Efficient Implementation of Anchored 2-core Algorithm

Babak Tootoonchi
University of Victoria
Victoria, BC, Canada
babakt@uvic.ca

Venkatesh Srinivasan
University of Victoria
Victoria, BC, Canada
srinivas@uvic.ca

Alex Thomo
University of Victoria
Victoria, BC, Canada
thomo@uvic.ca

***Abstract*— Often graph theory is used to model and analyze different behaviors of networks including social networks. Nowadays, social networks have become very popular and social network providers try to expand their networks by encouraging people to stay engaged and active. Studies show that engagement and activities of people in social networks influence engagement of their connections. This behavior has been modeled by the *k*-core problem in graph theory with the assumption that a person stays active in the network if he or she has *k* or more connections. In the above model if a person drops out, his or her friends can become discouraged and they might also drop out. An approach called anchored k-core algorithm has been introduced lately that prevents a cascade of drop-outs by finding nodes which have the most influence on their connections and rewarding them to stay in the network. In this work, an efficient implementation of the anchored 2-core approach has been proposed. The proposed implementation method was applied to a set of real world network data that includes very large graphs with millions of links. The results show that with only a few anchors, it is possible to save hundreds of nodes for the 2-core graph. Also, the execution time of our implementation is in order of minutes for huge datasets which proves the efficiency of our implementation.**

Keywords—social networks; anchored k-core; network unraveling

I. INTRODUCTION

One of the best ideas in the internet era is the development of social networks on World Wide Web. People use social network websites to share photos, videos, music and other information with either a select group of friends or with the public. In the past few years, social networks have played an important role in connecting people around the world and have become one of the most powerful media for communication.

Everyone's experience of a social network depends on the contents of the contributions of that person's connections. Interesting content and actively contributing friends provide an incentive for a user to continue logging into the site, and might encourage him or her to contribute more content of their own. Therefore, when an individual actively contributes to a social network, his or her friends become more active and there is a higher chance that they stay engaged and do not drop out [1]. This effect can propagate among peers and increase the connectivity and the number of users in a social network. Increasing the connectivity is one of the main goals of social network providers and is the key to keeping the network alive, growing and profitable. An individual is more likely to be engaged if many of his or her friends are engaged. This assumption is the main reason to consider social networks as one of the applications of algorithms that calculate the *k*-core organization of a network [2-5,10].

Assume that the individuals who have *k* or more engaged friends will stay in the network while others who have less active friends will leave the network eventually. If a person leaves the network, it influences his or her friends and may cause the number of their connections to drop below *k*. Hence, the friends that have less than *k* friends will also leave the network based on the above assumption. This effect spreads throughout the network and can cause a cascade of drop outs that can significantly reduce the number of users. At the end, what remains from the network is a subgraph in which every node has at least *k* adjacent nodes. This subgraph is called *k*-core graph of the original network and is unique in the sense that it does not depend on the order in which the nodes have dropped out of the original graph. *k*-core decomposition is a well-known concept in graph theory and has various applications such as in modeling real world web networks, protein structures, information retrieval, text summarization, etc. [6-8].

As described in the *k*-core model of the social networks, user dropouts can cause the network to be partitioned and similar effects over time can cause a social network's life to come to an end. In [9], Bhawalkar

et. al have introduced a method to prevent unraveling in social networks by locating the most valuable nodes, called "anchors", and rewarding those users to stay in the network. Anchors are the nodes which have the greatest impact on the network connectivity and the number of users if they are removed. The paper introduces an algorithm that solves the optimization problem of maximizing the network connectivity or graph size by finding the minimum number of anchors and rewarding them to stay engaged in the network.

The anchored *k*-core problem has been studied both empirically and theoretically by a few other works. The authors of [24] showed that the anchored *k*-core problem is W[1]-hard parameterized by the size of the core *p*. This improves the result of [9] which shows W[2]-hardness parameterized by *b*. The work in [25] extends the anchored *k*-core problem to directed graphs and provides new algorithmic and complexity results. There have been some empirical studies of the problem across multiple online social networks [26, 27]. These works have studied different factors that can contribute to the resilience of the social networks. The authors of [28] proposed a variation of the anchored *k*-core problem called peeling process in which the goal is to minimize the size of *k*-core. They show that the problem is NP-complete for all $k \geq 2$.

The approach that was introduced in [9] is a complex algorithm that requires a lot of computation cycles. The authors of [9] did not provide any details on possible implementation approaches for the proposed unraveling algorithms. Considering the complexity of the anchored *k*-core algorithm, the question is if it is viable to run it on a single consumer-grade PC. This work presents an efficient approach towards implementation of the anchored 2-core algorithm that makes it possible to run the algorithm on a single machine in reasonable time. The proposed approach utilizes fast algorithms and a chaining hash map to store the interim search results. It also uses an efficient implementation of the *k*-core algorithm, presented in [4], to compute the *k*-core decomposition at every iteration. For the graph database in our efficient implementation, we used Webgraph [11,12], which is a highly efficient graph compression framework. Due to the efficiency of Webgraph, it was used by other works to solve similar large scale problems on a single machine [21-23]. To prove the performance and scalability of the presented approach, experiments were run on a variety of real-world graph datasets ranging from a few thousands to billions of edges in size. All the experiments were performed on a single consumer-grade PC and the results showed that even for massive graphs with billions of edges, the elapsed time was in the order of minutes.

