

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



MATHEMATICAL FOUNDATION FOR COMPUTER SCIENCE

Assignment

Community Structure Identification: Book recommendation

Lecturer: **PhD. Nguyễn An Khuê**
Students: **2070423 - Nguyễn Thị Ngọc Phim**
1910006 - Nguyễn Ngọc Thái An
2170554 - Bạch Mai Tuyết Như
1911530 - Nguyễn Kim Lộc
2170551 - Vũ Thành Nhân

Ho Chi Minh city - May 2023

Contents

List of Figures

I. Introduction

1. Community Structure Identification Problem

The Community Structure Identification Problem is a fundamental challenge in network analysis and graph theory. It involves identifying groups or communities within a given network where nodes within the same community are densely connected to each other while having fewer connections to nodes in other communities.

The problem arises in various real-world scenarios, such as social networks, biological networks, and online communities. By uncovering the underlying community structure, researchers can gain valuable insights into the organization, dynamics, and functionality of complex systems.

Solving the Community Structure Identification Problem typically entails applying graph clustering algorithms to partition the network into cohesive communities. These algorithms aim to maximize the modularity, a measure that quantifies the quality of the community structure by comparing the observed connections within communities to those expected by chance.

Some example:

- **Social Networks:** Consider a social network platform like Facebook or Twitter. The problem involves identifying distinct groups or communities of users who share similar interests, have common social connections, or engage in similar discussions. By identifying these communities, the platform can better understand user behavior, personalize content recommendations, and detect influential users or potential communities of interest.
- **Biological Networks:** In molecular biology, the problem arises in analyzing protein-protein interaction networks or gene co-expression networks. Identifying communities within these networks can reveal groups of proteins or genes that work together to perform specific biological functions or participate in shared signaling pathways. This knowledge can aid in understanding disease mechanisms, drug discovery, and the design of targeted therapies.
- **Online Communities:** In online forums, discussion boards, or social media platforms, the Community Structure Identification Problem involves identifying clusters of users who interact and engage with each other on specific topics or interests. Uncovering these communities can help in mod-

erating online discussions, identifying influential users, and facilitating content recommendation or targeted advertising.

- **Citation Networks:** In academic research, the problem arises in analyzing citation networks, where papers cite other papers. By identifying communities within these networks, researchers can discover clusters of related papers that contribute to specific research topics or subfields. This information can aid in literature review, knowledge discovery, and identifying potential collaborators or experts in a particular domain.
- **Transportation Networks:** In transportation planning and optimization, the problem involves identifying communities or clusters of interconnected transportation hubs, such as airports or railway stations. By identifying these communities, transportation authorities can optimize routing, improve connectivity, and identify potential bottlenecks or areas for infrastructure development

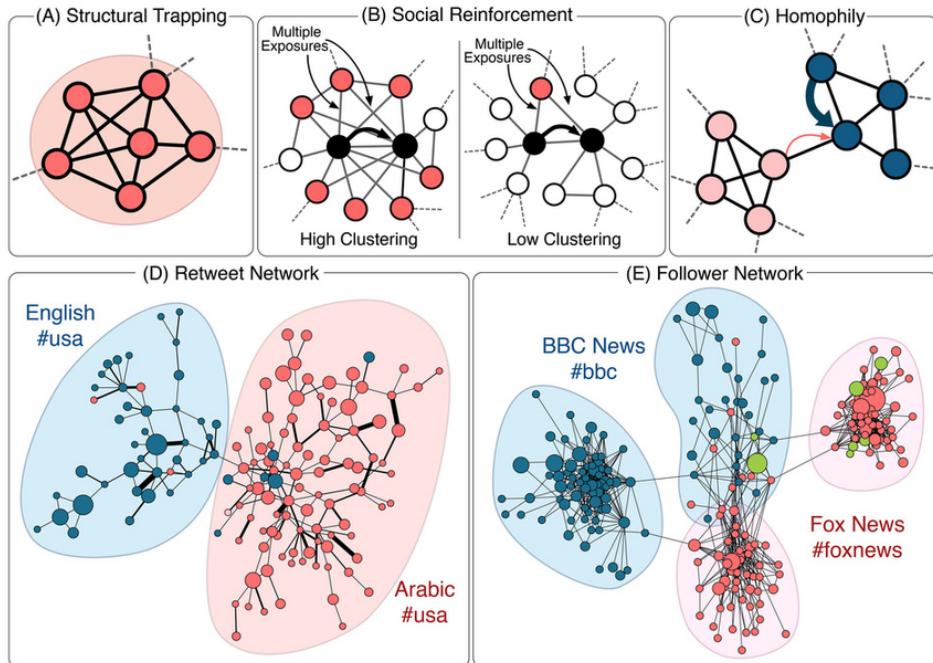


Figure 1: The importance of community structure in the spreading of social contagions

2. Community Structure Identification for book

2.1 Motivation

Consider a network where each node represents a book, and the connections between nodes represent similarities or relationships between books. These relationships can be based on various factors such as genre, author, themes, or even reader reviews and recommendations.

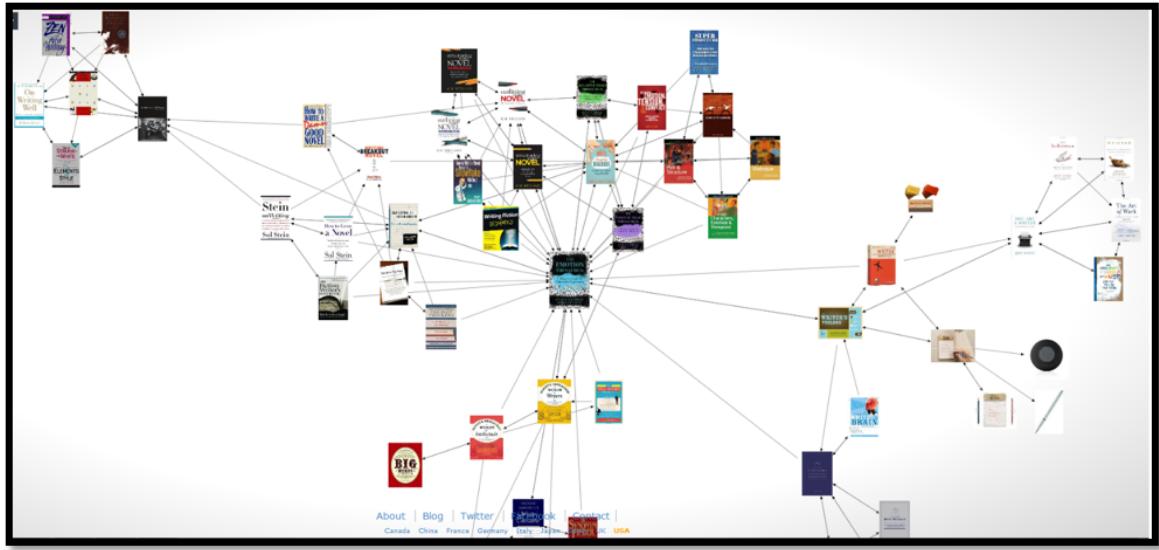


Figure 2: Book network example

The Community Structure Identification Problem in this scenario involves identifying clusters of books that share common characteristics, themes, or appeal to similar audiences. By uncovering these book communities, we can gain insights into the organization of the literary landscape and identify groups of books that are related in terms of content, style, or target readership.

