**THE UNIVERSITY OF MEMPHIS**

**Department of Computer Science**

**Efficient Core Maintenance on Dynamic Graphs**

**Yumi Kansakar**

**07/02/2021**

**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**

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 could not have been done.

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

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

**ABSTRACT**

This project aims to efficiently compute and maintain the core values of all the vertices in a dynamic graph. Although existing k-core decomposition approach can compute the core values of vertices in linear time, it is impractical to compute from scratch in case of graph updates, i.e., edge/vertex insertion and deletion. Techniques for core maintenance have been studied to reduce the computational cost to re-compute k-cores for a graph upon updates. In this project, we extend the classic peeling approach based on an observation of the k-core property and propose a complete solution for core maintenance. We validate our solution over a set of real-world graphs and report our findings when comparing with other two state-of-the-art methods.

The learning phase comprised of gathering knowledge about core decomposition and core maintenance. The implementation phase started with writing algorithms for core observation and maintenance followed by implementing with corresponding Java code. In the analysis phase, we ran the method over various graph datasets of different scales and analyzed the experimental results on a set of predefined parameters.

1. **INTRODUCTION**

Identifying subgraphs of the above discussed dynamic graphs can be one of the many challenges in graph analytics. Cohesive subgraphs such as k-core [1] is defined as the maximal subgraph of an undirected graph G, such that all the vertices in the subgraph have degree of at least k [5]. The core number of a vertex is the number of neighbor or the degree of the vertex. Computing the core numbers is known as core decomposition.

Maintaining the above discussed core numbers in a large real-world dynamic graph where the edges are inserted and deleted in real-time is the core maintenance. In other words, with the insertion and deletion of edges over time, the problem is to update the core numbers of the vertices in a timely and correct way so that the core numbers of the neighbor vertices are greater or at least equal to the core number of the vertex we are looking into.

The concept of dynamic graphs has been appearing in almost all the computer science fields with the change of data structures i.e., nodes in this case, with every execution of programs. Dynamic graphs are the graphs that change all the time. In the graphs, there are three cases. In case one, the edges are only added to the graph. In case 2, the edges are only deleted from the graph. And finally, edges can be either added or deleted from the graph. The final case can also be called a fully dynamic connectivity. Due to this concept of the dynamic graphs, the maintenance of the core values of the vertices in those graphs play an important role.

Dynamic graphs are playing a vital role in maintaining the graph network in real time data. For example, in social network like Facebook, Twitter, etc., in a graph representation, we can think of a node as a friend and the edge as the relationship that links the friends together [7]. Taking this example, there are many nodes inserted into the graph network and many nodes are deleted from this graph network. Updating the core values of the nodes is a major challenge and this project reflects upon the algorithm which can maintain the core of the nodes in an efficient way.

1. **RELATED WORK**

The concept of k-core in graphs was first introduced in [1]. Core decomposition in dynamic graphs has been extensively studied by many [10] [11] [12]. Core maintenance concepts were also discussed by many in [2] [3] [4] [5] [9]. Researchers have developed many algorithms to maintain the core of dynamic graphs in [2], [5] and [6] where the core value of the vertices in the graph are maintained. Authors in [6] have proposed algorithms that are linear in the size of the sub core which is the state-of-the-art approach for core maintenance. Authors in [5] have proposed a fast order-based approach to maintain the core values of the vertices of the graph by proposing a k-order with which the cost of core maintenance is reduced.

The goal of this project is to propose even more efficient algorithm to maintain the core of the vertices of the graph with respect to time and space. The algorithm that we have proposed is based on updating the core number of just the neighbors of the edges inserted or deleted in a real time graph and balancing the core values stands as per the observation. The computation is based upon the time taken to compute the core numbers of the vertices in the graph with insertion and deletion operations.

1. **PRELIMINARIES**

Let Graph be an undirected finite graph, where V denotes the set of vertices and E represents the set of edges in G. k is the number of core values in graph G. The degree or the number of edges connected to a vertex u is denoted as deg(u). The list of neighbors of each vertex is denote by N(u).

***Definition 1 (k-Core)***. Given a graph G = (V, E) and an integer k, a k-core is a maximal subgraph of G where the induced degree (the degree within the subgraph) of every vertex is at least k.

***Definition 2 (Core Number)***. Given a graph G = (V, E), the core-ness of a vertex is the maximum 𝑘 such that the vertex participates in a 𝑘-core, denoted by c(u).

***Definition 3 (Observation)***. Let c(u) be the core number of u, then there must be at least c(u) neighbors of u that have core number greater or equal to c(u).

