Layout and Visualization of Large Scale Citation Network

Li Jingyu

517030910318

1 Introduction

Nowadays, the Internet has created many larger and larger communities. While many of these communities can be represented as a directed graph, such as academic citation networks and social networks, how to visualize them is a big problem. Traditional force-directed graph layout algorithms often produce desirable results. However, these algorithms, such as ForceAtlas2 and Yifan Hu can only tackle a small scale graph about 100,000 nodes, while today's graph can be larger than millions or even billions of nodes. Instead of creating more and more powerful single layout algorithms, another way is to create a framework that can be applied to any scale of networks, which integrates the existing methods.

In this project, my work includes:

1. Design and implement an recursive and hierarchical large graph layout framework. The main idea is to partition and layout sub-graphs recursively.
2. Choose Louvain as the partition (community detection) algorithm and ForceAtlas2 as the single graph layout algorithm, and implement them. While ForceAtlas2 is implemented by existing toolkit interface, Louvain algorithm is implemented from scratch by myself.
3. Test the algorithms on DBLP database, which contains a citation network with about 5,000,000 nodes and 50,000,000 edges, and visualize the results in static images.
4. Build a web demo to visualize the layout results and interact with the layout network.

The code is open sourced at https://github.com/LiTrevize/Citation-Network

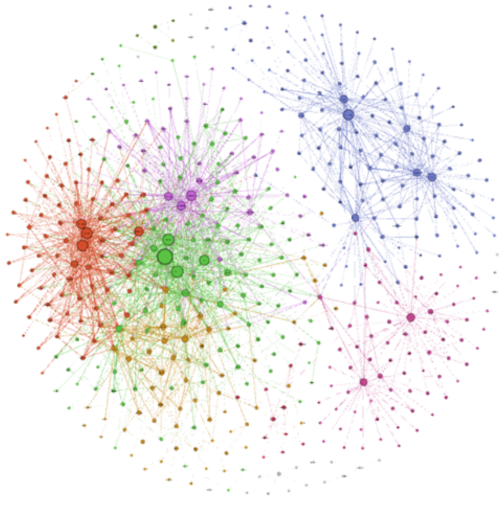
2 Approach

2.1 Problem Statement

We have a citation network $G = (V, E)$, where each node v denotes a paper and each edge e is a directed (v_i, v_j) pair, which means j cites i. We want to get a layout $L : G \to X \times Y \times R$, where X, Y are coordinates and R is the size of the nodes. The layout should provide structured information; e.g. It puts related nodes together and form clusters. Then how to layout G when $|V|$ is very large?

2.2 Algorithm Design

Existing force-directed graph layout can often produces desirable results on small networks, as shown in Figure 1. However, for larger graphs, these algorithms turn out to be inefficient and ineffective.


(a) ForceAtlas2

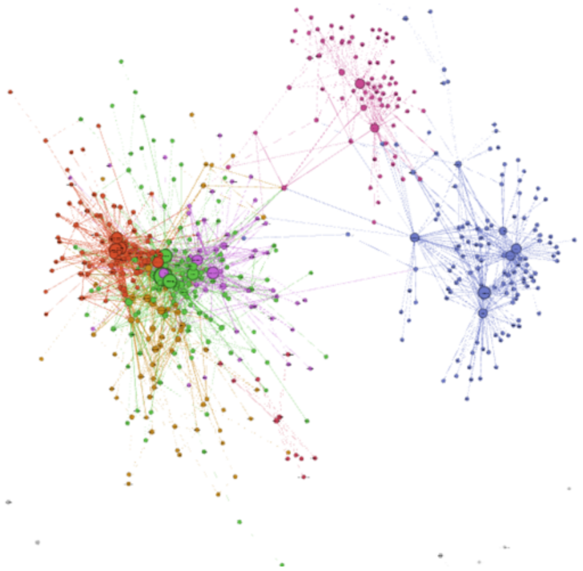

(b) Yifan Hu

Figure 1: Force-directed layout algorithms on small graphs

Therefore, to integrate these algorithms to larger graphs, we have to reduce the graph's size. We can first partition the graph into several small subgraphs (also called communities), and use small layout algorithms on each of these sub-graphs. Each subgraphs themselves can be regarded as a supernode in the community supergraph, so we can use algorithms on the supergraph as well. But either the subgraphs or the supergraph can be too large as well, so we repeat the above operations recursively. Finally, we need to merge all these sub-layout into one final layout. This is done by resize and squeeze the subgraph layout into its corresponding supernode (community). The whole algorithm is shown below.

Algorithm 1: Large Graph Layout Algorithm

```
Data: Citation Network $G = (V, E)$
Result: Layout $L = (X, Y, R) \in \mathbb{R}^{|V|} \times \mathbb{R}^{|V|} \times \mathbb{R}^{|V|}$
$L =$ LargeGraphLayout($G$);
begin
    if $|V| < V_{thres}$ then
        $L \leftarrow$ SmallGraphLayout($G$);
    else
        $\mathbb{G}_{sub}, G_{sup} \leftarrow$ Partition($G$);
        foreach $G_{sub,i}$ in $\mathbb{G}_{sub}$ do
            $L_i \leftarrow$ LargeGraphLayout($G_{sub,i}$);
        end
        $L_{sup} \leftarrow$ LargeGraphLayout($G_{sup}$);
        $n \leftarrow |\mathbb{G}_{sub}|$;
        $L \leftarrow$ MergeLayout($L_{sup}, L_1, \ldots, L_n$);
    end
end
```

So why would the proposed framework work? That is, why it's more efficient than directly using small graph layout algorithms on large networks. As shown in section 2.3.2, the implementation of partition algorithms can be optimized to $O(|V| + |E|)$, while force-directed layout algorithms is only $O(|V|^2)$. Therefore, this framework is based on a more efficient partition algorithm than single layout algorithm.

