Efficient Implementation of Anchored 2-core Algorithm

by

Babak Tootoonchi
B.Sc., Amirkabr University of Technology (Tehran Polytechnic), 1999

A Project Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisory Committee

Efficient Implementation of Anchored 2-core Algorithm

by

Babak Tootoonchi
B.Sc., Amirkabr University of Technology (Tehran Polytechnic), 1999

Supervisory Committee

Dr. Alex Thomo, (Department of Computer Science)
Supervisor

Dr. Venkatesh Srinivasan, (Department of Computer Science)
Departmental Member

Abstract

Supervisory Committee
Dr. Alex Thomo, Supervisor, (Department of Computer Science)
Supervisor
Dr. Venkatesh Srinivasan, (Department of Computer Science)
Departmental Member

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 assuming 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 on a set of real world network data that includes very large networks 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 larger datasets that proves the efficiency of our implementation.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my gratitude and appreciation to my supervisor Dr. Alex Thomo for his support, patience and guidance throughout this work. I would also like to thank my committee member Dr. Sirinivasan for his time and his constructive comments during my examination.

Finally, I would like to thank my wife for her endless support and love and my mother and sister for their encouragement and support.

Dedication

I would like to dedicate this work to my wife and my mother.

1. Introduction

One of the best ideas since the development of internet is the development of social networks on world wide web. A social network is a website that allows people to create profiles and connect with others including family, friends, or wider groups of people who have interest in a certain activity, art, etc. People can use social network websites to share photos, videos, music and other information with either a select group of friends or with public. In the past few years, social networks have played an important role in connecting people around the world and have turned to one of the most powerful media.

Social networks such as Facebook, Twitter and LinkedIn are great ways to keep in touch with friends and family around the world as well as making new connections with people based on similar interests or professions. 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 in to 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 in the social network and there is a higher chance that they stay in the social network 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 in social networks is one of the main goals for social network websites and is the key to keep them alive, growing and profitable.

Of course, now the question is how to design a social network and what strategies to use to maximize the participation and engagement of its users. One solution is to encourage

relationships among members and increase interpersonal attractions among members. Another solution to increase user engagement is to emphasize the community as an entity that have common interests, ethnicity, history, norms, or competition with other groups [2-4].

For example, Facebook, the social networking website that was launched in 2004, has become the world's largest and most popular social network website, with more than 1.79 billion monthly active users [5]. Facebook features an application platform which allows developers to implement different kinds of applications and integrate them into the website [5]. Games are among the most popular applications, attracting many users every day. The popularity of Facebook games provides the opportunity to have active users, and it encourages more users to be involved in the network.

Users in social networks can behave differently. While some may be very engaged and login and contribute daily, others may be rarely active and some may eventually leave the social network. Users, who are engaged, very often contribute to a social network by sharing contents, joining different groups, signing up for new features, and so on. An individual is more likely to be engaged in a social network 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 [6-9,14].

Assume in a social network, 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 meaning 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 www networks, protein structures, information retrieval, text summarization, etc. [10-12]. In Chapter 2, we discuss k-core decomposition in more detail and present an efficient *k*-core decomposition approach that we have used in our implementations for this work [8].

As described in the *k*-core model of the social networks, user drop outs can cause network to be partitioned and in similar effects over time can cause a social network's life to come to an end. In [13], authors have introduced a method to prevent unraveling in social networks by locating the most valuable nodes, called anchors, and rewarding them to encourage 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. Chapter 3 discusses the anchored *k*-core algorithm [13] in more detail.

The anchored *k*-core problem has been studied both empirically and theoretically by a few other works. The authors of [28] showed that the anchored *k*-core problem is W[1]-hard parameterized by the size of the core *p*. This improves the result of [13] which

shows W[2]-hardness parameterized by *b*. The work in [29] 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 [30, 31]. These works have studied different factors that can contribute to the resilience of the social networks. The authors of [32] 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 [13] is a complex algorithm that requires a lot of computation cycles. The authors of [13] 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 [8], to compute the *k*-core decomposition at every iteration. For the graph database in our efficient implementation, we used Webgraph, 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 [25-27].

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 the massive graphs with billions of edges, the elapsed time was in order of minutes. The proposed implementation approach and the experimental results can be found in Chapters 4 and 5 respectively.

The remainder of this dissertation is structured as follows.

Chapter 2 provides the basic concepts in graph theory that are required to understand the discussions in this thesis. Moreover, it describes the *k*-core algorithms in their general form.

Chapter 3 discusses unraveling in social networks and introduces the anchored k-core approach to prevent it as was proposed in [13].

Chapter 4 shows the details of our efficient implementation of the anchored 2-core algorithm [13].

Chapter 5 shows the experimental results of applying our efficient approach on a set of large graphs on a single machine.

Chapter 6 concludes the discussion and provides directions for possible future work.

2. K-Core Decomposition

The *k*-core decomposition of large networks provides the means to study properties of large networks, and in particular, it helps to measure the centrality and connectedness in large networks. In this chapter, we first introduce the basic definitions and describe the *k*-core decomposition of graphs and its properties. Then we introduce an algorithm that computes the *k*-core decomposition of a graph in polynomial time.

We also show a general form of *k*-core algorithms. In chapter 4, an efficient implementation of this algorithm [8] will be discussed that was used in this project. This efficiency is particularly useful for calculating the *k*-core decomposition of networks with billions of edges on a single machine. We show how we leveraged this implementation to implement an algorithm that solves the anchored *k*-core problem.

2.1 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 thesis, wherever we refer to a graph we mean an undirected graph. The number of vertices is shown by $|V| = n$ and the number of edges will be represented by $|E| = m$.

Definition 2.1 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*.

Definition 2.2 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 $\delta(G) = \min\{d_G(v) : v \in V\}$.

Definition 2.3 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 2.4 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 the maximal induced subgraph of *G*. 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 2.4, 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, $C_k(G) = G$, if $k \leq 0$ or $k \leq \delta(G)$. 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*.

