Advanced Topics in Graph Mining: Final Report

Xin Tan*
xtan22@cse.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong, China

Shaofeng Wu*
wsf123@cse.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong, China

1 OVERVIEW

Communities are ubiquitous in graphs abstracted from real-world scenarios. For example, users form communities by subscribing to the same content providers in social networks, such as the massive user network in Twitter. How to exploit this hidden information in real graphs that exhibits various forms(i.e. directed or undirected, with attribute or without attribute) motivates researchers to make continuous efforts in fast and accurate graph detection algorithms. Along with it, distributed graph processing frameworks and frameworks are also widely studied and proposed to provide efficient runtime support for community detection on large-scale graphs.

In this course project for CSCI5630 Advanced Topics in Graph Mining, we focus on community detection algorithms due to their importance in graph mining. The major contents of our course project are as follows:

(1) [1]A comprehensive survey about community detection algorithms, especially well-known traditional community detection ones. We summarize the key ideas and provide algorithm pseudo-code for representative algorithms including Label Propagation Algorithm(LPA), Walktrap and InfoMap. Technical relating topics to community detection algorithms, i.e. community detection datasets and distributed graph processing frameworks, are also discussed in our report.
(2) Implementation of two community detection algorithms. We implement LPA and InfoMap with a total of 1247 LOCs. The algorithms are implemented in scala to adapt to the programming model and abstraction of Spark platform.
(3) Evaluation of the implemented algorithms. For the evaluation, we sort several open-source datasets and set up our experiment testbed. Then We evaluate and compare the above mentioned algorithms based on two key metrics that are widely adopted in community detection: modularity and execution time, and propose several analysis based on empirical results.

The remaining part of the report is organized as follows:

(1) In section 2, we review the first-stage work of this project and introduce necessary preliminaries of community detection problem, community detection algorithms and relating technical issues such as distributed graph processing frameworks.
(2) In section 3, we briefly illustrate our algorithm implementation by walking through important functions and application logic in our source code.
(3) In section 4, we show the evaluation results of our target community detection algorithms with comparison, analysis and explanations.

*Both authors are in group 117. Both author contribute equally to the project. Xin Tan is mainly responsible for the paper survey, environment configuration and report writing. Shaofeng Wu mainly work on the implementation and evaluation of the algorithms.

[1]The major contents of this part are in our midterm report. Please check midterm report for more details.

2 BACKGROUND

In this part, we first review the major work of our midterm report. Then we briefly provide background knowledge about community detection algorithms and distributed graph processing frameworks.

2.1 Previous Work

In the first stage of our course project, we did a comprehensive survey about traditional community detection algorithms. Among these algorithms, we paid special attention to three representative algorithms: label propagation algorithm, walktrap and InfoMap, providing detailed algorithm explanation and pseudo-code. In addition, we also searched popular graph datasets and SOTA grpah processing systems that could be used for evaluating these algorithms. In the end, we chose Spark as the graph processing platform and showed a LPA demo on karate club dataset in local mode of Spark. The following two sections will briefly re-discuss the survey part of our project with additional information and insights.

2.2 Community Detection Algorithms

Community detection algorithms can be generally categorized into traditional algorithms and GNN-based algorithms[6]. Among traditional methods, different algorithms vary significantly with respect to their underlying mechanism. We list seven representative traditional community detection algorithms in Table 1 for reference. For this project, two algorithms, which are LPA and InfoMap, are selected for implementation and evaluation. These two algorithms represent message-passing and learning-based methods respectively, and with evaluation and comparison of these two algorithms based on two distinct mechanism, we hope to discover performance insights as well as advantages and disadvantages between these two algorithm categories.

LPA LPA is a community detection algorithm based on message-passing model in graphs[2]. The main idea of LPA is that a node's community label should be the same as the majority of its neighbours' label. The algorithm has the following major steps:

(1) Initialize each node V_i with a random community label L_i.
(2) Set k to 1.
(3) Sort all nodes in random order.
(4) For each node in the random sequence, update its label with the most labels of its neighbors'.