For instance, the algorithm might identify a community of classic literature, consisting of books by authors such as Jane Austen, Charles Dickens, and F. Scott Fitzgerald. Another community might comprise contemporary romance novels by popular authors like Nicholas Sparks and Jojo Moyes. The nodes within these communities would be densely connected, indicating strong similarities between the books within each community.

This information can be valuable for various stakeholders in the book industry. Publishers can identify potential target markets and optimize their marketing strategies by understanding which communities of books are likely to appeal

to similar readers. Online book retailers can enhance recommendation systems by suggesting books from the same community or providing "read-alike" recommendations to customers. Authors can gain insights into their genre or niche and better understand their position within the broader literary landscape.

In summary, the Community Structure Identification Problem in the context of books helps reveal clusters of books that share common attributes, themes, or readership, enabling targeted marketing, recommendation systems, and a deeper understanding of the literary world.

2.2 Dataset description

Context: Books read by users and ratings provided by them on Amazon

Content: Online data for books from Amazon along with user ratings and users who bought them. Totaly 3 files:

- Books.csv -> Has the book descescription, include some fields:
 1. Book ID: 10 digit number, ID of the book (unique value).
 2. Title: free text, title of the book.
 3. Author: free text, book author name.
 4. Publish year: number, the year that book published.
 5. Publisher: free text, the publisher name.
 6. Images: free text, link to image of the book.
- Ratings.csv -> Has the book ratings description, include some fields:
 1. User ID: 6 digit number, ID of the user.
 2. Book ID: 10 digit number, ID of the book.
 3. Book rating from user: number from 0 to 10, rating score from user to the book.
- Users.csv -> Has the user descriptions, include some fields:
 1. User ID: 6 digit number, ID of the user.
 2. Location: free text, location of the user.
 3. Age: number, user age.

II. Preliminaries

1. Networks and Node, Community Structure

A node is a unit of a network that represents an entity, such as a person, a group, or a concept. A network is a collection of nodes connected by edges or links, which can represent various types of relationships between the nodes. Networks are a common way of representing complex systems in a wide variety of scientific fields, from social sciences to biology, networks often exhibit a natural division into clusters or modules called communities. At the heart of these communities are nodes, which represent individual entities within the network. These nodes typically have unique attributes and connections to other nodes within the same community. A network is a collection of nodes, also called vertices, that are connected by edges or links, representing the relationships between them. The existence of community structure implies that vertices within a community share certain commonalities and exhibit similar behaviors within the larger graph.

For example, in a transportation network, nodes could represent cities or transportation hubs, with edges representing the links between them such as roads, railways, or airlines. Within this network, communities could emerge based on geographical proximity or shared transportation infrastructure. For example, cities located along the same highway or railway line may form a community, as they share similar transportation options and patterns of movement. Similarly, airlines may form communities based on shared routes or alliances, as they tend to have more connections and interactions with each other than with other airlines outside their community.

The term "community structure" refers to the composition of a given network, including the number and attributes of nodes, as well as the relationships between them. Community structure is a crucial aspect of complex networks, as it indicates that nodes tend to cluster together into distinct groups or communities. Within each community, the density of edges is typically higher than between communities, emphasizing the strength of connections among members of the same community.

Understanding the community structure of a network can have important

implications for many real-world applications, such as predicting the spread of diseases or identifying influential individuals in a social network. Moreover, community detection can also help to reveal underlying patterns and structures in complex systems, providing insights into the organization and behavior of these systems.

2. Community Detection

Community detection is a fundamental problem in network analysis that aims to identify groups of nodes that are more similar to each other than to the rest of the network. Community detection is widely used in various fields such as biology, sociology, and marketing to understand social structures, reveal user data within the network, and develop relevant recommendation systems. There are several approaches to community detection, including hierarchical clustering, spectral clustering, modularity maximization, and random walk methods. Each method has its strengths and weaknesses, and the choice of method depends on the specific properties of the network being analyzed.

- Modularity maximization is one of the most widely used methods for community detection, and it aims to partition the network into non-overlapping communities that maximize a quality function called modularity. Modularity measures the density of edges within communities relative to the expected density if edges were distributed randomly. Maximal modularity indicates strong community structure, while low modularity suggests a random network.
- Hierarchical clustering is a method that groups nodes based on their similarity and generates a hierarchical tree of communities. In contrast, spectral clustering uses eigenvectors of the graph Laplacian to cluster nodes and has been shown to perform well on networks with well-defined community structures.
- The Girvan-Newman algorithm is a popular method for community detection in networks. The algorithm works by iteratively removing edges from the network, with the aim of breaking it into smaller components that correspond to communities.

Community detection has many applications, including identifying functional modules in biological networks, understanding the structure of social networks,

and detecting communities in web graphs. In recent years, there has been growing interest in developing community detection algorithms that are scalable, efficient, and applicable to large-scale networks.

3. Modularity

A good community partition of a network should have fewer edges between communities than one would expect by chance. This indicates significant community structure within the network. On the other hand, if the number of edges between groups is what would be expected by chance, it is not evidence of significant community structure. The measure known as modularity quantifies this idea that true community structure corresponds to a statistically surprising arrangement of edges, with fewer edges between groups and more edges within groups than expected by chance.

3.1 Modularity Density

Li et al. have developed a new quantitative metric called modularity density (D), which is based on the density of subgraphs, to determine community structure of networks.

The higher the value of D , the better is a partition. The optimization of modularity density is also NP-hard. The modularity density is defined as:

$$D_\lambda = \sum_{i=1}^m \frac{2\lambda L(V_i, V_i) - 2(1-\lambda)L(V_i - \overline{V_1})}{|V_i|}$$

where, V_i is the subset of V $i=1, \dots, m$, such that $L(V_i) = \sum_{i \in V_i, j \in \overline{V_1}} A_{ij}$, where $\overline{V_1} = VV_i$

3.2 Modularity Q

Modularity Q is one of the most used measure. The modularity Q is, up to a multiplicative constant, the number of edges falling within groups minus the expected number in an equivalent network with edges placed at random. The modularity can be either positive or negative, with positive values indicating the possible presence of community structure. Thus, one can search for community structure precisely by looking for the divisions of a network that have positive, and preferably large, values of the modularity.

The original idea of modularity was given by Newman and Girvan, they have

defined modularity Q as:

$$Q = \frac{1}{2} \sum_{ij} (A_{ij} - \frac{k_i k_j}{2m}) \delta(\sigma_i, \sigma_j)$$

Here, m is the number of links, k_i is the degree of vertex i, k_j is the degree of vertex j, σ_i is the community to vertex i, σ_j is the community to vertex j, and $\delta(\sigma_i \sigma_j) = 1$ if i and j belong to the same community, otherwise it equals to 0.

An alternate formulation of this is as a sum over communities:

$$Q = \frac{1}{2m} \sum_c (A_c - \frac{K_c^2}{4m})$$

where m_c is the number of internal edges (or total internal edge weight) of community c and $K_c = \sum_{i|\sigma_i=c} k_i$ is the total (weighted) degree of nodes in community c.

III. Approaches

1. Community Detection Methods in Network Analysis: Agglomerative and Divisive Approaches

Community detection in network analysis is a crucial task known as community analytics. It involves identifying groups or communities within a network structure. There are two primary types of community detection methods: agglomerative methods and divisive methods.

