک‌ها را صفر می‌کنیم. برای اینکار می‌توان از XOR استفاده کرد. از آنجایی که این تابع جواب‌های نسبتا دوری از مبدا تولید می‌کند در اوایل استفاده از الگوریتم می‌تواند سرعت جستجو را افزایش دهد ولی در پایین به همین دلیل باعث می‌شد که بیشتر بار محاسباتی باشد بنابرین مقدار جهش‌ها در هر ۱۰ مرحله از الگوریتم ۰.۷ برابر خواهند شد.

کار جهش توسط تابع ***mutation*** انجام می‌شود.

```
def mutation(m, X):
    newHabitat = np.copy(X)
    n = np.random.randint(0, m)
    indices = np.random.randint(0, m, n)
    newHabitat[indices] = np.bitwise_xor(X[indices], 1)
    return newHabitat
```

ادغام دو کروموزوم

برای ترکیب با معنی‌ترین ترکیب مد نظر همان ترکیب تک یا چند نقطه‌ای است که باعث می‌شود قسمتی از گراف از تقسیم بندی والد اول و بقیه آن از والد دوم استفاده کند.

اما این روش به ترتیب یال‌ها وابسته‌است و می‌تواند در مراحلی جواب‌هایی دور از والدها تولید کند که مد نظر نیست. برای از بین بردن اثر ترتیب یال‌ها از ترکیب یکنواخت(uniform) استفاده می‌کنیم. در اینجا هم به طور احتمالی نصف یال‌ها از والد اول و بقیه از والد دوم به ارث برده می‌شوند اما دیگر ترتیب مانند قبل اهمیتی ندارد. می‌توان با انجام مکرر عمل تولید ادغام در هر مرحله از الگوریتم دید که تقریبا این روش روی گراف نمونه داده شده در حدود ۶۵ درصد مواقع بهتر از چند نقطه‌ای عمل می‌کند. تابع انجام دهنده این عمل به شکل زیر است:

```
def crossover(m, X, Y):
    return np.random.randint(0, 2, m) * (X - Y) + Y
```

اجرای برنامه

برای اجرای برنامه باید فایل *main.py* را اجرا کنید. ورودی‌های مسئله ابتدا فایل حاوی گراف با فرمتی شبیه به فرمت مثال داده‌شده است و در ادامه پارامترهای الگوریتم نیز باید وارد شوند.

برنامه در هر مرحله گراف را با بهترین کروموزوم جمعیت آن مرحله رنگ‌آمیزی می‌کند و بهترین کروموزوم و تجمع‌های متناظر با آن و همچنین مقدار تابع برازندگی را نیز به عنوان خروجی چاپ می‌کنند.

نیازمندی‌ها

برای اجرای الگوریتم‌های اصلی فقط نیاز به کتابخوانه *numpy* دارید اما از آنجایی که فایل *main.py* علاوه بر اجرای الگوریتم‌ها گراف بهترین کروموزوم و نمودار پیشفرفت الگوریتم را هم ترسیم میکنند به کتابخوانه‌های *matplotlib* و *networkx* نیاز است و باید نصب شده باشند.

بهبودها

هم اکنون می‌توان دید که بخشی از محاسبات مربوط به جهش و ادغام عملیات‌های خطی هستند و می‌توانند به شکل برداری در بیایند تا سرعت بسیار بیشتری به دست آید. به همین جهت در هر مرحله *population* به شکل لیست پایتونی بلکه به شکل ماتریس *numpy* ارائه شده است و فقط کافیست تا عملیات‌های *mutation* و *crossover* را نیز به صورت برداری تعریف کنیم.

با توجه به هزینه محاسبه تابع Q که از بقیه قسمت‌های الگوریتم بیشتر است می‌توان از caching استفاده کرد تا به ازای هر کروموزوم این مقدار فقط یکبار این تابع محاسبه شود.