NIMBLECORE: A Space-efficient External Memory Algorithm for Estimating Core Numbers

Priya Govindan
Rutgers University
priya.govindan@rutgers.edu

Sucheta Soundarajan
Syracuse University
susounda@syr.edu

Tina Eliassi-Rad
Northeastern University
eliassi@ccs.neu.edu

Christos Faloutsos
Carnegie Mellon University
christos@cs.cmu.edu

***Abstract*—We address the problem of estimating core numbers of nodes by reading edges of a large graph stored in external memory. The core number of a node is the highest k-core in which the node participates. Core numbers are useful in many graph mining tasks, especially ones that involve finding communities of nodes, influential spreaders and dense subgraphs. Large graphs often do not fit on the memory of a single machine. Existing external memory solutions do not give bounds on the required space. In practice, existing solutions also do not scale with the size of the graph. We propose $NimbleCore$, an iterative external-memory algorithm, which estimates core numbers of nodes using $O(n \log d_{max})$ space, where n is the number of nodes and d_{max} is the maximum node-degree in the graph. We also show that $NimbleCore$ requires $O(n)$ space for graphs with power-law degree distributions. Experiments on forty-eight large graphs from various domains demonstrate that $NimbleCore$ gives space savings up to 60X, while accurately estimating core numbers with average relative error less than 2.3%.**

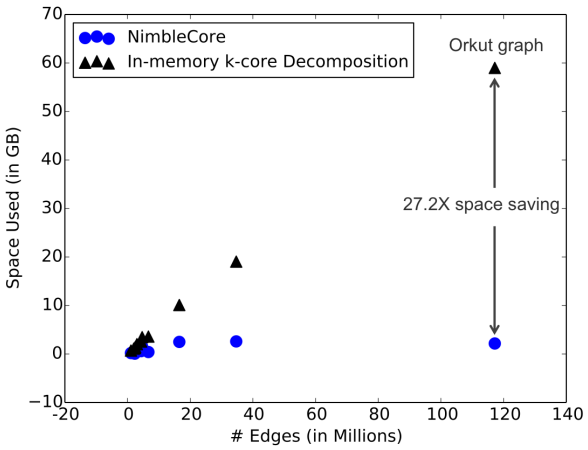


Fig. 1: ***NimbleCore*** **gives significant savings (8.04X average $\pm$ 7.5) while accurately estimating core numbers (0.011 average error $\pm$ 0.018).**

I. INTRODUCTION

Graphs are used to represent social connections, interactions, co-occurrences, and relationships between entities. Understanding the structure of graphs is key to building robust and efficient mining algorithms. Questions such as '*how central is a node?*', '*what is the most densely connected set of nodes?*', or '*which nodes lie on the periphery of the graph?*', are relevant in various graph-mining applications. Central to such applications is the concept of a *k-core* [17], which is defined to be the maximal subgraph such that all the nodes in the subgraph have degree at least k in the subgraph. Every node may be part of several k-cores, each corresponding to a different value of k. The *core number* of a node is the highest value k such that the node is a part of a k-core. Core numbers of nodes have wide applications in problems such as community detection, selecting nodes for network experiments, and modeling the spread of information. For example, in community detection it is helpful to know the set of nodes that are part of a k-core containing high degree nodes [14]. Another example is the use of k-core decomposition and core numbers to understand the structure of protein interaction networks [20].

Problem definition: Given a graph $G = (V, E)$, compute the core number C_u of each node u in V. Core numbers can be calculated by running an in-memory k-core decomposition algorithm [1], which takes $O(max(n, m))$ time and $O(m)$ space, where n and m are the number of nodes and edges in the graph respectively. Other existing algorithms perform k-core decomposition in a distributed setting [12], or estimate a node's core number from its local subgraph [13], or partition the graph and load the required partitions to compute k-cores [2]. Building the local subgraph [13] or finding the appropriate partition [2] can take large amounts of time and space, with significant numbers of I/O calls. In the worst case, these algorithms can require $O(m)$ space in memory. Similarly, the distributed algorithm by Montresor et al. [12] could require $O(m)$ space and up to n iterations over the data. As graph datasets become larger and larger, the need for algorithms that require much less than $O(m)$ space increases. The main **challenge** for estimating core numbers in large graphs is that the graph cannot be stored in main memory.

The major **contributions** of our work are as follows:

1) **Novel algorithm:** We present $\boldsymbol{NimbleCore}$, a space-efficient external memory algorithm that accurately core numbers in a large graph without storing the graph.
2) **Theoretical analysis:** $\boldsymbol{NimbleCore}$ requires only $O(n)$ space for graphs with power-law degree distributions and $O(n \log d_{max})$ space for general graphs, where n is the number of nodes in the graph and d_{max} is the maximum node degree.

IEEE/ACM ASONAM 2016, August 18-21, 2016, San Francisco, CA, USA 978-1-5090-2846-7/16/$31.00

3) **Real-world experiments:** Experiments on large real-world large graphs from various domains show that $\boldsymbol{NimbleCore}$'s achieves space savings on average of 8.04X ($\pm$ 7.5) while returning core number estimates with average relative error of 0.011 ($\pm$ 0.018).

The code for $\boldsymbol{NimbleCore}$ is available at http://eden.rutgers.edu/~priyagn/code.html

The remainder of this paper is organized as follows. Section II gives the background and notations used in the paper. In Section III, we present our proposed algorithm $NimbleCore$ and its theoretical analysis. Section IV discusses experiments, which show the efficiency and effectiveness of $NimbleCore$. Section V reports on the related work. Section VI concludes the paper.

