**THE UNIVERSITY OF MEMPHIS**

**Department of Computer Science**

**Interactive Visual Graph Exploration by Finding Dense Subgraphs**

**Neha Sinha**

**04/13/2020**

**Project Advisor:**

**Dr.** **Xiaofei Zhang**

**Assistant Professor, Department of Computer Science**

**A project report submitted to the faculty of The University of Memphis in partial fulfillment of the requirements for the Master of Science Degree in the college of Arts and Science.**

**ACKNOWLEDGEMENT**

On the very outset of this report, I would like to thank Dr. Xiaofei Zhang, Assistant Professor of Computer Science, the University of Memphis, for the cooperation and constant support extended to me in accomplishing the project. Instead of his busy schedule, Dr. Zhang was always prompt in assessing my progress and guided me in the right direction. I could make this endeavor possible with his valuable guidance, suggestions, help and cooperation without which it couldn’t have been done.

I would also like to thank Dr. Deepak Venugopal, Assistant Professor of Computer Science, University of Memphis and Dr. Xing Gao, Assistant Professor of Computer Science, University of Memphis for being part of my master’s project committee.

At last, I would like to thank my family for providing me moral support to complete my studies.

**ABSTRACT**

This project aims to provide the user a visualization tool for quickly analyzing graph data. Specifically, we ground our work on finding the dense subgraphs. It is motivated by the fact that, in many applications, it is a key algorithmic task to extract a densest subgraph from an input graph, according to a density definition. In this project, we compute the dense subgraphs based on the K-core [1] decomposition, which allows efficient analysis of graph beyond its mere degree distribution. In addition, we propose a sampling based preprocessing method, namely edge-weight method, to help capture high density subgraphs. We implement these methods in Java language and build an interactive visualization tool. We categorize the accomplished work as learning, implementing and analyzing.

The learning phase comprised of gathering knowledge about graph database. The implementation phase started with writing concrete algorithm of the dense component identification followed by writing the corresponding Java code. In the analysis phase, we ran the method against different graph datasets and analyzed the experimental results on a set of predefined parameters. We have been actively revising our implementation to gain more time efficiency.

**1. INTRODUCTION**

Graph-based representations became highly popular to model various real-world systems, as well as diverse types of knowledge and data, due to the simplicity and visually appealing nature of graph models. The specific task of extracting a dense subgraph from a large graph has received a lot of attention, since it has direct applications in many fields. To name a few:

* In a social network, such as Facebook, a typical graph representation has nodes corresponding to individuals, and the edges capture some relation or interaction between them, e.g., friendship. In this model, a dense subgraph represents a community.
* In a communication network, a dense subgraph can capture a congested part of the network, assuming that the edges correspond to those links that have traffic load over some threshold. Identifying such congested parts, and then taking appropriate action to relieve the congestion, can have a major impact on network performance.
* In the World Wide Web (WWW), a natural graph representation has the websites as its nodes, and the hyperlinks between them as edges. A densely interconnected part may indicate a web community, such as a group of content creators sharing a common interest.
* Other use cases of dense subgraphs include graph compression, graph visualization, clustering, real-time identification of important stories on Twitter, and many other data mining and knowledge discovery applications.

Among the many concepts that have been proposed for discovering dense subgraphs, k-cores [1] and graph density[17] are particularly attractive for the simplicity of their definition and linear time complexity.

* 1. **Contribution**

Having seen the importance of finding dense component of the graph, it would be interesting and helpful to quickly provide a “head-first impression” of the graph by identifying those large dense subgraphs, as well as a “zoom in” function for users to explore more if necessary. Thus, users will not be overwhelmed by visualizing all the details about the graph. This project provides an interactive visualization tool to interactively explore the graph structure. User can zoom into the dense component by the click functionality if he or she is interested in exploring more. Note that visualization results are computed in real time upon the user’s action. It eliminates the long waiting time cost to precompute everything, which is particular attractive to visualizing large graphs.

**2. RELATED WORK**

Dense subgraph extraction is a key problem in both algorithmic graph theory and graph mining applications [4][5][6]. K-Core model to compute dense subgraph has been extensively studied not just for static but also for streaming graph data [1] [7] [8] [9] [10]. Researchers have explored different definitions of density and examined the optimization problems corresponding to finding substructures that maximize a given notion of density. The complexity of such optimization problems varies widely with the specific choice of a definition [11].

Also, many graph analyzer tools are available [12] [13] [14] [15]. These tools precompute the graph data and gives the result to user to show graph structure. It doesn’t help much if the graph size is large.

The goal of this project is not to compete with these existing algorithms on computing k-core but to come up with an interactive tool to help discover and explore graph structure quickly. It computes the dense component in real time to improve the time and memory efficiency for large graph exploration. When required, user can select which dense component he need to explore more. Then the computation will only be limited to the component that he clicks on.

**3. PRELIMINARIES**

In this section, we introduce some basic concepts for the ease of discussion. We model a directed graph as , where and denote the sets of nodes and edges in A graph is a subgraph of if and .

**Definition 1:** A ***k*-core** of a graph  is a maximal connected subgraph of in which all vertices have degree at least *k*. [18]

A great advantage of the k-core is that it is very easy to compute algorithmically. The principle of the algorithm can be described in one sentence: delete all nodes with degree < k, and repeat this in the remaining graph, until either all nodes have degree ≥ k, or the remaining graph is empty. Note that the repetition is needed because the removal of nodes may decrease some degrees in the remaining graph.

**Definition 2:** Given a graph its ***edge-density*** is τ ()= | |/|2| which is total number of edges upon possible number of edges in the graph.

**Definition 3:** A directed graph is ***strongly connected*** if every vertex is reachable from every other vertex.

**4. ALGORITHMS**

There are mainly two algorithms that have been implemented to discover the dense component of the graph. One is K-core [1] and other is an edge density method [17].