2.3 Partition Algorithms

2.3.1 Modularity

In the partition part of the algorithm, we partition the graph into many communities. Rather than random partition, we want strong connection within each community and weak connection across different community. This is a problem that has long been studied and termed as community detection problem.

A commonly-used metric for evaluating the partition of communities is the modularity:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

where A_{ij} represents the weight of the edge between node i and j, $k_i = \sum_j A_{ij}$ is the sum; of weights of the edges attached to node i, c_i is the community to which node i is assigned, and $m = \frac{1}{2} \sum_{ij} A_{ij}$ is the number of edges.

While the formula may look complex, its physical meaning is very intuitive: The modularity tells the difference of weighted sum of edges within a community to that in a random graph where each possible edge has an equal probability to form.

So the problem now becomes a modularity maximization problem.

2.3.2 Louvain Algorithm

Louvain is a famous algorithm of modularity maximization problem. Itself is also a recursive and heirarchical algorithm. The algorithm works out as shown in Algorithm 2.

Algorithm 2: Louvain Algorithm

```
   Data: Citation Network $G = (V, E)$
   Result: Partition $\mathbb{G}_{sub}$ and supergraph $G_{sup}$
1  $\mathbb{G}_{sub}, G_{sup}$ = Louvain($G$);
2  begin
3      Assign each node $i$ to a community: $C(i) \leftarrow i$;
4      while the partition does not converge do
5          foreach node $i$ in $G$ do
6              Calculate $\Delta Q_{ic}$: the gain by removing node $i$ from its community and merge it to a neighboring
                community $c$;
7              $c^* \leftarrow \text{argmax}_c \Delta Q_{ic}$;
8              $C(i) \leftarrow c^*$;
9          end
10     end
11     Aggregate each community into a supernode, and construct a supergraph $G_s up$;
12     if $C(i)$ does not change after it's initialized then
13         Calculate $\mathbb{G}_{sub}$ from $C$;
14     else
15         $\mathbb{G}^*$ ,_ $\leftarrow$ Louvain($G_{sup}$);
16         Update $C$ by mapping from $\mathbb{G}^*$;
17         Calculate $\mathbb{G}_{sub}$ from $C$;
18     end
19 end
```

The difference of modularity ΔQ by removing i from its community and merge it to a neighboring community C can be calculated and simplified as:

$$\Delta Q = \frac{k_{i,in}}{2m} - \frac{k_i \Sigma_{tot}}{2m^2}$$

where Σ_{tot} is the sum of the weights of the links incident to nodes in C, k_i is the sum of weights of the links incident to node i, $k_{i,in}$ is the sum of weights of the links from i to all nodes in C.

As shown by the formula, Louvain only needs local information to update, which makes it suitable for parallel computation. The algorithm procedure in also shown in Figure 2.

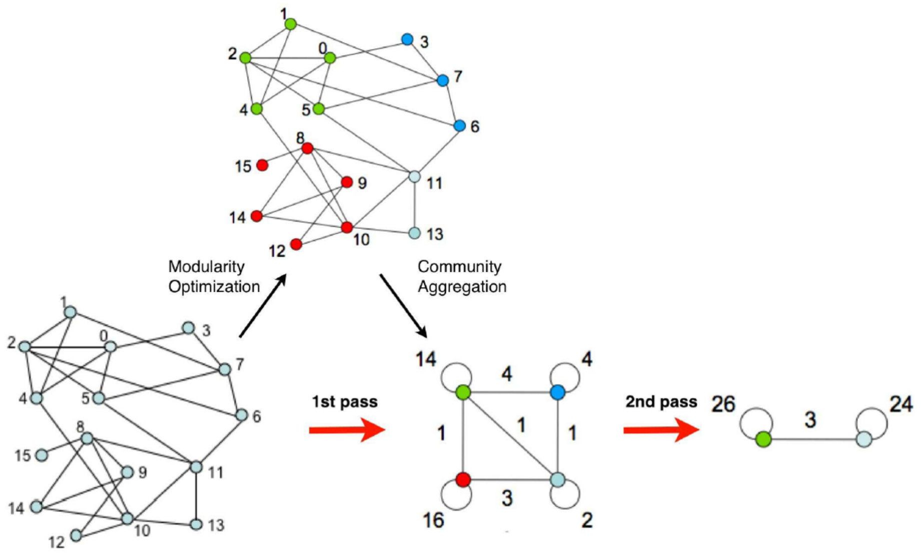


Figure 2: Louvain algorithm procedure

2.4 Small Graph Layout: ForceAtlas2

ForceAtlas2 is a force-directed graph layout algorithm. It imagines attractive force and repulsive force between any two nodes and layout the graph by letting the nodes move by the forces and get a balanced state. In the original version, the attractive force is proportional to the Euclidean distance by the order of 1:

$$F_a(n_1, n_2) = w(e)d(n_1, n_2)$$

and the repulsive force is proportional to the Euclidean distance by the order of -1:

$$F_r(n_1, n_2) = \frac{(deg(n_1) + 1)(deg(n_2) + 1)}{d(n_1, n_2)}$$

But the order of the two forces can have other combinations such as $(0, -2)$, where 0 means logarithm.

3 Implementation

In this section I will introduce how to implement the large graph layout algorithms and each of it components. I will also introduce some optimization methods to reduce time and space complexity.