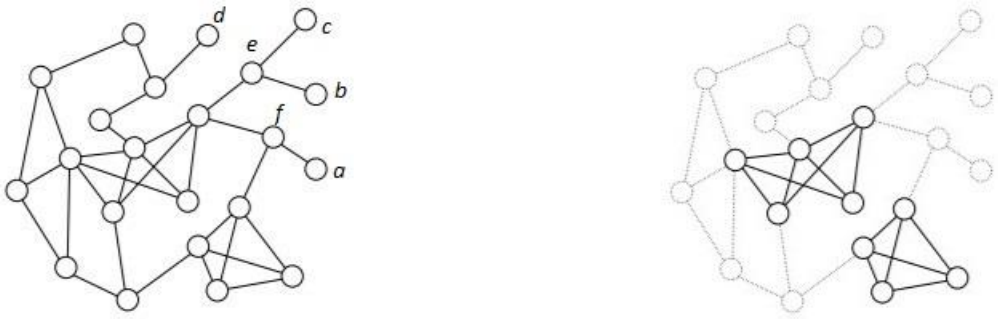
Definition 2.5 For a given vertex $v \in G$, the *core* number or coreness of *v* is the largest value for *k* such that $v \in C_k(G)$.

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, Equation 2.1 shows the generalization of this property.

$$C_{\Delta(G)}(G) \subseteq C_{\Delta(G)-1}(G) \subseteq \cdots \subseteq C_{\delta(G)}(G) \tag{2.1}$$

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 Figure 1(a). To extract the 3-core subgraph of *G* we need to eliminate nodes that have a degree less than 3. The result is illustrated in Figure 1(b).

This graph of Figure 1(b) is composed of two separate partitions. In each partition, every node has at least *k* neighbors. Note that nodes *e* and *f* have degree equal to 3, but they are not in the 3-core graph. This is because to achieve a 3-core graph, nodes with degrees less than 3 are iteratively removed. In the next section, we will discuss the *k*-core algorithms in more detail. In this case, the degrees of the nodes *a*, *b*, *c*, and *d* are 1, so they are eliminated. As a result, in the next iteration nodes *e* and *f* will have degrees equal to 2 and hence they get removed from the 3-core graph.



(a) The original graph *G* (b) The 3-core of *G*

Figure 1. *k*-core for *k* = 3 for an example graph *G*

2.2 K-Core Algorithm

As mentioned in the previous chapter, an important application of *k*-core graphs can be in modeling social networks. We assume that all individuals with at least *k* friends who are actively participating will remain engaged and the individuals who have less than *k* friends will drop out from the social network. Spreading this effect to their connections for whom the number of friends drops below *k* will cause them to leave the social network. These iterative withdrawals continue until the remaining subgraph of users shapes a *k*-core subgraph of all engaged individuals. This behavior is modeled by the *k*-core algorithm that is described in Algorithm 2.1.

In the next chapter, we show how finding potentially important engaged individuals among nodes that have degrees less than *k* in a social network and encouraging them to stay in the social network will benefit the overall size and usage of the network. These anchored nodes help to increase connectivity and to grow the network.

Algorithm 2.1 The k-core algorithm:

1. Given graph $G = (V, E)$

2. For a given value of k and the degree matrix of D for graph G.

3. Find all vertices $v \in V$ *with degrees* $d_G(v) < k$ *using the degree matrix D.*

4. For each vertex v found in Step 3 do:

4.1 Remove the vertex v from G.

4.2 Decrease the degree of all the neighbors of v by one.

5. If there is a node with $d_G(v) < k$ *go back to Step 3.*

6. The remaining subgraph represents the k-core of G.

3. Anchored *k*-Core Approach

In this chapter, we present an algorithm that was proposed in [13] to solve the unraveling that can happen in social networks because of a cascade of drop outs.

3.1 Anchored *k*-Core Problem

In our *k*-core model of social networks as discussed in the previous chapter, if a user leaves the network, not only will the number of nodes decrease by one because of his or her absence, but also the second level connections between his or her friends will be torn down. Also, the friends of the leaving individual will have one less connection that may discourage them to stay engaged in the network and provide incentives for them to leave perhaps for a newer product or another network.

In our *k*-core model, we 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 bellow *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 drop out 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. We call these nodes *anchors* and in this chapter, we discuss how it is possible to locate the anchors in a network in the most efficient way using the algorithms proposed in [13].

Before discussing the details of the solutions that were presented in [13], we need to formally define the *anchored k-core* problem. 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.

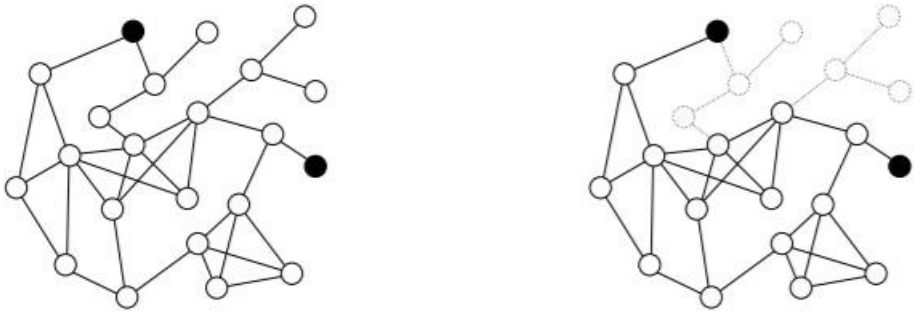
The anchored *k*-core problem is to find a set S of at most *b* nodes among all possible subsets of size *b*, where $S \subseteq V$ and $|S| \leq b$, such that keeping those nodes while calculating the *k*-core graph regardless of their degree results in a *k*-core with the maximum possible number of 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 assign them budget to maximize the size of network based on the *k*-core model. The budget can be any type of benefit including offering rewards, waving premiums, or giving some sort of rebates or points.

In Section 3.2 we will show that the anchored *k*-core problem for $k = 2$ can be solved in polynomial time. For k ≥ 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 ≥ 3, the problem is *W[2]-hard* with respect to the budget parameter *b* [13].

3.2 Anchored *k*-Core Algorithm

As described in Section 2.1, the *k*-core of a graph is the maximal induced subgraph with the minimum degree of k. It is easily shown that this subgraph is unique, the cores of a graph are nested, and that the *k*-core can be found by iteratively deleting vertices with degrees less than *k*. A deletion sequence is formed by iteratively deleting vertices with the smallest degrees.

The anchored *k*-core algorithm finds the anchors and assigns budget to them. It also iteratively removes nodes with degrees less than *k* unless they are anchored. Hence, the anchored *k*-core is computed similar to the *k*-core, but the anchored vertices are never deleted. *Non-anchored* vertices with degrees less than *k* are deleted iteratively, in any order.