The remainder of this paper is structured as follows. Section II provides the basic concepts in graph theory that are required to understand the discussions in this paper. Section III discusses unraveling in social networks and introduces the anchored *k*-core approach to prevent it as was proposed in [9]. Section IV shows the details of our efficient implementation of the anchored 2-core algorithm [9]. The experimental results of applying our efficient approach on a set of large graphs on a single machine are shown in Section V. Section VI concludes the discussion and provides directions for possible future work.

II. BASIC CONCEPTS

Consider an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Note that for the purpose of this paper, wherever we refer to a graph we mean an undirected graph. The number of vertices is denoted by $|V| = n$ and the number of edges will be represented by $|E| = m$. Vertices v and w are adjacent if there exists an edge $(v, w) \in E$ between v and w. In this case, v and w are called neighbors. For a given vertex v in graph G, the set of all the neighbors of v is shown as $N_G(v) = \{u : (u, v) \in E\}$. The number of all the neighbors of v is called the degree of vertex v and is shown by $d_G(v) = |N_G(v)|$. The maximum degree of a graph G is shown by $\Delta(G) = max\{d_G(v) : v \in V\}$. The minimum degree of graph G is denoted by $\delta(G) = min\{d_G(v) : v \in V\}$.

Let $S \subseteq V$ be a subset of the vertices of the graph $G = (V, E)$. The subgraph $C = (S, E_S)$ is called the subgraph of G that is induced by S where $E_S = \{(u, v) \in E : u, v \in S\}$.

Definition 2.1. For a given $k \in \{0, \dots, \Delta(G)\}$, a subgraph $C = (S, E_S)$ of $G = (V, E)$ induced by the set $S \subseteq V$ is called a *k-core* (or core of order *k*) of G if and only if the degree of every vertex $v \in S$ is equal or greater than *k*, i.e. $\{\forall v \in S: d_C(v) \geq k\}$ or $\delta(C) \geq k$, and C is a maximal induced subgraph of G with this property. We denote the *k*-core subgraph of G as $C_k(G)$.

Every node in a *k*-core subgraph has at least k neighbors. Since *k*-core is the maximal subgraph of G that holds the conditions of Definition 2.1, it is unique and for every graph G and for a given value of *k*, there exists exactly one *k*-core subgraph. Note that the *k*-core subgraph can be empty i.e. $C_k(G) = \emptyset$ depending on the value of *k*. The *k*-core will be empty for all the values of $k > \Delta(G)$. Any graph G is also its own *k*-core, $C_k(G) = G$, for $0 \leq k \leq \delta(G)$. For example, $C_0(G) = G$. $C_1(G)$ is a subgraph of G that is achieved by deleting all the isolated vertices in G.

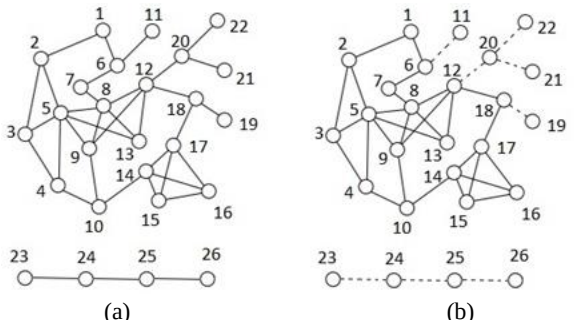

Fig. 1. (a) graph G with 26 nodes, (b) 2-core of graph G in solid lines

Definition 2.2. For a given vertex $v \in G$, the core number or *coreness* of v is the largest value for k such that $v \in C_k(G)$.

The maximum core number of a graph G is the maximum coreness of the vertices of G. The cores of a graph are nested which means for any $i < k$, $C_k(G) \subseteq C_i(G)$. Likewise, the generalization of this property is shown in the following equation.

$$C_{\Delta(G)}(G) \subseteq C_{\Delta(G)-1}(G) \subseteq \cdots \subseteq C_{\delta(G)}(G) \qquad (1)$$

Note that the *k*-core subgraph of a graph G is not necessarily a connected graph. For example, consider the graph G that is shown in Fig. 1(a). To extract the 2-core subgraph of G we need to eliminate nodes that have a degree less than 2. The result is illustrated in Fig. 1(b).