II. Background

Table I lists the notation used in this paper. We are given a graph $G = (V, E)$. A node $u \in V$ has degree d_u; its neighboring nodes are in Γ_u (i.e., $\Gamma_u = \{v \in V \text{ s.t. } (u, v) \in E\}$). The core number of node u is denoted by C_u and is the maximum value k of k-core in which u participates.

Given the core numbers of a node's neighbors, its core number can be exactly calculated as follows:

Lemma 1. *Let $\mathcal{S}_u$ contain the core numbers of neighbors of node u, sorted in descending order.*

$$C_u = \max_{1 \le i \le d_u} (\min(S_u[i], i))$$

Proof: See [12] [13] for the proof. ∎

From the above lemma, we can state that the core number C_u of a node u is the largest value k such that there are at least k neighbors with degree at least k. Note that the above computation of C_u is equivalent to the definition of h-index [6] used to measure the productivity and citation impact of a scholar. The h-index of a scholar with N papers, is defined as the highest value h such that the scholar has at least h papers with h or more citations. We can formally generalize the definition of h-index on a list of values as follows:

Definition 1. *The h-index of a list of positive integers L sorted in descending order is defined as*

$$h(L) = \max_{1 \le i \le length(L)} (\min(L[i]), i))$$

where $L_i \in \mathbb{Z}_{>0}$.

Given Lemma 1 and Definition 1, we can state that node u's core number is equal to the h-index of the core numbers of u's neighbors (see Figure 2a for a toy example). We will refer to the computation in Lemma 1 as h-index. The following lemma shows that for a node u, the h-index of the upper bounds on the core numbers of u's neighbors is an upper bound on the core number of u.

Lemma 2. *Let ψ be a function such that $\psi(x) \ge C_x$, for $x \in V$. Let $\Gamma_u = \{v_1, v_2 ... v_{d_u}\}$ be the neighbors of node $u \in V$. Let Ψ be the list of $\psi(v) \forall v \in \Gamma_u$ in descending order.*

$$C_u \le \max_{1 \le i \le d_u} (\min(\Psi(v_i), i)).$$

Notation	Description
V	set of nodes
E	set of edges
n	number of nodes
m	number of edges
d_u	degree of node u
d_{max}	maximum degree in the graph
C_u	core number of node u
$\hat{C}_u$	upper-bound on the core number of node u
$\bar{C}_u$	lower-bound on the core number of node u
C_u^i	estimate of node u's core number in i^{th} iteration
Γ_u	list containing neighbors of node u
S_u	sorted list (in descending order) containing core numbers of neighbors of node u
k_{max}	maximum core number (a.k.a. degeneracy) in G
$h(L)$	h-index of list L
$L[i]$	the i^{th} element of the list L

TABLE I: Notations used in the paper.

Proof: See [2]. ∎

Because a node in a k-core must have degree at least k, the degree of a node is a naive upper-bound for the node's core number (i.e., for a node u, $C_u \le d_u$). For graphs that can be stored on a single machine, Montresor *et al.* [12] show that we can use Lemmas 1 and 2, begin with degree as a naive upper-bound estimate for a node and iterate over the set of edges to compute the upper-bound estimate of each node's core number. They show that these values will converge to the true core number of each node. However, for graphs that cannot be stored in memory, this is not feasible.

III. Proposed Approach: $\boldsymbol{NimbleCore}$

This section is organized as follows. We first provide an overview of $NimbleCore$. Next, we describe its binning strategy, followed by its stopping condition. We then present the entire algorithm, and wrap-up the section by discussing a theoretical analysis and extensions of $NimbleCore$.

A. Overview of NimbleCore

We present an iterative external-memory algorithm, $NimbleCore$, that estimates each node's core number by binning its neighbors' core-number estimates and then estimating the h-index using these binned values. In other words, rather than storing a complete vector of core-number estimates for each node's neighbors, we divide these neighbor estimates into separate core-number bins. This allows for a large reduction in space requirements.

$NimbleCore$ iterates over the graph's edges. In the first iteration, it computes the degree of each node. In subsequent iterations, while the stopping condition is not satisfied (see Section III-C), $NimbleCore$ uses binning to keep approximate counts of the core numbers of neighbors of each node. At the end of each iterations, $NimbleCore$ updates the core number estimates of all the nodes.

The two main aspects of $NimbleCore$ are (1) the binning strategy and (2) the stopping condition. Bins are vectors

that are maintained to count the frequency of items. For example, while creating a histogram, one would be required to count items that fall in each range of values. Similarly, $NimbleCore$ uses bins to approximately count the frequency of core numbers of neighbors of a node. The binning strategy plays an important role because the number of bins determines the space requirement of $NimbleCore$ and the assignment of bin sizes affects the error on the estimated core numbers. Lastly, a good stopping condition results in fewer number iterations, thus saving runtime.

B. Binning Strategy

In this section, we describe $NimbleCore$'s binning strategy, which requires $O(n \log d_{max})$ space. The error resulting from the binning strategy of $NimbleCore$ is discussed in Section III-E.