3.1 Environment

- Java 8
- Python 3.7

3.2 Data Preparation

I choose DBLP database, a computer science bibliography that provides open bibliographic information on major computer science journals and proceedings. It has about 5,000,000 papers and 50,000,000 citations among themselves. I fetch the all the data from Acemap MongoDB database. The DBLP paper list are stored in document `DBLP` of collection `crawlerPaper`, while the paper information (such as title, citations) are stored in the document `paper` of collection `acemap`. And there is a mapping of paper id in DBLP to that in acemap. Since they are in different collections and MongoDB does not support table join, I have to fetch each part of data individually and process them on my local computer. I use the MongoDB API for python for this part. Below I show the code for fetching all the DBLP mapping in the `paper_mapping` document.

```
client = MongoClient(ADDR, PORT,
                     username=USERNAME,
                     password=PASSWORD)

def get_mapping():
    """get the mapping of dblp_id to acemap_id"""
    with open('dblp2am.csv', 'w') as f:
        for mapping in client.crawlerMapping.paper_mapping.find(
            {'_id': {'$regex': 'dblp'}}
        ):
            f.write('{}\t{}\n'.format(*mapping.values()))
```

After I fetch all the data, I store them in two csv files, one for all the node id and related title information, another for the edge lists, which specifies the citation.

3.3 Louvain

I implemented Louvain algorithm from scratch in Java. Why I choose Java is because it's a static language and it's more convenient to makes some optimization on it, such as memory usage, data structure and multi-threading.

First I define to inner class of `MyLouvain`: `Node` and `Community`.

```
private class Node {
    int id;
    int k_i; // sum of edge weight of all its neighbors, including self-loop
    int cid;
    int size; // for super-node, size is the number of atomic node
    int loop;
    List<Node> neighbors;
    List<Integer> weights;
}
private class Community {
    int id;
    int sum_tot;
    int size;
    int size_default;
}
```

Then I define the attributes of `MyLouvain`. Here I use a hash map for communities because its fast access. I use an array list for nodes because the algorithm only needs sequential access to nodes in the iteration, so there is no need to use a hash table and it can save memory.

```
public class MyLouvain {
    private List<Node> nodes;
    private List<Node> nodes_default;
    private Map<Integer, Community> communities; // community_id -> sum_tot: sum of k_i for
        all nodes in c
    private int num_edges; // sum of weights of all the edges <=> number of edges for the
        original network
}
```

The call function of Louvain algorithm is the `execute()`. Here `firstPhase()` and `secondPhase()` correspond to the modularity optimization part and community aggregation part in the algorithm. `firstPhase()` return a boolean value which denotes whether the community partition has changed in the last iteration, so it marks the end of the algorithm.

```
public void execute() {
    firstPhase();
    nodes_default = nodes;
    secondPhase();

    int round = 0;
    while (firstPhase()) {
        round++;
        updateNodesDefault();
        secondPhase();
    }
}
```

The implementation of `firstPhase()` and `secondPhase()` is complicated and long so I won't paste it here. They are in the source code directory attached with the report. Here I only emphasize some key points.

- The network are stored as adjacency table to have quick access to each node and its neighboring nodes
- The community partition are stored in a hash table to have quick access.
- My Louvain implementation converges in no more than several hundred iterations. And the time complexity for one iteration is $O(|V| + |E|)$

The above time complexity explains why the whole proposed framework will work.

3.4 ForceAtlas2: Gephi Toolkit

Gephi toolkit is a Java library that implements many graph layout algorithms, including ForceAtlas2. However, the whole package is written in a singleton classes and is inconvenient to use. So I write a `CitationNetwork` class that further encapsulates the Gephi library. Besides, Gephi library neither accepts input from csv files or databases, so I add these two supports as well.

The outline of `CitationNetwork` is shown below. Most of the attributes are Singleton classes defined in the library.

```
public class CitationNetwork {
    static private ProjectController pc = Lookup.getDefault().lookup(ProjectController.class
        );

    private Workspace workspace;
    private GraphModel graphModel;
    private PreviewModel previewModel;

    private AppearanceController appearanceController;
    private AppearanceModel appearanceModel;
    DirectedGraph graph;
}
```

Before the layout process, I first resize all the nodes. If the nodes are original nodes (i.e. papers), then I resize them by their out degrees (i.e. number of citations). If they are community nodes, I resize them by the community size.

```
public void rankSizeBy(String attr, int minSize, int maxSize) {
    // Rank size by degree
    Function sizeDegreeRanking;
    if (attr.equals("degree"))
        sizeDegreeRanking = appearanceModel.getNodeFunction(graph, AppearanceModel.
            GraphFunction.NODE_OUTDEGREE, RankingNodeSizeTransformer.class);
    else {
        Column attrCol = graphModel.getNodeTable().getColumn(attr);
        sizeDegreeRanking = appearanceModel.getNodeFunction(graph, attrCol,
            RankingNodeSizeTransformer.class);
    }
    RankingNodeSizeTransformer sizeDegreeTransformer = (RankingNodeSizeTransformer)
        sizeDegreeRanking.getTransformer();
    sizeDegreeTransformer.setMinSize(minSize);
    sizeDegreeTransformer.setMaxSize(maxSize);
    appearanceController.transform(sizeDegreeRanking);
}
```

The most import method of the class the to execute the layout algorithm.

```java
public void layout_fa2(int num_iter, boolean linLog, boolean adjustSize) {
    System.out.println("Layout␣" + getNodeCount() + "...");
    ForceAtlas2 fa2 = new ForceAtlas2(null);
    fa2.setGraphModel(graphModel);
    fa2.setAdjustSizes(adjustSize);
    if (linLog) {
        fa2.setLinLogMode(true);
        fa2.setScalingRatio(0.2);
    } else {
        fa2.setLinLogMode(false);
        fa2.setScalingRatio(1d);
    }

    fa2.initAlgo();
    for (int i = 0; i < num_iter && fa2.canAlgo(); i++) {
        fa2.goAlgo();
    }
    fa2.endAlgo();
    System.out.println();
}
```

3.5 Putting all together: File System and Multi-threading

In this section I will combine all components and implement the whole large graph layout algorithm.