Algorithm	Category	Time Complexity
LPA	message-passing	$O(E)$
Fastgreedy	optimization-based	$O(Nlog^2(N))$
Multilevel	optimization-based	$O(Nlog(N))$
Leading eigenvector	optimization-based	$O(kN(E+N))$
Spinglass	optimization-based	$O(N^{3.2})$
InfoMap	random walk & optimization-based	$O(E)$
Walktrap	random walk based	$O(N^2log(N))$

Table 1: A List of Representative Traditional Community Detection Algorithms

(5) If all labels equal the the most labels if the node's neighbors' or k exceeds limit of the iteration rounds, return the result. Else set $k = k + 1$ and go to step (3).

LPA algorithm is shown in Algorithm 1

As has been mentioned in Table 1, LPA has the advantages of low time complexity due to its simplicity. However, its limitations are also obvious as it does not consider overlapping communities and predefined iteration rounds can not provide convergence guarantee on arbitrary graphs.

Algorithm 1 Label Propagation

Input: $G(V, E)$, maximum iteration rounds K
Output: L, where L is the list of community labels of each vertice in V

Initialization: each $V_i \in V$, assign it with a random label L_i
$k \leftarrow 0$
while $k \leq K$ **do**
  $V' \leftarrow$ random sort of V
  for j to $len(V')$ **do**
    $L_j \leftarrow$ the most labels of V_j's neighbors
  end for
  $k \leftarrow k + 1$
  if each L_i == the most labels of V_i's neighbors **then**
    break
  end if
end while
return L

InfoMap Information Mapping(InfoMap)[4][3] is an optimization-based community detection algorithm that leverages Random Walk and hierarchical encoding. The main idea of InfoMap is: a long random walk in a graph can obtain a sample sequence, which contains implicit communities. The nodes that are in the same communities will reside closely in the random walk sequence. A good hierarchical encoding scheme, namely a community assignment scheme, will have relatively low coding length. The InfoMap algorithm(Algorithm 2) follows these steps:

$$L(M) = q_{\curvearrowright} H(\boldsymbol{Q}) + \sum_i p_{i\circlearrowright} H(\boldsymbol{\mathcal{P}}^i) \quad (1)$$

(1) Initialize each node as a separate community.

(2) Generate a long-enough random sequence from all nodes. Try merging each node into its neighbors' communities from the beginning of the sequence. Select the scheme that decrease 1 the most for each node. Keep visiting the next node in the random sequence until Equation 1 no longer decreases.

Algorithm 2 InfoMap

Input: $G(V, E)$
Output: Communities result

Initialization: encode each $V_i \in V$ forms a separate category
Generate a random sequence $S = V_0, V_1, ...$ of nodes
$i \leftarrow 0, l \leftarrow 0$
while $\Delta l > \epsilon$ **do**
  merge V_i into its neighbor's community that decrease Formula 1 most
  calculate l' with Formula 1
  $\Delta l = l - l'$
  $l \leftarrow l'$
  $i \leftarrow i + 1$
end while
return communities of all nodes

InfoMap can be viewed as a learning-based community detection method, where vertices are randomly selected as mini-batches to optimize total coding length. The study on derivatives of InfoMap is still active until today since its publishing in 2009. The time complexity of InfoMap is $O(kE)$ but it may suffer from high memory overhead due to the generation of complex encoding and prolonged training time on large-scale graphs due to poor convergence.

2.3 Graph Processing Frameworks

A great number of graph processing systems and frameworks have been proposed to provide efficient platforms for various graph tasks including community detection. A well-designed graph processing framework need to provide proper and complete programming abstraction for developers, while considering graph-specific and algorithm-specific optimizations such as graph partitioning with less communication overhead. Typical graph processing frameworks includes Spark GraphX, GraphLab, Neo4j, NetworkX, etc..

Among various graph processing frameworks, Spark(Figure 1) is a general purpose data processing framework for single node or clusters. The GraphX library extends Spark with graph-parallel computation and exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API[1]. For this project, we selected Spark as our implementation platform due to the simplicity of deployment and high processing efficiency reported.