III. Anchored k-Core Problem

In the *k*-core model of social networks, it is assumed that the threshold at which users will become disengaged is at the point where they have less than *k* friends. Hence when a person drops out, all of his or her connections who have exactly *k* friends will become disengaged because the number of their connections will drop below *k*. This effect can spread throughout the network and cause a cascade of withdrawals which is very unpleasant for the social network providers.

Depending on the location of the node in the network topology and the number and topology of his or her friends, the dropout cascade effect can be small or dramatic. The nodes, for whom leaving the network is very expensive in terms of reduction in the network connectivity and size, will have greater value in the network and it is to the benefit of the network provider to keep them engaged. These nodes are called anchors in [9] and it is shown how it is possible to locate the anchors in a network in the most efficient way.

For an undirected graph $G = (V, E)$ with size n, assume that values k and b are given where $k, 0 < k \leq \Delta(G)$, represents the maximum degree threshold for staying engaged and b, $b \in \{1,2,\dots,n\}$, denotes some kind of budget that a social network provider has to offer. The anchored *k*-core problem is to find a set S of at most b nodes among all possible subsets of size b, where $S \subseteq V$ and $|S| \leq b$, such that keeping those nodes regardless of their degrees while calculating the *k*-core graph results in a *k*-core with the maximum possible number of nodes. In other words, the problem is to find a subset of at most b nodes, which are the most valuable vertices and are called anchors, and give them incentive to stay in the network. This way, the social network provider maximizes the size of the *k*-core, subject to budget b. The incentive can be any type of benefit including offering rewards, waving premiums, or giving some sort of rebates or points, etc.

In [9], the authors showed that the anchored *k*-core problem for $k = 2$ can be solved in polynomial time and for $k \geq 3$, it is NP-hard to distinguish between instances in which $\Omega(n)$ vertices are in the optimal anchored *k*-core, and those in which the optimal anchored k-core has size only $O(b)$. Also, for every k $\geq$ 3, the problem is W[2]-hard with respect to the budget parameter b.

A. RemoveCore Subgraph

To compute the anchored *k*-core of graph G, a subgraph of G, called $RemoveCore(G)$ is computed as described below.

Definition 3.1. For graph G and a given k, let C_k be the set of vertices of the *k*-core of G. Removing all edges between all pairs of vertices $u, v \in C_k$ will result in a graph that is called RemoveCore(G) [9].

For $k = 2$ the $RemoveCore(G)$ subgraph of G is a forest where each tree in the forest contains at most one vertex from the 2-core.

Definition 3.2. Each tree in the $RemoveCore(G)$ graph of G is called *rooted* if it contains a node from the *k*-core graph. Otherwise, it is called *non-rooted*. The sets of rooted and non-rooted trees are denoted by R and S respectively [9].

To find the $RemoveCore(G)$, we first run the *k*-core algorithm for $k = 2$ and compute the set C_k of vertices of the 2-core of G. Then, we assume that the 2-core vertices shape a single virtual vertex named r. Note that C_k can be disjoint and hence *r* can include disjoint graphs. What remains is a single tree that has the vertex *r*, and zero or more other trees. The single tree that contains r is the aggregate of all the rooted trees assuming their roots fall on a single node. The rest of the trees represent the non-rooted trees.

Definition 3.3. An *anchor* is a vertex that is assigned a budget to be included in the *k*-core regardless of its degree.

Placing an anchor on vertex *v* will increase the degree of its neighbors by one and may cause their degrees to reach *k*. The nodes that are added to the *k*-core by assigning an anchor to *v* are considered to be *saved* by anchor *v*.

We can think of the C_k vertices as already being anchored because each vertex in the *k*-core subgraph would remain in the graph without the assistance of any anchors. Therefore, the anchored 2-core problem is reduced to finding a solution for the $RemoveCore(G)$ graph with an anchor placed on *r* for free.

B. Anchored 2-Core Algorithm

The anchored 2-core approach presented in [9] is exact in that it guarantees to find an anchored 2-core of the maximum size for a given budget. The first step of the algorithm is to find a vertex $v_1 \in \mathcal{R}$, where $\mathcal{R}$ is the set of rooted trees as described in Definition 3.2, such that placing an anchor on v_1 maximizes the number of vertices saved across all placements of a single anchor in $\mathcal{R}$. Also, another vertex $v_2 \in \mathcal{R}$ is found in a similar manner assuming an anchor has already been placed on v_1. In other words, considering the $RemoveCore(G)$ with the virtual vertex *r* on the 2-core nodes, v_1 will be the farthest vertex from *r*, and v_2 will be the second farthest vertex from *r* after v_1 has been selected and all the vertices on the $r - v_1$ path have been contracted into *r*. Next, two other vertices $v_3, v_4 \in S$ are found, where S is the set of non-rooted trees, such that placing two anchors at v_3 and v_4 simultaneously maximizes the number of vertices saved across all placements of the two anchors in S. In other words, v_3 and v_4 are on the endpoints of the longest path across all the trees in S.