1. **ALGORITHMS**

We have 3 algorithms implemented and compared to in our project. One is a Naïve k-core algorithm [1], order-based algorithm [5] and our proposed efficient algorithm for core maintenance.

The parameters use for the algorithms are given as follows:

|  |  |
| --- | --- |
| Parameters | Meaning |
| G (V, E) | An undirected finite graph G with vertices V and edges E. |
| K-CORE | A maximal connected subgraph of G |
| STATIC\_GRAPH | It indicates a minimum value for size to find a strongly connected component. |
| INSERT\_NODE\_PERCENT | Percentage of nodes which is inserted in the original graph in the process of updating the graph. |
| DELETE\_NODE\_PERCENT | Percentage of nodes which are deleted from the original graph in the process of updating the graph. |
| S | Empty set to add temporary vertices while edge insertion or deletion update. |

* 1. **Naïve Core Decomposition**

Naïve Core Decomposition algorithm is the base algorithm used to calculate the core value K, which is done in an iterative way. This algorithm starts with K = 1. It deletes all nodes having degree less than K. Next, the algorithm checks if the degree of all the vertices is greater or equal to K. Then with the increment of K by 1, the algorithm moves to the next iteration until the K has no more increment and stops.

**Algorithm: Naïve Core Decomposition**

Figure 1 Naive Core Decomposition Algorithm

**Input:** Graph

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

1. K = 1;

2. STATIC\_GRAPH = 60; ← 60% of all the graph dataset

3. degreeList ← contains calculated degree of each vertex u in

4. nodeDeleted ← holds percentage of nodes that been deleted

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

6. For each vertex u in

7. Repeat until (degree(u) ≥ K)

8. if (degree(u) < K)

9. = remove Vertex u from

10. degreeList ← calculate degree of each vertex in

11. End

12. End For

13. nodeDeleted ← calculate percentage of vertex deleted so far

14. K++;

15. End while

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

17. return Connected\_Component\_List

* 1. **Order-Based Algorithm**

The order-based algorithm is based on the paper [5], where the authors used an algorithm to update/maintain the core values of the vertices in a dynamic graph with insertion and deletion of nodes using the k-order in the graph. The authors proposed to reduce the computational cost to compute the k-cores for a graph when edges are inserted/deleted dynamically [5]. With the k-order, the algorithm can identify the set of vertices that need to be updated efficiently and can maintain the k-order for a graph with small overhead when edges are inserted/removed. We shall adopt this state-of-the-art method as a baseline to evaluate our solution.

* 1. **Modified Core Decomposition and Maintenance Algorithm**

The observation must hold for any vertices in the dynamic graph given the core value of the neighbors of a vertex must be greater than or equal to the main vertex. The pseudocode of the observation algorithm is shown in Figure 1. In this operation, the vertices are added to an adjacency list with its degree as the core value and its neighbor list with their degrees, respectively. The algorithm sorts the neighbors list in descending order and checks if the core value of the neighbor vertices of u has at least c(u) neighbors with greater or equal to the core value of u. If this holds, the operation will stop, else, the algorithm will deduct the neighbor one by one whose core value is lesser than u. With the deduction of each neighbor vertex, the observation is checked if it holds or not with the change in the degree of the vertex u.

For the insertion, let the edge to be inserted be (x, y) into the Graph G. We define an empty set S to hold the edges to be inserted and the common vertex whose neighbors are both x and y. First the algorithm checks if there is any vertex u whose list of neighbors contains both (x, y). If there exists any such vertex u, u is added to the empty set S. Then the algorithm checks if the core numbers of x and y are equal or not. If both the vertex has same core values, both the vertices are added to the set S. On the other hand, if c(x) is greater than c(y), the algorithm adds y to the set S. Finally, the vertices in S has its core value increased by 1. The pseudocode of the insertion algorithm is shown in Figure 2.

**Algorithm: Observation**

**Input**: G(V, E) , List of vertices and their neighbors

**Output**: Updated core values of all the vertices in G

1: if the Observation does not hold:

2: let N(u) = Sorted list of the neighbors of u (w.r.t. the core values in descending order)

3: int i = Max (1, deg(u));

4: c(Nk(u)) c(u) – t (where t is the number of vertices deducted from vertex u’s list of neighbors)

5: update c(u) = Min (i, c(Nk(u)));

6: Stop where there is no core number to be updated

Figure 2 Observation Algorithm

For the deletion algorithm, let the edge to be deleted be (x, y) from the Graph G. We define an empty set S to hold the edges to be deleted. The pseudocode of the deletion algorithm is shown in Figure 3.The algorithm first checks if the core values of x and y are equal or not. If they are equal both x and y are added into the set S. If core value of x is greater than y then the algorithm adds y to the empty set S. The algorithm updates the neighbor list of x and y be deleting x from y and y from x and checks if the recent vertices and it core values hold the observation or not. If the updated neighbor list violates the observation, then the core value of the vertex in set S is decreased by 1. After the core value update, now the neighbors core values also need to be checked and see if the neighbors core value holds the observation or not. So, by adding the neighbors of the deleted edge, the algorithm runs the observation and repeats the steps from 4 to 8 until the set S is empty.

