به نام خدا

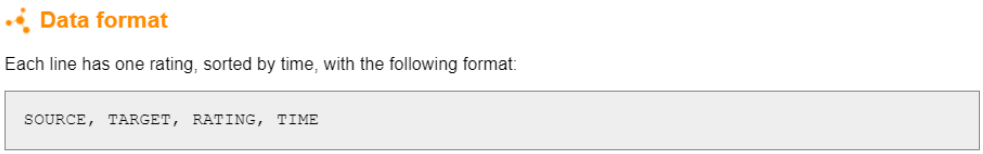
پروژه ی نهایی درس طراحی الگوریتم – اعضای گروه : فاطمه قیدی ، هلیا سادات محمدی

نکات مربوط به بخش اول پروژه:

معرفی کلی :

با دانلود گرافی وزن دار (weighted graph) از سایت
(https://snap.stanford.edu/data/index.html#socnets) که شامل مجموعه ای از مجموعه داده ها برای انواع مختلف شبکه های اجتماعی است که توسط پروژه تحلیل شبکه استنفورد (SNAP) ارائه شده است ، باید الگوریتم یافتن MST را روی آن اجرا کرده و هزینه درخت کمینه اعلام شود .

فرمت فایل CSV مربوط به گراف :



کد نهایی ارائه شده :

```
import pandas as pd
import networkx as nx
import time

 #Step 1: Read the CSV file
file = pd.read_csv('soc-sign-bitcoinotc.csv')

 #Step 2: Extract columns as source, target, and weight
source = file.iloc[:, 0].tolist() # Extract first column data as source
target = file.iloc[:, 1].tolist() # Extract second column data as target
weights = file.iloc[:, 2].tolist() # Extract third column data as weights
```

```
#Step 3: Create a graph
G = nx.Graph()
for i in range(len(source)):
    G.add_edge(source[i], target[i], weight=weights[i])

#Step 4: Compute Minimum Spanning Tree
start_time = time.time()
mst = nx.minimum_spanning_tree(G)
total_cost = sum([edge[2]['weight'] for edge in mst.edges(data=True)])
execution_time = time.time() - start_time
print(f"Total Cost of Minimum Spanning Tree: {total_cost}")
print(f"Execution Time: {execution_time} seconds")
```

معرفی کتابخانه های مورد استفاده در این بخش :

کتابخانه هایی که در این بخش از پروژه منظور شده اند عبارتند از :

```
import pandas as pd
import networkx as nx
import time
```

۱. کتابخانه ی Pandas : پانداز برای تغییر و تجزیه و تحلیل داده ها استفاده می شود. در این کد برای خواندن یک فایل CSV و استخراج ستون ها از دیتافریم مورد استفاده قرار میگیرد.
۲. کتابخانه ی Networkx : برای ایجاد، دستکاری و مطالعه شبکه های پیچیده است. در این کد برای ایجاد یک گراف و محاسبه MST استفاده می شود.
۳. کتابخانه ی Time : ماژول Time برای اندازه گیری زمان اجرای یک بلوک کد خاص استفاده می شود. در این کد برای محاسبه مدت زمان محاسبه MST استفاده می شود.

چهار قدم برای پیاده سازی و اجرای کد این بخش منظور گردیده است :

قدم اول : خواندن فایل CSV :

```
# Step 1: Read the CSV file
file = pd.read_csv('soc-sign-bitcoinotc.csv')
```

به منظور خواندن CSV از کتابخانه ی پانداز از دستور pd.read_csv('file_name') استفاده میشود . به این نکته توجه شود که فایل CSV حتما در سورس فایل import شود.

قدم دوم : استخراج ستون ها به عنوان source , target و weight :

```
# Step 2: Extract columns as source, target, and weight
source = file.iloc[:, 0].tolist() # Extract first column data as source
target = file.iloc[:, 1].tolist() # Extract second column data as target
weights = file.iloc[:, 2].tolist() # Extract third column data as weights
```

این خطوط کد، داده ها را از ستون های خاصی از یک فایل CSV استخراج می کنند و آن داده ها را به لیست های پایتون تبدیل می کنند.

`source = file.iloc[:, 0].tolist()`:

این خط داده ها را از ستون اول (ایندکس صفر) فایل CSV با استفاده از روش iloc استخراج می کند.