Assume $C_{\mathcal{R}}(v_1)$ and $C_{\mathcal{R}}(v_2)$ are the number of vertices saved by placing anchors v_1 and v_2 respectively. Similarly, let $C_S(v_3, v_4)$ be the number of vertices saved by placing two anchors on v_3 and v_4 simultaneously. If $C_{\mathcal{R}}(v_1) + C_{\mathcal{R}}(v_2) > C_S(v_3, v_4)$ or $b = 1$, an anchor will be placed on v_1 and the budget *b* will be reduced by 1. If $C_{\mathcal{R}}(v_1) + C_{\mathcal{R}}(v_2) \leq C_S(v_3, v_4)$, two anchors will be placed on at on v_3 and v_4, and the budget *b* will be decreased by 2. After the anchor placement, the saved vertices are added to the 2-core and the $RemoveCore(G)$ is calculated again. This process repeats until the budget is completely used $b = 0$.

IV. Efficient Implementation of Anchored 2-core Algorithm

The purpose of this work is to propose an efficient approach towards the implementation of the anchored 2-core algorithm. As mentioned before, one of the applications of this algorithm can be in studying unraveling and preventing it in social networks. Online social network graphs tend to be large with millions of nodes and connections. Hence, the question is if it is viable to run the anchored 2-core algorithm on real world network data on a single consumer-grade machine.

Studying web graphs is often difficult due to their large size. WebGraph is a framework for graph based databases that was designed to facilitate studying of web graphs [11]. Webgraph provides efficient usage of memory by utilizing compression techniques such as gap compression [15], referentiation [16] and intervalisation. It also provides a fast API for randomly accessing graph nodes and vertices [11,12]. The WebGraph framework also contains data sets for very large graphs with billions of links [13] which were either gathered from public sources [17] or obtained with UbiCrawler [18,19]. These features make Webgraph a suitable choice for dealing with large graph based databases and hence it was used for our implementations in this work.

In the anchored 2-core algorithm, to find the $RemoveCore$ subgraph of a graph G, first the k-core of graph G needs to be extracted. Different variations of *k*-core algorithm have been proposed so far [2,4,5,10, 14]. In this work, we used an efficient implementation of the Batagelj and Zaversnik (BZ) algorithm [14] that was proposed in [4]. This implementation is referred to as WG_BZ and uses WebGraph as the graph database framework. The main idea of WG_BZ [4] is to flatten the set of vertices of BZ into a few arrays, which make it significantly faster than other *k*-core algorithms.

To store the longest paths that are computed at each iteration of the anchored 2-core algorithm for rooted and non-rooted trees, the *SequentialSearchST* and *SeparateChainingHashST* structures [20] were used. The SeparateChainingHashST table maintains an array of SequentialSearchST objects that are accessible by using a hash function. SequentialSearchST is an unordered list of key-value pairs that uses sequential search. As will be shown below, we used the SeparateChainingHashST structure to implement the SC_hash table that we used in our implementation of the algorithm.

Algorithm 4.1 Anchored 2-core Algorithm

1. *function anchored_2-cores* (*Graph G, Integer budget*)
2. d, b, D ← *compute K-Core(G)*
3. *while* ($budget > 0$)
4. *initialize* (S, R)
5. R ← *compute* $RemoveCore(G, D, b, R, 2)$
6. *if* $size\ of\ R > 0$ *then*
7. *initialize* (SC_hash)
8. *for all* $i = 0$ *to* $size(R)$ *where* $d[R[i]] == 1$
9. $checkRootedTree(R[i], i, 0)$ *// builds SC_hash*
10. *end for*
11. *using SC_hash find:*
12. v_1 ← *farthest vertex from core on a rooted tree*
13. v_2 ← *second farthest from core on a rooted tree*
14. (v_3, v_4) ← *endpoints of the longest path across all non-rooted trees*
15. *if* $C_R(v_1) + C_R(v_2) > C_S(v_3, v_4)$ *then*
16. $d[v_1] = \Delta(G)$, *budget = budget – 1*
17. d, b, D ← *compute K-Core(G)*
18. *else*
19. $d[v_3] = \Delta(G)$, $d[v_4] = \Delta(G)$, *budget = budget – 2*
20. d, b, D ← *compute K-Core(G)*
21. *end if*
22. *else go to end*
23. *end if* *// size R>0*
24. *end while*
25. *end*

The following data structures were used in our implementation of anchored 2-core algorithm.

- Arrays d, D, p, and b, that were used in the WG_BZ *k*-core algorithm [4].
- Array R stores the set of vertices which are removed from a given graph to create the 2-core graph. In other words, R stores the vertices that are in the $RemoveCore$ of graph G.
- Array S holds the status of each vertex in the $RemoveCore$ graph which can be either rooted or non-rooted.
- Table SC_hash stores a hash table of vertices in the $RemoveCore$ graph. It is composed of a list of linked lists that store the longest paths on rooted and non-rooted trees. Each entry in the linked lists is a (key,value) pair where key is the index of the vertex and value is the distance of vertex from the vertex at the head of the corresponding linked list.