(a) An assignment of 2 anchors to *G*

(b) The vertices saved in an anchored 3-core of *G* with budget 2

Figure 2. Anchored 3-core with budget 2 on *G* [13]

Figure 2 shows an example of an anchored *k*-core algorithm applied to a graph *G* given *k*=3 and budget *b*=2. Figure 2(a) shows the location of the anchors that were identified by the algorithm on graph *G*. Figure 2(b) illustrates the anchored *k*-core subgraph of *G*

after applying the algorithm. In Figure 2(b), note that the anchored *k*-core subgraph contains 16 vertices (out of 22 vertices in graph *G*), which is much more than the 9 vertices remained in the *k*-core subgraph of Figure 2(b). The significant increase in the number of vertices is a result of keeping the two valuable anchors in the graph regardless of their degrees by assigning them budget *b*.

Let *G* be a graph for which we would like to compute the anchored *k*-core with budget *b*. First, we built a new graph called $RemoveCore(G)$ from *G* as described in Definition 3.1.

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 $RemoveCore(G)$.

An assignment of anchors has size *m* in $RemoveCore(G)$ if and only if it has size *m* in *G* [13]. We also assume that an anchor is placed on every vertex $v^* \in C_k$ for free (without using any budget). With this assumption, each $v^* \in C_k$ will remain in the graph after applying the *k*-core algorithm. Therefore, we can assume these vertices are already considered as anchors, and the internal structure of C_k has no impact on the anchored *k*-core of *G*. Thus, finding a solution for the graph $RemoveCore(G)$ will lead to finding a solution for the anchored *k*-core problem on *G*.

In the next section, an anchored 2-core algorithm will be introduced [13]. We will show how using the $RemoveCore(G)$ graph will significantly reduce the complexity of the problem for $k = 2$.

3.3 Anchored *2*-Core Approach

For all the discussions in this section, assume that $G = (V, E)$ is an undirected graph with size $|V(G)| = n$. Also, assume $b \in \mathbb{Z}$ and $k \in \mathbb{N}, k \leq \Delta(G)$, are the budget and threshold parameters respectively.

3.3.1 RemoveCore Subgraph

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 set of rooted and non-rooted trees are denoted by $\mathcal{R}$ and S respectively [13].

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.

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.

3.3.2 Anchored 2-Core Algorithm

In this section, we introduce an algorithm that solves the anchored 2-core problem as described in [13]. The proposed approach is greedy and exact in that it guarantees to find an anchored 2-core of the maximum size.

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, we find $v_3, v_4 \in S$, where S is the set of non-rooted trees as defined in Definition 3.2, 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 on 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 *k*-core and the $RemoveCore(G)$ is calculated again. This process repeats until the budget is completely used $b = 0$. The anchored 2-core algorithm is shown in Algorithm 3.1.

Algorithm 3.1 An exact algorithm for the anchored 2-core problem [13]

1. $G \leftarrow RemoveCore(G)$ *// G is now a forest*

2. $S \leftarrow \emptyset$

3. while $b > 0$ *do*

3.1 partition G *into sets* $\mathcal{R}$ *and* $\mathcal{S}$

3.2 $v_1 \leftarrow$ *the farthest vertex from* r

3.3 $v_2 \leftarrow$ *the farthest vertex from* r *after the* $r - v_1$ *is contracted*

3.4 $(v_3, v_4) \leftarrow$ *a pair of vertices on the endpoints of longest path across trees in* $\mathcal{S}$

3.5 if $C_{\mathcal{R}}(v_1) + C_{\mathcal{R}}(v_2) > C_S(v_3, v_4)$ *then*

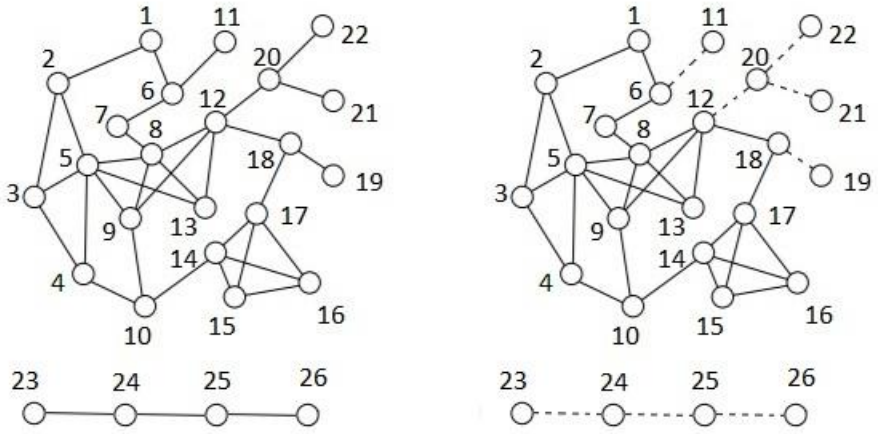
a. $S \leftarrow S \cup v_1, b \leftarrow b - 1$ *// place an anchor on* v_1

3.6 else

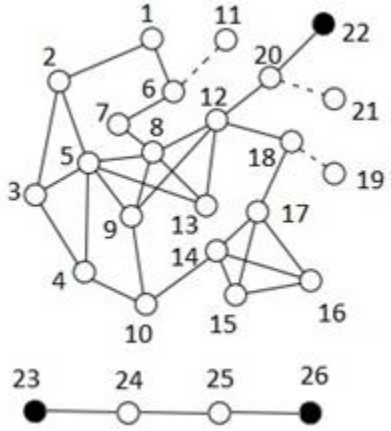
b. $S \leftarrow S \cup \{v_3, v_4\}, b \leftarrow b - 2$ *// place anchors on* v_3 *and* v_4

3.7 $G \leftarrow RemoveCore(G)$ *// G modified due to newly anchored vertices*

Example 3.1 Figure 3 shows an example of the anchored *2*-core algorithm applied to a graph *G* of Figure 3(a) given budget *b*=3. Figure 3(b) shows the 2-core subgraph of graph *G* after applying the *k*-core algorithm on graph *G* for *k*=2. The RemoveCore of *G* contains three rooted trees and one non-rooted tree as shown in dotted lines in Figure 3(b). Nodes 6, 12 and 18 are roots of the rooted trees.



a) Graph G with 26 nodes b) The 2-core graph



c) Anchor placement for budget 3

Figure 3 Anchored 2-core example with budget 3.