Since the overall data is huge, we cannot store all the intermediate and temporary data in the memory, or it will explode. Also, it is inefficient to put these all in a database, because our algorithm will generate hierarchical data and in each level, we only need sequential access to the data. Traditional databases cannot provide hierarchical storage and search of entries is time consuming and unnecessary. Therefore, I store all the intermediate data in the file systems and utilize the directory structure and sequential read of files. Although I/O access is slower than memory, it will not cause out of memory error. So for either partition or single layout algorithms, I export the results to csv files. Below is an example of the partition function:

```java
public static void partition(String nodeCsv, String edgeCsv, String outDir) {
    File dir = new File(outDir);
    if (dir.isDirectory()) return;
    MyLouvain lou = new MyLouvain(nodeCsv, edgeCsv);
    lou.execute();
    if (!dir.isDirectory()) dir.mkdir();
    lou.ensureOneNodeCommunityID();
    lou.saveCommunityPartition(outDir + "/partition.csv");
    lou.partitionAndSaveTo(outDir + "/community");
    lou.saveMegaGraph(outDir + "/mega_graph");
}
```

Alss, each recursion call of the top-level function can be regarded as a single task. In other words, layout of subgraphs can be carried out in parallel. Only when we merge all the sub-layouts should we wait for the tasks of the same level to finish. This architecture is the classical fork-join multi-threading pattern.

So first, I created a thread pool to better limit the number of threads and manage them.

```java
private static int nThreads = 6;
private static ExecutorService fixedThreadPool = Executors.newFixedThreadPool(nThreads);
private static ThreadPoolExecutor tpe = (ThreadPoolExecutor) fixedThreadPool;
```

And for layout function of all the subgraphs, I submit all the independent tasks to the thread pool and use the `Future` mechanism to wait for all tasks to join before the merge operation.

```java
public static void layoutCommunity(String baseDir) {
    File comDir = new File(baseDir + "/community");
    String[] fileNames = comDir.list();
    File layoutDir = new File(baseDir + "/layout");
    if (!layoutDir.exists()) layoutDir.mkdir();
    List<Future<?>> futures = new ArrayList<>();

    for (String fileName : fileNames) {
        File layoutFile = new File(baseDir + "/layout/" + fileName);
        if (layoutFile.exists()) continue;
        Future<?> future = layout(baseDir + "/community_node/" + fileName,
                comDir.getPath() + "/" + fileName,
                baseDir + "/layout/" + fileName,
                baseDir + "/" + fileName.substring(0, fileName.indexOf(".")), false);
        if (future != null) futures.add(future);
    }
    for (Future<?> t : futures)
        try {
            t.get();
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

The merge part is long because it has to iterate over all subgraph layout files and the supergraph layout to fit all subgraphs to the original large graph. Then I also store the layout results in a csv files. There are five columns: the node id, its x and y coordinates, its radius and the community id it belongs to. Since the code for merge is too long, I don't post it here.

3.6 Statistics

To get a grasp of how the algorithm goes, I first plot the community size distribution of the top-level graph in Figure 3

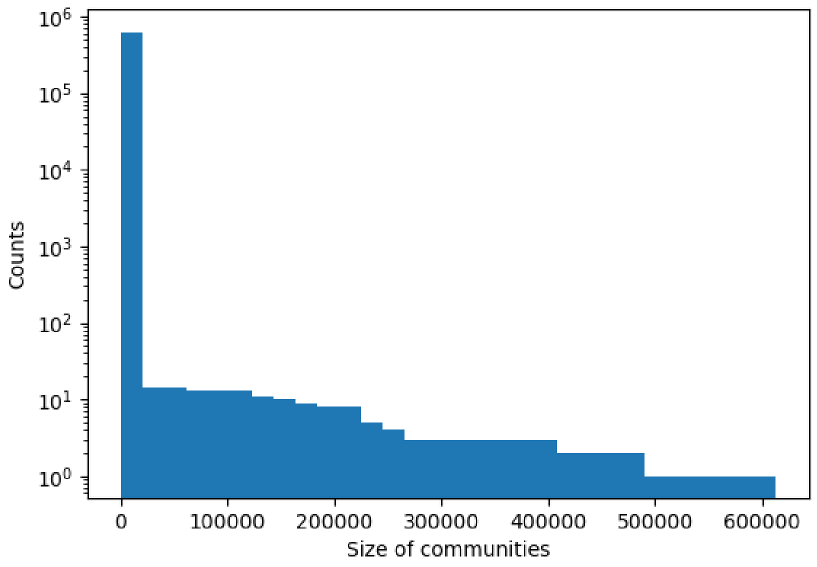


Figure 3: Community size distribution

We can see that the largest community has size over 600,000 while most communities have very small sizes. To get a clearer vision, I zoom in to the size smaller than 30 and get Figure 4. We can see that size one communities are most common, meaning that they have not citation within the DBLP database, maybe due to the incompleteness of the data source.

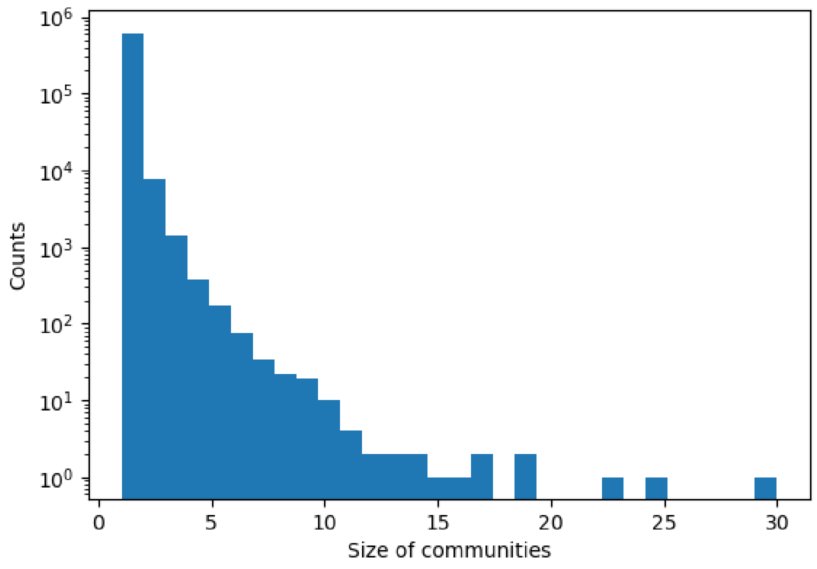


Figure 4: Community size distribution in range 0 to 30

4 Visualization

I first try visualization with SVG format images. SVG format utilize the xml format and defines its special element labels, such as `circle` and `line`. I write my own SVG generator from my layout files.

