КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ

Звіт до практичної роботи

на тему

Реалізація методів кластеризації на базі теорії графів

Виконав
студент II курсу магістратури
групи ОМ-2
Пишко Андрій

Київ – 2023

Зміст

1 Вступ

У сучасному світі величезна кількість даних генерується щодня в різних галузях, від науки до промисловості. Однак обробка і аналіз цих даних може бути важким завданням через їхній великий обсяг та складність. Методи кластеризації є одним з ключових інструментів у сфері обробки даних, що дозволяють групувати схожі об'єкти разом, роблячи дані більш зрозумілими та аналізованими.

В даній роботі я зосередився на використанні теорії графів для вирішення завдань кластеризації. Два розглянуті методи – HCS (Алгоритм виділення зв'язних компонент) та MST (Алгоритм мінімального кістякового дерева) – базуються на графовій структурі даних та дозволяють ефективно визначати групи об'єктів за їхніми взаємозв'язками.

Метод HCS базується на концепції зв'язних компонент графа. Він дозволяє визначати кластери, які представляють собою максимальні зв'язні підграфи. За допомогою цього методу, ми можемо ідентифікувати групи об'єктів, які мають сильні взаємозв'язки між собою.

Другий метод, MST, використовує алгоритм мінімального кістякового дерева для визначення кластерів. Цей алгоритм прагне об'єднати всі вершини графа, використовуючи найменшу можливу вагу ребер. Застосовуючи його до задачі кластеризації, ми робимо акцент на знаходженні оптимальних шляхів об'єднання об'єктів у кластери.

У цьому звіті ми розглянемо реалізацію обох методів, їхню теоретичну основу, а також результати експериментів з використанням реальних або синтетичних даних.

2 Метод HCS

2.1 Пояснення методу

Основним принципом, який допомагає вирішувати задачу кластеризації за допомогою графів, є визначення подібності між вершинами. У графі подібності, кількість ребер між вершинами служить мірою їхньої схожості. Інтуїтивно зрозуміло, що чим більше ребер пов'язує вершини, тим сильніше вони пов'язані між собою.

Якщо ми розглядаємо процес роз'єднання графа подібності, видаляючи ребра, виявляється, що кількість ребер, які треба видалити до того, як граф стане роз'єднаним, свідчить про ступінь схожості вершин. Цей принцип лягає в основу даного методу.

Основним елементом нашого підходу є поняття мінімального розрізу – це набір ребер, без якого граф стає роз'єднаним. Визначаючи мінімальний розріз, ми визначаємо ключові зв'язки між групами схожих об'єктів.

Даний алгоритм спрямований на визначення високозв'язних підграфів у графі з n вершинами. Основна мета полягає в тому, щоб знайти всі підграфи, де мінімальний розріз містить більше, ніж $n/2$ ребер, і ідентифікувати

їх як кластери. Кластери, утворені таким чином, відомі як високозв'язні підграфи (Highly Connected Subgraphs).

Враховуючи граф подібності $G(V, E)$, алгоритм кластеризації HCS перевірить, чи він уже сильно зв'язаний і, якщо так, повертає G, інакше використовує мінімальний розріз G для розділення G на два підграфи H і H' і рекурсивно запускає Алгоритм кластеризації HCS для H і H'.

2.2 Реалізація

Для реалізації алгоритму мною було розроблено наступні функції.

Листинг 1: Перевіряє граф G на високозв'язність

```
def highlyConnected(G, E):
return len(E) > len(G.nodes) / 2
```

Листинг 2: Видаляє всі ребра E з G

```
def remove_edges(G, E):
for edge in E:
    G.remove_edge(*edge)
return G
```

Листинг 3: Реалізація методу

```
def hcsClustering(G):
G_ = G.copy()
E = nx.algorithms.connectivity.cuts.minimum_edge_cut(G_)

if not highlyConnected(G_, E):
    G_ = remove_edges(G_, E)
    subGraphs = [G_.subgraph(c).copy() for c in nx.connected_components(G_)]

    if len(subGraphs) == 2:
        H = hcsClustering(subGraphs[0])
        _H = hcsClustering(subGraphs[1])

        G_ = nx.compose(H, _H)

return G_
```