سپس متد tolist() فراخوانی می‌شود تا داده‌های ستون اول را به لیست تبدیل کند و آن را در متغییر source ذخیره کند .

`target = file.iloc[:, 1].tolist()`:

این خط مشابه خط اول است اما داده ها را از ستون دوم (ایندکس ۱) فایل CSV استخراج می کند و آن را به عنوان لیست پایتون در متغیر target ذخیره می کند.

`weights = file.iloc[:, 2].tolist()`:

این خط داده ها را از ستون سوم (ایندکس ۲) فایل CSV استخراج می کند و آن را به لیست پایتون تبدیل می کند که سپس آن را به عنوان متغییر weight ذخیره مینماید.

قدم سوم : ساخت گراف :

```python
# Step 3: Create a graph
G = nx.Graph()
for i in range(len(source)):
    G.add_edge(source[i], target[i], weight=weights[i])
```

این قطعه کد وظیفه ایجاد یک گراف با استفاده از کتابخانه NetworkX در پایتون را بر عهده دارد.

`G = nx.Graph()`:

این خط نمونه ای از یک گراف به نام G را با استفاده از تابع nx.Graph () از کتابخانه NetworkX ایجاد می کند.

`for i in range(len(source)):`:

این خط یک حلقه for را شروع می کند که روی ایندکس های های متغییر source تکرار می شود. تابع len(source) تعداد عناصر موجود در لیست source را برمی‌گرداند که محدوده لوپ را تعیین می‌کند.

`G.add_edge(source[i], target[i], weight=weights[i])`:

در هر تکرار لوپ ، این خط یک یال به گراف G با پارامترهای زیر اضافه می کند:

source[i]: گره source برای لبه، از فهرست source در ایندکس i گرفته شده است.

target[i]: گره target برای لبه، از فهرست target در ایندکس i گرفته شده است.

weight=weights[i]: وزن یال، از فهرست weight ها در ایندکس i گرفته شده است. این برای نشان دادن قدرت بین گره های منبع و هدف استفاده می شود.

قدم چهارم : محاسبه ی MST :

```python
# Step 4: Compute Minimum Spanning Tree
start_time = time.time()
mst = nx.minimum_spanning_tree(G)
total_cost = sum([edge[2]['weight'] for edge in mst.edges(data=True)])
execution_time = time.time() - start_time
print(f"Total Cost of Minimum Spanning Tree: {total_cost}")
print(f"Execution Time: {execution_time} seconds")
```

این قطعه کد مسئول محاسبه MST گراف G است که قبلاً با استفاده از NetworkX ایجاد شده بود.

`start_time = time.time()`:

این خط زمان فعلی را درست قبل از شروع محاسبه MST ثبت می کند. این مقدار برای محاسبه کل زمان اجرا استفاده خواهد شد.

`mst = nx.minimum_spanning_tree(G)`:

این خط از تابع nx.minimum_spanning_tree(G) از NetworkX برای محاسبه MST گراف G استفاده می کند. MST محاسبه شده در متغیر mst ذخیره می شود.

`total_cost = sum([edge[2]['weight'] for edge in mst.edges(data=True)])`:

این خط با جمع‌آوری وزن تمام یال‌ها در گراف mst، هزینه کل MST را محاسبه می‌کند. روی هر یال در MST تکرار می‌شود، وزن هر یال را بازیابی می‌کند و آن‌ها را برای محاسبه هزینه کل جمع می‌کند.

`execution_time = time.time() - start_time`:

این خط کل زمان اجرای بلوک کدی را که MST را محاسبه می کند را بدست می آورد . زمان شروع ثبت شده در ابتدای محاسبه را از زمان فعلی کم می کند تا کل زمان صرف شده برای محاسبه تعیین شود.

و در نهایت مقادیر total_cost و execution_time برای نمایش در غالب خروجی کد با دستور print ، پرینت میشوند.

به طور کلی، این قطعه کد به طور موثر MST گراف G را محاسبه می کند، هزینه کل MST را محاسبه می کند و زمان اجرای محاسبات را گزارش می دهد و امکان تجزیه و تحلیل بیشتر داده های شبکه را بر اساس این نتایج فراهم می کند.