Below are the parameters that have been used in these algorithms.

|  |  |
| --- | --- |
| **Parameters** | **Meaning** |
|  | a directed graph with vertex sets and edge set |
|  | a subgraph of with vertex set and edge set |
| K\_start | Value that is used to find K-core of a graph. Suppose K=5, then every vertex of this graph has degree of minimum 5. |
| K\_increment | Increment the value of K by K\_increment to check if graph has all the vertices with degree at least (K + K\_increment) |
| MIN\_SCC\_SIZE | It indicates a minimum value for size to find a strongly connected component. |
| DELETED\_NODE\_PERCENT | Percentage of nodes which got deleted from the original graph in the process of finding k-core. |
| VERTEX\_PERCENTAGE | Percentage of randomly selected vertices of a graph. |
| WEIGHT\_THRESHOLD | A threshold value for weight of edge. It is used to delete all the edges whose weight are more than this threshold. |

Table 1: Parameters

**4.1 KCore**

**4.1.1 Naïve K-core**

Here goal is to find a subgraph with some highest K valueby deleting the nodesuntil the percentage of deleted nodes is equal to or more than WEIGHT\_THRESHOLD. We start with K=1, and delete all the nodes having degree less than K. Next step is to scan the degrees of all the remaining vertices again to ensure every vertex has degree equal to or greater than K. And finally increment the K to move to next iteration.

**Algorithm: Naïve K-core**

**Input:** Graph

**Output:** Strongly Connected Component List i.e Subgraph

Initialize:

K = 1;

MIN\_SCC\_SIZE = 50;

DELETED\_NODE\_PERCENT = 50.0;

degreeList ← contains calculated degree of each vertex u in

nodeDeleted ← holds percentage of nodes that been deleted

while (nodeDeleted < *DELETED\_NODE\_PERCENT*)

For each vertex u in

Repeat until (degree(u) ≥ K)

if (degree(u) < K)

= remove Vertex u from

degreeList ← calculate degree of each vertex in

End

End For

nodeDeleted ← calculate percentage of vertex deleted so far

K++;

End while

Connected\_Component\_List ← from subgraph compute all the connected component whose size > MIN\_SCC\_SIZE

return Connected\_Component\_List

**4.1.2 Modified K-core**

This is modified version of naïve K-core algorithm. Below are the two modifications done to improve time efficiency.

1. Value of K – Instead of assigning K equal to 1, start with some greater value (like k = 5). This start value which is referred as K\_start is a user assigned value.
2. No need of repeated loop to check modified degree of vertices –After deleting all the nodes with degree less than K in an iteration directly jump to next iteration by incrementing the value of K by K\_increment.

**Algorithm**: **KCoreModified**

**Input:** , K\_start, K\_increment, MIN\_SCC\_SIZE, DELETED\_NODE\_PERCENT

**Output:** Strongly Connected Component List i.e Subgraph

Initialize:

K = K\_start;

degreeList ← contains calculated degree of each vertex u in

nodeDeleted ← holds percentage of nodes that been deleted

while (nodeDeleted < DELETED\_NODE\_PERCENT)

For each vertex u in

if (degree(u) < K)

= remove Vertex u from

End For

degreeList ← calculate degree of each vertex in

nodeDeleted ← calculate percentage of vertex deleted so far

K = K + K\_increment;

End while

Connected\_Component\_List ← from subgraph compute all the connected component whose size > MIN\_SCC\_SIZE

return Connected\_Component\_List

**4.2 Edge- weight method**

Here goal is to find all the edges whose weight is more than WEIGHT\_THRESHOLD. Those edges are removed from the input graph. The output of this pre-processing is a subgraph which then serve as an input to modified K-core algorithm. Algorithm starts with finding some randomly selected nodes equal to VERTEX\_PERCENTAGE which is a user assigned value. Next step is to update the weight of all the edges of those nodes. Lesser the weight higher is the edge density. Now take all the edge whose weight is more than WEIGHT\_THRESHOLD and delete it from input graph. This new subgraph is then passed to modified k-core algorithm.

**Algorithm:** **EdgeWeight**

**Input**: , VERTEX\_PERCENTAGE, WEIGHT\_THRESHOLD

**Output**: Subgraphs of graph

random\_vertex\_set ← Randomly select vertices equal to VERTEX\_PERCENTAGE from

For each u in random\_vertex\_set

For each edge e of u

weight(e) += 1/(degree(u))

End For

End For

For each edge e in

If(weight(e) > WEIGHT\_THRESHOLD)

← remove e contained from

End If

End For

return KCoreModified ()

**5. IMPLEMENTATIONS**

Our program reads a graph dataset file selected by the user. Then it preprocesses the data using selected algorithm and presents a dense component on UI. This framework gives user easy to explore feature where user can zoom into a dense component for more information. Here data is computed in real time so that if graph is large then result can be shown in minimum time possible. Every click functionality by user on dense component initiate a callback to java program which does the required computation and return the JSON data back to UI. I have used JGraphT [2], which is a Java library of graph theory data structures and algorithms. Also, D3.js [3] which is a JavaScript library for visualizing graph, is used to convert JSON data into visual graph.

**5.1 Code Design**

Here I have used **Spring framework** which is advance Java application framework. Advantages of using this framework is that is follows MVC design pattern. MVC Design Pattern is a software design that separates the following components of a system or subsystem:

* **Model –** It consists of application data. Here I have simple java POJO classes.
* **View -** It is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.Here I have used JSP with D3.js**.**
* **Controller –** It is responsible for processing user requests and building an appropriate model and passes it to the view for rendering. Here I have used ‘Rest Controller’ which is used to define the RESTful web services.