**Algorithm: Insertion Update**

**Input**: G(V, E), (x, y) inserted edge, empty set S

**Output**: Updated core values of all the vertices with new edge insertion in G

1: for any u V, has both (x and y) V as the neighbors:

s = [u];

2: if c(x) = c(y): S = [u, x, y];

3: else if c(x) > c(y): S = [u, y];

4: for any vertex s in S: c(s) = c(s) + 1;

5: check is the observation holds with the above update in line 4 and update if necessary

Figure 3 Insertion algorithm

**Algorithm: Deletion Update**

Figure 4 Deletion algorithm

**Input**: G(V, E), (x, y) deleted edge, empty set S

**Output**: Updated core values of all the vertices with edge deletion in G

1: if c(x) = c(y): S = [x, y];

2: else if c(x) > c(y): S = [y];

3: while S [] do:

4: update the neighbor list of vertices in S;

5: if observation does not hold:

6: c(s) = c(s) - 1;

7: S = [list of neighbors of s]

8: remove s from S;

9: end while

1. **IMPLEMENTATIONS**

I have implemented the above algorithms in Java and have calculated the estimated time taken for the operation to be done with the memory usage of the algorithms. The processor used for this program to implement has Intel(R) Core (TM) i5-8250U CPU @ 1.60GHz 1.80 GHz.

1. **EXPERIMENT**

The experiments are performed on 5 datasets from SNAP library [8]. These graphs are undirected graphs, and the experiments are carried out to compute the time efficiency of the modified algorithm. The experiment is held into 2 stages. The first half of the experiment is done to compare the computation of k-cores of the static graph between the Naïve k-core algorithm and the proposed algorithm in this project. The second half of the experiment is done to compare the core maintenance in a dynamic graph while edges insertion and removal with the algorithm proposed in [5], the order-based algorithm and the algorithms from Figure 1, Figure 2 and Figure 3.

The experiments have been carried out to evaluate the following metrics – time and space efficiency of the outcome of the maintained core values in a graph. Note that due to the size of the selected datasets, we need to increase the heap size according to the dataset. Following sections discuss about these in detail.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File name | Graph Type | # Of nodes | # Of edges | File size in Mb |
| com-facebook.combined.txt | Social circles from Facebook | 63,731 | 817,035 | 12 |
| com-dblp.ungraph.txt | DBLP collaboration network | 1,314,050 | 5,362,414 | 13.2 |
| com-youtube.ungraph.txt | YouTube online social network | 3,223,589 | 9,375,374 | 36 |
| com-lj.ungraph.txt | LiveJournal online social network | 4,846,609 | 42,851,237 | 478 |
| com-orkut.ungraph.txt | Orkut online social network | 3,072,441 | 117,185,083 | 1640 |

Table 1 Datasets

* 1. **Experiment 1**

The first experiment is conducted to compare the Naïve core decomposition algorithm and our proposed new algorithm from Figure 1, to compute the k-core of the static graph. In the table below, we can see the results based on time efficiency and space both the algorithm takes to compute k-core. We have started k from 1 until the algorithm finds the maximum k and the static graph that we are using is 60% of each dataset.

* + 1. **Time Efficiency**

Here we will see the comparison between both the algorithms based on the time efficiency.

The above line chart shows the efficiency of the modified algorithm compared to the naïve core decomposition algorithm from Figure 1. Here we can see the modified algorithm runs much faster in terms of time computation. Note that the - - in the Orkut dataset is because the algorithm could not compute on the high server and crashed due to its larger dataset even when we took 30% of the whole dataset to observe the static graph.

* + 1. **Space Efficiency**

Here we will see the comparison between both the algorithms based on the memory efficiency.

|  |  |  |
| --- | --- | --- |
| File | Modified Core Observation (mb) | Naïve Core Decomposition (mb) |
| com-facebook | **143** | 167 |
| com-dblp | **175** | 634 |
| com-youtube | **714** | 2317 |
| com-lj | **3714** | 8821 |
| com-orkut | **874** | * - |

Table 2 Performance Comparison with Naive Core Decomposition (memory used in mb)

Table 2 clearly shows the memory usage of the modified algorithm is much more lesser than the naïve algorithm. Reason is the modified algorithm does not delete its edges but just provides the core value of all the graph data with its neighbors.