در نهایت خروجی کد خواهد بود :

```python
import pandas as pd
import networkx as nx
import time

# Step 1: Read the CSV file
file = pd.read_csv('soc-sign-bitcoinotc.csv')

# Step 2: Extract columns as source, target, and weight
source = file.iloc[:, 0].tolist() # Extract first column data as source
target = file.iloc[:, 1].tolist() # Extract second column data as target
weights = file.iloc[:, 2].tolist() # Extract third column data as weights

# Step 3: Create a graph
G = nx.Graph()
for i in range(len(source)):
    G.add_edge(source[i], target[i], weight=weights[i])

# Step 4: Compute Minimum Spanning Tree
start_time = time.time()
mst = nx.minimum_spanning_tree(G)
total_cost = sum([edge[2]['weight'] for edge in mst.edges(data=True)])
execution_time = time.time() - start_time
print(f"Total Cost of Minimum Spanning Tree: {total_cost}")
print(f"Execution Time: {execution_time} seconds")
```

```
Total Cost of Minimum Spanning Tree: -5174
Execution Time: 0.1575925350189209 seconds
```

نکات مربوط به بخش دوم پروژه:

معرفی کلی :

در این بخش گراف را خوشه بندی می کنیم و به صورت موازی درخت MST را روی هر خوشه اجرا می کنیم. در نهایت هزینه ی درخت کمینه ی هر خوشه را اعلام و زمان اجرا را ثبت می کنیم.

کد نهایی ارائه شده :

```
import csv
import timeit
import threading
import networkx as nx
from sklearn.cluster import KMeans


# Read the CSV file
csv_file_path = 'Soc-sign-bitcoinotc.csv'
source_col = []
target_col = []
rating_col = []


with open(csv_file_path, mode='r') as file:
    csv_reader = csv.reader(file)
    next(csv_reader)  # Skip the header row
    for row in csv_reader:
        source_col.append(row[0])
        target_col.append(row[1])
        rating_col.append(int(row[2]))  # Convert rating to integer
```

```
# Create a feature matrix (combine source and target columns)
feature_matrix = [[s, t] for s, t in zip(source_col, target_col)]

# Perform K-means clustering
n_clusters = 100
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(feature_matrix)

# Assign cluster labels to each data point
for i, label in enumerate(cluster_labels):
    source_col[i] = f"{source_col[i]}_cluster{label}"
    target_col[i] = f"{target_col[i]}_cluster{label}"

# Compute MST for each cluster
def compute_mst(cluster_source, cluster_target, cluster_rating):
    G = nx.Graph()
    for s, t, w in zip(cluster_source, cluster_target, cluster_rating):
        G.add_edge(s, t, weight=w)

    mst = nx.minimum_spanning_tree(G)
    return mst

# Create threads for parallel processing
```

```
threads = []
start_time = timeit.default_timer()

for i in range(n_clusters):
    thread = threading.Thread(target=compute_mst,
args=(source_col[i::n_clusters], target_col[i::n_clusters], rating_col[i::n_clusters]))
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()
end_time = timeit.default_timer()
execution_time = end_time - start_time
print(f"Execution time: {execution_time:.4f} seconds")

# Calculate cost for each cluster
cluster_costs = {}
for i in range(n_clusters):
    cluster_ratings = [r for r, label in zip(rating_col, cluster_labels) if label == i]
    cluster_costs[f"Cluster {i}"] = sum(cluster_ratings)

# Print cluster costs
for cluster, cost in cluster_costs.items():
    print(f"{cluster}: Cost = {cost}")
```

معرفی کتابخانه های مورد استفاده در بخش دوم :

```
import csv
import timeit
import threading
import networkx as nx
from sklearn.cluster import KMeans
```

۱. کتابخانه ی csv : csv یک فرمت استاندارد برای قالب‌بندی محتوای متنی است. معمولاً داده‌های ساختاریافته و جدولی را می‌توان به شکل بسیار ساده‌ای در این فرمت قرار داد. هر خط از فایل، یکی از رکوردهای اطلاعاتی ماست و داده‌ها در هر خط با علامت یکسانی جدا شده‌اند. کتابخانه ی csv عملکردی را برای خواندن و نوشتن در فایل‌های csv فراهم می‌کند. برای کار کردن با فایل های csv تولید شده توسط Excel طراحی شده است، اما به راحتی برای کار با انواع فرمت های csv سازگار است. این کتابخانه حاوی اشیاء و کدهایی برای خواندن، نوشتن و پردازش داده ها از فایل های csv است.