Lets first consider a naive binning algorithm, as shown in Figure 2a, where a bin is maintained for each distinct value of a neighbor's core number estimate, for each node. This naive binning strategy essentially build a histogram and thus requires $O(nd_{max})$ space, which is equivalent to $O(m)$; and hence not feasible for large graphs. One could use fewer counters (or bins) to build an approximate histogram. ***But, how many bins do we choose, and of what sizes, to get the best estimates of core numbers?***

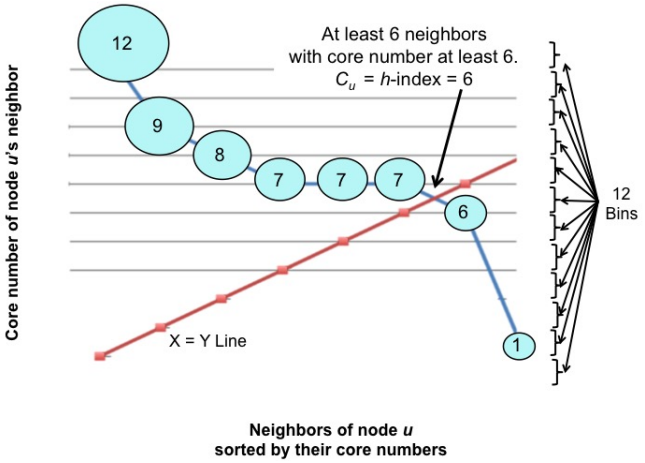
Given the core numbers of a node's neighbors, $NimbleCore$'s binning, called *reverse log-binning*, selects larger bins for smaller values and smaller bins for larger values. In this way, $NimbleCore$ allows for more granularity in the core-number estimates of u's neighbors that are closer to the estimated upper-bound core number $\hat{C}_u$.

Reverse log-binning sets the number and sizes of bins (for the computation of approximate h-index) as follows:

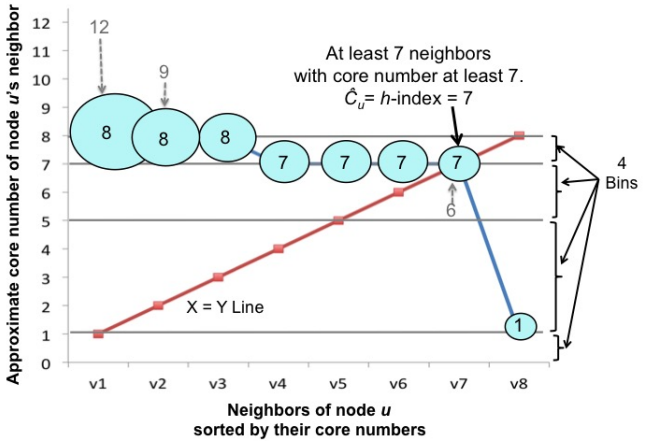
- Let g be an upper bound on the estimate of node u's core number. g is initially set to the degree of u.
- Let B be the number of bins: $B = \log(g) + 1$.
- Set $BinCount$ and $BinValue$ of length B such that $\forall 0 \leq i < B$, $BinValue\ (i) = g - 2^{\log(g)-i} + 1$.
- A value $S_u[j]$ is counted in $BinCount\ (i)$ if $BinValue\ (i-1) < S_u[j] \leq BinValue\ (i)$.

Figure 2b demonstrates how $NimbleCore$'s reverse log-binning would estimate $\hat{C}_u$ using $BinCount$ and $BinValue$. As shown in the figure, there are two kinds of approximations done depending on whether the values are higher or lower than the current estimate. In Figure 2b, the current estimate, $\hat{C}_u$, is the degree, which is 8. Values higher than $\hat{C}_u$ are approximated to the last bin (i.e., the current upper-bound estimate). For example, 12 being higher than 8, is approximated to 8 and counted in the last bin corresponding to 8. Values that are at most the current estimate $\hat{C}_u$, are assigned to an appropriate bin as described above. For example, 6 is assigned to third bin and its value is approximated to 7.

Next, we show that the above approximation of core numbers of a node's neighbors and the subsequent h-index computation result in an upper-bound estimate of the node's core number. For values in S_u that are greater than the given



(a) Let the core numbers of the 8 neighbors of node u be [12, 9, 8, 7, 7, 7, 6, 1]. A node's core number is the h-index of the core numbers of its neighbors. The h-index is the value at which the descending order of a set of values intersect with the 'X=Y' line. A node's exact core number can be computed by storing the exact core numbers of its neighbors in arrays $BinValue$ = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] and $BinCount$ = [1, 0, 0, 0, 0, 1, 3, 1, 1, 0, 0, 1]. This requires $O(d_u)$ space per node.



(b) Proposed idea: Reverse Log-binnng. A node's approximate core number can be computed by storing approximate values and counts of its neighbors's core numbers in arrays, $BinValue$ = [1, 5, 7, 8] and $BinCount$ = [1, 0, 4, 3]. This requires $O(log(d_u))$ space per node.

Fig. 2: Example: Calculating node u's core number using the core numbers of u's neighbors

upper bound of the h-index, we count them in the last bin (i.e., in BinCount(B)). Lemma 3 shows that this procedure does not change the h-index result.

Lemma 3. *Let L be an ordered list (in descending order). Let j be the highest index such that $L[j] > h(L)$, where $h(L)$ is the h-index of the list L (see Definition 1). Furthermore, let L' be an ordered list (in descending order) such that $\forall i : 0 \leq i < length(L)$, $L'[i]$ is set to $h(L)$ if $i \leq j$; otherwise, $L'[i]$ is set to $L[i]$. Then, $h(L') = h(L)$.*

Proof: If $\nexists j$ such that $L[j] > h(L)$, then L' and L are

equivalent. Hence $h(L') = h(L)$. On the other hand, if $\exists j$ such that $L[j] > h(L)$, then the maximum value in L' is $h(L)$. So, $h(L') \leq h(L)$. The values at indices $i \leq j$ are replaced with $h(L)$ because j is the highest index (i.e., $L[j] > h(L)$). Thus, there are at least $h(L)$ values in L' that are at least $h(L)$; i.e. $h(L') \geq h(L)$. Hence $h(L') = h(L)$. ■

For values in S_u that are less than the given upper bound of the h-index, reverse log-binning assigns them to one of the bins and approximates their values to the upper bound of the range of values held by the bin. Hence for values in S_u that are less than the current estimate, their approximation is always an upper bound of their exact value. By Lemma 2, we know that the upper bounds on the core numbers of u's neighbors give the upper bound of u's core number.