In the first iteration of the anchored 2-core algorithm, the farthest vertex from *r* can be either of vertices 21 or 22. Assume that vertex 22 is selected as v_1, the second farthest node can be any of the vertices 11, 21 or 19. For example, if the algorithm picks vertex 11 as the second farthest, v_2, the total number of nodes saved by v_1 and v_2 will be 3. On the other hand, assigning two anchors to the endpoints of the longest path on a non-rooted tree, nodes 23 and 26, will save 4 nodes. Hence, two anchors are placed on nodes 23 and

26 and the nodes on the non-rooted tree are added to the 2-core graph. In the next iteration, the budget is reduced to 1 and the RemoveCore is computed again with the new 2-core. The new RemoveCore contains the three rooted trees and no more non-rooted trees. The two vertices, which have the longest path to the 2-core graph, are nodes 21 and 22. Since they have the same distance, one will be picked by algorithm and an anchor will be placed on it, for example in Figure 3(c) node 22 is selected. Figure 3(c) illustrates the anchor placement for the three anchors in of *G*. In Figure 3(c), note that the anchored *k*-core subgraph contains 23 vertices (out of 26 vertices in graph *G*), which is much more than the 17 vertices remained in the *2*-core subgraph of Figure 3(b). The significant increase in the number of vertices is a result of keeping the three valuable anchors in the graph regardless of their degrees by assigning them budget *b*.

4. Implementation

The first part of this chapter is devoted to introduction of *WebGraph*, a framework for large graph datasets, which was utilized in our implementations. In the next section, an efficient implementation of the *k*-core algorithm [8] will be introduced that was used as part of our implementations. Finally, the proposed approach for implementing the anchored *2*-core algorithm of [13] and its implementation results will be presented.

4.1 WebGraph Framework

WebGraph is a framework for graph based databases that was designed to facilitate studying of web graphs. Webgraph provides efficient usage of memory by utilizing compression techniques. It also provides fast API for randomly accessing graph nodes and vertices [15,16]. 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.

Studying web graphs is often difficult due to their large size. The WebGraph framework consists of various algorithms and tools that allow storing and manipulating large graphs [15]. The framework exploits modern compressing techniques such as gap compression [19], referentiation [20] and intervalisation to manage very large graphs. WebGraph uses some lazy techniques to access compressed graphs without decompression unless it is necessary.

The WebGraph framework also contains data sets for very large graphs with billions of links [17] which were either gathered from public sources [21] or obtained with UbiCrawler [22,23]. WebGraph was implemented in different programming languages such as, Java, Python, C++, and Matlab. We use the Java distribution of WebGraph under GNU public license. The WebGraph Java library includes a few jar files that can be easily installed and used.

The WebGraph framework contains different classes; among those, some important classes are as follows:

- *ImmutableGraph* is an abstract class representing an immutable graph.
- *BVGraph* allows to store and access web graphs in a compressed form.
- *ASCIIGraph* is used to store the graph in a human-readable ASCII format.
- *ArcLabelledImmutableGraph* is an abstract implementation of a graph with labeled arcs.
- *Transform* returns the transformed version of an immutable graph. We can use the transpose method of this class if we want to create the transpose graph.

Besides the data sets that are available in the WebGraph website [17], there is a class in the framework that creates sample web graph datasets. The class name is *Text2ASCII*. The steps for creating a web graph are as follows:

1. Create a file in the *ASCIIGraph* format.
2. Convert the file format from *ASCIIGraph* to *BVGraph*

4.2 Project Design

In this project, we have implemented the anchored 2-core algorithm that was described in Algorithm 3.1. The implementation consists of three different layers. These three layers are as follows.

- **Graph Database:** The implemented algorithm deals with graph databases. For our graph database, we used the WebGraph framework that was introduced in Section 4.1. As discussed in the previous section, WebGraph provides a memory efficient approach and fast APIs, which make it easy for us to deal with larger datasets.
- **Kcore Algorithm:** In steps 1 and 3.7 of Algorithm 3.1, the RemoveCore of the given graph has to be computed. As described in section 3.2.1, to find the RemoveCore subgraph of a graph G, first the k-core of graph G needs to be extracted. In this work, the Batagelj and Zaversnik Algorithm was used to calculate the k-core at each iteration of Algorithm 3.1. The Batagelj and Zaversnik algorithm will be introduced in Section 4.2.2.
- **Anchored *k*-core Algorithm:** The main contribution of this work is an efficient implementation of the anchored 2-core algorithm of [13] as described in Algorithm 3.1. The details of our implementation will be presented in Section 4.2.3.
- **Utils:** A third party utility package [24] was used in our implementation of the anchored *k*-core algorithm. We used a data structure called *SeparateChainingHash* as our hash storage that utilizes a linked list class called

SequentialSearchST. *SequentialSearchST* is an unordered list of key-value pairs that uses sequential search.

4.2.1 Graph Layer

Graph layer is an interface for WebGraph. There are two java files in this layer. One of them is an interface, “Graph”, to create some abstract methods like *getNeighbors* and *vertexIteretor*. The other file has the *GraphWebgraph* class which implements the *Graph* interface. This class is used to create and access immutable graphs, that is, graphs that are computed once for all, stored conveniently, and then accessed repeatedly. Moreover, immutable graphs are usually very large which means such graphs may not fit into a central memory.

There are different methods to load immutable graphs such as, *load(), loadMapped()* and *loadOffline()* which are supported by the *GraphWebgraph* class. The *load()* method is the standard way to load an immutable graph from disk into memory. The *loadMapped()* method creates a new Immutable graph by memory-mapping a graph file. Metadata could be read from disk, but the graph will be accessed by memory mapping, and the class should guarantee that random access is possible [17].

In *loadOffline*, the immutable graph should be set up, and possibly some metadata could be read from disk, but no actual data is loaded into memory. The class should guarantee that offline sequential access is still possible [17]. The *GraphWebgraph* class has some methods to provide some functionalities to work with the loaded *webGraph*. These methods are *maxDegree(), size(), getNeighbors(), outdegree(),* and *vertrxItrator()*.