Algorithm 4.1 shows the implementation details of the anchored 2-core algorithm. The input of the algorithm is a graph, G, and the maximum available budget. In the first step of the algorithm (line 2), WG_BZ algorithm is used to compute the 2-core of graph G, which is effectively defined by arrays D and b. The main loop of the algorithm iteratively finds anchors and assigns budgets to them until it runs out of budget.

After the initialization of the S and R arrays, the $RemoveCore$ of G is computed for $k = 2$ (line 5). The $RemoveCore$, R, is computed by simply copying the first bucket of vertices from D to R i.e. indices 0 to $b[k] - 1$. If R is non-empty, a SeparateChainingHash, SC_hash, is initialized based on the vertices in R. A list of pointers is created in the SC_hash such that each pointer corresponds to a leaf vertex in R and points to the head of an empty linked list. Note that we only compute the longest paths from the leaves as shown in lines 8 and 9 of the algorithm.

The $checkRootedTree(u)$ method at line 9 fills up the SeparateChainingHash table and array S. Algorithm 4.2 shows a detailed explanation of this function. The $checkRootedTree$ function recursively searches the tree where u is located and persists each node and its distance from u in $SC\_hash[i]$ where $u = R[i]$. If a root (a node on the 2-core) is found, then the tree is marked as rooted and the distance of root will be set to 0 (line 14 of Algorithm 4.2). At this point another recursive function is called to update the distances of all nodes to be from the root rather than u (line 16). This second function is called $computeDistance$. Note that there is at most one root on each tree as otherwise all the nodes on the path between the two or more roots would be on the 2-core and could not be in the $RemoveCore$ graph.

In Algorithm 4.1, after the SC_hash is filled, it is easy to find the most valuable nodes in the $RemoveCore$ graph. The first two nodes which are on two separate rooted trees and have the longest distance to the root are selected (v_1 and v_2 at line 12-13 of Algorithm 4.1). Then, the two endpoints of the longest path among all non-rooted trees are identified, v_3 and v_4 (line 14). If assigning anchors to v_1 and v_2 saves more nodes, an anchor is assigned to v_1, and again the 2-core is calculated to update D (lines 16-17). Otherwise, two anchors are assigned to v_3 and v_4 and the 2-core is called again (lines 19-20).

Note that to simulate the impact of budget on the nodes, the degree of the anchored nodes is assigned to the maximum degree of graph G. This guarantees that the 2-core algorithm will not place these nodes amongst the 1-core nodes or first bin in D. The loop continues finding anchors and assigning budget until it runs out of budget, or if there are no more vertices left to save in the $RemoveCore$ graph.

Algorithm 4.2 Fill up the SeparateChainingHash table

1. *function* $checkRootedTree(u, i, distance)$
2. *if* $u \in R$ *and u is not visited*
3. *if* $SC\_hash[i]$ *does not contain u*
4. $SC\_hash[i].put(u, distance)$, $S[u] = "non-rooted"$
5. *mark u as visited*
6. *for each neighbor v of u*
7. *if v is not visited and* $SC\_hash[i]$ *does not contain v*
8. $SC\_hash[i].put(v, distance + 1)$
9. $S[v] = "non-rooted"$
10. *end if*
11. *if* $v \in R$ *then*
12. $checkRootedTree(v, i, distance + 1)$
13. *else*
14. $S[v] = "rooted", distance = 0$
15. $SC\_hash[i].put(v, distance)$
16. $computeDistance(v, i, 0)$
17. *go to end //line 21*
18. *end if*
19. *end for*
20. *end if*
21. *end*

V. Experimental Results

This section, provides the implementation results of examining the proposed approach on real world network data ranging from small to large networks with millions of links. All the implementations were done using the Java programming language and WebGraph framework. The experiments were run on a system with a 64-bit Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz and 12 GB RAM. The evaluation was performed by applying the proposed implementation on a set of twelve test suites. The execution time proves the efficiency and speed of the proposed implementation methods.

Table I shows the result of applying our anchored 2-core implementation on twelve datasets ranging from small to large networks with budget b=5. The first column shows the name of test suits and the second column shows the number of nodes for each network. The size of the 2-core graph in each network is shown in column 3. Column 4 shows the number of nodes that have been saved by finding the three most valuable anchors and assigning the budget to them. The last column shows the execution time of the algorithm in seconds.

As shown in the fourth column, the number of nodes that are saved by placing three anchors can vary based on the network topology not network size. For example, the number of nodes which were saved in data7_web_berk_stan is much higher than the number of nodes which were saved in data12_uk-2005-edgeList while uk-2005 is significantly larger than web_berk_stan. On the other hand, in the graph of network data5_soc_slashdot, the number of nodes that are saved is the same as the number of anchors that are assigned to them.