* 1. **Experiment 2**

The second experiment is conducted to compare our proposed modified algorithm for insertion/removal or vertices and edges with the proposed Order-Based algorithm in [5]. The experiment is based on time efficiency and memory usage. For the experiment, we base our findings based upon the experiments done in [5]. We insert 20% of the latest sampled edges for all 5 datasets and compute its’ time and the memory usage. For removal, we sample 20% randomly selected vertices to remove from the dynamic graph and compute its’ time and memory usage in average. Following are the outcomes of our experiment.

* + 1. **Time Efficiency**

Even though the modified algorithm run faster than the naïve core decomposition algorithm. Order-based algorithm runs much faster than our proposed algorithm in the case of edges insertion and removal. This may be because our algorithm run the observation and must update the core values of the inserted edges in all other occurrence and for removal likewise.

* + 1. **Memory Usage**

|  |  |  |
| --- | --- | --- |
| File | Modified Edge Insert/Deletion (mb) | Order Insertion/Deletion (mb) |
| com-facebook | 18.6 | 21 |
| com-dblp | 132 | 157 |
| com-youtube | 199 | 200 |
| com-lj | 469 | 539 |
| com-orkut | 1123 | * - |

Table 3 Performance comparison (memory usage in mb)

For the performance of memory usage, we have computed the memory usage of our algorithm with the order-based insertion/removal algorithm. Note, for Orkut data the order-based algorithm could not load due to high amount data in it.

1. **CONCLUSION**

In this project, we studied about the core maintenance and proposed a modified algorithm to maintain the core values in a dynamic changing real-world graph. Maintaining core values can cost a lot of computational time but with this proposed algorithm, we were able to run the core maintenance of the dynamic graphs in an efficient time in comparison to other algorithms proposed in [5]. From the experiment we learn that our proposed modified observation algorithm runs faster and better in terms of memory usage than that or a naïve core decomposition algorithm. In terms of insertion and removal of edges in the dynamic graph, the order-based algorithm has shown better results based on time whereas our proposed algorithm uses little space than the order-based algorithm.

1. **FUTURE WORK**

Further for the future works related to this project, we aim to test run on larger dynamic data using a much faster server/computer to verify the scalability and memory utilization. We plan to generate a hands-on user experience where the user can visually interact with the graph data and its subgraph with the core values maintain for the vertices in the graph.

**REFERENCE**

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

[2] Wang, Na, et al. "Parallel algorithm for core maintenance in dynamic graphs." 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2017.

[3] Jin, Hai, et al. "Core maintenance in dynamic graphs: A parallel approach based on matching." IEEE Transactions on Parallel and Distributed Systems 29.11 (2018): 2416-2428.

[4] Li, Rong-Hua, Jeffrey Xu Yu, and Rui Mao. "Efficient core maintenance in large dynamic graphs." IEEE Transactions on Knowledge and Data Engineering 26.10 (2013): 2453-2465.

[5] Zhang, Yikai, et al. "A fast order-based approach for core maintenance." 2017 IEEE 33rd International Conference on Data Engineering (ICDE). IEEE, 2017.

[6] Saríyüce, Ahmet Erdem, et al. "Streaming algorithms for k-core decomposition." Proceedings of the VLDB Endowment 6.6 (2013): 433-444.

[7] "Using Facebook To Learn Some Basics About Graph Theory". Medium, 2021, [https://medium.com/@ja3606/using-facebook-to-learn-some-basics-about-graph-theory-ecadb31cd127. Accessed 23 Apr 2021](https://medium.com/@ja3606/using-facebook-to-learn-some-basics-about-graph-theory-ecadb31cd127.%20Accessed%2023%20Apr%202021).

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

[9] Aridhi, Sabeur, et al. "Distributed k-core decomposition and maintenance in large dynamic graphs." Proceedings of the 10th ACM international conference on distributed and event-based systems. 2016.

[10] Jakma, Paul, et al. "Distributed k-core decomposition of dynamic graphs." Proceedings of the 2012 ACM conference on CoNEXT student workshop. 2012.

[11] Sarıyüce, Ahmet Erdem, et al. "Incremental k-core decomposition: algorithms and evaluation." The VLDB Journal 25.3 (2016): 425-447.

[12] Malliaros, Fragkiskos D., et al. "The core decomposition of networks: Theory, algorithms and applications." The VLDB Journal 29.1 (2020): 61-92.