Having small bin sizes for values that are close to but not larger than the current upper-bound estimate gives lower error in the h-index computation.[1] For example, for a node with degree five to have a core number of five, five of its neighbors must have core numbers equal to or greater than five. Thus, having smaller bins close to and not greater than five, would result in lower error in the h-index for our example node.

C. Stopping Condition

In every iteration, $NimbleCore$ calculates better estimates for the core numbers of nodes than that in the previous iteration. Although more iterations give better estimates and lower error, this increases the runtime of $NimbleCore$. In this section, we discuss how $NimbleCore$ decides to stop iterating over the list of edges, making a good trade-off between number of iterations and error.

Our stopping condition is based on estimating the average relative error at every iteration and stopping when the estimated error drops below a threshold. The relative error (also referred to as 'error' in the paper) of these estimates can be calculated as follows,

$$\text{Relative Error} = \frac{\hat{C}_u - C_u}{C_u} \tag{1}$$

where C_u is u's exact core number and $\hat{C}_u$ is the upper-bound estimate for u's exact core number. Since C_u is not known, we estimate the lower bound $\bar{C}_u$ and thus estimate the error.

$$\text{Estimated Relative Error} = \frac{\hat{C}_u - \bar{C}_u}{\bar{C}_u} \tag{2}$$

Since $\bar{C}_u \leq C_u$, Estimated Relative Error $\geq$ Relative Error. We compute the lower bound as follows: *Step 1:* Sample a set of wedges for each node. *Step 2:* Count the number of closed wedges i.e. triangles. *Step 3:* Given the number of triangles of node u, using Turan's theorem,[2] compute the size of the lower bound of the size of largest clique, r, that the node u participates in. If a node participates in a clique of size r, it is a part of the r-core and hence the lower bound of core number of u is r.

Similar to tightening the upper bound, we can tighten the lower bound estimate for a node's core number in each iteration. Better lower bounds give a better estimate of the error. In every iteration, for all $u \in V$, the lower-bound $\bar{C}_u$ and the upper-bound $\hat{C}_u$ of u's core number are updated. $\bar{C}_u$ helps estimate the error, as described above. Since the upper bound monotonically decreases and the lower bound monotonically increases, the estimated error also monotonically decreases. As the number of the iterations increases, the estimates of core numbers converge (i.e., stop changing). The goal of the stopping condition is to terminate $NimbleCore$ when the change in the estimates falls below a negligible value. Specifically, when the difference in the estimated error, averaged over all nodes, between consecutive iterations falls below a threshold, $NimbleCore$ terminates.[3]

D. The Complete Algorithm

In the first iteration, the degree of each node u is calculated by incrementing a counter for u when an edge containing u is read. The second and the third iterations are used to count triangles, which are used in calculating a lower-bound estimate on core numbers. Specifically, in the second iteration, the neighbors of a node are sampled with probability p. A wedge is comprised of a pair of sampled neighbors. In the third iteration, each edge is checked to see if it closes any of the sampled wedges. A closed wedge is a triangle. From the triangles observed in the third iteration, $NimbleCore$ estimates the lower bound on each node's core number and calculates an estimated error (described in Section III-C). The subsequent iterations estimate and update the upper and lower-bounds of core numbers of the nodes. The method terminates when the drop in the estimated error falls below a threshold. The upper-bound estimates from the last iteration are then returned as the final estimates.

E. Performance Analysis of NimbleCore

The main challenge that $NimbleCore$ is trying to address is that of space. By cleverly choosing bin sizes, $NimbleCore$ is able to achieve a dramatic reduction in required space, as compared to existing methods. In this section, we provide bounds on $NimbleCore$'s space and time requirements.

1) Space: $NimbleCore$ requires $O(n \log d_{max})$ space, which is less than the $O(m)$ space required for a k-core decomposition. Many real graphs have power-law degree distributions [3]. For these graphs, $NimbleCore$ requires only $O(n)$ space.

Theorem 1. *$NimbleCore$ requires $O(n \log d_{max})$ space.*

Proof: $NimbleCore$ requires $\log d_u$ bins per node, where d_u is the degree of node u in the graph. So, it requires $O(\log d_{max})$ space per node, where d_{max} is the maximum degree. Given n nodes in the graph, the total space required is then $O(n \log d_{max})$. ■

[1] Recall that values larger than the current estimate are all placed in the last bin.

[2] Turan's theorem [18] about a graph without a clique of size $r+1$ is as follows: Let G be any graph with n vertices and m edges, such that G does not have a clique of size $r+1$. Then $m \leq \left(\frac{r-1}{r} \cdot \frac{n^2}{2}\right)$

[3] The experiments in Section IV use a threshold of 0.01.