```java
public class SVG {
    private Document document;
    Element svg;
    Element eNode, eEdge;

    public SVG() {
        try {
            // create the builder factory
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = factory.newDocumentBuilder();
            document = db.newDocument();
            // do not display standalone="no"
            document.setXmlStandalone(true);

            svg = document.createElement("svg");
            svg.setAttribute("xmlns", "http://www.w3.org/2000/svg");
            svg.setAttribute("version", "1.1");
            eEdge = document.createElement("g");
            eEdge.setAttribute("id", "edges");
            svg.appendChild(eEdge);
            eNode = document.createElement("g");
            eNode.setAttribute("id", "nodes");
            svg.appendChild(eNode);

            document.appendChild(svg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

For function `addNode`, it creates circles for layout entries.

```java
public void addNode(String id, String x, String y, String r) {
    Element e = document.createElement("circle");
    e.setAttribute("id", id);
    e.setAttribute("cx", x);
    e.setAttribute("cy", y);
    e.setAttribute("r", r);
    e.setAttribute("fill", "black");
    e.setAttribute("fill-opacity", "0.5");
    eNode.appendChild(e);
}
```

I run my code and the generated svg file of the whole DBLP has more than 1 GB, which is too large to be displayed. So I limit the nodes to that with citation larger than 50 and edge size to be no more than double the node size. Then the generated svg file has 37 MB, and is shown in Figure 5.