4.2.2 Batagelj and Zaversnik Algorithm

As part of the implementation of the anchored *k*-core algorithm, Algorithm 3.1, the *k*-core of the given graph needs to be calculated at each iteration. *k*-core graph is a subgraph *C* of an undirected graph in which the minimum degree for every vertex is *k*, i.e. $\delta(C) \geq k$. A more detailed explanation can be found in Definition 2.4.

Different variations of *k*-core algorithms have been proposed so far [6,8,9,14, 18]. For this work, we used the efficient implementation of the Batagelj and Zaversnik (BZ) algorithm [18] that was proposed in [8]. This implementation is referred to as WG_BZ and uses WebGraph as the graph database framework.

The BZ algorithm finds the *k*-core decomposition by recursively deleting the vertices with degrees less than *k* starting from the smallest degree nodes. The deletions are not physically done on the graph, but instead a couple of sets and an array are used to store the state of the core subgraphs after deletions [18]. The main challenge with efficient implementation of the BZ algorithm is how to implement the sets of vertices *D*. The authors in [8] experimented with different hashing approaches such as Java standard library, Google's Guava collections library, Trove high-performance collections for Java, and the modern Cuckoo-hashing. None of these implementations of hash maps seemed to be fast enough for handling large datasets in a reasonable time. In contrast, the WG_BZ algorithm, which was proposed in [8] and is a modification of the BZ algorithm, seems to be significantly faster and could process large datasets in order of 2-5 minutes as opposed to many days using BZ.

The main idea in [8] is to flatten the set of vertices of BZ into a few arrays. There are five arrays in the WG_BZ implementation of the *k*-core algorithm.

- Array *D* contains the indices of vertices sorted by their degrees.
- Array d holds the degree of each vertex.
- Array *p* stores the position of each vertex in *D*.
- Array b has size $\Delta(G)$ and stores the indices at which blocks of vertices with the same degrees start in *D*.

Algorithm WG_BZ of [8] is shown in Algorithm 4.1. Arrays *d*, *b*, *D*, and *p* are initialized first. Array *d* is initialized by the degrees of each vertex, and array *D* is filled by indices 1 to *n*. Then, *d* is sorted with *D* rows tagging along so that *D* contains the sorted list of vertices from the lowest degree to the highest degree. Finally, *p* is populated by setting indices for position of each vertex in *D*, i.e. $P\big[D[i]\big] = i,\ i \in \{1, \dots, n\}$.

Then the algorithm iterates over all the entries in *D*, which are vertices with indices from 1 to *n*. The coreness of each vertex *v* is equivalent to its degree $d[v]$. Iteratively, vertex *v* is selected (assuming it was logically deleted). Then, for each neighbor *u* of *v* with a higher degree, the degree of *u* is decreased by one as shown in line 13. Decreasing the degree of *u* leads to it falling in another block in *D*. So, before that *u* is moved one block up in *D* since its degree will be one less. This is achieved in constant time by swapping *u* with the first vertex, *w*, in the same block in *D* (lines 10-11) and then moving the block head index one position up in b to move *u* to the previous block as shown in line 13. Thus, *u* becomes the last element of the previous block.

Algorithm 4.1 k-core computation using a flat array *D* [8]

1. *function k-cores* (*Graph G*)
2. *initialize* (d, b, D, p, G)
3. *for all* $i \leftarrow 1$ *to* n *do*
4. $v \leftarrow D[i]$
5. *for all* $u \in N_G(v)$ *do*
6. *if* $d[u] > d[v]$ *then*
7. $du \leftarrow d[u], pu \leftarrow p[u]$
8. $pw \leftarrow b[du], w \leftarrow D[pw]$
9. *if* $u != w$ *then*
10. $D[pu] \leftarrow w, D[pw] \leftarrow u$
11. $p[u] \leftarrow pw, p[w] \leftarrow pu$
12. *end if*
13. $b[du] + +, d[u] - -$
14. *end if*
15. *end for*
16. *end for*
17. *return d*

For example, consider graph *G* as shown in Figure 4 with vertices that are labeled by numbers 1 to 22. Table 2 illustrates the initialization of arrays *d*, *b*, *D* and *p* for graph *G*. For instance, vertex 1 has a degree of 2, so $d[1] = 2$. Vertex 1 is in the fifth row of array *D* so its position is 5, i.e. $p[1] = 5$.

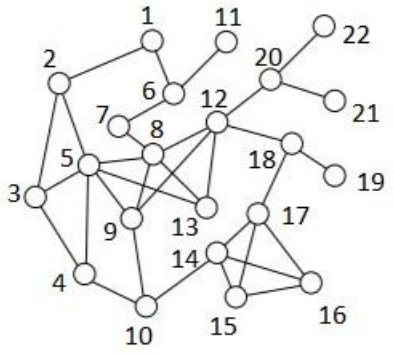


Figure 4. Graph *G* with 22 nodes

Array *D* shows the vertices in order of their degrees. *D* is partitioned into sets of nodes with the same degrees which are referenced by indices in array *b*. For example, nodes with degree 2 start at index 5 and the block of vertices with degree 3 starts at index 7 hence $b[2] = 5$ and $b[3] = 7$.

Table 1. Initialization of the arrays in WG_BZ algorithm for the graph of Fig. 3.

index	D	B	D	P
1	2	1	11	5
2	3	5	19	7
3	3	7	21	8
4	3	17	22	9
5	6	20	1	22
6	3	22	7	10
7	2		2	6
8	5		3	20
9	4		4	17
10	3		6	11
11	1		10	1
12	5		13	21
13	3		15	12
14	4		16	18
15	3		18	13
16	3		20	14
17	4		9	19

18	3		14	15
19	1		17	2
20	3		8	16
21	1		12	3
22	1		5	4

After applying Algorithm 4.1 to graph *G*, the contents of array *D* represent the *k*-core of graph *G* for each value *k.* In other words, each bin in array *D* shows nodes that belong to the corresponding *k*-core of graph *G.*

For example, Table 2 shows the result of applying Algorithm 4.1 to graph *G* with initialization values as shown in Table 1.