3 IMPLEMENTATION

In our project, we implemented LPA and InfoMap with scala. We use scala to implement these algorithms due to its affinity to Spark[5]. The implementation contains a total of 1247 LOCs. Due to the strict directory constraint of sbt[2], each algorithm has its distinct project

[2] sbt is the compilation tool for scala.

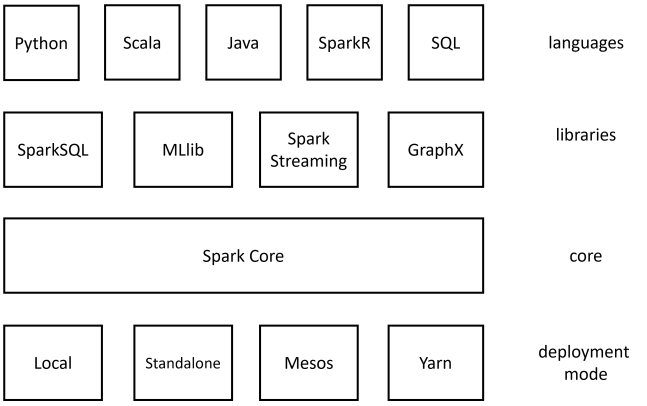


Figure 1: Architecture of Spark.

directory with all of the source code, datasets, building configurations and related contents. To maintain the conciseness of the report, we will only discuss the most important part of the source code to facilitate better understanding of our implementations. More details of the implementation can be found in our source code.

3.1 LPA

The source code of LPA algorithm is stored in "/LPA/src/main/scala". The main function of LPA will read the runtime configuration from "/LPA/config.json", which specifies Spark configurations, algorithm parameters and I/O directories. Then the LPA algorithm will be called on targeted graph. Built-in timer of scala is used for recording execution time of the algorithm. Finally, community detection results will be shown in standard I/O or in file. The main function of other algorithms follow the same steps as mentioned above, so we will not repeatedly discuss them for the rest part of this section.

```
object LPA {
    def main(args: Array[String]): Unit = {
        // load configuration file
        val config_file = "config.json"
        val config = new JsonReader(config_file)
        val graph_file =
             config.getObj("Graph").value.toString
        val( spark, context ) = init_spark(
             config.getObj("spark configs") )
        val LPA_config = config.getObj("LPA")
        val num_iter = LPA_config.getObj("num
             iterations").value.toString.toInt
        val log_file = init_log( context,
             config.getObj("log") )

        // load the input graph file
        val G = GraphLoader.edgeListFile(context, graph_file)

        // run lpa algorithm for several iterations and
             record running time
        val t1 = System.nanoTime
        val G_result = lpa(G,num_iter)
        val duration = (System.nanoTime - t1) / 1e9d
        println(s"Running time: $duration s")

        // print community assignment result
        G_result.vertices.sortBy(_._2).foreach {
            case (id, (group)) => println(s"$id $group")
        }
    }
    // other contents are omitted
}
```

For the LPA algorithm, the implementation follows the message passing model of Pregel to enable concise programming. There are three important functions:

(1) `msg_send`: A typical routing to send message to other nodes. The message is the community number of the vertex.
(2) `msg_merge`: Merge all the message sent to the verte into one message. It will count how many times a community number appears in all of the messages and store the result as a key-value tuple (community id, count).
(3) `vprogram`: The vertex program. It finds the community id with highest count as the community assignment for the vertex in this round.

```
def vprogram(vid: VertexId, attr: Long, msg:
     Map[VertexId, Long]): VertexId = {
    if (msg.isEmpty) attr else msg.maxBy(_._2)._1
}
```

The LPA function will use these user-defined functions together with other necessary arguments, i.e. iteration rounds, to define the Pregel instance.

```
Pregel(lpaGraph,init_msg,maxIterations=max_round)(
    vprog = vprogram,
    sendMsg = msg_send,
    mergeMsg = msg_merge
)
```