۲. کتابخانه ی timeit : با استفاده از ماژول timeit می توانید به راحتی زمان اجرای یک فرآیند را در کد خود اندازه گیری کنید.

۳. کتابخانه ی threading : در حوزه‌ی برنامه نویسی، یک نخ (thread) کوچک ترین واحد اجرایی مستقل است. یک برنامه با استفاده از نخ‌ها می‌تواند به صورت همزمان به اجرای چندین عملیات در یک فضای فرایند بپردازد. توانایی یک فرایند برای اجرای چندین نخ به صورت موازی را می‌توان چند نخی (multithreading) نامید. کتابخانه‌ی threading در پایتون ابزارهای لازم برای مدیریت و همگام‌سازی تاپیک‌ها (threads) را ارائه می‌دهد.

۴. کتابخانه ی networkx : این کتابخانه یک کتابخانه‌ی متن‌باز و پرکاربرد برای تحلیل و شبیه‌سازی شبکه‌ها و گراف‌ها در پایتون است. با استفاده از این کتابخانه، شما می‌توانید گراف‌های مختلف را تعریف، ساخت، تحلیل و ویرایش کنید.

۵. کتابخانه ی Scikit Learn : sklearn.cluster از کتابخانه‌های متن‌باز، مفید، پرکاربرد و قدرتمند در زبان برنامه‌نویسی پایتون است که برای اهداف یادگیری ماشین به کار می‌رود. این کتابخانه ابزارهای کاربردی زیادی به منظور یادگیری ماشین و مدل‌سازی آماری داده‌ها همچون طبقه‌بندی (classification)، رگرسیون، خوشه‌بندی و کاهش ابعاد فراهم می‌کند.

قدم هایی که برای پیاده سازی و اجرای کد این بخش منظور گردیده است :

قدم اول : خواندن فایل csv ، استخراج داده های هر ستون و قرار دادن هر یک از آنها در یک لیست :

```
# Read the CSV file
csv_file_path = 'soc-sign-bitcoinotc.csv'
source_col = []
target_col = []
rating_col = []

with open(csv_file_path, mode='r') as file:
    csv_reader = csv.reader(file)
    next(csv_reader)  # Skip the header row
    for row in csv_reader:
        source_col.append(row[0])
        target_col.append(row[1])
        rating_col.append(int(row[2]))
```

به منظور این کار برای هر ستون یک لیست ایجاد می کنیم. سپس با دستور with open و با قرار دادن mode روی r (read) اطلاعات هر ردیف را می خوانیم. اطلاعات ستون اول (row[0]) را به عنوان source، اطلاعات ستون دوم (row[1]) را به عنوان target و اطلاعات ستون سوم (row[2]) را به عنوان rating ذخیره می کنیم.

قدم دوم : ساخت ماتریس ویژگی بر پایه ی ترکیب source و target :

```
# Create a feature matrix (combine source and target columns)
feature_matrix = [[s, t] for s, t in zip(source_col, target_col)]
```

با این خط کد در واقع یک ماتریس (feature_matrix) ساخته می شود که یک لیست از لیست هاست. در هر لیست داخلی دو عنصر قرار دارد، یکی s متناظر با source_col و یکی t متناظر با target_col.

قدم سوم : خوشه بندی K-means :

```
# Perform K-means clustering
n_clusters = 100
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(feature_matrix)
```

در این قسمت تعداد خوشه ها را ۱۰۰ تا تعیین کرده ایم و با استفاده از روش K-means عمل clustering را روی گراف انجام داده ایم.

الگوریتم K-means یک مجموعه داده را به K خوشه مجزا تقسیم می کند. هر خوشه نشان‌دهنده گروهی از نقاط داده‌ای است که شباهت‌های مشترکی دارند و امکان بینش معنادار و کشف الگو را فراهم می‌کنند. الگوریتم خوشه‌بندی-k میانگین از گروه روش‌های خوشه‌بندی تفکیکی (Partitioning Clustering) محسوب می‌شود. در خوشه‌بندی-k میانگین از بهینه‌سازی یک تابع هدف (Object Function) استفاده می‌شود. پاسخ عملیات خوشه‌بندی، پیدا کردن خوشه‌هایی است که فاصله بین اشیاء هر خوشه کمینه باشد. در مقابل، اگر از تابع مشابهت (Dissimilarity Function) برای اندازه‌گیری مشابهت اشیاء استفاده شود، تابع هدف را طوری انتخاب می‌کنند که پاسخ خوشه‌بندی مقدار آن را در هر خوشه بیشینه کند.