Table 2. *k*-core of graph *G* after applying WG_BZ algorithm

index	**D**	**B**	**D**	**P**
1	2	1	11	13
2	3	6	19	7
3	3	14	21	8
4	3		22	9
5	6		20	22
6	3		7	10
7	2		2	6
8	5		3	20
9	4		4	17
10	3		6	11
11	1		10	1
12	5		18	21
13	3		1	15
14	4		16	18
15	3		13	16
16	3		15	14
17	4		9	19
18	3		14	12
19	1		17	2
20	3		8	5
21	1		12	3
22	1		5	4

It is shown that the maximum coreness of graph *G* is 3 which is the size of array *b*. Also, array *b* shows that the bin containing 1-core vertices starts at index 1 in *D*. The 2-core vertices start at index 6 and finally the 3-core vertices bin starts at index 14 in array *D*.

4.2.3 Efficient Implementation of Anchored *2*-core Algorithm

As discussed in Section 4.2, the main part of this project is an efficient implementation of the anchored 2-core algorithm, Algorithm 3.1. Before entering the implementation details of the algorithm, it is worthwhile to introduce a third-party utility package that was used in our implementation [24].

The *SequentialSearchST* class represents an unordered linked list of nodes that contain key-value pairs. It supports the usual *put*, *get*, *contains*, *delete*, *size*, and *is-empty* methods. It also provides a *keys* method for iterating over all keys in the list. The class also uses the convention that values cannot be null. Setting the value associated with a key to null is equivalent to deleting the key from the symbol table [24].

The *SeparateChainingHashST* class implements a symbol table with a separate-chaining hash table. It maintains an array of *SequentialSearchST* objects and implements *get()* and *put()* by computing a hash function to choose the corresponding *SequentialSearchST* list and applying *get()* and *put()* on it. As will be shown below, we used the *SeparateChainingHashST* class to implement the *SC_hash* table that we used in our implementation of the algorithm.

The following data structures were used in our implementation of Algorithm 3.1.

- Arrays *d, D, p,* and *b*, that were used in the *k*-core Algorithm 4.1.

- 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 vertex and value is the distance of vertex from the vertex at the head of the corresponding linked list.

Algorithm 4.2 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), Algorithm 4.1 is used to compute the *k*-core of graph *G*, which is effectively defined by arrays *D* and *b.* The main loop of the algorithm iteratively finds anchors and assigns budget to them as long as there is budget available.

Algorithm 4.2 Anchored 2-core Algorithm

1. *function anchored_2-cores* (*Graph G, Integer budget*)
2. $d, b, D \leftarrow$ *compute K-Core(G) // using Algorithm 4.1, fills arrays d, b and D*
3. *while* $(budget > 0)$
4. *initialize* (S, R)
5. $R \leftarrow$ *compute* $RemoveCore(G, D, b, R, 2)$

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

After initialization of *S* and *R* arrays, the RemoveCore of G is computed for $k = 2$ (line 5). The RemoveCore is found by simply copying the first bucket of vertices from *D* to *R* i.e. indices 0 to $b[k] - 1$. The RemoveCore function is shown in Algorithm 4.3.

Algorithm 4.3 RemoveCore for $k = 2$

1. *function* $RemoveCore(G, D, b, R, k)$
2. *for* $i = 0$ *to* $b[k] - 1$
3. $R \leftarrow D[i]$
4. *end for*
5. *end*

After finding the RemoveCore graph, *R*, if *R* is non-empty, a SeparateChainingHash will be initialized based on the vertices in *R* in that a list of pointers is created in which each pointer corresponds to a vertex in *R* and points to the head of an empty linked list.

The $checkRootedTree(u)$ method at line 9 fills up the SeparateChainingHash table and the *S* array. Algorithm 4.4 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 *k*-core) is found, then the tree is marked as rooted and the distance of root will be set to 0 (line 14). At this point another recursive is called to update the distances of all nodes to be from the root rather than *u* (line 16). This second function is called $computeDistance$ and is shown in Algorithm 4.5. 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.

Algorithm 4.4 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

Algorithm 4.5 which computes distances to root on a rooted tree is shown below.

Algorithm 4.5 Find the longest path from root on a rooted tree

1. function $computeDistance(u, i, distance)$

2. mark u as visited

3. for each neighbor v of u

4. if v *is not visited and* $v \in R$

5. $SC\_hash[i].put(v, distance + 1)$

6. $S[v] = "rooted"$

7. $computeDistance(v, i, distance + 1)$

8. end if

9. end for

10. end

In Algorithm 4.2, 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). 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 for simulating the impact of budget on the nodes, the degree of the anchored nodes is assigned to the maximum degree of the 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.

5. Experimental Results

This chapter, 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 3 shows the result of applying our anchored 2-core implementation on twelve datasets ranging from small to large networks with budget $b = 3$. 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.

Table 3. Results of our 2-core implementation with b=3

Data set	# of nodes	2-core size	# of nodes saved	Execution time (s)
data1_astrocnet	133,280	17,440	8	0.136
data2_condmatcnet	108,300	20,613	10	0.094
data3_p2pgnutella	62,586	33,816	7	0.231
data4_soc_sign_slashdot	82,144	52,103	10	0.528
data5_soc_slashdot	82,168	80,365	3	0.942
data6_amazon	403,394	390,938	13	2.305
data7_web_berk_stan	685,231	629,459	1052	12.22
data8_wiki_talk	2,394,385	622,999	11	1081.199
data9_soc-LiveJournal	4,847,571	3,784,309	14	44.162
data10_roadnet_tx	1,393,383	1,093,520	38	2.007
data11_roadnet_ca	1,971,281	1,591,795	86	4.108
data12_uk-2005-edgeList	39,459,923	35,580,606	28	547.423

Table 4 shows the result of applying the anchored 2-core algorithm on the same set of networks with budget 5. As it is shown in Table 4, adding more budget leads to saving more nodes. For example, 1663 nodes will be saved by assigning 5 anchors to the data7_web_berk_stan graph.

Table 4. 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 5. 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

The results of running the anchored 2-core algorithm with budget 10 are shown in Table 5. Adding more budget, linearly increases the iteration cycles in the algorithm which increases the execution time as illustrated in Table 5. For dataset data8_wiki_talk, the execution time is higher than data12_uk-2005-edgeList 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_wiki_talk is much smaller than data12_uk-2005-edgeList, 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 could be because of the trees with large number of branches in the RemoveCore of data8. Assigning more budgets breaks down these trees into smaller pieces and reduces the number of backtrackings. For example, the execution time becomes more comparable to data12 for $b = 10$.