3.2 InfoMap

For InfoMap algorithm, we adopt a recursive structure to naturally adapt to the requirement that the algorithm needs to walk along a long random sequence. The major part of the InfoMap algorithm is shown below.

```
def merge_recur(loop: Int,qsum: Double,graph: Graph,part:
     Partition,mergeList: RDD[((Long,Long),Merge)]):
     (Graph, Partition) = {
    if(mergeList.count == 0 )
    return terminate(loop, logFile, graph, part)

    val merge = merge_find(mergeList)
    if(merge._2.dL > epsilon )
    return terminate(loop, logFile, graph, part)

    val new_qsum = qsum_calculate(part, merge, qsum)
    val new_part = part_calculate(part, merge)
    val new_graph = graph_calculate(graph, merge)
    val new_mergel = mergelist_update(merge, mergeList,
         new_part, new_qsum)
```

```
13      merge_recur(loop+1, new_qsum, new_graph, new_part,
            new_mergel)
14  }
```

The merge_recur is a recursive function that greedily merge vertex into the community that decrease the code length the most and recursively walk onto next random vertex. As shown from line 2 to line 7, the recursive algorithm will terminate when the whole graph has been merged into one module or the greedy merge cause the code length to increase beyond a certain threshold. From line 9 to line 13, the algorithm update several necessary values, i.e. the transmit possibility of the graph after merge, the community result, the merge list of vertices, etc., and then calls next merge based on these updated values. It can be observed from the source code of the algorithm that InfoMap relies on a hyper-parameter(epsilon in the code) for convergence. The convergence speed of the algorithm are heavily affected by this parameter, especially on large-scale graphs.

4 EVALUATION

4.1 Experiment Configuration

In the experiment part, we evaluate LPA and InfoMap from the aspects of their clustering quality and time complexity with multiple configurations. Our experiments are conducted in both local and remote settings. For local setting, Spark uses dedicated threads as workers to emulate distributed mode. The distributed setting is achieved through standalone mode of Spark, where a master machine and several worker machines consist a computing cluster. All machines used in distributed setting have the same hardware and software configurations as the machine used in local mode. And the communication between host and worker or workers are achieved with 100Gb/s Ethernet. The hardware and software configurations of our master and worker machines are shown in Table 2. For each

CPU	Intel E5-2683 v3 14-core CPUs at 2.00 GHz
Memory	256GB ECC Memory
NIC	Dual-port Intel 10Gbe NIC (X520)
Kernel	4.15.0-169-generic
OS	Ubuntu 18.04.1 LTS
Java version	11.0.17
Spark version	3.3.1 Pre-built for Hadoop 3.3 and later
sbt version	1.8.0
Scala version	2.12.17

Table 2: Hardware and Software Configurations

different configuration, we run the experiment for five times and take the average of results as the final result. Each algorithm is evaluated with three datasets with the size ranging from small($|V| < 100$), medium($100 \leq |V| < 10000$) and large($10000 \leq |V|$). The details of these datasets are shown in Table 3.

4.2 Performance

Table 4 shows the execution time and modularity of LPA and InofMap using one single-core Spark executor. The execution time reflects the end-to-end performance of these two algorithms, while

Dataset	Vertex Number	Edge Number
Zachary's karate club[7]	34	78
political blogs	1490	19090
DBLP	317080	1049866

Table 3: Information of Graph Datasets Used in Evaluation

modularity is a quantative metric that evaluates the quality of the community assignment result. Modularity can be calculated with Equation 2, where m is the number of edges in the graph, l_c is the inter-edges of community c and d_c is the sum of vertices' degree within the community c. A good clustering scheme usually has a Q between 0.3 and 0.7.

$$Q = \sum_{c=1}^{k} (\frac{l_c}{m} - (\frac{d_c}{2m})^2) \tag{2}$$

For all three datasets, LPA achieves significantly lower execution

Datasets	Execution Time(s)		Modularity	
	LPA	InfoMap	LPA	InfoMap
Zachary's karate club	1.99	4.17	-0.056	0.21
political blogs	3.16	223.66	0.43	0.33
DBLP	43.24	923.83	0.56	0.00013

Table 4: Performance comparison between LPA and InfoMap. We run LPA algorithm for 5 rounds in this experiment. Both algorithms are tested with three datasets using 1 executor, 1 core and local mode of Spark.