Theorem 2. *$NimbleCore$ requires $O(n)$ space for any graph with power-law degree distribution.*

Proof: Suppose a graph $G = (V, E)$ has $n = |V|$ nodes, such that the degrees of the nodes, arranged in decreasing order, follow a Zipf distribution. The i^{th} node with degree d_i is given as $d_i = i^R/n^R$ for all $i = 1, \ldots, n$, where $R < 0$. Space required by $NimbleCore$ per node is $\log_\epsilon d_i = \log(i^R/n^R)$, where $\epsilon > 1$. The base of the log is 2 (i.e., $\epsilon = 2$). Total space used by the algorithm for computation of core numbers (in each iteration) is then:

$$\sum_{i=1}^{n} \log(d_i) = \sum_{i=1}^{n} \log(i^R/n^R)$$

Approximating the summation with integral we get,

$$\int_{x=1}^{n} \log(x^R/n^R)dx = -nR + R + R\log n$$

Since $R < 0$, $\forall n > 1$, $R \log n < 0$. Thus, the total space required by $NimbleCore$ is less than $-nR$, i.e. $O(n)$. ■

2) Runtime: The runtime required by $NimbleCore$ depends on the number of iterations. Each iteration takes $O(m + n \log d_{max})$ time.

Theorem 3. *$NimbleCore$ requires $O(m + n \log d_{max})$ time per iteration.*

Proof: Each iteration requires $O(m)$ time to read the edges. After an iteration, $NimbleCore$ takes $O(\log d_{max})$ time per node to estimate its core number. Hence, the time taken per iteration by $NimbleCore$ is $O(m + n \log d_{max})$. ■

The number of iterations depends on the stopping condition (described in Section III-B), which in turn depends on the estimates of the lower bounds on core numbers. Thus, the number of iterations, and hence the runtime for $NimbleCore$, depends on the quality of the lower-bound estimates.

F. Extensions

$NimbleCore$ easily extends to directed graphs, by considering either the in-degree or out-degree to find the corresponding k-cores. $NimbleCore$ can also be extended to weighted graphs $G(V, E, W)$, where $W(u, v)$ is the weight on edge (u, v) as described by Giatsidis et.al [4], by replacing the h-index function in Lemma 1 with

$$C_u = \max(\min_{1 \le i \le d_u} (S_u[i], \sum_{j=1}^{i} W(u, \Lambda_u[j])))$$

where Λ_u is the list of neighbors corresponding to the sorted list of core numbers of neighbors S_u.

IV. Experiments

This section is organized as follows: datasets, baseline and competing approaches, and results. All experiments were run on a CentOS machine with 2.4 GHz (x 80) and 1024 GB memory, running Linux 2.6 and using Python 2.7.

Graph	# Nodes	# Edges	k_{max}	Graph type
Web-NotreDame	326K	1.1M	155	Unipartite
Web-Stanford	282K	2.M	71	Unipartite
Flickr	106K	2.3M	573	Unipartite
Amazon	403K	2.4M	10	Projected Bipartite
YouTube	1M	3.M	51	Unipartite
Web-Google	876K	4.3M	44	Unipartite
Wiki-Talk	2M	4.7M	131	Unipartite
Web-BerkStan	685K	6.6M	201	Unipartite
Cit-Patents	4M	16.5M	64	Unipartite
LiveJournal	4M	34.7M	360	Unipartite
Orkut	3M	117.2M	253	Unipartite

TABLE II: Graphs used in our experiments. The graph's degeneracy is denoted by k_{max}.

A. Datasets

Tables II lists the graphs from the Stanford Large Network Dataset Collection [10], used in our experiments. We consider a variety of graph types: *social networks* (Flickr, LiveJournal, Orkut, and YouTube), *Web graphs* (Web-BerkStan, Web-Google, Web-NotreDame and Web-Stanford), *citation graph* (Cit-Patent), *co-purchasing graphs* (Amazon), and *communication networks* (Wiki-Talk).

B. Baseline and Competing Methods

Figure 3 provides a summary of $NimbleCore$ and the baseline and competing methods. Specifically, we compare $NimbleCore$ against the following four methods.

(1) k-core decomposition: Batagelj and Zaversnik [1] propose an in-memory algorithm to find the k-cores of a graph, and thus the core numbers of nodes. To find a k-core, their method recursively deletes all nodes with degree less than k, until there are no nodes of degree k left. This method requires $O(m)$ space and $O(max(n, m))$ runtime.

(2) Shaving Method: k-core decomposition as described above can be implemented in an out-of-memory fashion by maintaining the current degree of each node in a vector of length n and by reading edges from the disk. To find a k-core, Shaving deletes nodes that have degree less than k and re-computes the degree of the remaining nodes, requiring two iterations on the edge list. As this step is performed repeatedly, a node that is removed when finding a k-core will have a core number equal to $k-1$. This exact method requires $O(n)$ space and could require up to $O(m \times k_{max})$ iterations, resulting in a worst case time complexity of $O(m^2 \times k_{max})$.

(3) EMcore: Cheng et al. [2] propose an out-of-memory exact k-core decomposition method that partitions the graph into blocks and loads the required blocks of the graph into memory for the k-core decomposition. Unfortunately the space guarantees given in their paper was shown to be incorrect by Goodrich and Pszona [5] and by Khaouid et al. [7]. In worst case, EMcore could require $O(m)$ space.