Figures 5 and 6 illustrate the number of nodes saved by only a few anchors in the above datasets. As it is shown in Figure 5, 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. For example, as shown in Figure 6, for the network of graph data7_web_berk_stan, assigning budget to only 10 anchors will save 2633 nodes.

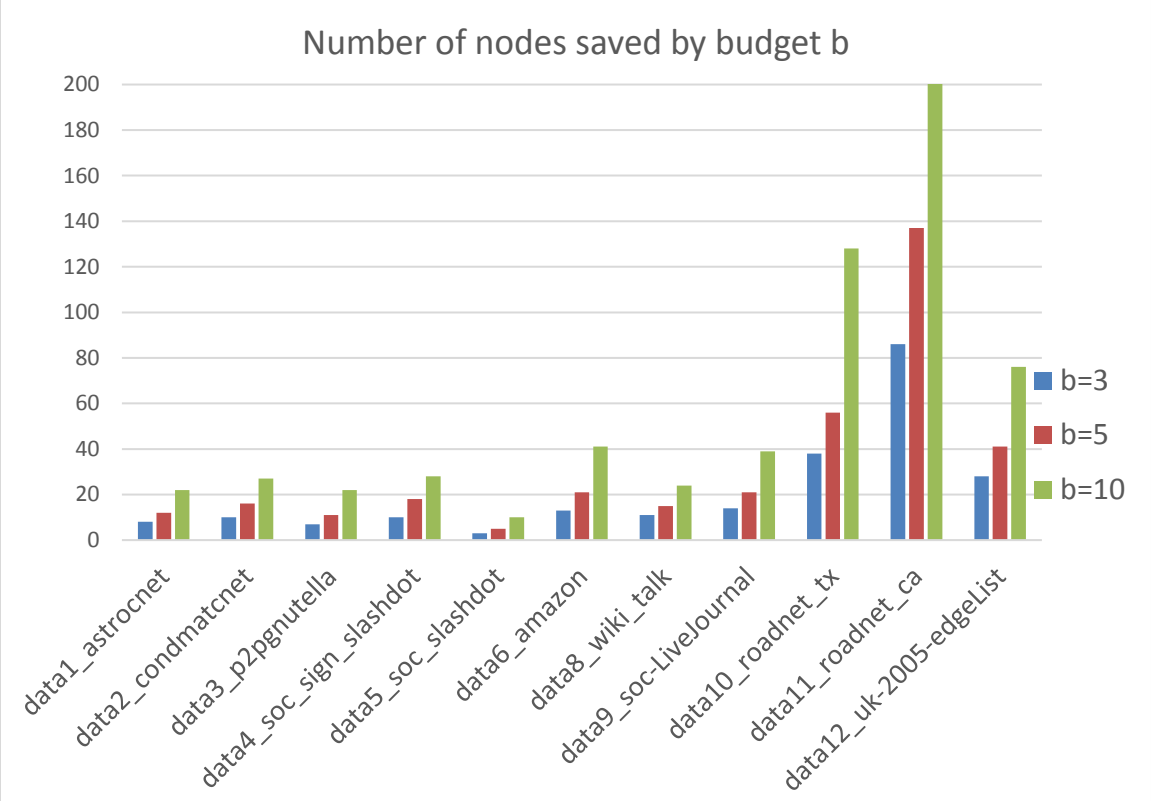


Figure 5. Number of nodes saved by assigning budget *b* for smaller datasets

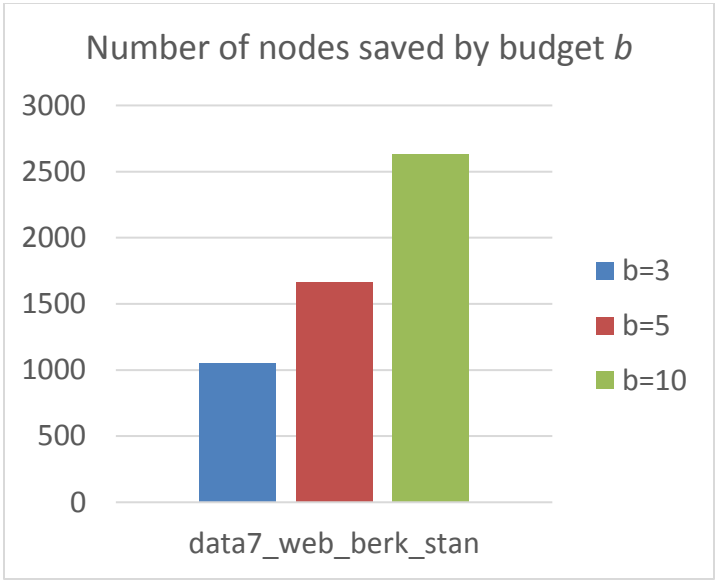


Figure 6. Number of nodes saved by assigning budget *b* for larger datasets

To illustrate the execution times, the results were divided into three datasets of similar ranges for better viewing. The charts are shown in Figures 7, 8, and 9. As it is shown in Figure 7, for graphs of size 50k to 150k the execution time is in order of a few seconds. For the larger graphs shown in Figure 8 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. 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.