time compared with InfoMap, namely 0.47×, 0.014× and 0.047× of InfoMap's training time for karate club, political blogs and DBLP dataset respectively. Although theoretically both LPA and InfoMap has linear($O(kE)$) time complexity, the coefficient k of these two algorithms vary significantly. For LPA, k is proportional to the number of running round, which is a pre-defined parameter and can be very small if LPA converges fast on specific dataset. However, InfoMap doess not have pre-defined training round and the k in InfoMap depends on the convergence speed of the algorithm, which is slow on large-scale datasets.

For the quality of community detection, LPA outperforms InfoMap on political blogs and DBLP. For small-scale datasets, LPA suffers from over diffusion problem, leading the whole graph to form less modules. This problem can not be avoided sometimes since the vanilla implementation LPA algorithm is coarse-grained, where broadcasting will be conducted in each round for every vertex.

One issue to note there is that InfoMap has poor convergency on large-scale datasets. For DBLP dataset, setting the parameter epsilon in InfoMap to zero, which is the vanilla implementation of InfoMap, causes the training time to exceed 24 hours. Therefore, we set epsilon to a relatively low value(-0.00002 in this case) to achieve tradeoff between execution time and modularity. However, the modularity is still very low for this configuration, showing the poor convergency of InfoMap on large datasets.

4.3 Scalability

To investigate the scalability of LPA and InfoMap when deployed on Spark, we submit community detection job on political blogs dataset to a cluster of 1, 2 and 4 single-core Spark executors respectively. As shown in Figure 2, our implementation of LPA does not benefit from the parallelism provided by multi-executor in Spark, but InfoMap algorithm is accelerated 1.2× and 1.4× times with 2 and 4 executors compared with single executor scenario. This is mainly due to the computation intensive nature of InfoMap[4], which makes it benefit from more CPU cycles provided by more executors. For LPA, as the dataset can be fitted into a single executor, message passing between executors may bring overhead and can not benefit from more executors with our current implementation. This result also implies that for even larger graphs, InfoMap can exploit more parallelism from Spark's executor-based scheduling model.

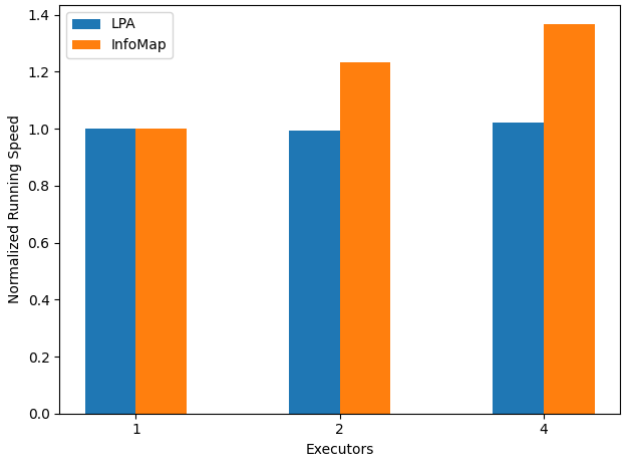


Figure 2: Running speed of LPA and InfoMap with 1,2 and 4 Spark executors.

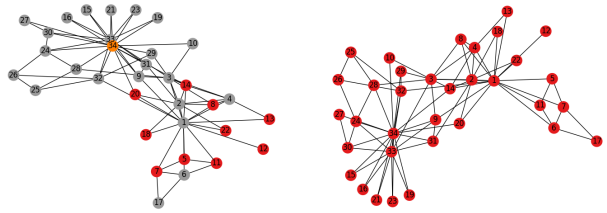
4.4 Convergency

We also record necessary metrics to study the convergence speed of LPA and InfoMap. The modularity in each round of label propagation is shown in Table 5. From a theoretical perspective, LPA may fail to converge on some special graphs. Our experiment result demonstrates that LPA can quickly converge on political blogs dataset, which does not conflict with theoretical convergecy of LPA since real-world graphs hardly has the special form that prevents LPA from converging, and sometimes we do not even require convergency for LPA since it can achieve a fair community assignment result without convergence. Despite the quick convergence of LPA, over diffusion, which has been mentioned in Section 4.2, may occur especially for small graphs(Figure 3). This is because the major labels are easy to be propagated across small graphs, which has small graph diameter.