(4) Egonet-based core number estimation: O'Brien and Sullivan [13] propose an approximate method to estimate a node's core number by using the induced subgraph up to h hops away from the node. Specifically, for each such subgraph, they run k-core decomposition to estimate the node's the core

Guaranteed space required < *O(m)*

	NimbleCore:* *Our Proposed Method	**Shaving Method**
Space	General graphs: ***O(n log d_{max})*** Graphs with power law degree dist: ***O(n)***	*O(n)*
Runtime	A minimum of **40x fewer iterations** and **2.8x faster in runtime** than Shaving. Uses estimated error to terminate.	*O($m^2 \times k_{max}$)* iterations
Error	**Average relative error = 0.011** (SD = 0.018)	Exact

No guarantee of space required < *O(m)*

	K-core Decomposition [2]	**EMcore [6]**	**Egonet-based Method [20]**
Space	*O(m)* Graph in memory	No guarantees. Worst case: *O(m)* *	No guarantees. Worst case: *O(m)*
Runtime	*O(max(n, m))*	Partition graph: *O(m)* + k_{max} × (Build subgraph + *k*-core decomposition)	*O($n \times d_{max}^2$)* for 1-hop egonet.
Error	Exact	Exact	No guarantees

Fig. 3: Comparison of other methods to $NimbleCore$. Unlike $NimbleCore$, several of the existing methods (listed in the right-hand side table) do not guarantee space less than $O(m)$. Shaving is the only method that provides a guaranteed space of less than $O(m)$, but it requires many more iterations than $NimbleCore$ and has slower runtime than $NimbleCore$. ** Note that the lower space guarantee in EMcore [2] was shown to be incorrect by Khaouid et al. [7] and by Goodrich and Pszona [5].*

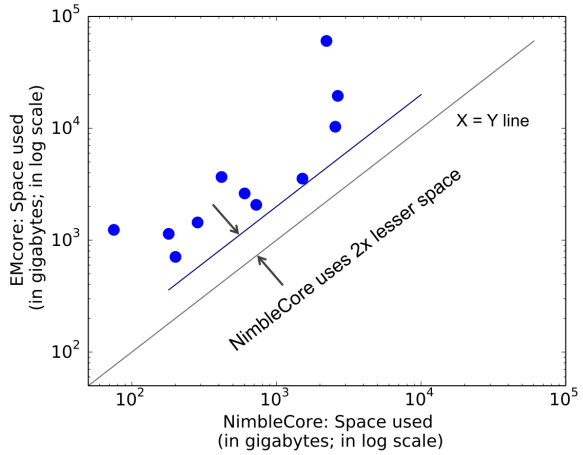


Fig. 4: ***NimbleCore* uses $2\times$ lesser space than EMcore [2].** The space required by $NimbleCore$ for a given graph is $O(n \log d_{max})$, while the space requirement for EMcore depends on the k-core being identified.

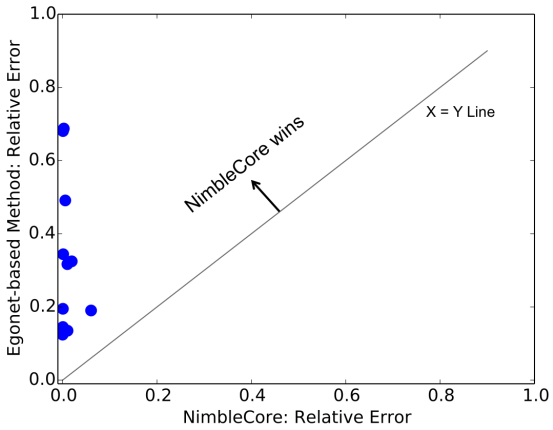


Fig. 5: ***NimbleCore*'s error is less than the 1-hop Egonet-based method [13].** Each point on the plot represents a graph. The average ratio of error on $NimbleCore$ to error on the Egonet-based method is 0.051 ($\pm$ 0.097). (A tie would have an average ratio of error = 1).

number. When the number of hops is set to the diameter of the graph, the core number of a node is exactly calculated. The performance of their algorithm in terms of space, runtime, and error depends on the number of hops considered for each node. In our comparisons, we use the 1-hop induced subgraph (i.e., the egonet) of a node.

C. Results

This section contains two parts. In part 1, we demonstrate the performance of $NimbleCore$ in terms of space, error, runtime, and stopping condition. In part 2, we discuss observations about $NimbleCore$ in terms of the drop in error on core-number estimates and error on core-number estimates of high-degree nodes.

1) Performance of NimbleCore:

Q1: Space: How much space does *NimbleCore* save? Figure 1 shows that on average, $NimbleCore$ saves 8.04X ($\pm$ 7.5) of the space as compared to the k-core decomposition [1] on the graphs in TableII. These observations support the theoretical results that real graphs, with power-law degree distributions, the $NimbleCore$'s space complexity is $O(n)$ and for general graphs is $O(n \log(d_{max}))$ while the space requirement for k-core decomposition is $O(m)$ For example, for Orkut graph with 3 million nodes and 117.2 million edges, $NimbleCore$ requires only 2.1 GB space, where as the k-core decomposition requires 59 GB space.

EMcore [2] is an out-of-memory exact k-core decomposition method, based on graph-partitioning that requires $O(m)$ space in worst case. Figure 4 shows that the space required by EMcore is at least $2\times$ higher than $NimbleCore$ (on average $8\times$ higher, with $\pm$ 7.49). In contrast to EMcore, $NimbleCore$ guarantees the space requirement to be $O(n \log d_{max})$, which is smaller than $O(m)$.

Q2: Error: How accurate are *NimbleCore*'s estimated core numbers?