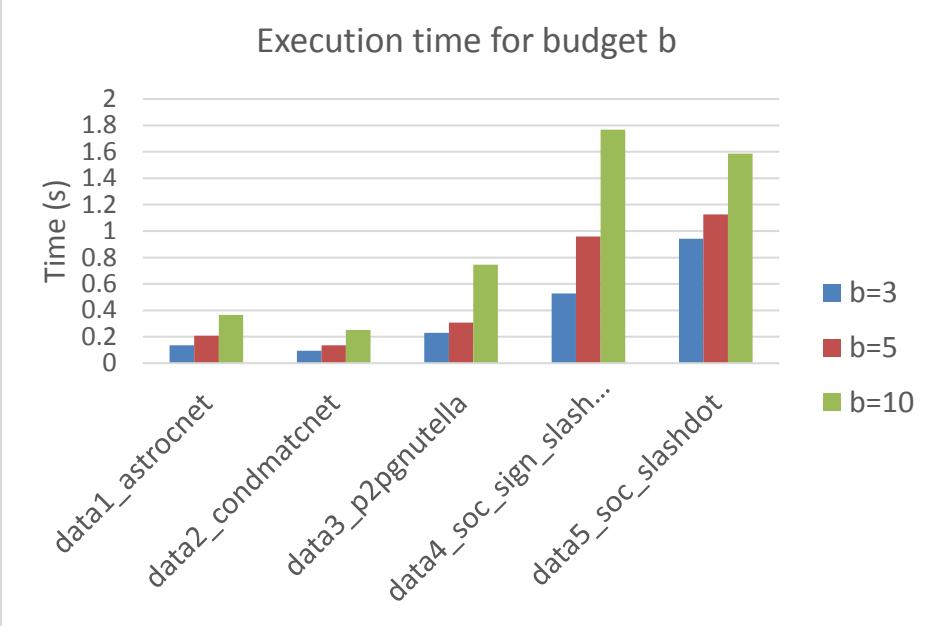


Figure 7. Execution times of Algorithm 4.2 on dataset 1

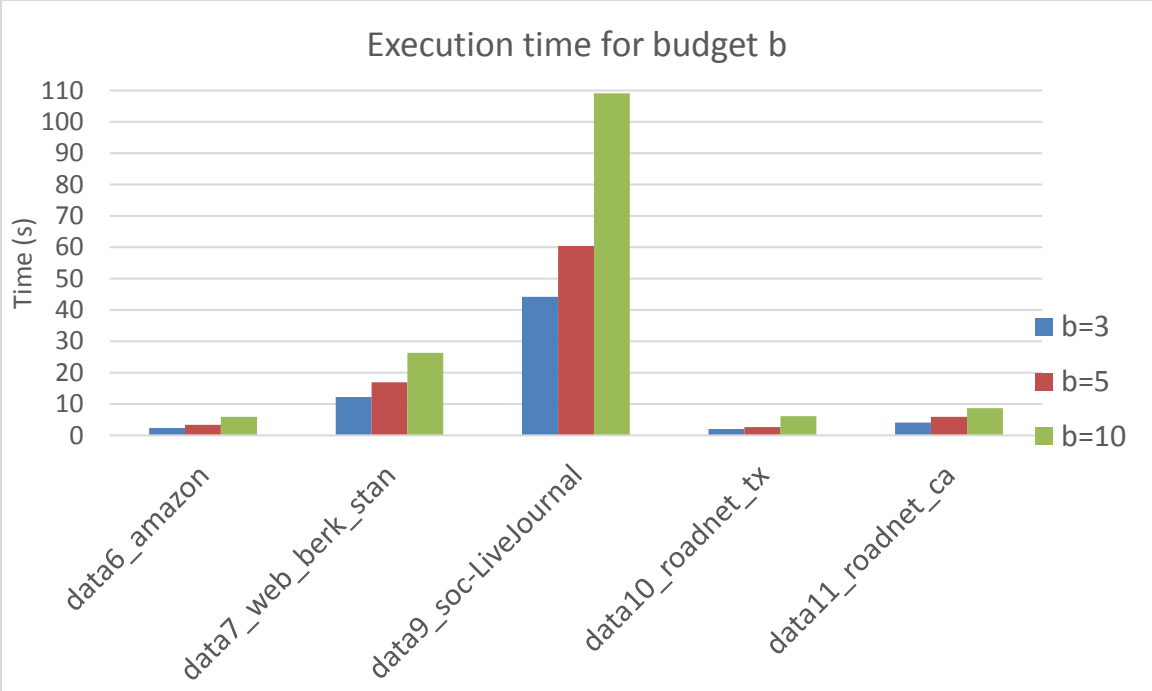


Figure 8. Execution times of Algorithm 4.2 on dataset 2

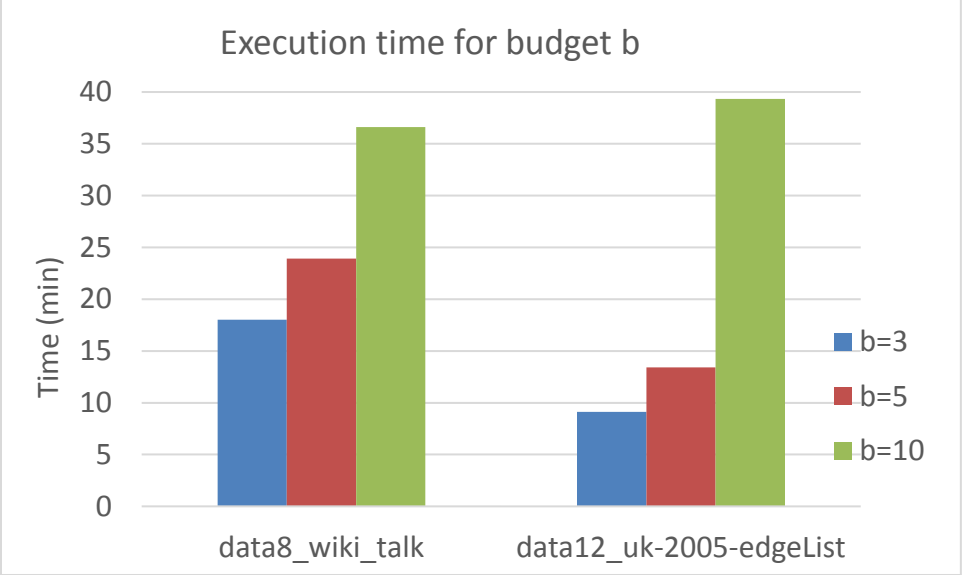


Figure 9. Execution times of Algorithm 4.2 on dataset 3

6. 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 [13] 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 [13] 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 [13]. 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.

Bibliography

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

[2] R. Farzan, L. A. Dabbish, R. E. Kraut, and T. Postmes, "Increasing commitment to online communities by designing for social presence," *In CSCW*, 2011.

[3] D. A. Prentice, D. T. Miller, and J. R. Lightdale, "Asymmetries in attachments to groups and to their members: Distinguishing between common-identity and common-bond groups," *Personality and Social Psychology Bulletin*, Vol 20, No 5, pages 484–493, 1994.

[4] Y. Ren, R. Kraut, and S. Kiesler, "Applying common identity and bond theory to design of online communities," *Organization Studies*, Vol 28, No 3, pages 377–408, 2007.

[5] FACEBOOK. *News room - statistics*. http://newsroom.fb.com/company-info/, 2016.

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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* (*CIKM)* pages 1553-1562, 2016.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

Appendix

The implementation codes for the anchored 2-core algorithm can be found at https://github.com/btootoonchi/KCore-WebGraph.