Figure 4 shows the convergence curve of InfoMap algorithm also running on political blogs datasets. InfoMap converges at 250 s after the training starts. Based on previous results, LPA usually converges faster than InfoMap due to their different mechanism. For each random step, InfoMap has to conduct massive computations. What's more, each step of InfoMap merges at most one vertex into other communities. These two factors significantly slows down the optimization process of InfoMap. A mini-batch based optimization may help the algorithm to achieve quicker convergence and higher training speed.

Round	Modularity
0	0
1	0.038
2	0.25
3	0.40
5	0.43
10	0.43

Table 5: Modularity in each round of LPA running on political blogs dataset.



(a) 5th round (b) 10th round

Figure 3: Over diffusion of LPA on Zachary's karate club dataset.

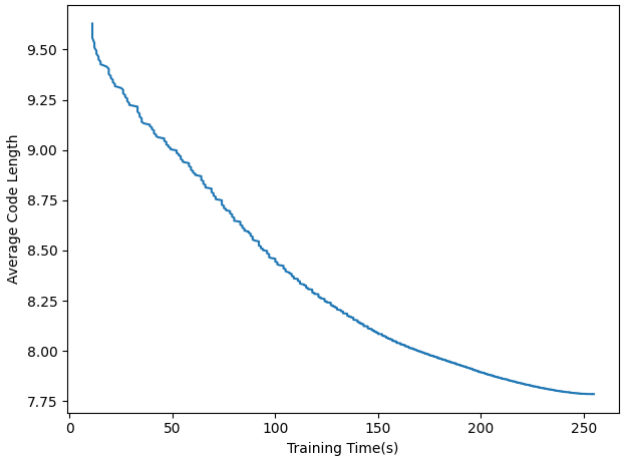


Figure 4: Convergence curve of InfoMap running on political blogs dataset. The algorithm converges when average code length starts to increase with the current merge operation.

5 SUMMARY

In this course project, our major focus is on community detection algorithms. Firstly, we did comprehensive survey on the traditional community detection algorithms and distributed graph processing frameworks with special focus on key ideas and characteristics of three algorithms, which are LPA and InfoMap. Then, we implement these algorithms with scala. Lastly, we evaluate these algorithms on Spark and compare their clustering quality and time complexity on various datasets. This course project provides insights on the empirical performance comparison of these three representative algorithms, which are based on three different underlying mechanism to discover community structure respectively. Additionally, we also learnt valuable lessons about the design principle, programming abstraction and limitations of SOTA graph processing frameworks such as Spark, which could hopefully inspire our future work in system studies.

REFERENCES

[1] Apache Spark 2022. Official Document. https://spark.apache.org/docs/.

[2] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76 (Sep 2007), 036106. Issue 3. https://doi.org/10.1103/PhysRevE.76.036106

[3] Martin Rosvall, D. Axelsson, and Carl T. Bergstrom. 2007. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences* 104, 18 (2007), 7327–7331. https://doi.org/10.1073/pnas.0611034104 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.0611034104

[4] Martin Rosvall and Carl T. Bergstrom. 2007. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences* 104, 18 (2007), 7327–7331. https://doi.org/10.1073/pnas.0611034104 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.0611034104

[5] Spark 2022. Official Github Repository. https://github.com/apache/spark/.

[6] Zhao Yang, René Algesheimer, and Claudio J. Tessone. 2016. A Comparative Analysis of Community Detection Algorithms on Artificial Networks. *Scientific Reports* 6 (Aug 2016), 30750. Issue 1. https://doi.org/10.1038/srep30750

[7] Wayne W. Zachary. 1977. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research* 33, 4 (1977), 452–473. https://doi.org/10.1086/jar.33.4.3629752 arXiv:https://doi.org/10.1086/jar.33.4.3629752