The results of running the anchored 2-core algorithm with budget 10 on the same set of networks are shown in Table II. As it is shown in Table II, adding more budget leads to saving more nodes. For example, 2633 nodes were saved by assigning 10 anchors to the web_berk_stan graph. Adding more budget, linearly increases the iteration cycles in the algorithm which increases the execution time as illustrated in Table II.

TABLE I. Results of our 2-core implementation with b=5

Data set	# of nodes	2-core size	# of nodes saved	Execution time (s)
data1_astrocnet	133,280	17,440	12	0.209
data2_condmatcnet	108,300	20,613	16	0.136
data3_p2pgnutella	62,586	33,816	11	0.307
data4_soc_sign_slashdot	82,144	52,103	18	0.96
data5_soc_slashdot	82,168	80,365	5	1.125
data6_amazon	403,394	390,938	21	3.352
data7_web_berk_stan	685,231	629,459	1663	16.959
data8_wiki_talk	2,394,385	622,999	15	1435.891
data9_soc-LiveJournal	4,847,571	3,784,309	21	60.391
data10_roadnet_tx	1,393,383	1,093,520	56	2.686
data11_roadnet_ca	1,971,281	1,591,795	137	5.93
data12_uk-2005-edgeList	39,459,923	35,580,606	41	805.293

TABLE II. Results of our 2-core implementation with b=10

Data set	# of nodes	2-core size	# of nodes saved	Execution time (s)
data1_astrocnet	133,280	17,440	22	0.366
data2_condmatcnet	108,300	20,613	27	0.252
data3_p2pgnutella	62,586	33,816	22	0.747
data4_soc_sign_slashdot	82,144	52,103	28	1.768
data5_soc_slashdot	82,168	80,365	10	1.587
data6_amazon	403,394	390,938	41	5.898
data7_web_berk_stan	685,231	629,459	2633	26.322
data8_wiki_talk	2,394,385	622,999	24	2196.921
data9_soc-LiveJournal	4,847,571	3,784,309	39	109.137
data10_roadnet_tx	1,393,383	1,093,520	128	6.067
data11_roadnet_ca	1,971,281	1,591,795	201	8.682
data12_uk-2005-edgeList	39,459,923	35,580,606	76	2359.54

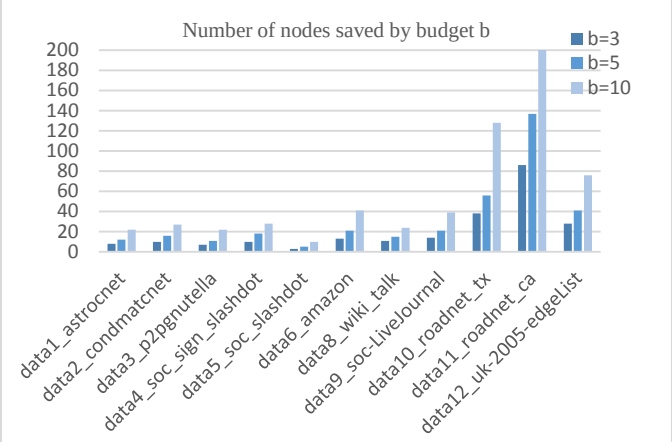

Fig. 2. Number of nodes saved by assigning budget b

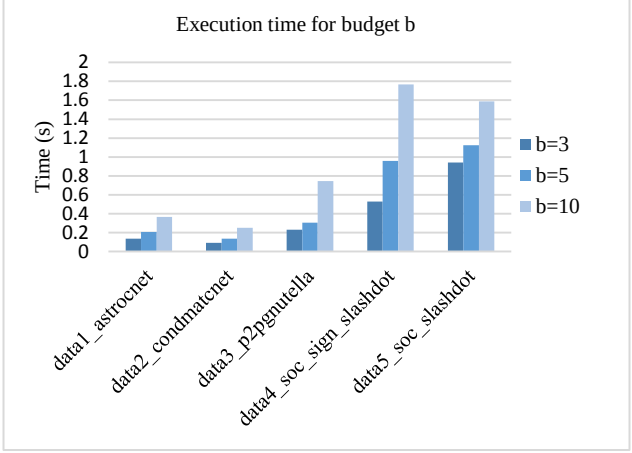

Fig. 3. Execution times of Algorithm 4.1 on data set 1

For dataset data8, the execution time is higher than data12 for b=5 and more comparable for b=10, which shows the impact of the *RemoveCore* topology on the execution time. Note that although data8 is much smaller than data12, its *RemoveCore* size is almost half of the *RemoveCore* size in data12 (i.e. 1.8 million compared to 3.8 million). The high execution time in data8 is because of trees with a large number of branches in the *RemoveCore* of data8. Assigning more budget breaks down these trees into smaller pieces and reduces the number of backtrackings. For example, the execution time becomes more comparable to data12 for b=10.

