باسمه تعالی

دانشگاه صنعتی شریف

دانشکده مهندسی برق

درس سیستم های توزیع شده

تمرین سری پنجم

علی محرابیان96102331

استاد: دکتر صالح

تابستان1399

در این گزارش به شرح کامل تمرین می پردازیم.

در ابتدا کتابخانه های مورد نیاز را import کرده و فایل result را که خروجی نهایی در آن ذخیره می شود، اگر از قبل در محیط کار ما باشد،پاک می کنیم.سپس ورژن پایتون pyspark را به ورژن 3 که ورژن محیط کار است، تغییر می دهیم.

```
import shutil ##we mustn't have result file at first

try:
    shutil.rmtree('./results')
    print('Deleted the \'results\' directory')
except:
    pass
```

```
os.environ['PYSPARK_PYTHON'] = '/usr/bin/python3'
```

سپس تمامی داده ها را به 4 پارتیشن تقسیم بندی کرده و همچنین پوشه checkpoint را برای ذخیره rdd های ضروری ایجاد می کنیم.ورودی های مطرح شده در مسئله را نیز در متغیر های مذکور ذخیره می کنیم.

```
conf = SparkConf()
spark = SparkSession.builder.master('local[4]').appName('ranking').getOrCreate()
sc = spark.sparkContext
sc.setCheckpointDir("./checkpoints")

lines = sc.textFile(sys.argv[1],4) # Inputs "dataset.txt"
ap = float(sys.argv[2]) # Inputs are always in string form
num = int(sys.argv[3])
beta = float(sys.argv[4])
```

از تابع parse برای جدا کردن گره های مبدا از گره های مقصد و وزن ها استفاده می کنیم.از تابع های node و Nod برای ذخیره تمامی گره های موجود در مسئله استفاده می کنیم.از تابع filter برای نرمالیزه کردن وزن های خروجی از گره ها استفاده می کنیم.

```
def parse(x):#for parsing source node and its datas
    r=x.split()
    id=r[0]
    data=r[1:]
    return id,data


def node(x):
    r=x.split()
    f=r[1]
    return f

def nod(x):
    r=x.split()
    f=r[0]
    return f


def filterr(l):#filter and just keeping outbound wieghts and normalizing weights

    x1=l[0]
    x2=l[1][::2]
    c=len(x2)
    t=0

    for i in range(c):
        t=t+float(x2[i][1])

    for i in range(c):

        x2[i][1]=float(x2[i][1])/t



    return x1,x2
```

دو rdd مهمی که در ادامه از آنها استفاده خواهیم کرد،rank و use هستند.rank،رتبه های اولیه گره های مسئله هستند.use، گره های مبدا به همراه وزن های نرمالیزه شده و گره های خروجی هستند.چون از use بسیار استفاده خواهیم کرد،به کمک cache آن را در memory ذخیره می کنیم.

```
##
rank=n3.map(lambda y:[y,1/a])#initial ranks
use=link.join(rank).reduceByKey(lambda x,y:x+y).map(lambda r:filterr(r)).cache()
#nodes with outband weights,here we use cache because we need this rdd later
```

حال به توضیح روش استفاده شده در کد می پردازیم.رتبه هرگره در هر مرحله به کمک رابطه زیر به دست می آید.

$$r_{t+1} = (1-\alpha)\left(\sum_j r_j w_{Nj} + \sum_i \frac{r_i}{n-1}\right) + \alpha\frac{\sum_k r_k}{n}$$

در داخل پرانتز،ترم اول نشان دهنده گره هایی است که از قبل وزن تعریف شده دارند که تعداد آن ها کم است. ترم دوم هم برای گره هایی است که وزن تعریف شده از قبل ندارند.برای پیاده سازی،ابتدا فرض می کنیم که همه گره ها از جنس ترم دوم بوده و جمع مذکور را روی همه گره ها انجام می دهیم.حال با توجه به اطلاعاتی که در Use ذخیره کردیم،برای هر مورد که وزن خروجی وجود دارد،ابتدا ترم دومی را که قبلا جمع زدیم، کم کرده و سپس مقدار مربوطه از عبارت اول را اضافه می کنیم.در نهایت هم با توجه به مقدار α،رتبه جدید هر گره را برای مرحله بعد حساب می کنیم.

تابع compute،موارد گفته شده در صفحه قبل را توضیح می دهد.

در این جا از yield برای ذخیره خروجی استفاده می کنیم که حالت تابع را پس از خروجی در هر مرحله نگاه می دارد و سپس از آخرین yield قبلی شروع به ادامه کار می کند.

```
def compute(x):
    id=x[0]
    x1=x[1][0][0]
    x2=x[1][0][1]
    x3=x[1][1]
    c=len(x1)

    for i in range(c):

        if x1[i][0]!=id:

            yield(x1[i][0],x3*x1[i][1]-x2)

        else:

             yield(x1[i][0],x3*x1[i][1])
```

در هر مرحله نیز موارد مورد نیاز را محاسبه می کنیم و سپس π را در هر مرحله با قبلی مقایسه می کنیم.در صورتی که از β کمتر باشد،آن را ذخیره کرده و از حلقه خارج می شویم.

```
    for j in range(num):

        o=rank.map(lambda x:(x[0],x[1]*(1/(a-1)))).map(lambda x:x[1]).sum()

        h=rank.map(lambda x:(x[0],o-x[1]*(1/(a-1))))

        p=rank.map(lambda x:(x[0],(x[1]*(1/(a-1)))))

        y=use.join(p).join(rank).flatMap(lambda x:compute(x)).reduceByKey(add)

        pi=y.join(h).map(lambda x:(x[0],(1-ap)*(x[1][0]+x[1][1])+(o*ap*(1-(1/a)))))
        pi.checkpoint()
        if pi.join(rank).map(lambda x:abs(x[1][0]-x[1][1])).sum() <= beta:#for comp

            rank=pi
            break;

        rank=pi
```

یکی از مواردی که قصد پیاده سازی آن را داشته ولی موفق نشدیم،استفاده از ضرب خارجی برای ضرب ماتریس G در π می باشد.به طور مثال برای محاسبه ماتریس خروجی زیر،از ضرب خارجی استفاده می کنیم.

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 28 & 18 \\ 11 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 9 & 18 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 9 & -15 \\ 3 & -5 \end{bmatrix} + \begin{bmatrix} 10 & 15 \\ 4 & 6 \end{bmatrix}$$

چون در ماتریس H ما،تعداد زیادی از خانه ها در هر سطر مقدار 0 دارند،در نتیجه محاسبه ماتریس های اولیه ساده است و خروجی به دست می آید.

در نهایت فایل های result و checkpoint ساخته می شود و فایل های مدنظر داخل آن ها ذخیره می شوند.

در باب سوال آخر،می توان گفت که حالت بهینه این است که ارتباط پارتیشن ها با یکدیگر کم باشد چرا که جمع آوری داده از پارتیشن های جدا از هم کمتر است.داشتن موضوعات یکسان،لزوما به این معنا نیست که وبسایت ها به همدیگر اشاره می کنند.در واقع پارتیشن های ما باید به گونه ای باشند که افراد داخل یک پارتیشن،بیشترین ارجاع به هم را داشته باشند.