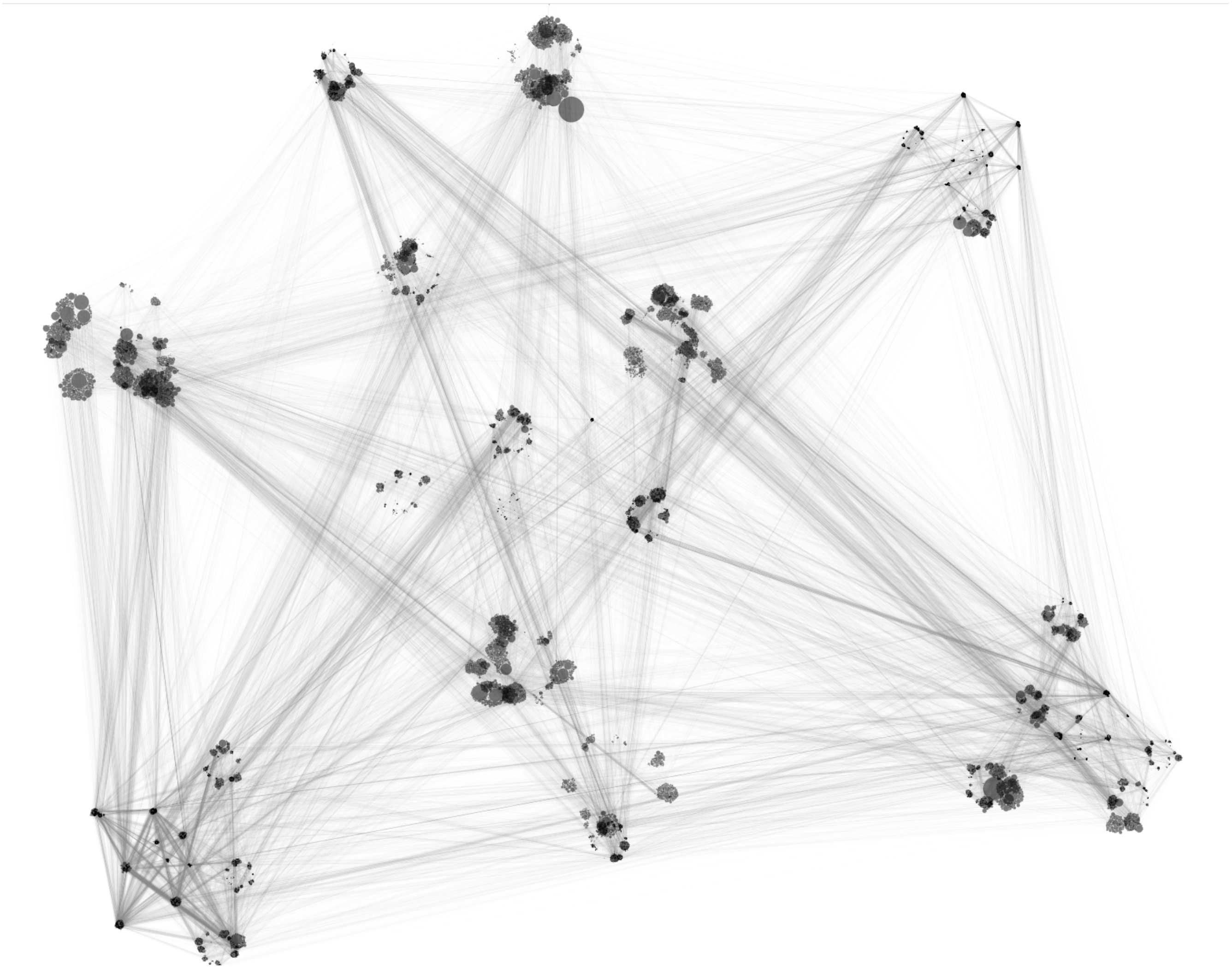
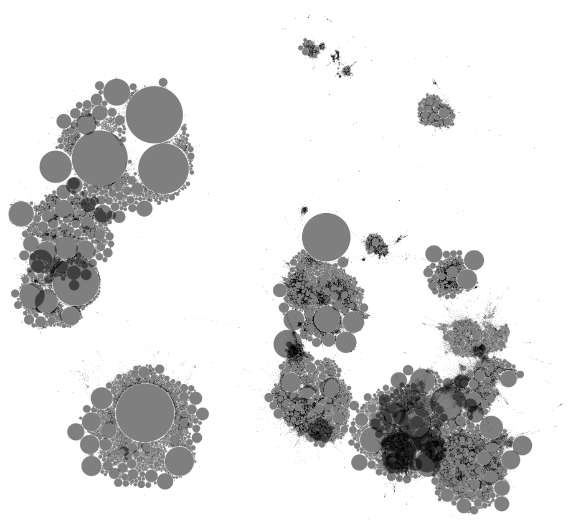
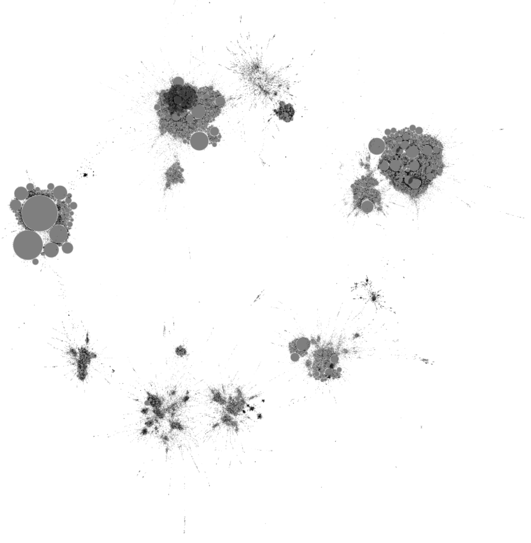
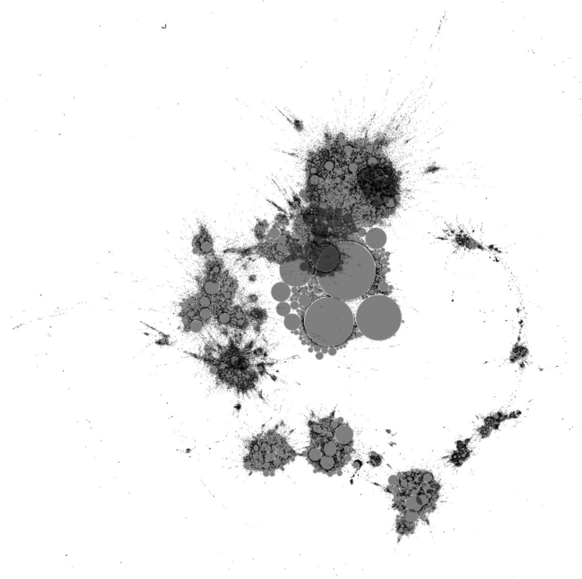


Figure 5: The whole DBLP citation network

To get a more detailed view, I choose some largest communities and show them below.


(a) size 250418


(b) size 227781


(c) size 211922

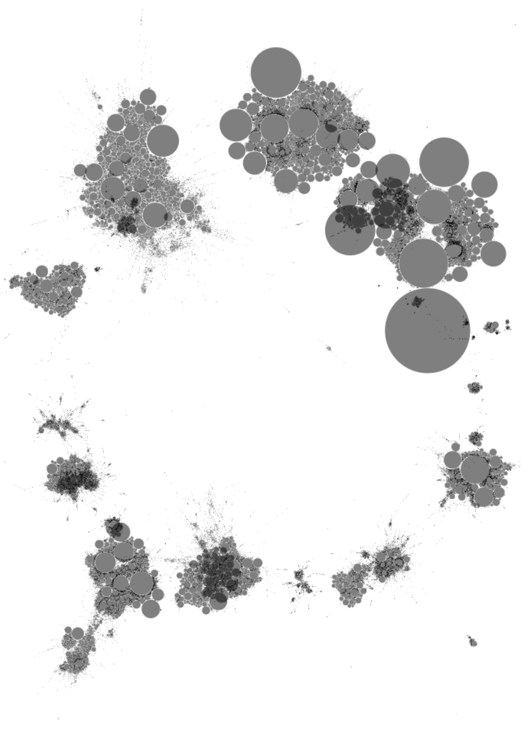

(d) size 113955

Figure 6: Layout of selected communities

5 Web Demo

I write a web demo where users can see the layout network, and find out what papers are grouped together and have great influence.