Fig. 2 illustrates the number of nodes saved by 3, 5, and 10 anchors in the above datasets. Data7 is not shown because of difference in order of magnitude. As it is shown in Fig. 2, the number of nodes that were saved is not a function of the size of the network, but rather it depends on the topology of the network. Also, there is no linear relation between the number of saved nodes and the assigned budget because it is heavily dependent on the network topology. However, the impact of using only a few anchors is impressive.

To illustrate the execution times, the results were divided into three data sets of similar ranges for better viewing. The charts are shown in Fig. 3, 4, and 5. As it is shown in Fig. 3, for graphs of size 50k to 150k the execution time is in order of a few seconds. For the larger graphs shown in Fig. 4 with up to 4.8 million nodes, the execution time is in order of a few minutes. For our largest dataset with 39.5 million nodes, the execution time is 39 minutes for budget 10. Note that the execution time is not directly a function of the graph size, but it depends on the topology of the graph as well. For example, for data11_roadnet_ca with almost 2 million nodes the execution time is 8.7 seconds which is almost one third of the execution time of data7_web_berk_stan with 685k nodes.

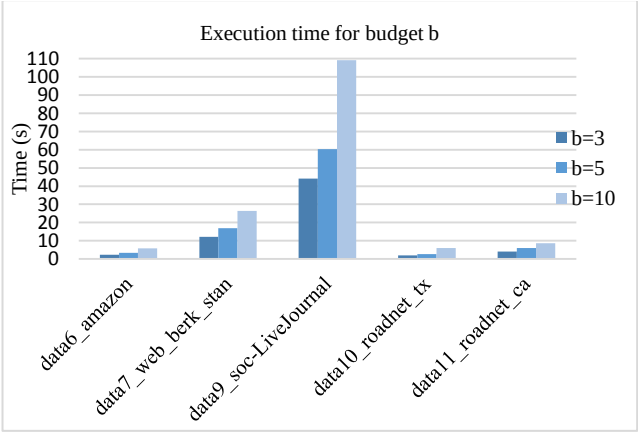

Fig. 4. Execution times of Algorithm 4.1 on data set 2

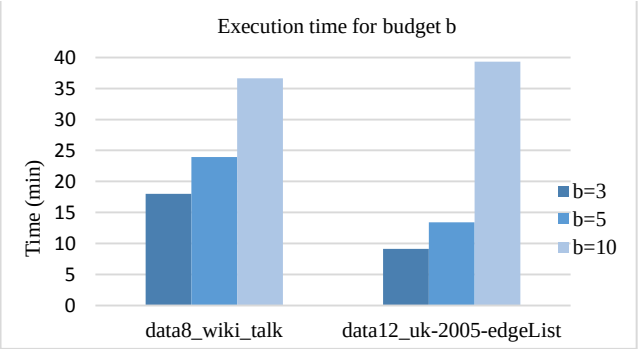

Fig. 5. Execution times of Algorithm 4.1 on data set 3

As mentioned earlier and can be seen in the charts, adding more budget linearly increases the iteration cycles of the algorithm, but the increase in the execution time is less than linear growth. The results prove that our implementation scales very well considering the size and complexity of these graphs and the reasonable execution times.

VI. CONCLUSION

Unraveling in social networks can happen due to losing connections between different network partitions. It is to the benefit of the social network provider to encourage the nodes connecting network partitions to stay in the network. The anchored *k*-core algorithm of [9] introduces a mechanism to model unraveling in social networks. Also, it provides an approach to tackle this problem by locating the valuable nodes, called anchors, and rewarding them to stay active in the network.

An exact algorithm was proposed in [9] for $k = 2$ that guarantees finding the most valuable nodes which save the most number of vertices by staying engaged. In this work, we proposed an efficient approach for implementation of the anchored 2-core algorithm of [9]. Our goal was to present an efficient approach that could process large datasets on a single consumer-grade machine in a reasonable time. We ran our implementation on a set of large graphs with millions of connections. The results proved that our approach is fast and despite the complexity of the algorithm, the execution time was in order of minutes even for the larger circuits with millions of connections for budget 10 or less. Increasing the budget will increase the execution time as it adds to the iteration cycles.

The results show that assigning a few budgets can save significant number of nodes in the network. These networks did not have examples of long chains with smaller degrees to prove the significant impact that this approach could have on saving the social network from falling apart. The future work will be studying the anchored 3-core problem and investigating the possible approximation approaches for solving the 3-core problem. This problem cannot be solved in polynomial time for k>2; however, heuristics can be used to find optimized solutions.

Source Code: https://github.com/btootoonchi/Anchored_2-Core/

REFERENCES

[1] M. Burke, C. Marlow, T. Lento, "Feed me: motivating newcomer contribution in social network sites," In CHI, 2009.