پارامتر random_state با ثابت کردن random seed تکرار پذیری را تضمین می کند.

پارامتر n_init تعداد دفعاتی را که الگوریتم با مرکز ثقل های اولیه مختلف اجرا می شود را مشخص می کند. بهترین نتیجه (کمترین اینرسی) را انتخاب می کند.

kmeans.fit_predict(feature_matrix) :

مدل K-means را با ماتریس ویژگی مطابقت می دهد و هر نقطه داده را به یک خوشه اختصاص می دهد.

قدم چهارم : اختصاص دادن برچسب های خوشه ای به هر نقطه داده :

```
# Assign cluster labels to each data point
for i, label in enumerate(cluster_labels):
    source_col[i] = f"{source_col[i]}_cluster{label}"
    target_col[i] = f"{target_col[i]}_cluster{label}"
```

این قطعه کد برای تغییر نام یا قالب بندی مجدد مقادیر ستون های source_col و target_col بر اساس برچسب های cluster اختصاص داده شده آنهاست.

با افزودن پسوند cluster{label} مقادیر ستون جدیدی ایجاد می شود که نشان می دهد هر نقطه داده متعلق به کدام خوشه است.

قدم پنجم : محاسبه کردن MST برای هر خوشه :

```
# Compute MST for each cluster
def compute_mst(cluster_source, cluster_target, cluster_rating):
    G = nx.Graph()
    for s, t, w in zip(cluster_source, cluster_target, cluster_rating):
        G.add_edge(s, t, weight=w)

    mst = nx.minimum_spanning_tree(G)
    return mst
```

در خط اول تابع compute_mst یک گراف خالی به نام G با استفاده از networkx ایجاد می کند.

در داخل حلقه، لبه هایی به گراف اضافه می شود. s راس مبدا، t راس مقصد و w وزن لبه را نشان می دهند.

قدم ششم : ایجاد کردن threads برای پردازش موازی :

```
# Create threads for parallel processing
threads = []
start_time = timeit.default_timer()

for i in range(n_clusters):
    thread = threading.Thread(target=compute_mst, args=(source_col[i::n_clusters],
             target_col[i::n_clusters], rating_col[i::n_clusters]))
    threads.append(thread)
    thread.start()
```

در داخل حلقه یک رشته جدید با استفاده از threading.Thread() ایجاد می شود.

Target رشته بر روی تابع compute_mst تنظیم شده است. پارامتر args آرگومان هایی را برای ارسال به compute_mst مشخص می کند.

قدم هفتم : صبر کردن برای تمام شدن تمام threads و چاپ کردن زمان اجرای عملیات :

```
# Wait for all threads to finish
for thread in threads:
    thread.join()

end_time = timeit.default_timer()
execution_time = end_time - start_time
print(f"Execution time: {execution_time:.4f} seconds")
```

در داخل حلقه thread.join() فراخوانی می شود. این کار برای این است که برای عملیات چند پردازشی قبل از اینکه حلقه به تکرار بعدی برود، منتظر می ماند تا رشته مشخص شده اجرای خود را به پایان برساند.

قدم هشتم : محاسبه هزینه برای هر خوشه :

```
# Calculate cost for each cluster
cluster_costs = {}
for i in range(n_clusters):
    cluster_ratings = [r for r, label in zip(rating_col, cluster_labels) if label == i]
    cluster_costs[f"Cluster {i}"] = sum(cluster_ratings)
```

قدم نهم : پرینت کردن هزینه ی هر خوشه:

```
# Print cluster costs
for cluster, cost in cluster_costs.items():
    print(f"{cluster}: Cost = {cost}")
```

نکات مربوط به قسمت سوم پروژه :

معرفی کلی این قسمت :

این قسمت ادامه ی قسمت دوم پروژه میباشد ، در این قسمت باید درخت های MST باهم مرج شوند و سپس final cost درخت MST باید حساب شود.