The average relative error of core number estimates of the graphs in Table II is 0.011 ($\pm$0.018). To demonstrate the

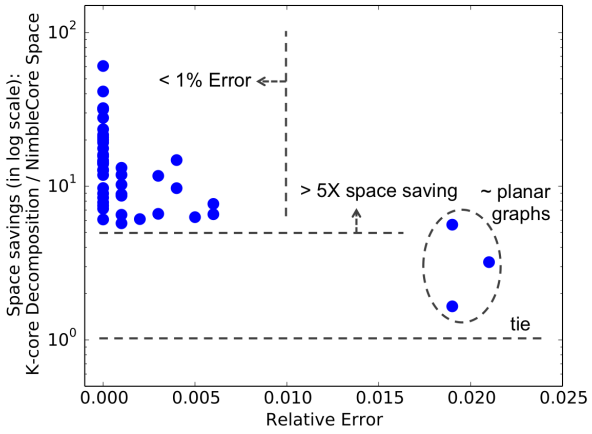


Fig. 6: ***NimbleCore*gives upto 60.5X space savings for error less than 2.3%** Each point corresponds to one of the 48 large graphs from KONECT [9]. (0.002 average error $\pm$ 0.005 and 14.6X average space saving $\pm$ 10.8X).

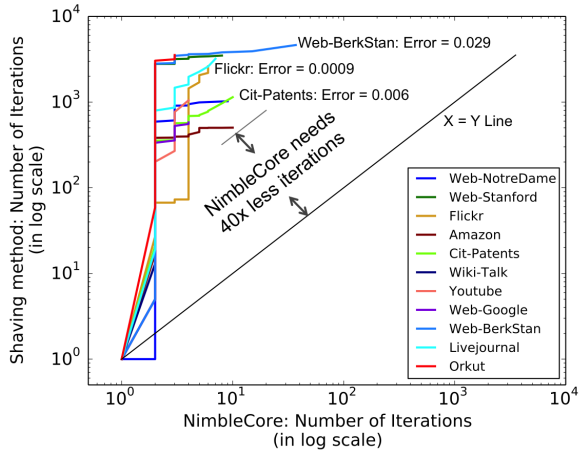


Fig. 7: ***NimbleCore* requires fewer iterations than Shaving.** Each point on the line in the plot corresponds to the number of iterations required by $NimbleCore$ and the number of iterations required by Shaving to obtain the same relative error. ***NimbleCore*** has 2.8$\times$ faster runtime than Shaving.

quality of the estimates returned by $NimbleCore$, we ran experiments on additional 48 graphs[4] from KONECT [9][5] with more than a million nodes and for which the *competition* (namely, the in-memory k-core decomposition) took up to 36 hours to terminate (in contrast $NimbleCore$ took less than 2.5 hours to terminate, for all 48 graphs). Figure 6 shows that up to 60X space savings (14.6X average $\pm$10.8) with relative error less than 2.3% (0.002 average error $\pm$ 0.005) was achieved. The three points to the right represent road networks (with maximum degree of 12), which are similar to planar graphs.

O'Brien and Sullivan's egonet-based method [13] extracts

[4]List of the additional 48 graphs: http://eden.rutgers.edu/~priyagn/code.html

[5]http://konect.uni-koblenz.de/networks/

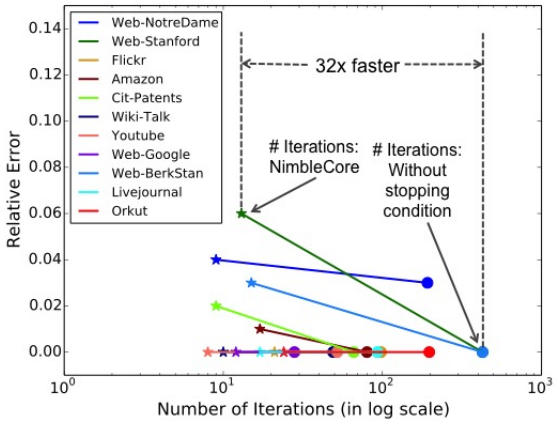


Fig. 8: ***NimbleCore*'s stopping condition produces significant speed-up.** The stopping condition described in III-C helps terminate $NimbleCore$ in fewer iterations. The average speed-up is 11.5$\times$ ($\pm$ 10.7). The average difference (i.e. sacrifice) in error is 0.012 ($\pm$ 0.019).

a h-hop egonet for every node and computes its core number based on the egonet. The value for h has to be set by the user. Constructing egonets for every node can be time-consuming depending on the value of h. Figure 5 compares the relative error of the 1-hop Egonet-based method with $NimbleCore$ on graphs in Table II. We observe that the average error of the Egonet-based method is 0.29 (with $\pm$ of 0.18), while that of $NimbleCore$'s is 0.011 (with $\pm$ of 0.018).

Q3: Runtime: How much faster is *NimbleCore* than the Shaving method? The Shaving method is an out-of-memory baseline that uses $O(n)$ space and iterates over the edges by reading them off the disk. Since $NimbleCore$ also requires $O(n)$ space for graphs with power-law degree distributions and iterates over the edges from the disk, we compare the number of iterations required by the two methods. Figure 7 shows the number of iterations required by $NimbleCore$ and the Shaving method to achieve the same error, on graphs in TableII. It shows that the Shaving method requires at least 40X more iterations over the edges than $NimbleCore$, to achieve the same error. In terms of wall-clock runtime, $NimbleCore$ was found to be at least 2.8$\times$ faster than the Shaving method.

Q4: Stopping Condition: How effective is *NimbleCore*'s stopping condition? The stopping condition helps $NimbleCore$ terminate in a small number of iterations, making a trade-off between error and runtime. Recall that the stopping condition depends on the estimated error. Figure 8 presents the number of iterations vs. relative error under two scenarios: (1) using $NimbleCore$'s stopping condition (See Section III-C), and (2) using no stopping condition (i.e. terminating $NimbleCore$ when there is no change in the core number of all nodes between consequent iterations). $NimbleCore$'s stopping condition provides an average speedup of 11.5$\times$ ($\pm$10.7) with an average difference in error of 0.012 ($\pm$ 0.019). Note that the error for 'Web-

NotreDame' graph is higher than 0 because $NimbleCore$ without a stopping condition is still an approximate algorithm.