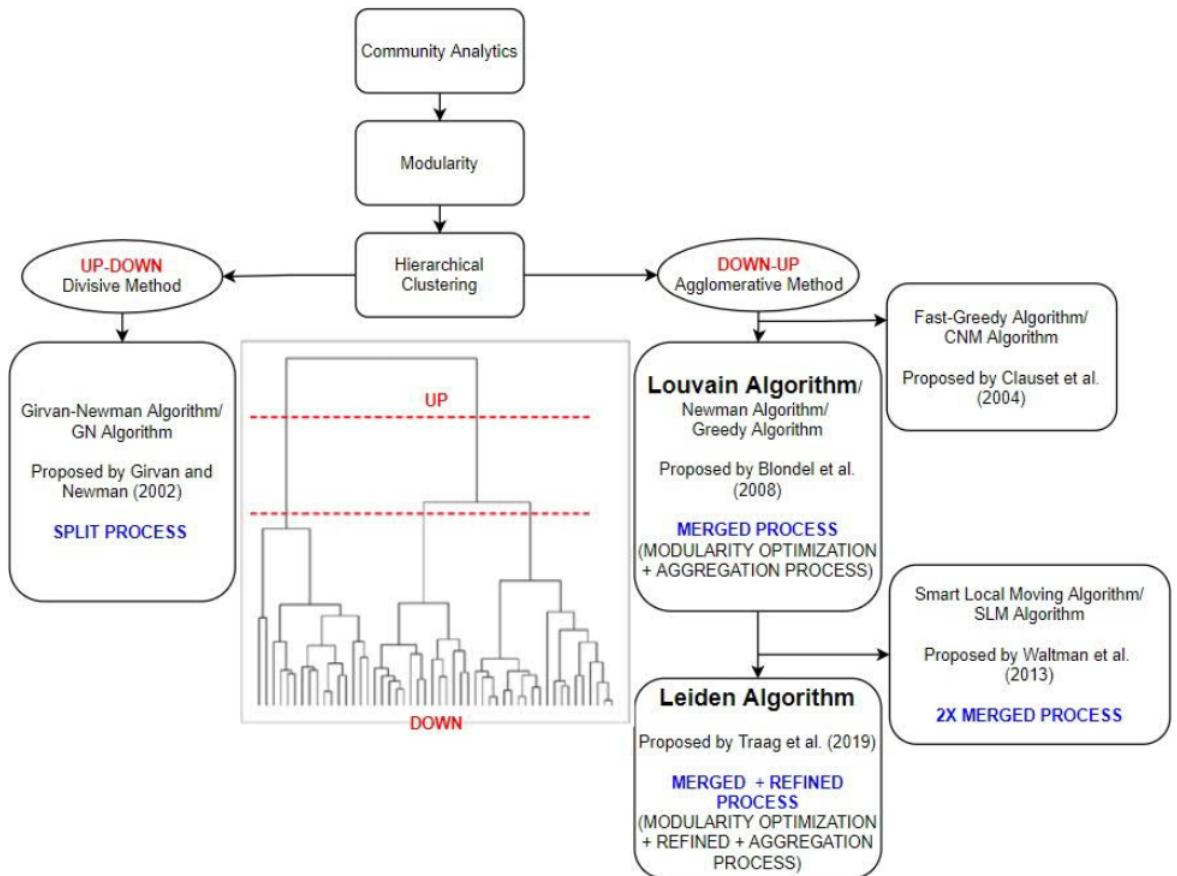


Figure 1. Taxonomy of community analytics.

In agglomerative methods (shown in the right wing of Figure 1), the process starts from individual nodes and progressively merges them based on their similarity or connectivity, resulting in a hierarchical clustering. This means that nodes are merged from the bottom up, creating larger communities as the process continues. The Louvain and Leiden algorithms, which are based on modularity and hierarchical clustering, fall into the agglomerative category. Modularity, denoted as Q , measures the strength of communities and guides the algorithm. Higher values of modularity indicate better communities, and a value below 1 suggests that each node is considered a separate community. The dendrogram in

Figure 1 represents the hierarchy of clusters generated by hierarchical clustering.

On the other hand, divisive methods (depicted in the left wing of Figure 1) start with a single partition containing all nodes and iteratively split it by removing edges with low similarity. This process is known as the split process. Divisive methods aim to find smaller, more cohesive communities by progressively dividing the network.

Both agglomerative and divisive methods have different approaches to community detection, but they ultimately seek to uncover meaningful groups of nodes based on their connectivity patterns within a network.

2. Girvan-Newman algorithm

The Girvan-Newman algorithm is a well-known community detection algorithm in network analysis. Proposed by Michelle Girvan and Mark Newman, it aims to identify communities or clusters within a network by iteratively removing edges with high betweenness centrality. This algorithm follows a divisive approach, progressively dividing the network into smaller communities based on the importance of edges.

Edge Betweenness: It uses edge betweenness to find and remove central edges that connect communities within a larger graph. The formula for edge betweenness is:

$$B(e) = \sum_{u,v \in V(G)} \frac{\sigma_{u,v}(e)}{\sigma_{u,v}} \quad (1)$$

where $\sigma_{u,v}$ is the number of shortest paths between two distinct vertices and $\sigma_{u,v}(e)$ is the corresponding number of shortest paths containing a particular edge.

Modularity: After removing an edge, the Girvan-Newman algorithm calculates the modularity (Q) of the graph, which is a value between the range $[-0.5, 1]$. A higher value suggests a more significant community structure. Therefore, we can identify communities by maximizing modularity.

The formula for modularity is:

$$Q = \sum_{s=1}^m \left[\frac{l_s}{L} - \left(\frac{d_s}{2L} \right)^2 \right] \quad (2)$$

(m = number of modules, l_s = the number of edges inside module s , L = the number of edges in the network, d_s = total degree of the nodes in module s)

This process of removing an edge and calculating the modularity is iteratively repeated. The algorithm will stop when the new modularity is no longer greater than the modularity from the previous iteration. The ending modularity is usually around 0.6.

2.1 Pseudo Code

Algorithm 1 Girvan-Newman Algorithm

Input : Graph G

Output: Disconnected components of G

```

1 do
2     Let  $n$  be the number of edges in  $G$ 
3     for  $i = 0$  to  $n - 1$  do
4         Let  $B[i]$  be betweenness centrality of edge  $i$ 
5         if  $B[i] > max_B$  then
6              $max_B \leftarrow B[i]$ 
7              $max_B(edge) \leftarrow i$ 
8         end
9         Remove edge  $i$  from graph
10    end
11 while number of edges in  $G$  is not 0;

```

2.2 Implement Code

```

1 def girvan_newman_algorithm(G):
2     # calculate initial betweenness centrality of all edges
3     betweenness = nx.edge_betweenness_centrality(G)
4
5     while len(G.edges()) > 0:
6         # remove edge with highest betweenness centrality
7         max_edge = max(betweenness, key=betweenness.get)
8         G.remove_edge(*max_edge)
9
10        # recalculate betweenness centrality of remaining edges
11        betweenness = nx.edge_betweenness_centrality(G)
12
13        # check if the graph has been split into disconnected
14        components

```

```

14     if nx.number_connected_components(G) > 12:
15         # if so, return the list of communities
16         communities = list(nx.connected_components(G))
17         return communities
18
19     # if no disconnected components are found, return the original
20     # graph as a single community
21     return [set(G.nodes())]

```

2.3 Pros, Cons

The Girvan-Newman algorithm offers several advantages and disadvantages in community detection within network analysis.