5.1 Environment and Dependencies

- Java 8

- Apache Maven 3.6.3
- Spring Boot 2.3.0

5.2 Architecture

The web demo uses the three-layer MVC pattern architecture.

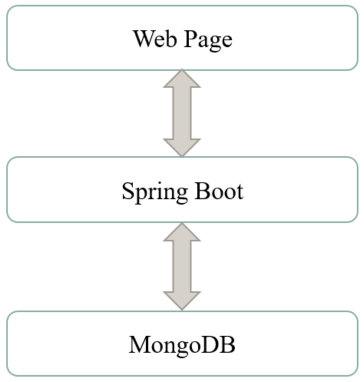
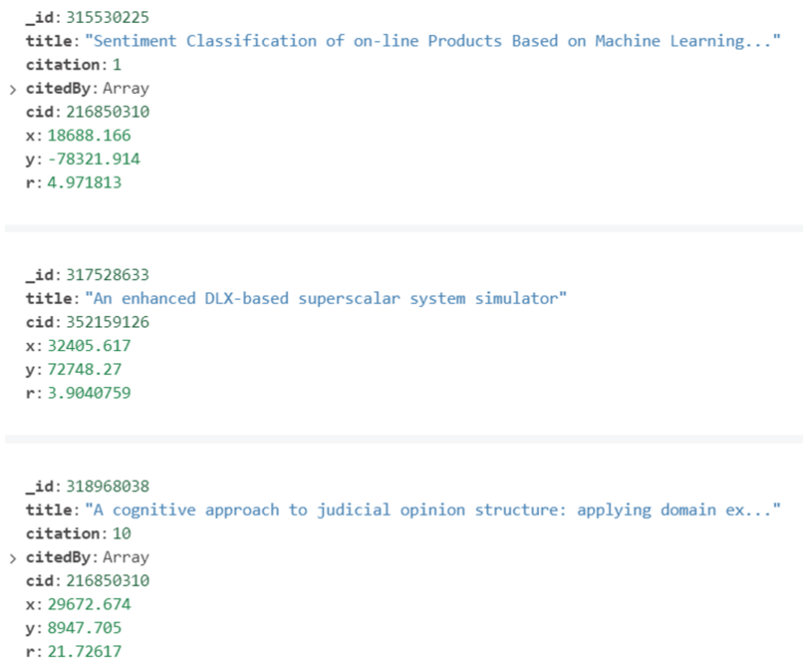


Figure 7: Web demo architecture

5.3 Export to MongoDB

Since in the web demo, search of certain nodes would be frequent, so it is best to export the layout data to the database. Here I choose MongoDB. Similar to the data fetch part, I use python to do the export.



```
_id: 315530225
title: "Sentiment Classification of on-line Products Based on Machine Learning..."
citation: 1
> citedBy: Array
cid: 216850310
x: 18688.166
y: -78321.914
r: 4.971813

_id: 317528633
title: "An enhanced DLX-based superscalar system simulator"
cid: 352159126
x: 32405.617
y: 72748.27
r: 3.9040759

_id: 318968038
title: "A cognitive approach to judicial opinion structure: applying domain ex..."
citation: 10
> citedBy: Array
cid: 216850310
x: 29672.674
y: 8947.705
r: 21.72617
```

Figure 8: Demonstration of layout MongoDB structure

5.4 Spring Boot

To write the logic of the web application, we use spring boot as backend server and programming platform. First, add database dependencies and configurations in `application.properties`.

```
spring.data.mongodb.uri=mongodb://root:pwd@127.0.0.1:27017/admin
spring.data.mongodb.database=citationNetwork
```

Then define a class `Node` to serve as the bridge variable between controller and the database.

```
@Document(collection = "node")
public class Node {
    @Id
    public int id;
    public int citation;
    public float x, y;
    public float r;
    public String title;
    public int cid;
    public List<Integer> citedBy;
}
```

In the main controller, the code takes in http `GET` request and return the specified layout information to the front end page.