Листинг 4: Функція для повернення результатів у зручний для подальшого використання формат

```
def hcsClusteringLabelled(G):
_G = hcsClustering(G)

subGraphs = (G.subgraph(c).copy() for c in nx.connected_components(_G))
```

```
    labels = np.zeros(shape=(len(G)), dtype=np.uint16)

    for _class, _cluster in enumerate(subGraphs, 1):
        c = list(_cluster.nodes)
        labels[c] = _class

    return labels
```

2.3 Результати

Зазвичай для кластеризації використовують реальні дані. Проте для спрощення я використаю завичайну матрицю суміжності, яка попередньо була отримана з реальних даних. Маємо наступні вхідні дані:

Листинг 5: Вершини графа

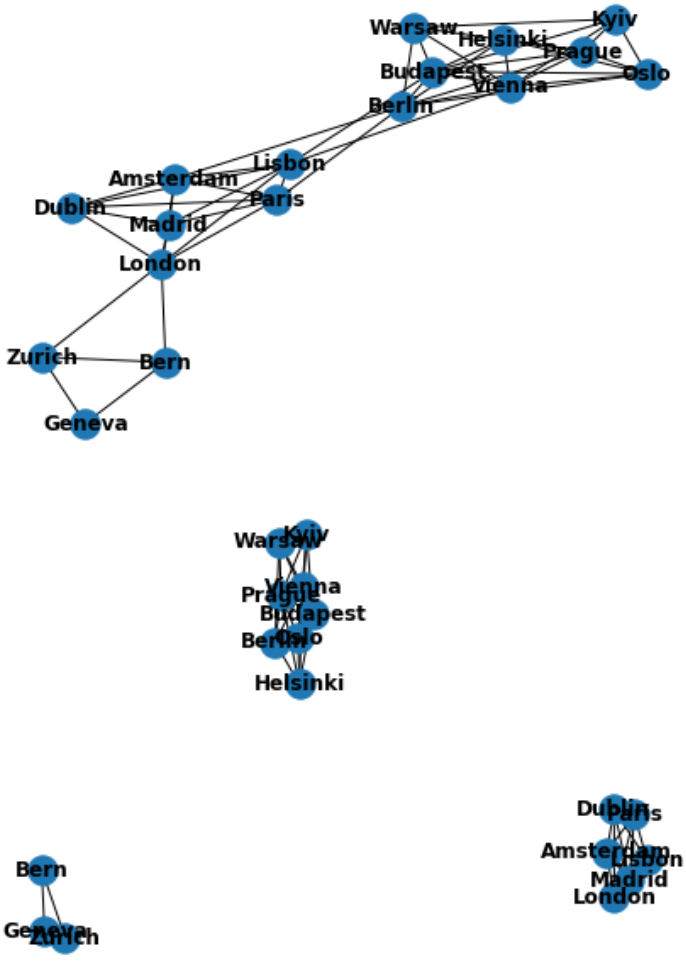
```
v = {'Dublin': 0,
     'London': 1,
     'Paris': 2,
     'Amsterdam': 3,
     'Madrid': 4,
     'Lisbon': 5,
     'Zurich': 6,
     'Geneva': 7,
     'Bern': 8,
     'Berlin': 9,
     'Vienna': 10,
     'Budapest': 11,
     'Prague': 12,
     'Warsaw': 13,
     'Kyiv': 14,
     'Oslo': 15,
     'Helsinki': 16}
```

Листинг 6: Матриця суміжності

```
[[0  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0]
 [1  0  1  1  1  1  1  0  1  0  0  0  0  0  0  0  0]
 [1  1  0  1  1  1  0  0  0  1  0  0  0  0  0  0  0]
 [1  1  1  0  1  1  0  0  0  1  0  0  0  0  0  0  0]
 [1  1  1  1  0  1  0  0  0  0  0  0  0  0  0  0  0]
 [1  1  1  1  1  0  0  0  0  0  1  1  0  0  0  0  0]
 [0  1  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0]
 [0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  0  0]
 [0  1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0]
 [0  0  1  1  0  0  0  0  0  0  1  1  1  1  0  1  1]
```

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
$$

Для заданих значень отримаємо наступні результати



3 Метод MST

3.1 Пояснення методу

Алгоритм кластеризації MST побудований на основі мінімального кістякового дерева і використовує ідею побудови графу, де вузли представляють точки даних, а ребра визначаються їх відстанями чи схожістю. Основна мета - створити мінімальне кістякове дерево, яке забезпечує з'єднаність графу і має мінімальну загальну вагу ребер.