Also, for **graph visualization**, I have used **D3.js** which is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. It makes use of Scalable Vector Graphics, HTML5, and Cascading Style Sheets standards.

For software development **version control** I have used **GIT.** It offers the distributed version control and source code management functionality using GitHub.

**Flow Diagram**

Below diagram represents a basic flow diagram of the application. Here user submit a form where he selects a graph dataset file, choose which method he wants to use i.e. K-core or edge-weight, then he enters parameters values. After form submit action, java classes are called and based on chosen method graph is generated.

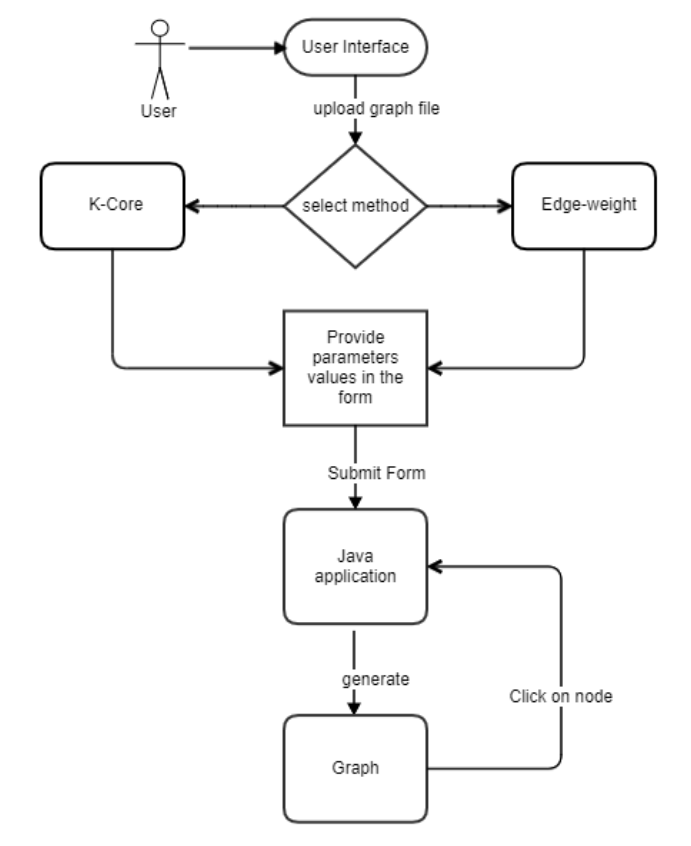


Figure 1: Flow Diagram

**Sequence Diagram**

This diagram shows sequence of events that takes place through UI. Here controller and services are java classes.

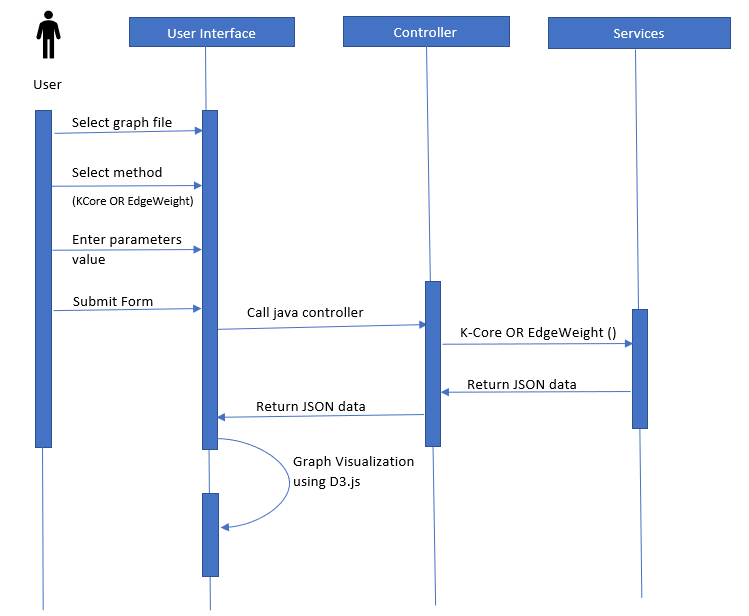


Figure 2: Sequence Diagram

**Java Class Description**

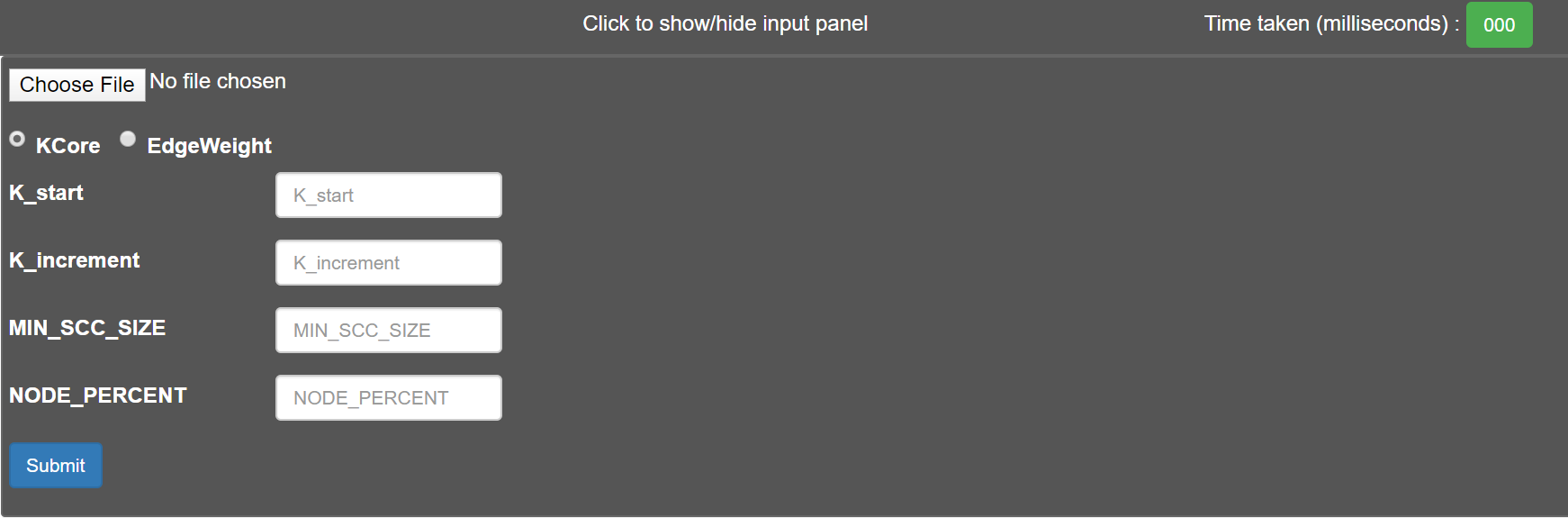
Below are the main class of the framework:

1. graphController.java – It receives request from UI after form submission. Based on input values it calls appropriate method of service class and return JSON data back to UI.
2. GraphService.java – It does necessary computation to find dense component and other related information about the graph and return the result back to controller.
3. FormSub.java – It is the model class which holds all the application data required for computation.

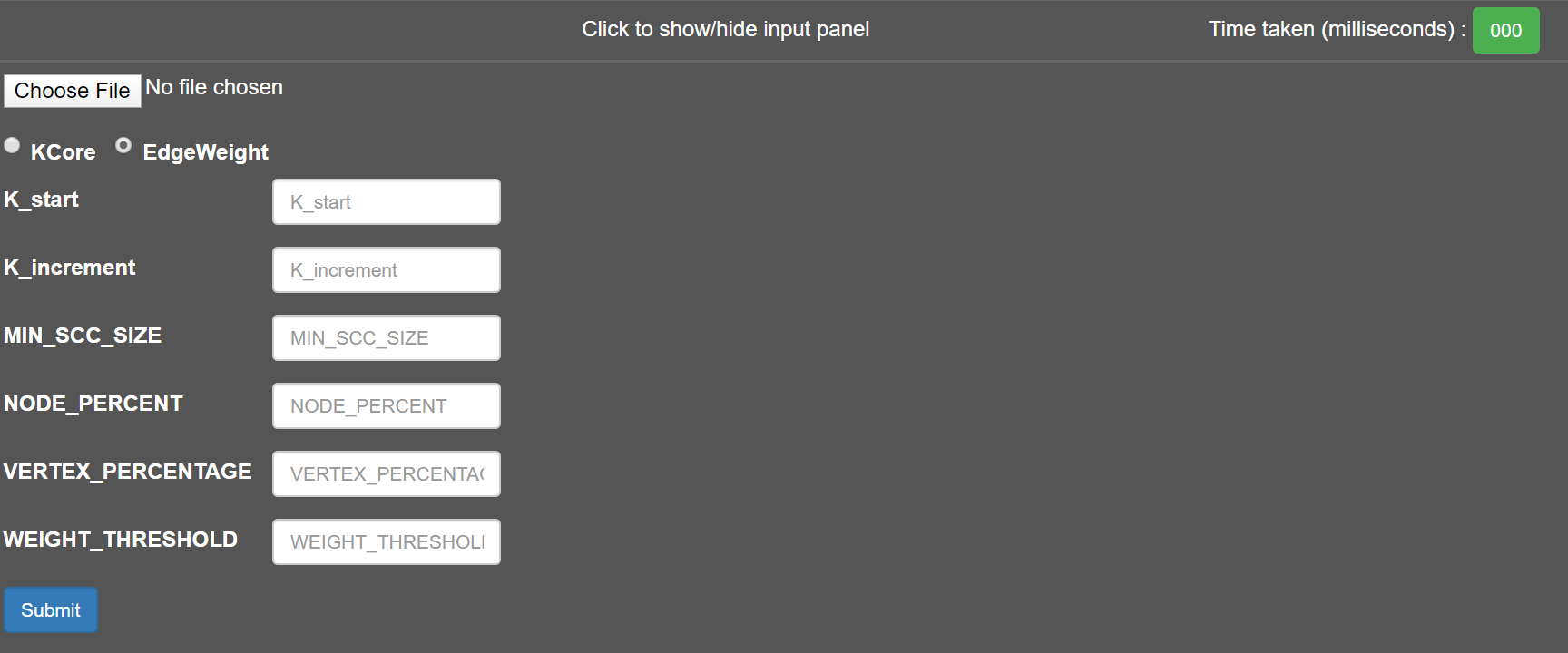
**5.2 User Interface**

In this section, we will go through the UI screens.