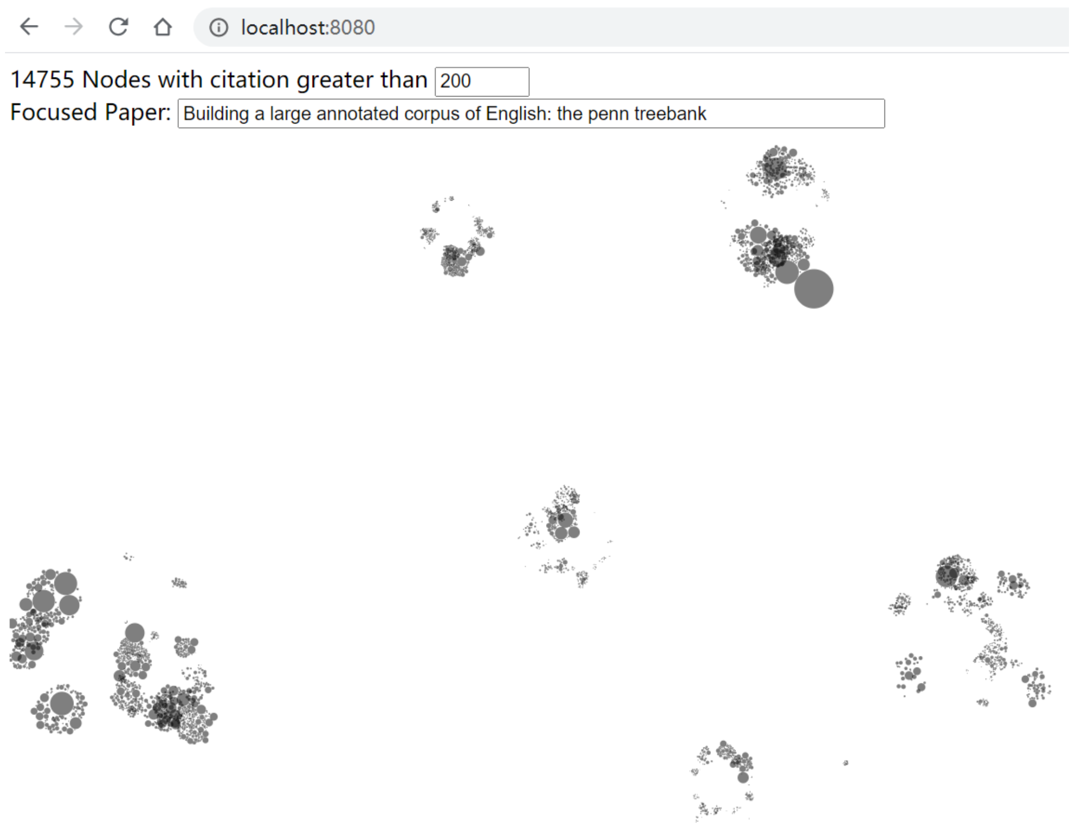
```
@Controller
public class MainController {
    @Autowired
    private MongoTemplate mongoTemplate;
    private String defaultPath = "src/main/resources/static/";

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public ModelAndView index(@RequestParam(value = "cite", required = false, defaultValue =
        "200") int cite) {

        List<Node> nodes = getCiteGte(cite);
        ModelAndView mv = new ModelAndView("index");

        mv.addObject("nodes", nodes);
        mv.addObject("view", viewBox(nodes));
        mv.addObject("cite",cite);
        mv.addObject("num",nodes.size());
        return mv;
    }
```

5.5 Web Page Template

In the front page the HTML elements uses thymeleaf template pattern to generate a new web page. Here I write an svg element and use the data from backend to create the nodes. I also define mouse move over event to display the paper title.

```
<div>
    <?xml version="1.0" encoding="UTF-8"?>
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1" th:viewBox="${view}" width="100%"
        height="100%">
        <g id="nodes" th:each="n: ${nodes}">
            <circle th:cx="${n.x}" th:cy="${n.y}" th:id="${n.title}" th:r="${n.r}" th:label=
                "${n.title}" onclick="sub(event)" onmouseover="show_title(event)" fill="
                black" fill-opacity="0.5"/>
        </g>
    </svg>
</div>
```

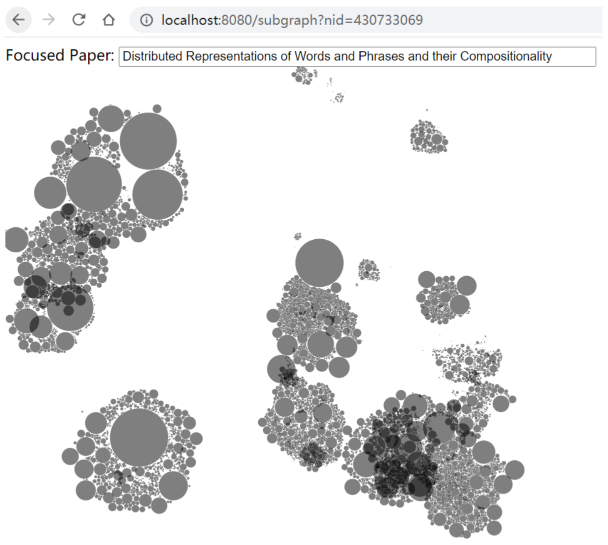
5.6 Demonstration



14755 Nodes with citation greater than 200

Focused Paper: Building a large annotated corpus of English: the penn treebank

(a) Main Page



Focused Paper: Distributed Representations of Words and Phrases and their Compositionality

(b) Subgraph Page

Figure 9: Web Demo. When the mouse move over a node, the above will show the title for the paper. When clicking it, it will jump to the subgraph page, where it will show the partitioned community

Checking the community in the Figure 9(b). We found these papers: Distributed Representations of Words and Phrases and their Compositionality, Glove: Global Vectors for Word Representation, Bleu: a Method for Automatic Evaluation of Machine Translation, Class-based n -gram models of natural language, etc. So it means the proposed algorithm indeed group together papers of similar fields.

6 Conclusion

In this project, I implement an recursive and hierarchical large graph layout framework in Java and successfully layout the DBLP database of size 5,000,000. I write a svg generator that accepts input from csv files and MongoDB database. I build a basic web demo to show what papers are grouped together and what their influence is. For future work, I will add more features to the web demo, such as search.

References

[1] Mathieu Jacomy, Sebastien Heymann, Tommaso Venturini, and Mathieu Bastian. ForceAtlas2, A Continuous Graph Layout Algorithm for Handy Network Visualization. 2012.

[2] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Generalized Louvain method for community detection in large networks. *International Conference on Intelligent Systems Design and Applications*. 2011

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*. 2008.

[4] Gephi Tutorial Quick Start. Retrieved from https://gephi.org/toolkit/

[5] Spring Boot Guide. Retrieved from https://www.springcloud.cc/spring-boot.html