On the positive side, the algorithm is effective in uncovering the community structure of a network. By targeting edges with high betweenness centrality and iteratively removing them, it reveals the underlying modular organization of the network. This capability allows for a comprehensive understanding of how communities are formed and interconnected.

Additionally, the Girvan-Newman algorithm does not require any prior information or assumptions about the network's structure or the number of communities. It dynamically adapts to the unique characteristics of the network during the divisive process, making it versatile and applicable in various scenarios.

The algorithm also provides valuable insights into bridge edges, which are edges with high betweenness centrality that act as connections between different communities. By identifying these bridge edges, researchers can gain a deeper understanding of the interconnections and relationships between communities.

However, there are also some drawbacks to consider. One major limitation is the computational expense associated with the Girvan-Newman algorithm, especially for large networks. Calculating betweenness centrality for all edges in the network requires significant computational resources, making it impractical for networks with millions of nodes.

Another limitation is the resolution limit of the algorithm. It may struggle to detect smaller communities or communities with weaker connectivity. In some cases, the algorithm tends to merge smaller communities into larger ones, resulting in the loss of fine-grained community structure.

Additionally, the Girvan-Newman algorithm lacks scalability for very large networks. As the network size increases, the algorithm's runtime and memory

requirements grow significantly, posing challenges in analyzing massive datasets.

Despite these limitations, the Girvan-Newman algorithm remains a valuable tool for community detection, providing insights into the modular organization of networks. Researchers continue to explore enhancements and alternative approaches to address its drawbacks and improve its applicability in different network analysis scenarios.

2.4 Time Complexity

Despite Girvan-Newman's popularity and quality of community detection, it has a high time complexity, increasing up to $O(m^2n)$ on a sparse graph having m edges and n nodes. As a result, Girvan-Newman is generally not used on large scale networks. Its optimal node count is a few thousand nodes or less.

Because of this, there exist greedy algorithms for detecting communities to reduce the time but at the same time sacrificing the most accurate results. One such example is the Louvain algorithm.

3. Louvain Algorithm

The Louvain algorithm is a fast implementation of community detection. It can be used to analyze a network of 2 million nodes in only 2 minutes

It is a hierarchical clustering algorithm that involves two phases: modularity optimization and community aggregation

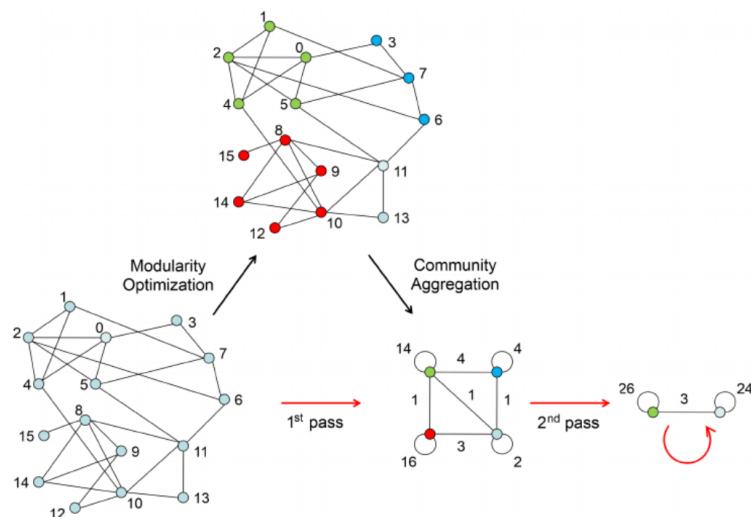


Figure 3: Louvain Algorithm

3.1 Modularity Optimization

As seen in the figure, the first step is to optimize the modularity of the entire graph. In this example, it splits the nodes into four communities.

To find these clusters, each node is moved into its neighboring community. If the change in modularity (ΔQ) is greater than 0, it is moved into the neighboring community. Otherwise, it remains in its current community. This process is repeated until $\Delta Q = 0$ for all nodes.

3.2 Community Aggregation

After modularity optimization, super nodes are created to represent each cluster. After the initial phase of the algorithm, there will exist many communities.

However, the two phases repeat, creating larger and larger communities. The algorithm stops only when no improvement can be made by any of the two operations

3.3 Pros, Cons

Pros of the Louvain algorithm:

- Fast and scalable for large networks.
- Optimizes modularity, which measures community quality.
- Detects hierarchical community structures.
- Allows flexibility in resolution for different levels of detail.
- Widely used with extensive resources and documentation.

Cons of the Louvain algorithm:

- Resolution limit may merge small communities into larger ones.
- Results can be sensitive to initial community assignments.
- Assumes non-overlapping communities.
- Bias towards detecting large, cohesive communities.
- Lacks theoretical guarantees of optimality.

We need to keep in mind these points when considering the Louvain algorithm for community detection.

3.4 Pseudo Code

Algorithm 2 Louvain Algorithm

Input : Graph G

Output: Communities

- 1 Initialize each node in G as a separate community
- 2 Initialize the change indicator $modularity_{improvement}$
- 3 **repeat**
- 4 modularity_improvement $\leftarrow 0$
- 5 **for** each node i in G **do**
- 6 **for** each neighbor j of node i **do**
- 7 Merge community of node i with community of node j
- 8 Compute the modularity improvement
- 9 Undo the merge if the improvement is not positive
- 10 Track the maximum improvement
- 11 **end**
- 12 **end**
- 13 **until** modularity_improvement is not positive;

3.5 Implement Code

```
1
2 def initialize_communities(graph):
3     """
4         Initialize each node into its own community.
5     """
6     return {node: [node] for node in graph.nodes()}
7
8
9 def calculate_modularity(graph, communities, m):
10    """
11        Calculate the modularity of the graph given the communities.
12    """
13    Q = 0.0
14    for community in communities.values():
15        community_subgraph = graph.subgraph(community)
16        L_c = sum(dict(community_subgraph.degree(weight='weight')).values())
17        d_c = 2 * L_c / m
```

```

18         Q += (L_c / m) - np.square(d_c)
19     return Q
20
21
22 def louvain(graph):
23     """
24     Perform community detection using the Louvain algorithm.
25     """
26     m = sum(dict(graph.degree(weight='weight')).values()) / 2
27
28     # Step 1: Create initial communities
29     communities = initialize_communities(graph)
30
31     # Track the best modularity
32     best_modularity = calculate_modularity(graph, communities, m)
33
34     while True:
35         # Step 2: Move nodes between communities
36         moved = False
37
38         for node in graph.nodes():
39             # Get the current community of the node
40             current_community = None
41             for c, nodes in communities.items():
42                 if node in nodes:
43                     current_community = c
44                     break
45
46             # Calculate the modularity gain of moving the node to
47             # each neighboring community
48             neighbor_communities = set()
49             for neighbor in graph.neighbors(node):
50                 for c, nodes in communities.items():
51                     if neighbor in nodes:
52                         neighbor_communities.add(c)
53
54             modularity_gains = {}
55             for neighbor_community in neighbor_communities:
56                 communities[current_community].remove(node)
57                 communities[neighbor_community].append(node)
58
59                 modularity_gains[neighbor_community] =
60                 calculate_modularity(graph, communities, m)