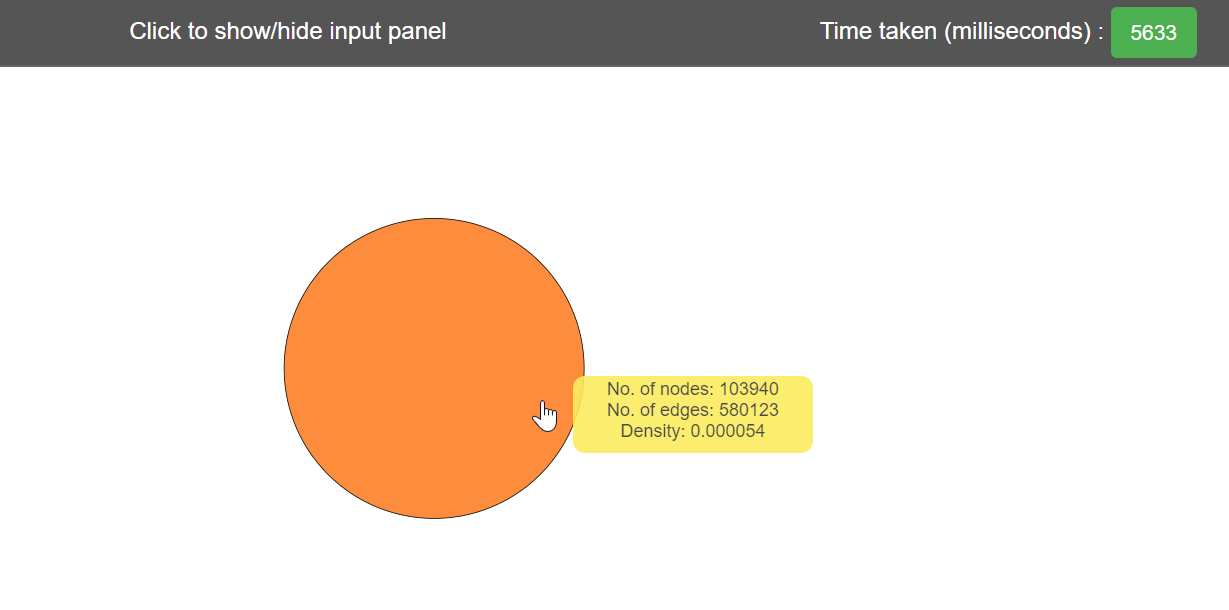
**Screen 1:** Input form for K-core method