Після побудови мінімального кістякового дерева використовують його для формування кластерів. Це здійснюється шляхом розрізання дерева на піддерева або групи за допомогою певного порогового значення. Вузли, які залишаються разом після розрізання, утворюють окремі кластери.

Отже, вхідні дані представляють собою граф, де кожна точка даних є вузлом. Ребра між вузлами вагуються відстанню чи схожістю відповідних точок даних.

Алгоритм будує мінімальне кістякове дерево (МКД) з графа. Зазвичай для цього використовують алгоритми Пріма або Крускала. МКД є структурою типу дерева, яка з'єднує всі вузли з мінімальною можливою сумою ваг ребер, забезпечуючи з'єднаність графа.

МКД використовується як основа для кластеризації. Кластери формуються за допомогою визначення гілок або піддерев в МКД. Зазвичай відсікають гілки на підставі порогового значення, такого як максимальна вага ребра. Кластери формуються розділенням дерева на частини після відсічення.

Після того, як МКД розділено на кластери, кожен з'єднаний компонент формує окремий кластер. Точки даних всередині одного кластера вважаються більш взаємозв'язаними на основі структури МКД.

Особливої уваги потребує метод Пріма. Це алгоритм для побудови мінімального кістякового дерева в зваженому з'єднаному графі. Основна ідея полягає в поетапному додаванні ребер до дерева, починаючи з будь-якого випадкового вузла, і виборі на кожному етапі ребра з найменшою вагою, яке з'єднує поточне дерево з вершиною, яка ще не включена до дерева.

Спочатку вибирається випадкова початкова вершина графа. Усі ребра, що з'єднують початкову вершину з іншими, помічаються як потенційні ребра для додавання до МКД. Ваги цих ребер визначають їхню пріоритетність.

На кожному кроці алгоритм обирає ребро з найменшою вагою серед потенційних ребер. Вибране ребро додається до МКД, і вершина, до якої веде це ребро, тепер вважається частиною дерева.

Після кожного додавання ребра до МКД, оновлюються потенційні ребра для кожної вершини, яка вже включена до дерева. Ребра, які з'єднують ці вершини з вершинами, що ще не включені, розглядаються як нові потенційні ребра.

Процес триває до того моменту, коли всі вершини графа включаються до МКД, тобто коли дерево стає з'єднаним.

Метод Крускала - це інший алгоритм для побудови мінімального кістякового дерева (МКД) в зваженому з'єднаному графі. На відміну від методу Пріма, Крускал працює шляхом послідовного додавання ребер з найменшою вагою, не обираючи конкретну початкову вершину.

У ньому усі ребра графа сортуються в порядку зростання їх ваги. Кожна вершина графа в початковий момент розглядається як окрема множина. Починаючи з ребра з найменшою вагою, алгоритм додає це ребро до МКД, якщо воно не утворює цикл разом із ребрами, що вже додані. Додається ребро до МКД, і множини вершин, які з'єднує це ребро, об'єднуються в одну множину. Процес повторюється, додаючи на кожному кроці ребро з наступною найменшою вагою, яке не утворює цикл з раніше доданими ребрами. Процес триває до того моменту, коли кількість доданих ребер дорівнює (V-1), де V - кількість вершин у графі. В цей момент утворюється зв'язне МКД.