[2] S. N. Dorogovtsev, A. V. Goltsev, and J. Mendes, "K-core organization of complex networks," Physical review letters, Vol 96, No. 4, 2006.

[3] A. V. Goltsev, S. N. Dorogovtsev, and J. Mendes, "k-core (bootstrap) percolation on complex networks: Critical phenomena and nonlocal effects," Physical Review E, Vol. 73, No. 5, 2006.

[4] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," In Proceedings of the VLDB Endowment, Vol. 9, No. 1, pages 13-23, 2015.

[5] F. Bonchi, F. Gullo, A. Kaltenbrunner, andY. Volkovich, "Core decomposition of uncertain graphs," In Proceedings of the 20th ACM international conference on Knowledge discovery and data mining, pages 1316-1325, 2014.

[6] J. I. Alvarez-Hamelin, L. Dall´Asta, A. Barrat, and A. Vespignani, "K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases," Networks and Heterogeneous Media, Vol. 3, No. 2, pages 371-393, 2008.

[7] S. Wuchty, and E. Almaas, "Evolutionary cores of domain co-occurrence networks," BMC Evolutionary Biology, Vol. 5, No.4, 2005.

[8] L. Antiqueira, O. N. Oliveira, L. da Fontoura Costa, and M. d. G. V. Nunes, "A complex network approach to text summarization," Information Sciences, Vol. 179, No. 5, pages 584-599, 2009.

[9] K. Bhawalkar, J. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma, "Preventing Unraveling in Social Networks: The Anchored k-Core Problem," Automata, Languages, and Programming (ICALP), Vol. 7392, pages 440-451, 2012.

[10] Allan Bickle , "The k-Cores of a Graph," PhD dissertation, Western Michigan University, 2010.

[11] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," In Proceedings of the 13th International World Wide Web Conference (WWW), pages 595-601, 2004.

[12] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks." In Proceedings of the 20th International World Wide Web Conference (WWW), pages 587-596, 2011.

[13] P. Boldi and S. Vigna, "WebGraph Framework", [Online]. Available: http://webgraph.di.unimi.it/. [Accessed: 05-Jan-2017].

[14] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks," Advances in Data Analysis and Classification, Vol. 5, No. 2, pages 129-145, 2011.

[15] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, "The Connectivity Server: Fast access to linkage information on the Web," In Proceedings of the 7th International World Wide Web Conference (WWW), pages 469–477, 1998.

[16] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener, "The LINK database: Fast access to graphs of the Web," Research Report 175, Compaq Systems Research Center, 2001.

[17] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, "WebBase: A repository of Web pages," In Proceedings of the 9th International World Wide Web Conference (WWW), Pages 277-293, 2000.

[18] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed Web crawler," Software: Practice & Experience, Vol. 34, No. 8, pages 711-726, 2004.

[19] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: Scalability and fault-tolerance issues," In Poster Proceedings of the 11th International World Wide Web Conference (WWW), 2002.

[20] R. Sedgewick and K. Wayne, "Algorithms, 4th Edition," Section 3.4, [Online]. Available: http://algs4.cs.princeton.edu/34hash/, [Accessed: 05-Jan-2017].

[21] S. Chen, R. Wei, D. Popova, and A. Thomo, "Efficient Computation of Importance Based Communities in Web-Scale Networks Using a Single Machine," In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pages 1553-1562, 2016.

[22] M. Simpson, V. Srinivasan, and A. Thomo, "Efficient Computation of Feedback Arc Set at Web-Scale," In Proceedings of the VLDB Endowment, Vol 10, No. 3, pages 133-144, 2016.

[23] M. Simpson, V. Srinivasan, and A. Thomo, "Clearing Contamination in Large Networks," IEEE Transactions on Knowledge and Data Engineering, Vol 28, No. 6, pages 1435-1448, 2016.

[24] R. Chitnis, F. V. Fomin, and P. A. Golovach, "Preventing unraveling in social networks gets harder," In Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, pages 1085-1091, 2013.

[25] R. Chitnis, F.V. Fomin, and P. A. Golovach, "Parameterized complexity of the anchored k-core problem for directed graphs," Information and Computation Journal, Vol. 247, No. C, pages 11-22, 2016.

[26] D. Garcia, P. Mavrodiev, and F. Schweitzer, "Social resilience in online communities: the autopsy of Friendster," In Proceedings of the first ACM conference on Online social networks, pages 39-50, 2013.

[27] S. Wu, A. D. Sarma, A. Fabrikant, S. Lattanzi, and A. Tomkins, "Arrival and departure dynamics in social networks," In Proceedings of the 6th ACM international conference on Web search and data mining, pages 233-242, 2013.

[28] M. Mitzenmacher, and V. Nathan, "Hardness of peeling with stashes," Information Processing Letters, Vol. 116, No. 11, pages 682-688, 2016.