**Screen 2:** Input form for EdgeWeight method

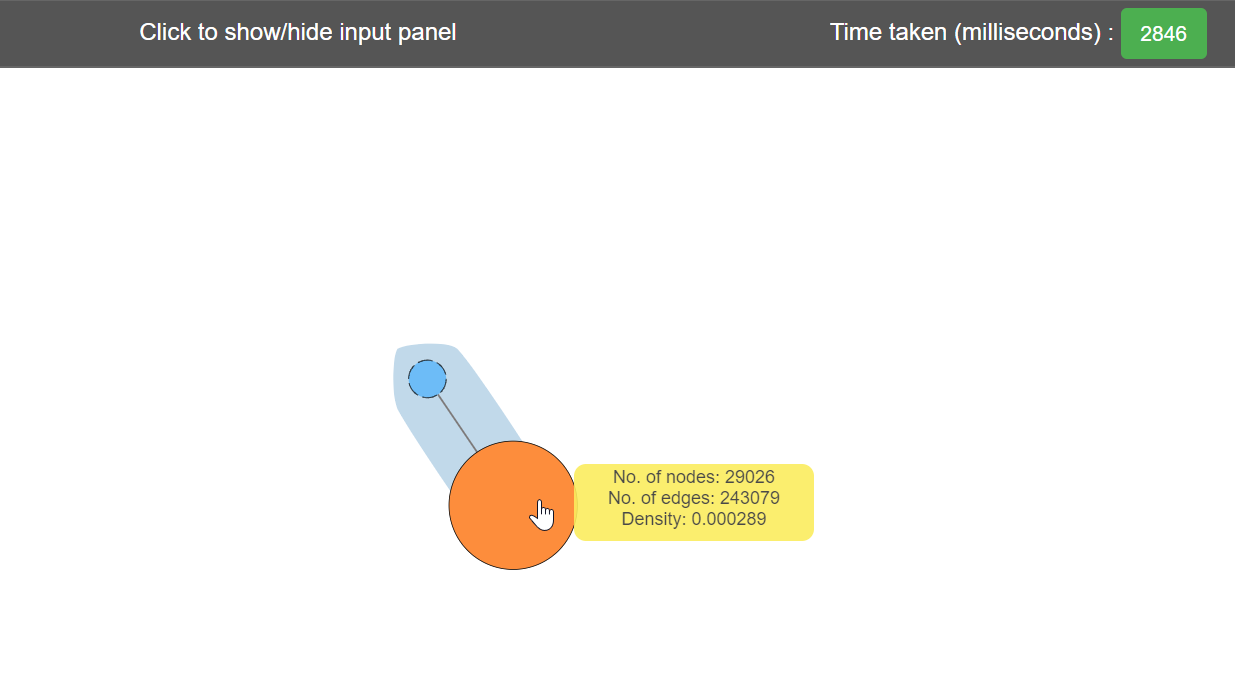


**Screen 3:** First subgraph after user submits the form



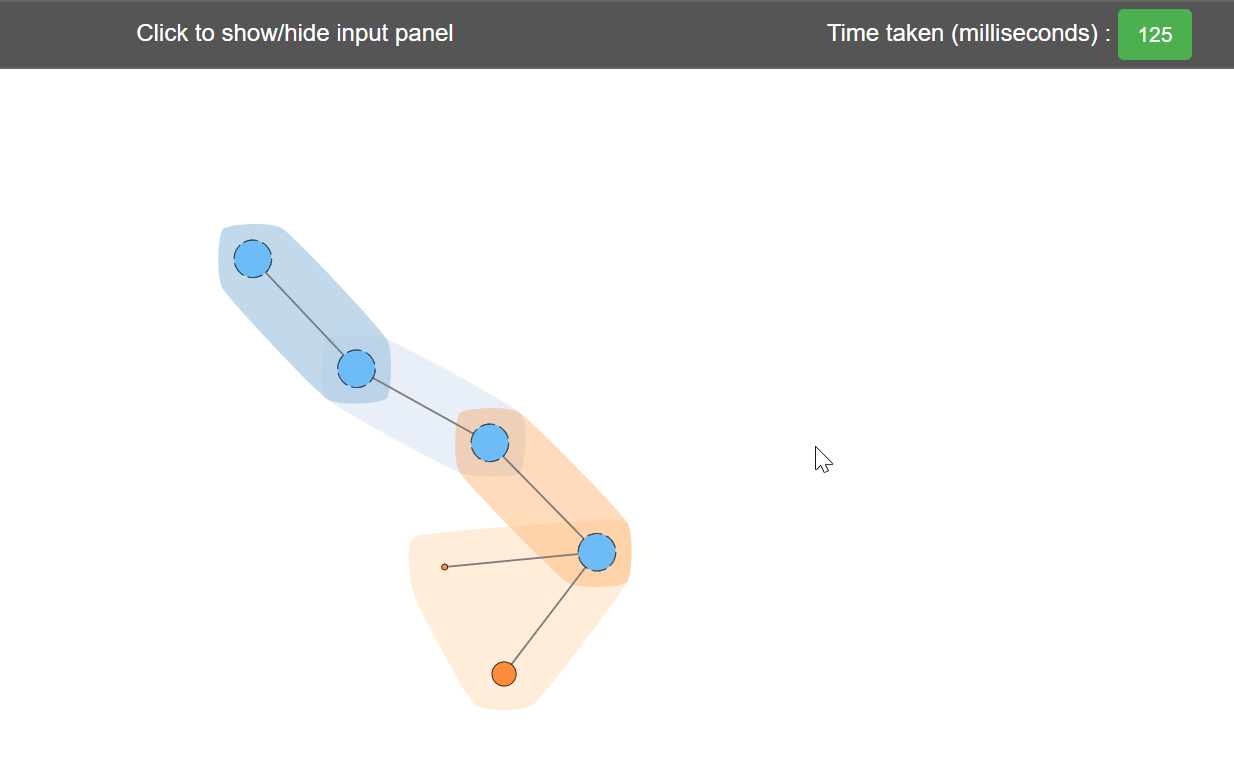
Here orange node is first dense subgraph. Yellow region shows some data. Time taken filed in top right shows execution time to generate dense graph.

**Screen 4:** Second subgraph after user clicks on first subgraph



First dense subcomponent shrinks to blue node and new denser subgraph is represented by orange node. Size of node depends on number of nodes present. Highlighted region in the background (in this case light blue) shows these two subgraphs are connected.

**Screen 5: Expanded graph**



When user keep exploring dense component this is what it looks like.

**6. EXPERIMENT**

I have performed experiments on five graphs given in Table 2 below. These datasets are from SNAP library [16]. These graphs cover various domains, such as collaboration networks (e.g., Ca-AstroPh), bibliographical graphs (e.g., DBLP), web graphs (e.g., google), social networks (e.g., Twitter).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.no** | **File Name** | **Graph Type** | **#Nodes** | **#Edges** | **File Size (MB)** |
| 1. | CA-AstroPh.txt | Physics collaboration network | 18,772 | 396,160 | 5 |
| 2. | Com-amazon.ungraph.txt | Amazon product co-purchasing network | 334,863 | 925,872 | 12 |
| 3. | Com-dblp.ungraph.txt | DBLP bibliographical network | 317,080 | 1,049,866 | 14 |
| 4. | Twitter \_combined.txt | Social circles | 81,306 | 1,768,149 | 43.5 |
| 5. | Web-google.txt | web graph | 875,713 | 5,105,039 | 73.6 |

Table 2: Graph Datasets

Experiments have been carried out to evaluate the following metrics – time efficiency and density of the outcome subgraph. Following sections discuss about these in detail.

**6.1 TIME EFFICIENCY**

Here we will see how two algorithms and different parameters affect the time efficiency for finding the dense component.

**6.1.1 Comparison between algorithms**

Below table shows different values of time taken by the two algorithms on the five input graphs. The parameters are set as K\_start = 5, K\_increment = 2, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, VERTEX\_PERCENTAGE = 0.4 and WEIGHT\_THRESHOD = 0.2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Graph** | **KCoreModified Algorithm (ms)** | | | **EdgeWeight Algorithm (ms)** | | |
| *First subgraph* | *Second subgraph* | *Third subgraph* | *First subgraph* | *Second subgraph* | *Third subgraph* |
| CA-AstroPh | **2474** | 2376 | 1993 | 5716 | 2179 | 2177 |
| Com-amazon | **9392** | 1356 | 30 | 15704 | 1160 | 28 |
| Com-dblp | **10393** | 5350 | 3142 | 17948 | 4485 | 3343 |
| Twitter | **19388** | 17383 | 18979 | 36486 | 18853 | 18118 |
| Google | **66320** | 22867 | 8415 | 119440 | 98002 | 91220 |

Table 3: Time efficiency of algorithms

Note: For Google datasets, as the size of the graph is 73.6 MB, the program was throwing Java heap space error, so the parameters I have used to make it run are K\_start =15, K\_increment = 10. Rest parameters are same for all the graphs. Also, we didn’t consider Naïve K-Core algorithm because it is much slower than the modified version.

As EdgeWeight algorithm involves an extra pre-processing step to remove some edges from the input graph based on a threshold value, it usually takes more time to execute in comparison to KCoreModified algorithm. Bold values in table indicates lesser time taken by that algorithm.

**6.2.2** **Experiment based on parameters**

Below are the experiments results based on different parameters values. Here I have highlighted the impact of VERTEX\_PERCENTAGE and WEIGHT\_THRESHOLD parameter. For all other parameters results have been included in [Appendix A](#APPENDIXA). It helps to understand how different input parameters increase or decrease the execution time.