2) Observations about NimbleCore:

Observation 1: Error drops quickly over a small number of iterations. The first, fifth, tenth and twentieth iterations, respectively, give an error of 0.49 ($\pm$ 0.19), 0.04 ($\pm$ 0.03), 0.017 ($\pm$0.021) and 0.01 ($\pm$ 0.017), on graphs in TableII. This demonstrates that even without a stopping condition, in real graphs, low error can be obtained by running $NimbleCore$ for as low as 5 iterations.

Observation 2: The error of high degree nodes is low. In many applications [19] [14], it is useful to know the core number of high degree nodes. We found that the average error of $NimbleCore$ on the top 1% of the high-degree nodes in the graphs in TableII, is 0.012 ($\pm$ 0.008), as compared to an average of 0.011 ($\pm$ 0.018) for the whole graph.

V. Related Work

Core numbers and k-cores of a graph, introduced by Seidman [17], have been shown to have many applications in problems such as community detection [4], finding dense subgraphs [15], designing network experiments [19], modeling spread in networks [8], predicting protein function [20] and graph coloring problems [11].

Batagelj and Zaversnik [1] proposed a k-core decomposition algorithm that requires $O(max(n, m))$ runtime and $O(m)$ space to perform k-core decomposition. Cheng et al. [2] presented a method to partition the graph and compute core number in a distributed fashion. O'Brien and Sullivan [13] introduced a recursive algorithm that computes core numbers for each node based on a h hop subgraph around the node. We compare $NimbleCore$ against these three methods. See Figure 3 for a summary comparison and Section IV-C for empirical comparisons.

Montresor et al. [12] proposed a distributed k-core decomposition algorithm, which is similar to $NimbleCore$. In [16], Saríyüce et al. propose algorithms for updating a k-core decomposition of a graph, as edge insertions and deletions are given as a stream, by storing certain subgraphs in memory. $NimbleCore$ does not store any subgraph or list of neighbors of any node. In recent work by Khaouid et al. [7], they implement the algorithms in [1], in [2] and in a distributed setting. Goodrich and Pszona [5] present a method that returns an approximate degeneracy-ordering of nodes. Given the core numbers of nodes, a degeneracy-ordering is simply an ordering of nodes by their core number, but the core number of nodes cannot be obtained from a degeneracy-ordering.

VI. Conclusion

We presented $\boldsymbol{NimbleCore}$, a novel space-efficient external-memory algorithm, which estimates core numbers of nodes in graphs too large to be stored in main memory. Some of $\boldsymbol{NimbleCore}$'s properties are as follows:

1) **Novel algorithm:** $\boldsymbol{NimbleCore}$ iterates over the edges and computes accurate estimates of core numbers in a large graph without storing the graph in memory.
2) **Theoretical analysis:** $\boldsymbol{NimbleCore}$ requires $O(n)$ space for graphs with power-law degree distributions and $O(n \log d_{max})$ for general graphs.
3) **Real-world experiments:** In real graphs, $\boldsymbol{NimbleCore}$ gave significant space savings (average 8.04X $\pm$7.5) and consistently low error (0.011 average error $\pm$ 0.018).

The code for $\boldsymbol{NimbleCore}$ is available at http://eden.rutgers.edu/~priyagn/code.html

VII. Acknowledgments

This work was supported by NSF under Grant No. CNS-1314603, CNS-1314632, IIS-1408924, and by DTRA HDTRA1-10-1-0120.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, DTRA, or other funding parties.

References

[1] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.
[2] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.
[3] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.
[4] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *ASONAM*, pages 87–93, 2011.
[5] M. T. Goodrich and P. Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *ESA*, pages 664–676, 2011.
[6] J. E. Hirsch. An index to quantify an individual's scientific research output. *PNAS*, 102(46):16569–16572, 2005.
[7] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single PC. *PVLDB*, 9(1):13–23, 2015.
[8] M. Kitsak, L. Gallos, S. Havlin, F. Liljeros, L. Muchnik, E. Stanley, and H. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, 2010.
[9] J. Kunegis. KONECT: The koblenz network collection. In *Proc. Int. Web Observatory Workshop*, pages 1343–1350, May 2013.
[10] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[11] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.
[12] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE TPDS*, 24(2):288–300, 2013.
[13] M. P. O'Brien and B. D. Sullivan. Locally estimating core numbers. In *ICDM*, pages 460–469, 2014.
[14] C. Peng, T. G. Kolda, and A. Pinar. Accelerating community detection by using k-core subgraphs. *CoRR*, abs/1403.2226, 2014.
[15] Y. Qian, G. Zhang, and K. Zhang. FACADE: A fast and effective approach to the discovery of dense clusters in noisy spatial data. In *SIGMOD*, pages 921–922, 2004.
[16] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.
[17] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.
[18] P. Turán. On an extremal problem in graph theory. *Matematikai és Fizikai Lapok*, 48:436–452, 1941.
[19] J. Ugander, B. Karrer, L. Backstrom, and J. M. Kleinberg. Graph cluster randomization: Network exposure to multiple universes. In *KDD*, pages 329–337, 2013.
[20] S. Wuchty and E. Almaas. Peeling the yeast protein network. *PROTEOMICS*, 5(2):444–449, 2005.