3.2 Реалізація

Листинг 7: Використані бібліотеки

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

import heapq
```

Листинг 8: Реалізація алгоритму Прима для пошуку мінімального кістякового дерева

```
def primsAlgorithm(graph):
graphDict = nx.to_dict_of_dicts(graph)

treeNodes = set()
startNode = list(graphDict.keys())[0]
treeNodes.add(startNode)
treeEdges = []

edges = [(data['weight'], startNode, neighbor) for neighbor, data in graphDi
heapq.heapify(edges)

while edges:
    weight, currentNode, nextNode = heapq.heappop(edges)


    if nextNode not in treeNodes:
        treeEdges.append((currentNode, nextNode, weight))
        treeNodes.add(nextNode)
```

```
            for neighbor, data in graphDict[nextNode].items():
                if neighbor not in treeNodes:
                    heapq.heappush(edges, (data['weight'], nextNode, neighbor))

    return treeEdges
```

Листинг 9: Реалізація алгоритму Крускала для пошуку мінімального кістякового дерева

```
    def kruskalAlgorithm(graph):

    minSpanningTree = []
    edgeHeap = []
    nodeSets = []

    for edge in graph.edges(data=True):
        heapq.heappush(edgeHeap, (edge[2]['weight'], edge[0], edge[1]))

    for node in graph.nodes():
        nodeSets.append({node})

    while edgeHeap:
        weight, u, v = heapq.heappop(edgeHeap)

        u_set = None
        v_set = None

        for node_set in nodeSets:
            if u in node_set:
                u_set = node_set
            if v in node_set:
                v_set = node_set

        if u_set != v_set:
            minSpanningTree.append((u, v, weight))

            u_set.update(v_set)
            nodeSets.remove(v_set)

    return minSpanningTree
```

Листинг 10: Фукнція реалізовує видалення ребер у порядку спадання доти

```
    def removeEdgesForClusters(minSpanningTreeGraph, clustersNumber):
    sortedEdges = sorted(minSpanningTreeGraph.edges(data=True), key=lambda x: x[
```

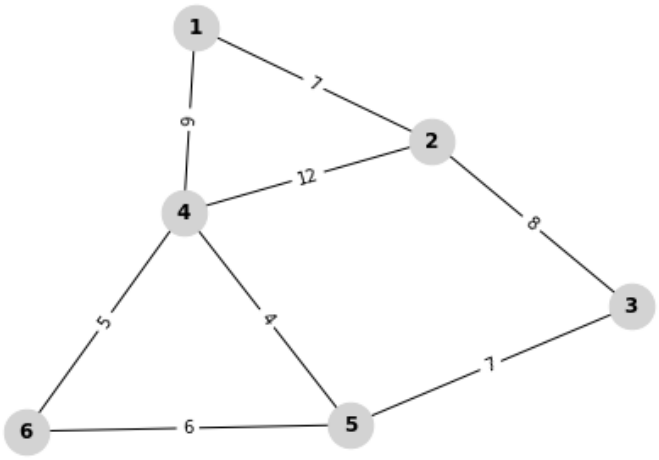
```
for edge in sortedEdges:
    minSpanningTreeGraph.remove_edge(edge[0], edge[1])
    if nx.number_connected_components(minSpanningTreeGraph) == clustersNumbe
        break

resultEdges = [(edge[0], edge[1], edge[2]['weight']) for edge in minSpanning
return resultEdges
```

Листинг 11: Фінальна функція

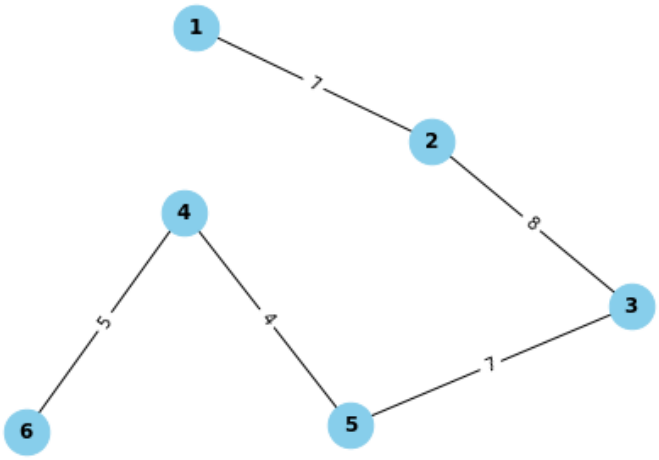
```
def mstClustering(graph, clustersNumber):
minSpanningTree = primsAlgorithm(graph)