**VERTEX\_PERCENTAGE:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, WEIGHT\_THRESHOD = 0.2

|  |  |
| --- | --- |
| **VERTEX\_PERCENTAGE** | **EdgeWeight Algorithm (ms)** |
| 0.3 | 9717 |
| 0.4 | 9909 |
| 0.5 | 10010 |
| 0.6 | 9970 |
| 0.7 | 10211 |

Table 8: VERTEX\_PERCENTAGE Result

This parameter is used in EdgeWeight algorithm. More the percentage value means more part of the graph will be considered for finding the weight of the edge. So more the value more will be execution time.

**WEIGHT\_THRESHOD:**

For this experiment, parameters values are graph dataset = DBLP, K\_start = 5, K\_increment = 1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, VERTEX\_PERCENTAGE = 0.4

|  |  |
| --- | --- |
| **WEIGHT\_THRESHOD** | **EdgeWeight Algorithm (ms)** |
| 0.1 | 9931 |
| 0.2 | 9277 |
| 0.3 | 9161 |
| 0.4 | 9868 |
| 0.5 | 9807 |

Table 9: WEIGHT\_THRESHOD Result

Smaller the value of this threshold the more edges get pruned. So, it takes more time to execute.

**6.2 DENSITY**

In this section, we will see how these two algorithms and different parameters affects the density of the subgraph.

**6.2.1 Comparison between algorithms**

Below table shows different values of density of subgraphs generated by the two algorithms executed on the five graphs. Different parameters are set as K\_start =5, K\_increment = 1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, VERTEX\_PERCENTAGE = 0.4 and WEIGHT\_THRESHOD = 0.2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Graph** | **KCoreModified (Density: )** | | | **EdgeWeight (Density: )** | | |
| *First subgraph* | *Second subgraph* | *Third subgraph* | *First subgraph* | *Second subgraph* | *Third subgraph* |
| CA-AstroPh | 4155 | 4579 | 5055 | 4155 | 4579 | 5055 |
| Com-amazon | 32 | 564 | - | **48** | **1961** | **-** |
| Com-dblp | 54 | 139 | 300 | **69** | **157** | **314** |
| Twitter | 132 | 1466 | 1643 | **1344** | **1486** | **1666** |
| Google | 65 | 187 | 467 | **97** | **268** | **498** |

Table 10: Density based on algorithm

Note: Blank value in the table indicates no subgraph is present. Bold values represent higher densities.

Table data clearly shows that the EdgeWeight method gives denser subgraph than the K-core. Reason is the pre-processing step that deletes all the edges with lesser weight.

**6.2.2 Experiment on parameters**

Here I have highlighted the impact of K\_start, VERTEX\_PERCENTAGE and WEIGHT\_THRESHOLD parameter. Only these three parameters have impact on density, for all other parameters, results have been included in [Appendix B](#APPENDIXB).

**K\_start:**

For this experiment, parameters values are graph dataset = DBLP, K\_increment=1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **K\_start** | **KCoreModified (Density: )** | **EdgeWeight (Density: )** |
| 1 | 32 | 44 |
| 5 | 54 | 69 |
| 15 | 291 | 357 |
| 25 | 751 | 938 |
| 35 | 1548 | 1974 |

Table 11: K\_start result for density

For any given K\_start value, EdgeWeight method will always outperform the K-core method. Also, density increases with higher K value.

**VERTEX\_PERCENTAGE:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, WEIGHT\_THRESHOD = 0.2

|  |  |
| --- | --- |
| **VERTEX\_PERCENTAGE** | **EdgeWeight (Density: )** |
| 0.3 | 63 |
| 0.4 | 69 |
| 0.5 | 76 |
| 0.6 | 85 |
| 0.7 | 97 |

Table 13: VERTEX\_PERCENTAGE result for density

Higher the percentage value larger the portion of graph covered for execution.So, with more vertices and edges, density bound to increase. Same result can be seen from table, with the increase in value the density also increases.

**WEIGHT\_THRESHOD:**

For this experiment, parameters values are graph dataset = DBLP, K\_start = 5, K\_increment = 1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT = 50, VERTEX\_PERCENTAGE = 0.4

|  |  |
| --- | --- |
| **WEIGHT\_THRESHOD** | **EdgeWeight (Density: )** |
| 0.1 | 94 |
| 0.2 | 69 |
| 0.3 | 62 |
| 0.4 | 57 |
| 0.5 | 54 |

Table 14: WEIGHT\_THRESHOLD result for density

Smaller the value of this threshold the higher is the number of edges to get pruned. So, the density will be high for less threshold as seen in the table.

**7. CONCLUSION**

Inspired by k-core method and density-based graph mining, two algorithms have been implemented to come up with a new tool for quickly analyzing graphs. Like k-core decomposition, implemented approach decomposes a given graph into a nested sequence of subgraphs. These subgraphs have the property that the inner subgraphs are always denser than the outer ones. The EdgeWeight method further improved the density of the subgraph. We saw in the experiment that the removal of low-density edges from graph results into denser subgraph. Also, graph visualization helps in better understanding the structure of graph.

**8. FUTURE WORK**

There are few points based on which this project work can be extended. We can plan to optimize the code by improving the data structure used for graph and algorithms. Also, we can have a server to handle the large graph and verify for scalability and memory utilization.

**REFERENCE**

[1] V. Batagelj and M. Zaversnik, “An o(m) algorithm for cores decomposition of networks,” arXiv preprint cs/0310049, 2003

[2] JGraphT – URL: <https://jgrapht.org/javadoc/>

[3] D3.js – URL: <https://d3js.org/>

[4] A. Angel et al. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. PVLDB, 5(6):574–585, 2012.

[5] E. Fratkin et al. Motifcut: regulatory motifs finding with maximum density subgraphs. Bioinformatics, 22(14): e150–e157, 2006

[6] S. Khuller et al. On finding dense subgraphs. In ICALP, pages 597–608. Springer, 2009.