```

```

59
60             # Move the node to the community with the maximum
61             modularity gain
62             max_gain_community = max(modularity_gains, key=
63             modularity_gains.get)
64             max_gain = modularity_gains[max_gain_community]
65
66             if max_gain > 0:
67                 communities[max_gain_community].append(node)
68                 moved = True
69                 break
70
71             # If no improvement, revert the move
72             communities[neighbor_community].remove(node)
73             communities[current_community].append(node)
74
75             # If no nodes have moved communities, stop iterating
76             if not moved:
77                 break
78
79             # Recalculate modularity
80             modularity = calculate_modularity(graph, communities, m)
81
82             # If modularity has improved, update the best modularity
83             if modularity > best_modularity:
84                 best_modularity = modularity
85
86
87         return communities.values()

```

4. Leiden algorithm

The Leiden algorithm is an algorithm for detecting communities in large networks. It separates nodes into disjoint communities so as to maximize a modularity score for each community. Modularity quantifies the quality of an assignment of nodes to communities, that is how densely connected nodes in a community are, compared to how connected they would be in a random network. The Leiden algorithm extends the Louvain algorithm, which is widely seen as one of the best algorithms for detecting communities. However, the Louvain algorithm can lead to arbitrarily badly connected communities, whereas the Leiden algorithm guarantees communities are well-connected.

This algorithm is processed included 3 step:

- Step 1: local moving of nodes.
- Step 2: Refinement of the partition.
- Step 3: Aggregation of the network based on the refined partition.

We can see this process in the picture from (a) to (f) as below. It starts from a singleton partition (a). The algorithm moves individual nodes from one community to another to find a partition (b), which is then refined (c). An aggregate network (d) is created based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network. For example, the red community in (b) is refined into two subcommunities in (c), which after aggregation become two separate nodes in (d), both belonging to the same community. The algorithm then moves individual nodes in the aggregate network (e). In this case, refinement does not change the partition (f). These steps are repeated until no further improvements can be made.

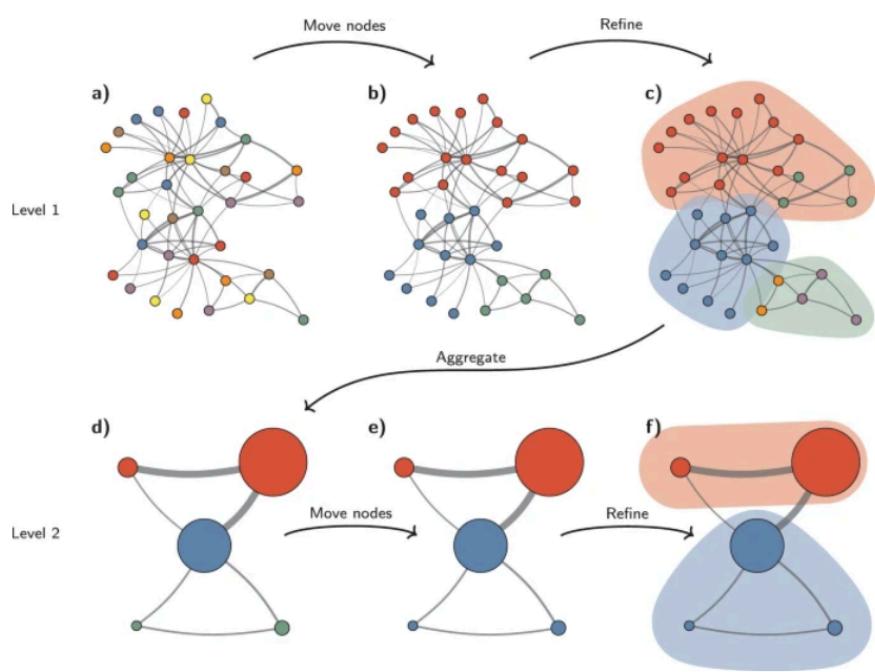


Figure 4: The Leiden algorithm

4.1 Pseudo code

Algorithm 3.2: Leiden

Input : Graph G , Partition P

```

1 function Leiden(Graph  $G$ , Partition  $P$ )
2   do
3      $P \leftarrow$  FastLocalMoving( $G, P$ )
4      $done \leftarrow |P| == |V(G)|$  // Every community consists of one node
5     if not  $done$  then
6        $P_{refined} \leftarrow$  RefinePartition( $G, P$ )
7        $G \leftarrow$  Coarsen( $G, P_{refined}$ )
8        $P \leftarrow$  SingletonPartition( $G$ )
9   while not  $done$ 
10  return flatten( $P$ )
11 function FastLovalMoving(Graph  $G$ , Partition  $P$ )
12    $Q \leftarrow$  Queue( $V(G)$ ) // Insert nodes in random order
13   do
14      $v \leftarrow Q.pop()$ 
15      $C_{new} \leftarrow \text{argmax}_{C \in P \cup \emptyset} \Delta_{v \rightarrow C}$ 
16     if  $\Delta_{v \rightarrow C_{new}} > 0$  then
17        $v \rightarrow C_{new}$  // Move  $v$  to Community  $C_{new}$ 
18        $Neighbours \leftarrow \{u \mid \{v, u\} \in E(G), u \notin C_{new}\}$ 
19        $Q.push(Neighbours \setminus Q)$ 
20   while  $Q \neq \emptyset$ 
21   return  $P$ 
22 function RefinePartition(Graph  $G$ , Partition  $P$ )
23    $P_{refined} \leftarrow$  SingletonPartition( $G$ )
24   for  $C \in P$  do
25      $P_{refined} \leftarrow$  MergeNodesSubset( $G, P_{refined}, C$ )
26   return  $P_{refined}$ 
27 function MergeNodesSubset(Graph  $G$ , Partition  $P$ , Subset  $S$ )
28    $R \leftarrow \{v \mid v \in S, \text{cut}_w(v, S - v) \geq \deg_w(v)[\text{vol}_w(S) - \deg_w(v)]\}$ 
29   for  $v \in R$  do
30     if  $v$  is singleton then
31        $T \leftarrow \{C \mid C \in P, C \subseteq S, \text{cut}_w(C, S - C) \geq \text{vol}_w(C)[\text{vol}_w(S) - \text{vol}_w(C)]\}$ 
32        $C_{new} \leftarrow \text{argmax}_{C \in T} \Delta_{v \rightarrow C}$ 
33        $v \rightarrow C_{new}$ 

```

4.2 Pros, Cons

The Leiden algorithm is a popular community detection algorithm that has been used in various fields, including social network analysis and bioinformatics. Some of the advantages of the Leiden algorithm are:

- It is a fast and scalable algorithm that can handle large and complex networks.
- It can detect communities of different sizes and densities.

- It can handle directed and weighted networks.
- It has been shown to perform well in comparison to other community detection algorithms.

However, there are also some potential limitations and drawbacks of the Leiden algorithm, such as:

- It may not be suitable for detecting communities with overlapping nodes.
- It may not perform as well in networks with low modularity.
- It may be sensitive to the choice of resolution parameter.
- It may require some expertise in parameter tuning and interpretation of results.

IV. Experiments

1. Preprocessing

The dataset is quite big to handle contain 3 file:

- Books.csv: 271379 records
- Users.csv: 276271 records
- **Ratings.csv: 1149781 records**

Therefore, we are using the map-reduce technique to split the data in order to fit the computer memory.

The map reduce is using to group and count all pair of books bought by at least 2 users and running on file Ratings.csv.