کد نهایی ارائه شده که در ادامه ی کد نهایی قسمت دوم پروژه قرار میگیرد :

```
# Create a function to merge two MSTs based on their lowest weighted edge
def merge_msts(msts):
    final_mst = nx.Graph()
    for mst in msts:
        for edge in mst.edges(data=True):
            source, target, weight = edge
            if final_mst.has_edge(source, target):
                if weight['weight'] < final_mst[source][target]['weight']:
                    final_mst[source][target]['weight'] = weight['weight']
            else:
                final_mst.add_edge(source, target, weight=weight['weight'])
    return final_mst


# Merge the resulting MST trees togethermst_tree = []
for i in range(n_clusters):
    mst = compute_mst([source_col[j] for j in range(i, len(source_col), n_clusters)],
                      [target_col[j] for j in range(i, len(target_col), n_clusters)],
                      [rating_col[j] for j in range(i, len(rating_col), n_clusters)])
    mst_trees.append(mst)
```

final_mst = merge_msts(mst_trees)

#Calculate cost of the final MST

final_cost = sum(rating_col)

print(f"Final MST cost: {final_cost}")

#Print complete execution time

print(f"Complete execution time: {execution_time:.4f} seconds")

چون این قطعه کد در ادامه ی بخش دوم قرار میگیرد ، پس کتابخانه های مورد استفاده در آن یکی یکی در بخش قبل معرفی شده اند.

چهار قدم برای پیاده سازی و اجرای کد این بخش منظور گردیده است :

قدم اول : ادغام کردن دو MST براساس کمترین وزن آنها :

```
# Create a function to merge two MSTs based on their lowest weighted edge
def merge_msts(msts):
    final_mst = nx.Graph()
    for mst in msts:
        for edge in mst.edges(data=True):
            source, target, weight = edge
            if final_mst.has_edge(source, target):
                if weight['weight'] < final_mst[source][target]['weight']:
                    final_mst[source][target]['weight'] = weight['weight']
            else:
                final_mst.add_edge(source, target, weight=weight['weight'])
    return final_mst
```

این تابع چندین MST را با در نظر گرفتن لبه های هرکدام و انتخاب یکی با کمترین وزن برای هر یال منحصر به فرد، ادغام می کند. final_mst به دست آمده حاوی MST است که همه ی راس ها را در سراسر MST های ورودی پوشش می دهد.

def merge_msts(msts):

این خط یک تابع merge_msts را تعریف می کند که لیستی از ساختارهای MST (msts) را به عنوان ورودی می گیرد.

`final_mst = nx.Graph()`:

این یک گراف خالی `final_mst` با استفاده از کتابخانه NetworkX ایجاد می کند. این گراف به عنوان ساختار نهایی MST مرج شده عمل می کند.

`for mst in msts:`:

این لوپ روی هر استراکچر MST در لیست msts تکرار می شود.

`for edge in mst.edges(data=True):`:

در هر ساختار MST، این لوپ روی هر یال تکرار می شود، جایی که data=True شامل اطلاعات وزن لبه می شود.

`source, target, weight = edge`:

این خط گره source، گره target و وزن لبه فعلی را که در حال تکرار است unpack می کند.

`if final_mst.has_edge(source, target):`:

این دستور شرطی بررسی می کند که آیا لبه از قبل در گراف final_mst وجود دارد یا خیر.

`if weight['weight'] < final_mst[source][target]['weight']:`:

اگر لبه از قبل وجود داشته باشد، این شرط بررسی می کند که آیا وزن یال فعلی کمتر از وزن یال مربوطه در final_mst باشد.

`final_mst[source][target]['weight'] = weight['weight']`:

اگر لبه جدید وزن کمتری داشته باشد، وزن یال مربوطه را در final_mst آپدیت میکند.

`else: final_mst.add_edge(source, target, weight=weight['weight'])`:

اگر یال از قبل در final_mst وجود نداشته باشد، یک یال جدید با وزن آن به نمودار اضافه می شود.

`return final_mst`:

هنگامی که تمام یال ها پردازش و مرج شدند، تابع گراف نهایی MST مرج شده final_mst را برمی گرداند.

