تمرین سری 4 هوش

بکتاش انصاری

99521082

سوال تئوری :

آ)

برای حل این مسئله به روش الگوریتم ژنتیک ابتدا نیاز است که ما کروموزوم های خود را مشخص کنیم. (جامعه)

که در ابتدای مسئله نیاز است جامعه اولیه مشخص شود. برای این کار باید یک روش Encoding معرفی کنیم که مسئله مدل سازی و ساده شود.

طبق گراف هر عضو جامعه اولیه 15 نود از گراف را شامل میشود. برای رنگ آمیزی هر کروموزوم اگر ما از بین این 15 نود، نودی که بیشترین همسایه (درجه) را دارد پیدا کنیم به طور یقین میتوان گفت این گراف با [درجه این نود + 1] رنگ متفاوت رنگ آمیزی میشود.

پس ابتدا به این اندازه رنگ را Encode میکنیم (به هر رنگ عددی اختصاص میدهیم.) و رنگ های تخصیص داده شده به هر نود را داخل لیست ذخیره میکنیم.(Index لیست شماره نود و عدد داخل هر خانه ی لیست رنگ تخصیص داده شده است.)

برای مثال در گراف عدد بزرگترین درجه این گراف 7 میباشد که با 8 (1 تا 8) رنگ میتوان گراف را رنگ آمیزی کرد که برای جامعه اولیه به صورت رندوم 6 لیست شامل اعداد یک تا 8 برای 15 نود گراف ایجاد میکنیم.

جامعه اولیه :

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 3, 3, 1, 2]

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 4]

[7, 7, 4, 7, 5, 3, 7, 4, 5, 1, 6, 8, 4, 3, 4]

[2, 6, 5, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6]

[7, 5, 7, 8, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7]

[1, 7, 1, 2, 2, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7]

ب)

برای تابع fitness هر کروموزوم باید تابعی معرفی کنیم که هر چه مقدار آن کمتر باشد مارا به جواب نزدیک تر میکند. برای این کار این تابع را طوری تعریف میکنیم که تعداد Conflict های رنگ بین نود های هر کروموزوم را محاسبه کند.

به طوری که اگر هر دو نود گراف که همسایه هستند دارای یک رنگ باشند مقدار را بعلاوه یک کند. که در انتها این تابع جمع این مقادیر را به ما برمیگرداند. که هر چه این مقدار به صفر نزدیک تر باشد تعداد Conflict ها کمتر است. حال این مقدار را برای تمام اعضای جامعه انجام میدهیم و آن مقادیر را به صورت صعوری سورت میکنیم که **هر چه مقدار برای کروموزومی کمتر باشد عضو بهتری از جامعه است و نسل آن باید ادامه پیدا کند.**

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 3, 3, 1, 2] ==> 12

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 4] ==> 11

[7, 7, 4, 7, 5, 3, 7, 4, 5, 1, 6, 8, 4, 3, 4] ==> 14

[2, 6, 5, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6] ==> 13

[7, 5, 7, 8, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7] ==> 14

[1, 7, 1, 2, 2, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7] ==> 11

Fitness = [11,11,12,13,14,14]

کد این بخش :

```
arr = [
    [1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 3, 3, 1, 2],
    [7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 4],
    [7, 7, 4, 7, 5, 3, 7, 4, 5, 1, 6, 8, 4, 3, 4],
    [2, 6, 5, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6],
    [7, 5, 7, 8, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7],
    [1, 7, 1, 2, 2, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7]
    ]
for member in arr :
    fit = 0
    for i in range(15) :
        for j in range(i+1,15) :
            if member[i]==member[j] :
                fit+=1
    print(member,"==>",fit)
```

ج)

حال در این مرحله نسل اعضای جامعه قبلی را ادامه میدهیم. برای ساخت بچه های آن ها از روی خودشان از دو روش mutation و crossover استفاده میکنیم.