[7] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin. Efficient probabilistic k-core computation on uncertain graphs. In International Conference on Data Engineering (ICDE), pages 1192–1203. IEEE, 2018.

[8] L. Lu, T. Zhou, Q.-M. Zhang, and H. E. Stanley. The h-index of a network node and its relation to degree and coreness. Nature communications, 7:10168, 2016.

[9] Shin, K., Eliassi-Rad, T., Faloutsos, C.: CoreScope: graph mining using k-core analysis—patterns, anomalies, and algorithms. In Proceedings of IEEE ICDE (2016)

[10] H. Esfandiari, S. Lattanzi, and V. S. Mirrokni, Parallel and streaming algorithms for k-core decomposition, in ICML, 2018.

[11] R. Kannan and V.Vinay. Analyzing the Structure of Large Graphs. manuscript, August 1999.

[12] Gephi, The Open Graph Viz Platform – URL: <https://gephi.org/>

[13] Graphviz – URL: <http://graphviz.org/>

[14] NetworkX – URL: <http://networkx.github.io/>

[15] GraphStream – URL: <http://graphstream-project.org/>

[16] SNAP. Stanford network analysis package– URL: <https://snap.stanford.edu/data/index.html>

[17] S. Khuller and B. Saha. On finding dense subgraphs. In ICALP, 2009.

[18] S. Seidman. Network structure and minimum degree. Social Networks, 5(3):269–287, 1983.

**APPENDICES**

**Appendix A –Time efficiency based on parameters**

**K\_start:**

For this experiment, parameters values are graph dataset = DBLP, K\_increment=1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **K\_start** | **KCoreModified Algorithm (ms)** | **EdgeWeight Algorithm (ms)** |
| 1 | 5929 | 9257 |
| 5 | 5159 | 8192 |
| 15 | 4354 | 8095 |
| 25 | 4199 | 8089 |
| 35 | 5014 | 7769 |

Table 4: K\_start result

Higher the value of K\_start lesser is the execution time. It is because many nodes with degree less that K\_start get deleted in the preprocessing step.

**K\_increment:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **K\_increment** | **KCoreModified Algorithm (ms)** | **EdgeWeight Algorithm (ms)** |
| 1 | 6184 | 10430 |
| 5 | 6364 | 10194 |
| 15 | 6682 | 9895 |
| 25 | 6788 | 10161 |
| 35 | 6805 | 10264 |

Table 5: K\_increment result

We observe that when K\_increment values increases more nodes get deleted in the next iteration causing little increase in execution time of the program.

**MIN\_SCC\_SIZE:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2.

|  |  |  |
| --- | --- | --- |
| **MIN\_SCC\_SIZE** | **KCoreModified Algorithm (ms)** | **EdgeWeight Algorithm (ms)** |
| 50 | 6007 | 9582 |
| 100 | 6332 | 9826 |
| 150 | 5713 | 10190 |
| 200 | 5988 | 9303 |
| 250 | 6072 | 9630 |

Table 6: MIN\_SCC\_SIZE result

This parameter is used as a threshold parameter for the size of the subgraph. It doesn’t affect the execution time much as it is just used to check the size of the strongly connected component.

**DELETED\_NODE\_PERCENT:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, MIN\_SCC\_SIZE = 50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **DELETED\_NODE\_PERCENT** | **KCoreModified Algorithm (ms)** | **EdgeWeight Algorithm (ms)** |
| 40 | 11997 | 19344 |
| 50 | 11548 | 17675 |
| 60 | 11251 | 12658 |
| 70 | 12361 | 18653 |
| 80 | 12416 | 18522 |

Table 7: DELETED\_NODE\_PERCENT result

This parameter indicates when the iteration should stop while finding the k-core subgraph. More the percentage value the more the number of iterations. So, based on result, it can be said that more the value higher is the execution time.

**Appendix B – Density based on parameters**

**K\_increment:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=1, MIN\_SCC\_SIZE = 50, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **K\_increment** | **KCoreModified (Density: )** | **EdgeWeight (Density: )** |
| 1 | 32 | 44 |
| 5 | 32 | 44 |
| 15 | 32 | 44 |
| 25 | 32 | 44 |
| 35 | 32 | 44 |

Table 12: K\_increment result for density

Based on results it can be seen K\_increment doesn’t impact density.

**MIN\_SCC\_SIZE:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, DELETED\_NODE\_PERCENT =50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2.

|  |  |  |
| --- | --- | --- |
| **MIN\_SCC\_SIZE** | **KCoreModified (Density: )** | **EdgeWeight (Density: )** |
| 50 | 54 | 69 |
| 100 | 54 | 68 |
| 150 | 54 | 69 |
| 200 | 54 | 69 |
| 250 | 54 | 69 |

Table 11: MIN\_SCC\_SIZE result for density

Based on results it can be seen MIN\_SCC\_SIZE doesn’t impact density.

**DELETED\_NODE\_PERCENT:**

For this experiment, parameters values are graph dataset = DBLP, K\_start=5, K\_increment =1, MIN\_SCC\_SIZE = 50, VERTEX\_PERCENTAGE=0.4, WEIGHT\_THRESHOD = 0.2

|  |  |  |
| --- | --- | --- |
| **DELETED\_NODE\_PERCENT** | **KCoreModified (Density: )** | **EdgeWeight (Density: )** |
| 40 | 54 | 69 |
| 50 | 54 | 69 |
| 60 | 54 | 69 |
| 70 | 54 | 69 |
| 80 | 54 | 133 |

Table 12: DELETED\_NODE\_PERCENT result for density

Based on results it can be seen DELETED\_NODE\_PERCENT doesn’t impact density.