T = nx.Graph()
for edge in minSpanningTree:
    T.add_edge(edge[0], edge[1], weight=edge[2])

result = removeEdgesForClusters(T, clustersNumber)
return result
```

3.3 Результати

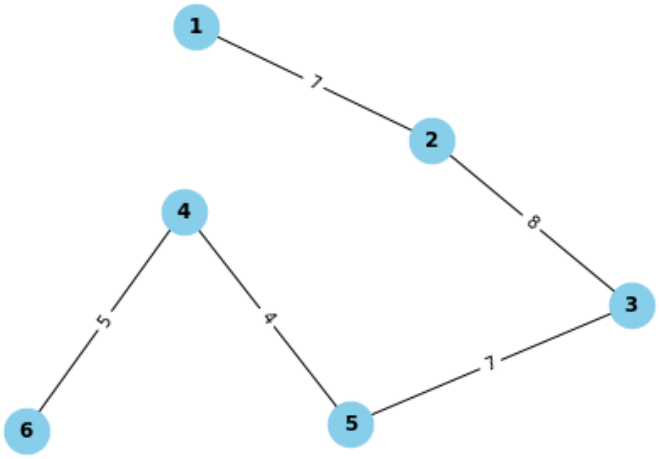
Аналогічно до попереднього прикладу, розглянемо спрощену модель інформації, яку будемо кластеризувати. В даному випадку розглянемо маленький граф G зі зваженими ребрами.



Після використання матоду Прима отримаємо наступне кісткове дерево:



Після використання методу Крускала результат маємо аналогічний:

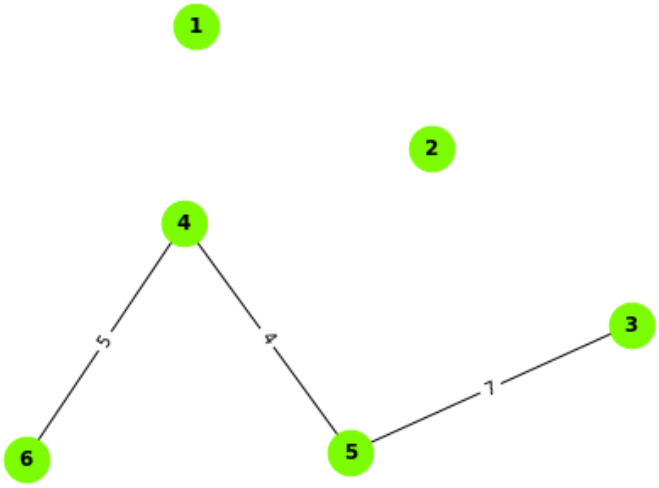


Після застосування кластеризації загалом

Листинг 12: Застосування кластеризації

```
clustersNumber = 3

clustersEdges = mstClustering(G, clustersNumber)
```

отримаємо задану кількість кластерів



4 Висновки

У даній практичній роботі було реалізовано та застосовано методи кластеризації побудовані на базі теорії графів.

Перший алгоритм, HCS, є ефективним у визначенні груп взаємозв'язаних вершин у графі. Використання мінімального розрізу дозволяє виявляти стійкі та густі зв'язки між вершинами. Важливо враховувати, що параметр n/2 може бути налаштований в залежності від конкретного контексту задачі. Цей алгоритм знаходження високозв'язних підграфів може бути застосований у різних областях, таких як аналіз соціальних мереж, виявлення спільнот у великих мережах або класифікація об'єктів за їхніми взаємозв'язками.

Кластеризація з мінімальним кістяковим деревом особливо корисна для даних із природною ієрархічною або деревоподібною структурою. Вона ефективна з обчислювальної точки зору, особливо коли кількість точок даних велика.

Метод Пріма є ефективним і простим алгоритмом для знаходження мінімального кістякового дерева, і його застосування часто знаходиться в різних областях, таких як мережеве проектування, транспортні системи та аналіз даних.

Метод Крускала є не менш ефективний, оскільки використовує принцип "з'єднання-запитання"(union-find) для ефективної перевірки циклів під час додавання ребер. Він широко використовується у транспортних системах, телекомунікаціях та інших областях для вирішення завдань оптимізації мереж та з'єднань.