(Crossover

در این روش دو تا از والد ها را با هم ترکیب میکنیم و دو فرزند تولید میکنیم. به طوری که یک ایندکس از لیست را انتخاب میکنیم. و اعضای قبل آن ایندکس را از لیست اول و اعضای بعد آن را از لیست دوم انتخاب میکنیم و فرزند اول را میسازیم و برعکس همین کار را برای ساخت فرزند دوم انجام میدهیم.

[1,2,5,2,4]

[5,3,1,3,4]

⇨ [5,3,1,2,4]

⇨ [1,2,5,3,4]

(Mutation

در این روش یک ایندکس به صورت رندوم از کروموزوم انتخاب میشود که رنگ آن عوض میشود. این روش زمانی مناسب است که قصد داریم تنها یک Conflict را از بین ببریم.

[1,2,5,2,4] --> [1,2,5,3,4]

که برای نسل اول داریم :

crossover :

[1, 1, 2, 8, 5, 7, 6, 4, 2, 4, 5, 3, 8, 1, 4]

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 3, 3, 3, 4]

[1, 1, 2, 8, 5, 7, 6, 6, 7, 2, 3, 7, 7, 1, 6]

[1, 1, 2, 8, 5, 7, 6, 6, 6, 8, 4, 7, 6, 2, 7]

[1, 1, 2, 8, 5, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7]

[7, 3, 2, 3, 5, 3, 7, 4, 5, 1, 6, 8, 4, 3, 4]

[7, 3, 2, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6]

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 7]

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 4, 5, 7]

[7, 7, 5, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6]

[7, 7, 4, 7, 5, 3, 7, 1, 7, 8, 4, 7, 6, 2, 7]

[7, 7, 4, 7, 5, 3, 7, 4, 5, 1, 3, 2, 4, 5, 7]

[2, 6, 5, 2, 7, 4, 1, 6, 7, 2, 4, 7, 6, 2, 7]

[2, 6, 5, 2, 7, 4, 1, 6, 7, 3, 3, 2, 4, 5, 7]

[7, 5, 7, 8, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7]

mutation :

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 5, 3, 1, 2]

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 1]

[7, 7, 4, 7, 5, 3, 7, 6, 5, 1, 6, 8, 4, 3, 4]

[2, 6, 5, 5, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6]

[7, 5, 7, 7, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7]

[1, 4, 1, 2, 2, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7]

کد این بخش :

```
#crossover
next_generation1 = []
for i in range(len(arr)) :
    for j in range(i+1,len(arr)) :
        index = randrange(15)
        next_generation1.append(arr[i][:index]+arr[j][index:])

#mutation
next_generation2 = []
for member in arr :
    index = randrange(15)
    color = randrange(1,9)
    while(color == member[index]) :
        color = randrange(1,9)
    new_member = copy.deepcopy(member)
    new_member[index] = color
    next_generation2.append(new_member)
```

د)

حال با توجه به روش های تولید نسل باید از روی اعضای جامعه قبلی یک جامعه جدیدی تولید کنیم. مقداری از جامعه جدید را توسط روش mutation و مقداری را توسط روش crossover ایجاد میکنیم. که این کار را انجام دادیم.

و حال دوباره فرآیند محاسبه تابع fitness را روی جامعه جدید اجرا میکنیم. اگر روش های تولید نسل ما درست باشند به طور تدریجی باید مقدار توابع fitness ما طی چند نسل کمتر شده و ما به مقدار بهینه نزدیک تر شویم.