Map reduce first phase:

Map phase	Reduce phase
<User-ID1, ISBN1>	<User-ID1, [ISBN1, ISBN2, ISBN3, ...]>
<User-ID1, ISBN2>	
<User-ID1, ISBN3>	

Table 1: Map reduce first phase find all books were bought by single users

Map second phase:

Map phase	
<User-ID1, [ISBN1, ISBN2, ISBN3, ...]>	<[ISBN1, ISBN2], User-ID1> <[ISBN1, ISBN3], User-ID1> <[ISBN2, ISBN3], User-ID1>

Table 2: Map phase to find all books pair of book bought by users

Reduce second phase:

Reduce phase	
<[ISBN1, ISBN2], User-ID1>	<[ISBN1, ISBN2], Count1>
<[ISBN1, ISBN3], User-ID1>	<[ISBN1, ISBN3], Count2>
<[ISBN2, ISBN3], User-ID1>	<[ISBN2, ISBN3], Count3>

Table 3: Reduce phase to find all books pair of books and number of users bought it

2. Build application

We built a web application to demonstrate the process of recommendation in book store. Web are built with VueJS as a Frontend and Flask as a framework by Python language programming. Backend is in charge of query similar books and return it to Frontend via Restful api.



Figure 5: Technologies used

2.1 Build Backend

Flask introduction: Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.

All required libraries:

Create file requirements.txt:

```
1 Flask==2.2.3
2 Flask-Cors==3.0.10
3 pandas==2.0.0
4 networkx==2.8.8
```

Install all libraries via command:

```
1 pip install -r requirements.txt
```

Run backend locally on port 5000

```
1 python main.py
```

2.2 Build Frontend

VueJS introduction: Vue.js is an open-source model–view–viewmodel front end JavaScript framework for building user interfaces and single-page applica-

tions. It was created by Evan You, and is maintained by him and the rest of the active core team members