به طور کلی، این تابع به طور موثر چندین ساختار MST را با آپدیت کردن یال‌ها با وزن‌های کمتر و افزودن لبه‌های جدید برای ایجاد یک نمایش تلفیقی از ساختارهای ورودی MST ترکیب می‌کند.

قدم دوم : مرج کردن درخت های MST :

```
# Merge the resulting MST trees together
mst_trees = []
for i in range(n_clusters):
    mst = compute_mst([source_col[j] for j in range(i, len(source_col), n_clusters)],
                      [target_col[j] for j in range(i, len(target_col), n_clusters)],
                      [rating_col[j] for j in range(i, len(rating_col), n_clusters)])
    mst_trees.append(mst)

final_mst = merge_msts(mst_trees)
```

۱. ابتدا یک لیست خالی به نام "mst_trees" را مقداردهی اولیه میشود.

۲. سپس در محدوده ای از ۰ تا «n_clusters» در یک for تکرار می شود وMST برای هر cluster محاسبه می شود . در داخل لوپ ، زیرمجموعه های ««source_col، target_col»» و «rating_col» را بر اساس ایندکس کلاستر های فعلی استخراج می شود و سپس MST را برای آن زیر مجموعه محاسبه می کند. سپس MST حاصل به لیست «mst_trees» اضافه می‌شود.

به نظر می‌رسد تابع «compute_mst» سه لیست را به عنوان ورودی دریافت می‌کند: گره‌های source، گره‌های target، weight های لبه (یا رتبه‌بندی‌ها در این مورد)، و MST را برمی‌گرداند.

۳. پس از اجرای لوپ ، کد ، تابع "merge_msts" را با لیست "mst_trees" به عنوان ورودی فراخوانی می کند تا همه MST ها را با هم ادغام کند و نتیجه را در متغیر "final_mst" ذخیره کند.

به طور خلاصه، کد MST ها را برای هر کلاستر ایجاد می کند و سپس این MST ها را با هم در یک MST واحد و نهایی مرج می کند.

قدم سوم : حساب هزینه ی MST نهایی :

```
# Calculate cost of the final MST
final_cost = sum(rating_col)
print(f"Final MST cost: {final_cost}")
```

۱. متغیر "final_cost" را برای ذخیره کل هزینه MST نهایی مقداردهی میشود.

۲. سپس از تابع "sum" برای محاسبه مجموع همه عناصر در "rating_col" استفاده می شود که وزن یا هزینه یال ها را در MST نشان می دهد.

۳. پس از به دست آوردن هزینه کل، کد از print برای نمایش نتیجه استفاده می شود.

به طور خلاصه، این قطعه کد هزینه کل MST نهایی را با جمع‌بندی وزن‌ها یا هزینه‌های مربوط به یال‌ها در MST محاسبه می‌کند و سپس این هزینه را با استفاده از تابع «print » نمایش می‌دهد.

قدم چهارم : در نهایت زمان اجرای نهایی پرینت میشود :

```
# Print complete execution time
print(f"Complete execution time: {execution_time:.4f} seconds")
```

مقایسه هزینه کل MST و زمان اجرای بخش اول و دوم پروژه:

هزینه کل قسمت اول پروژه کمتر از قسمت دوم است، زیرا هزینه نهایی MST آن بر اساس weight های انفرادی از فایل CSV محاسبه می شود.

در قسمت دوم پروژه، کل هزینه MST بر اساس تمام weight های لبه از مجموعه داده محاسبه می‌شود.

زمان اجرای قسمت اول پروژه کمتر از قسمت دوم است زیرا میدانیم در قسمت دوم پروژه هزینه ی تک تک کلاستر ها باید جدا جدا حساب شوند و این برروی زمان اجرا تاثیر میگذارد (زمان اجرا طولانی تر خواهد بود) بنابراین زمان اجرای الگوریتم MST ممکن است زمانی که گراف به cluster های مختلف تقسیم می‌شود، در مقایسه با زمانی که یک گراف جداگانه است، طولانی‌تر باشد، زیرا پیچیدگی و محدودیت‌های موجود در یافتن MST افزایش می‌یابد

لینک خروجی کد ها :

https://drive.google.com/file/d/1rJ8Pbp16QMkf1vzVPt7ooSBaVb9TpGzI/view?usp=sharing