که در نسل جدید با محاسبه تابع fitness داریم :

[1, 1, 2, 8, 5, 7, 6, 4, 2, 4, 5, 3, 8, 1, 4] ==> 9

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 3, 3, 3, 4] ==> 11

[1, 1, 2, 8, 5, 7, 6, 6, 7, 2, 3, 7, 7, 1, 6] ==> 13

[1, 1, 2, 8, 5, 7, 6, 6, 6, 8, 4, 7, 6, 2, 7] ==> 12

[1, 1, 2, 8, 5, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7] ==> 8

[7, 3, 2, 3, 5, 3, 7, 4, 5, 1, 6, 8, 4, 3, 4] ==> 11

[7, 3, 2, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6] ==> 16

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 7] ==> 9

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 4, 5, 7] ==> 12

[7, 7, 5, 2, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6] ==> 18

[7, 7, 4, 7, 5, 3, 7, 1, 7, 8, 4, 7, 6, 2, 7] ==> 22

[7, 7, 4, 7, 5, 3, 7, 4, 5, 1, 3, 2, 4, 5, 7] ==> 17

[2, 6, 5, 2, 7, 4, 1, 6, 7, 2, 4, 7, 6, 2, 7] ==> 16

[2, 6, 5, 2, 7, 4, 1, 6, 7, 3, 3, 2, 4, 5, 7] ==> 10

[7, 5, 7, 8, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7] ==> 14

[1, 1, 2, 8, 5, 7, 6, 6, 6, 6, 7, 5, 3, 1, 2] ==> 12

[7, 3, 2, 3, 1, 6, 4, 4, 2, 4, 5, 3, 8, 1, 1] ==> 10

[7, 7, 4, 7, 5, 3, 7, 6, 5, 1, 6, 8, 4, 3, 4] ==> 12

[2, 6, 5, 5, 7, 4, 1, 6, 7, 2, 3, 7, 7, 1, 6] ==> 12

[7, 5, 7, 7, 1, 4, 2, 1, 7, 8, 4, 7, 6, 2, 7] ==> 18

[1, 4, 1, 2, 2, 7, 6, 4, 1, 3, 3, 2, 4, 5, 7] ==> 11

که در مقادیر جدید مقدار های 8 و 9 و 10 نیز دیده میشود که نشان میدهد نسل جدید ما عملکرد بهتری نسبت به نسل قبلی نشان داده است.

این فرآیند را تکرار میکنیم تا به نسلی برسیم که دارای ژنی است که مقدار تابع fitness آن برابر صفر است که با رسیدن به این مقدار متوجه میشویم با 8 رنگ میتوان این گراف را رنگ کرد. حال باید دوباره الگوریتم ژنتیک را برای 7 رنگ پیاده سازی کنیم. این فرآیند را انقدر تکرار میکنیم تا در عددی به جواب نمیرسیم که عدد بزرگتر از آن عدد را به عنوان جواب اعلام میکنیم.

گزارش بخش عملی

تابع value :

در این تابع ما چک میکنیم که در استیت مورد نظر چند تا تهدید بصورت مستقیم یا غیر مستقیم داریم.

که برای این کار کافیست چک کنیم که در tuple مورد نظر برای تهدید های عمودی چند تا از جفت وزیر ها دارای ایندکس برابر هستند و برای تهدید ضربدری چک میکنیم اختلاف ایندکس تاپل چند تا از جفت وزیر ها با اختلاف ایندکس خود دو وزیر برابر است.

تابع goal_test :

در این تابع چک میکنیم که آیا state فعلی جواب مسئله هست یا نه که برای این کار باید چک کنیم که در state مورد نظر آیا وزیری وزیر دیگر را تهدید میکند یا نه که برای این کار Value آن استیت را توسط تابع value محاسبه میکنیم.

تابع neighbors :

در این تابع همسایه های هر state را خروجی میدهیم که برای اینکار عدد هر خانه تاپل را با توجه به مقیاس های جدول عوض کنیم و در لیستی ذخیره کنیم که تعداد همسایه های هر استیت برابر است با N*(N-1)

تابع hill_climbing :

در این تابع ما باید الگوریتم hill climbing را اجرا کنیم به گونه ای که هر بار همسایه های یک state را پیدا کرده و آن همسایه ای که مقدار کمتری دارد را خروجی میدهیم. که این کار را با توجه به توابعی که پیاده سازی کردیم انجام میدهیم.