2.2.1 Components

Include 3 pages

1. Homepage: Some categories, and popular items among users

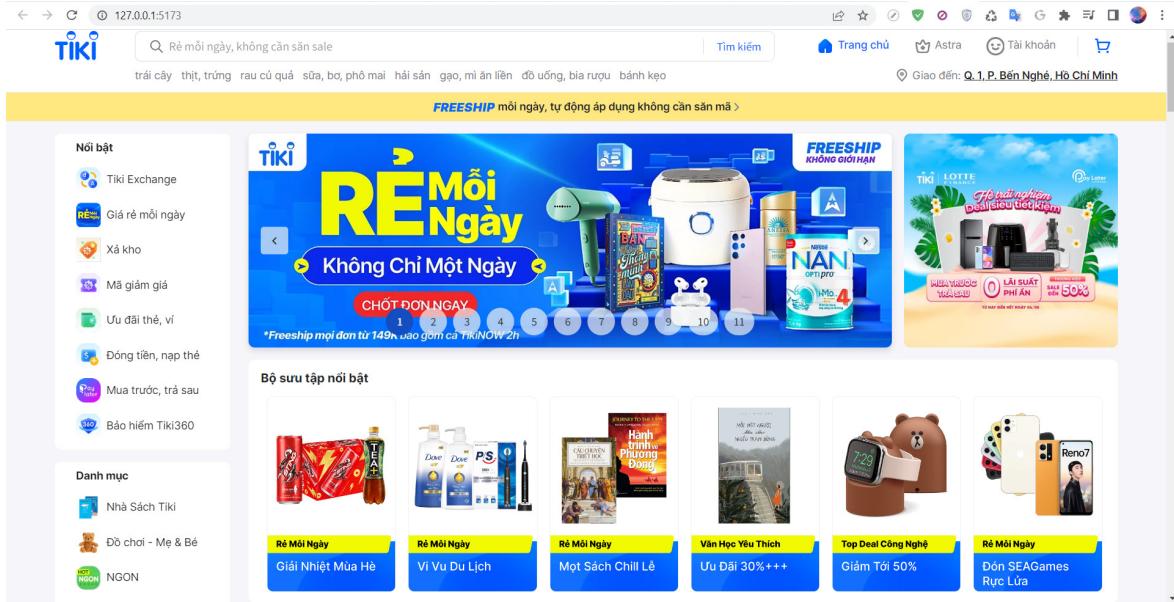


Figure 6: Homepage

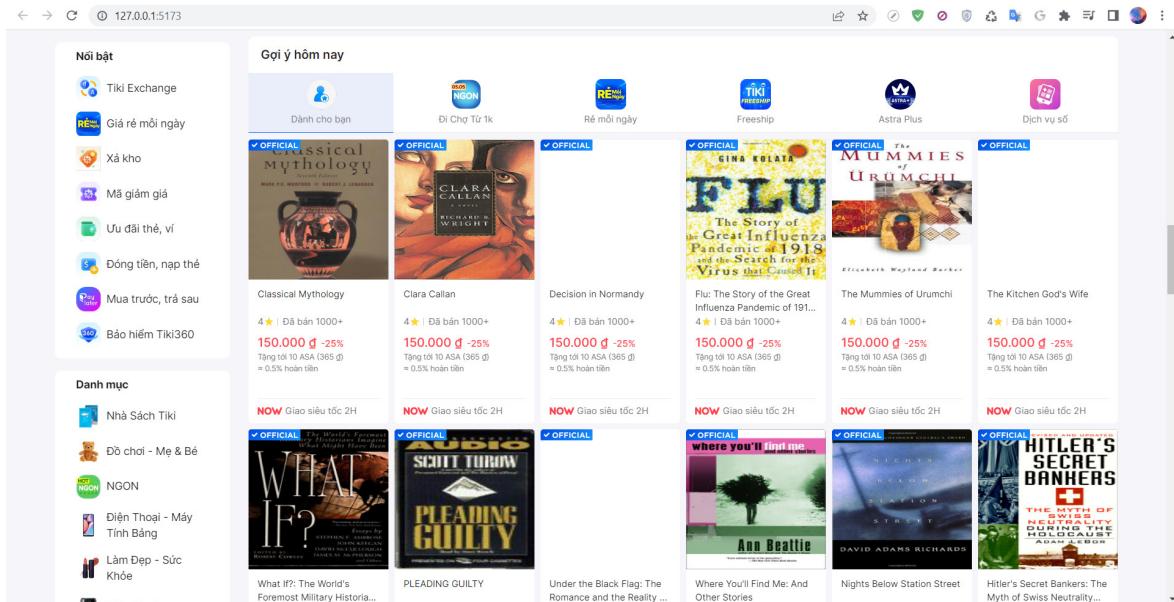


Figure 7: Book category in homepage

2. Book category page: All books in E-Commerce using pagination.

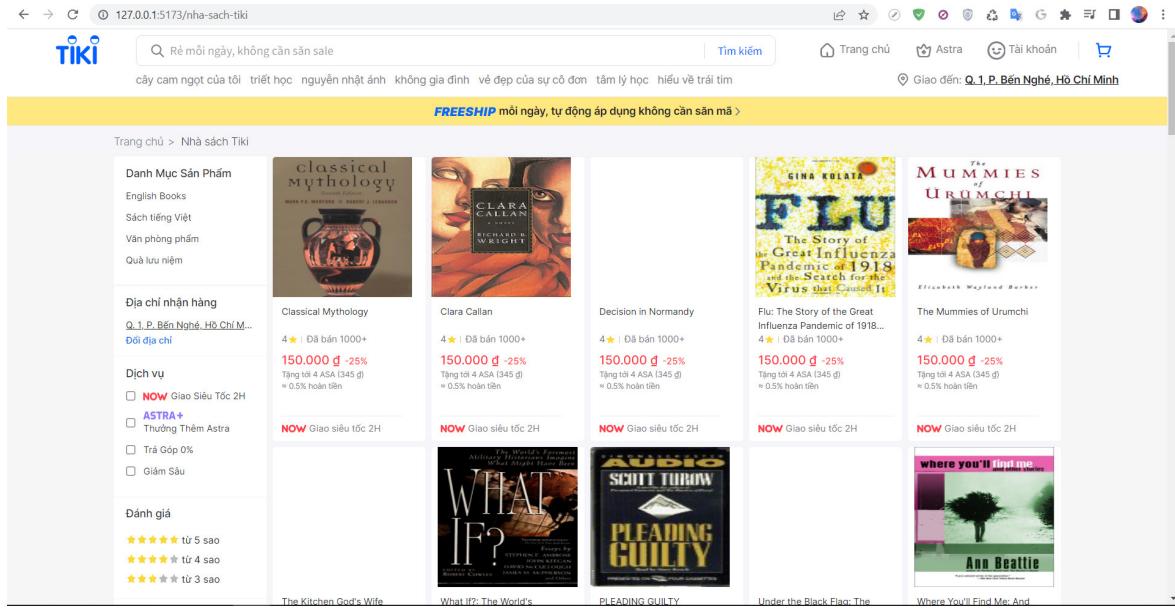


Figure 8: Book category page

3. Book detail page: Detail of a book and all similar book using Louvain, Leiden, Girvan Newman.

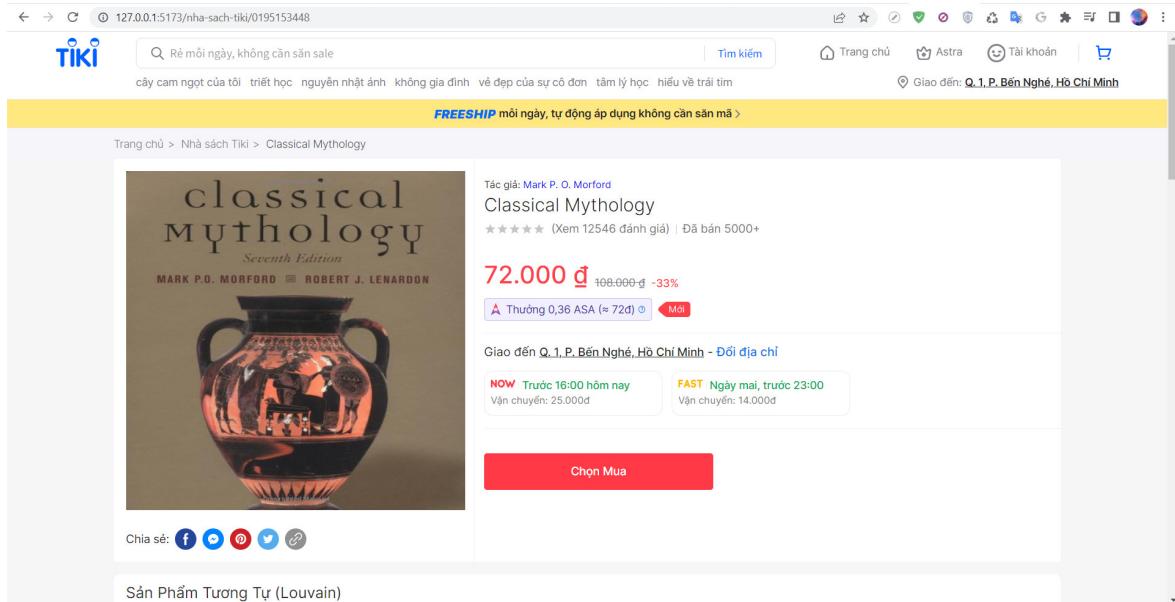


Figure 9: Book detail page

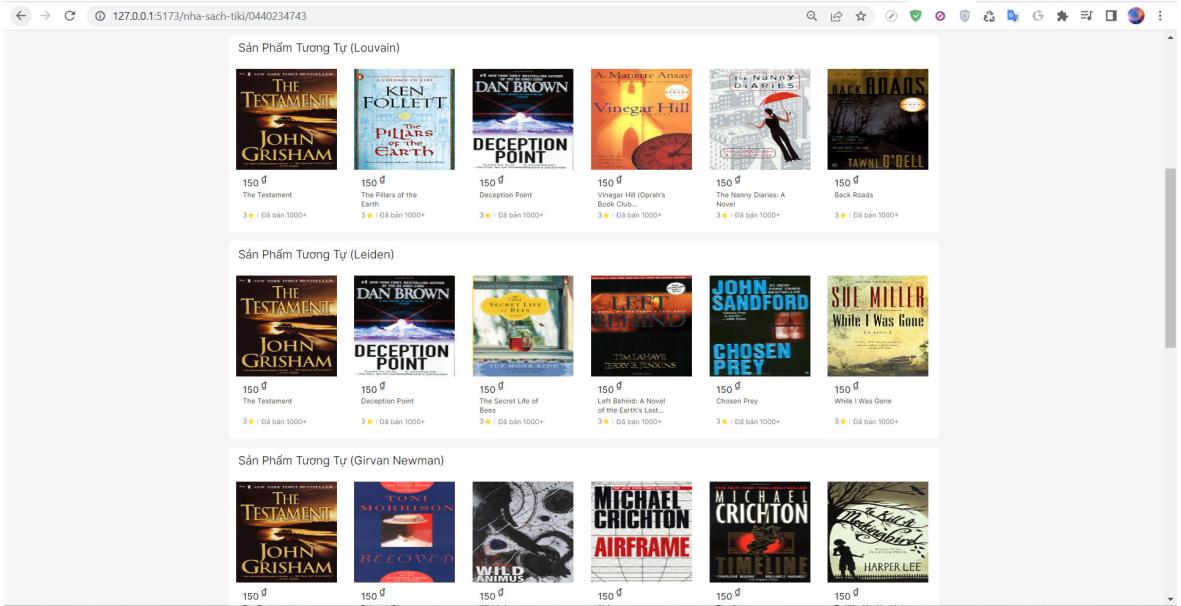


Figure 10: Book detail page recommendation algorithms

3. Modularity score comparison

To improve the efficiency of measuring algorithm modularity, we implemented a strategy of dividing the dataset into smaller subsets. Our approach involved selecting the top 500 books that appeared in all book pairs from the previous Map Reduce step. This selection criterion was chosen over random selection from all pairs to avoid generating a sparse graph that would result in significantly lower modularity scores for all three algorithms, making it challenging to measure them accurately.

By ensuring that the selected nodes had the highest number of edges, we aimed to facilitate faster execution of all three algorithms

```

▶ # print number of edges and nodes
print("Number of edges:", G.number_of_edges())
print("Number of nodes:", G.number_of_nodes())

```

⇨ Number of edges: 15652
Number of nodes: 500

Figure 11: Number of edge and node for modularity measurement

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'visualize.py', 'girvan_newman.ipynb', 'Leiden.ipynb', and 'louvain.ipynb'. The 'louvain.ipynb' tab is active. The code cell contains the following Python code:

```
from networkx.algorithms.community.quality import modularity

mod = modularity(G, partition)

print('Modularity:', mod)
```

The output of the code is:

```
... Execution time: 0.1195535659790039 seconds
...
Modularity: 0.27143231973645493
```

Figure 12: modularity measurement for Louvain algorithms

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'visualize.py', 'girvan_newman.ipynb', 'Leiden.ipynb', and 'louvain.ipynb'. The 'Leiden.ipynb' tab is active. The code cell contains the following Python code:

```
from networkx.algorithms.community.quality import modularity

mod = modularity(G, partition)

print('Modularity:', mod)
```

The output of the code is:

```
... Execution time: 0.11145472526550293 seconds
...
Modularity: 0.2750441061038088
```

Figure 13: modularity measurement for Leiden algorithms

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'visualize.py', 'girvan_newman.ipynb' (which is the active tab), 'Leiden.ipynb', and 'louvain.ipynb'. Below the tabs, the file path is displayed: 'C: > Users > maing > Desktop > girvan_newman.ipynb > from networkx.algorithms.community.modularity'. There are buttons for '+ Code', '+ Markdown', and '...'. The main area shows a code cell with the following content:

```
from networkx.algorithms.community.quality import modularity
mod = modularity(G, communities)

print('Modularity of the best partition:', mod)
```

The cell has a status bar indicating '[167]'. Below the code cell, the output is shown:

```
Modularity of the best partition: 0.0021023578934315575
```

Figure 14: modularity measurement for Girvan Newman algorithms

Louvain: The Louvain algorithm is recognized for its ability to generate community partitions with high modularity scores. Its iterative optimization process helps identify dense subgraphs and optimize modularity locally and globally.

Leiden: The Leiden algorithm slightly higher score than Louvain.

Girvan-Newman: The Girvan-Newman algorithm tends to have a lower modularity score compared to both Louvain and Leiden. This is because its divisive edge removal approach may not capture the community structure as effectively as the other two algorithms in most cases.

4. Time Efficiency comparison

To improve the efficiency of measuring algorithm modularity, we implemented a strategy of dividing the dataset into smaller subsets. Our approach involved selecting the top 500 books that appeared in all book pairs from the previous Map Reduce step. To ensure that all three algorithms were executed under similar conditions with large input size.

```
# print number of edges and nodes
print("Number of edges:", G.number_of_edges())
print("Number of nodes:", G.number_of_nodes())
```

Number of edges: 15652
Number of nodes: 500

Figure 15: Number of edge and node for time efficiency measurement

The screenshot shows a Jupyter Notebook interface. The title bar says "girvan_newman.ipynb". The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. On the left, there are icons for file operations like new, open, save, and run. A code cell is selected, indicated by a grey background. The code inside the cell is:

```
start_time = time.time()
communities = girvan_newman(G, 15)
end_time = time.time()
execution_time = end_time - start_time
print("Execution time:", execution_time, "seconds")
```

Below the code cell, the output is displayed in a separate box:

```
Execution time: 3748.941060781479 seconds
```

Figure 16: modularity measurement for Girvan Newman algorithms

The screenshot shows a Jupyter Notebook interface with the title "louvain.ipynb". The code cell contains the following Python script:

```
start_time = time.time()
partition = nx.community.louvain_communities(G)
end_time = time.time()
execution_time = end_time - start_time
print("Execution time:", execution_time, "seconds")
```

The output cell shows the execution time:

```
Execution time: 0.34261059761047363 seconds
```

Figure 17: modularity measurement for Louvain algorithms

The screenshot shows a Jupyter Notebook interface with the title "Leiden.ipynb". The code cell contains the following Python script:

```
execution_time = end_time - start_time
print("Execution time:", execution_time, "seconds")
```

The output cell shows the execution time:

```
Execution time: 0.5217368602752686 seconds
```

Figure 18: modularity measurement for Leiden algorithms

Leiden: The Leiden algorithm is known for its efficiency, particularly in handling large-scale networks. It utilizes techniques such as smart local moving and graph contraction to improve performance.

Louvain: The Louvain algorithm is also efficient and widely used for community detection. It employs a two-phase approach, iteratively optimizing modularity at the local and global levels. However, it can face scalability challenges

with very large networks. In this comparison, the execution times of Louvain and Leiden are similar, possibly because the Louvain algorithm has been improved with support from libraries or optimized implementations.

Girvan-Newman: The Girvan-Newman algorithm uses a divisive approach, iteratively removing edges with the highest betweenness centrality to break the graph into communities. This approach is significantly more computationally expensive compared to Louvain and Leiden (3748 seconds), especially for large networks. This is because calculating edge betweenness centrality for all edges can be time-consuming and resource-intensive.

5. Overall comparison

We have the result of each algorithms:

	Girvan Newman	Louvain	Leiden
Modularity	0.0021	0.2714	0.2750
Time(s)	3748.9411	0.3426	0.5217
Number of communities	12	21	11

In conclusion, the Louvain and Leiden algorithms generally perform well in terms of modularity score and time efficiency. They are efficient and effective for community detection in various network sizes. However, the Girvan-Newman algorithm, while effective, may not achieve as high modularity scores and much more computationally expensive. It is important to consider the specific characteristics of the network and the desired trade-offs between modularity and runtime when selecting an appropriate algorithm for community detection.

V. Conclusion

1. Achievement

The algorithms that the group used to solve the community search problem for the artist dataset above accomplished the following:

- Our team have implemented and runned 3 algorithms (Girvan Newman, Louvain, Leiden) on Python, from that, we can see some comparisions and how they works.
- Using the above data clusters, we can suggest similar books to the user while they are seeing the information of the books they want to buy, increasing the diversity of the e-commerce website such as Tiki.
- Analyzing the tastes of the readers community in specific clusters to promote better suggestions and apply them to run ads or PR is another strength that this algorithm brings.
- Besides, we also build a website to demonstrate our work and its application. The demonstration video of the website is here: <https://youtu.be/nnhuGfggjb4>

2. Drawback

Because of the limited research time, the group did not generate many algorithms or make very specific comparisons with the algorithms sought.

However, with the above three algorithms,Girvan-Newmana and Louvain, classical algorithms, and Leiden, an improved algorithm widely used today, the group has also made appropriate comparisons of performance and reliability to generalize them.

3. Future work

The team plans to use more algorithms in the future to get a thorough understanding of the approaches used by businesses and researchers to address problems.

The team also intends to work with more actual data sets in order to fully assess the algorithm' s efficacy and its potential for use in other contexts. This deals with, for instance, developing techniques for providing suggestions or advertising using these data clusters.

References.

- [1] Dataset source: <https://www.kaggle.com/datasets/arashnic/book-recommendation-dataset>
- [2] Lilian Weng, Filippo Menczer Yong-Yeol Ahn (2013). "Virality Prediction and Community Structure in Social Networks". Last access 26/05/2023 https://www.researchgate.net/publication/256188948_Virality_Prediction_and_Community_Structure_in_Social_Networks
- [3] Fabian Nguyen (2021). "Leiden-Based Parallel Community Detection". Last access 26/05/2023 https://i11www.iti.kit.edu/_media/teaching/theses/ba-nguyen-21.pdf
- [4] How Authors Can Find Their Ideal Reading Audience. Last access 26/05/2023 <https://www.janefriedman.com/how-authors-can-find-readers/>