Streaming Algorithms for k-core Decomposition

Ahmet Erdem Sarıyüce[†◇], Buğra Gedik[‡], Gabriela Jacques-Silva[*], Kun-Lung Wu[*], Ümit V. Çatalyürek[†◦]
sariyuce.1@osu.edu, bgedik@cs.bilkent.edu.tr, g.jacques@us.ibm.com, klwu@us.ibm.com, umit@bmi.osu.edu
[†]*Department of Biomedical Informatics, The Ohio State University*
[◇]*Department of Computer Science and Engineering, The Ohio State University*
[‡]*Department of Computer Engineering, İhsan Doğramacı Bilkent University*
[◦]*Department of Electrical and Computer Engineering, The Ohio State University*
[*]*IBM Thomas J. Watson Research Center, IBM Research*

ABSTRACT

A k-core of a graph is a maximal connected subgraph in which every vertex is connected to at least k vertices in the subgraph. k-core decomposition is often used in large-scale network analysis, such as community detection, protein function prediction, visualization, and solving NP-Hard problems on real networks efficiently, like maximal clique finding. In many real-world applications, networks change over time. As a result, it is essential to develop efficient incremental algorithms for streaming graph data. In this paper, we propose the first incremental k-core decomposition algorithms for streaming graph data. These algorithms locate a small subgraph that is guaranteed to contain the list of vertices whose maximum k-core values have to be updated, and efficiently process this subgraph to update the k-core decomposition. Our results show a significant reduction in run-time compared to non-incremental alternatives. We show the efficiency of our algorithms on different types of real and synthetic graphs, at different scales. For a graph of 16 million vertices, we observe speedups reaching a million times, relative to the non-incremental algorithms.

1. INTRODUCTION

Relationships between people and systems can be captured as graphs where vertices represent entities and edges represent connections among them. In many applications, it is highly beneficial to capture this graph structure and analyze it. For instance, the graph may represent a social network, where finding communities in the graph [14] can facilitate targeted advertising. As another example, the graph may represent the web link structure and finding densely connected regions in the graph [12] may help identify link spam [24]. In telecommunications, graphs are used to capture call relationships based on call detail records [22], and locating closely connected groups of people for generating promotions. Graph structures are widely used in biological systems as well, such as in the study of proteins. Locating cliques in protein structures can be used for comparative modeling and prediction [25].

Many real-world graphs are highly dynamic. In social networks, users join/leave and connections are created/severed on a regular basis. In the web graph, new links are established and severed as a natural result of content update and creation. In customer call graphs, new edges are added as people extend their list of contacts. Furthermore, many applications require analyzing such graphs over a time window, as newly forming relationships may be more important than the old ones. For instance, in customer call graphs, the historic calls are not too relevant for churn detection. Looking at a time window naturally brings removals as key operations like insertions. This is because as edges slide out of the time window, they have to be removed from the graph of interest. In summary, dynamic graphs where edges are added and removed continuously are common in practice and represent an important use case.

In this paper, we study the problem of incrementally maintaining the k-core decomposition of a graph. A k-core of a graph [26] is a maximal connected subgraph in which every vertex is connected to at least k other vertices. Finding k-cores in a graph is a fundamental operation for many graph algorithms. k-core is commonly used as part of community detection algorithms [16], as well as for finding dense components in graphs [2, 4, 19], as a filtering step for finding large cliques (as a k-clique is also a k-1-core), and for large-scale network visualization [1].

The k-core decomposition of a graph maintains, for each vertex, the max-k value: the maximum k value for which a k-core containing the vertex exists. This decomposition enables one to quickly find the k-core containing a given vertex for a given k. Algorithms for creating k-core decomposition of a graph in time linear to the number of edges in the graph exist [6]. For applications that manage dynamic graphs, applying such algorithms for every edge insertion and removal is prohibitive in terms of performance. Furthermore, batch processing takes away the ability to react to changes quickly – one of the key benefits of stream processing [28].

In this paper, we develop streaming algorithms for k-core decomposition of graphs. In particular, we develop algorithms to update the decomposition as edges are inserted into and removed from the graph (vertex additions and removals are trivial extensions). There are a number of challenges in achieving this. The first is a theoretical one: determining a small subset of vertices that are guaranteed to contain all vertices that may have their max-k values changed as a result of an insertion or removal. The second is a practical one: finding algorithms that can efficiently update the max-k values using this subset. Last but not the least, we have to understand the impact of the graph structure on the performance of such streaming algorithms.

We address these challenges by developing the first incremental k-core decomposition algorithm for streaming graph data, where we efficiently process a small subgraph for each change. We develop a number of variations of our algorithm and empirically show that incremental processing provides a significant reduction in runtime compared to non-incremental alternatives, reaching 6 orders of magnitude speedup for a graph of size of around 16 million.

Proceedings of the VLDB Endowment, Vol. 6, No. 6

We showcase the efficiency of our algorithms on different types of real and synthetic graphs at different scales and study the impact of graph structure on the performance of algorithm variations.

In summary, we make the following major contributions:

- We identify a small subset of vertices that have to be visited in order to update the max-k values in the presence of edge insertions and deletions.
- We develop various algorithms to update the k-core decomposition incrementally. To the best of our knowledge, these are the first such incremental algorithms.
- We present a comparative experimental study that evaluates the performance of our algorithms on real-world and synthetic data sets.

The rest of this paper is organized as follows. Section 2 gives the background on k-core decomposition of graphs. Section 3 introduces our theoretical findings that facilitate incremental k-core decomposition. Section 4 introduces several new algorithms for incremental maintenance of a graph's k-core decomposition. Section 5 provides discussions on implementation details. Section 6 gives a detailed experimental evaluation of our algorithms. Section 7 reports related work, Section 8 provides possible directions for future work, and Section 9 concludes the paper.

2. BACKGROUND

In this work, we focus on incremental maintenance of k-core decomposition of large networks modeled as undirected and unweighted graphs. Here, we start by giving several definitions that are used throughout the paper as part of our theorems and proofs.

Let G be an undirected and unweighted graph. For a vertex-induced subgraph $H \subseteq G$, $\delta(H)$ denotes the minimum degree of H, defined as the minimum number of neighbors a vertex in H has (i.e., $\delta(H) = min\{\delta_H(u) : u \in H\}$, where $\delta_H(u)$ denotes the number of neighbors of a vertex u in H). As a result, any vertex in H is adjacent to at least $\delta(H)$ other vertices in H and there is no other value larger than $\delta(H)$ that satisfies this property.

DEFINITION 1. *If H is a connected graph with $\delta(H) \geq k$, we say that H is a* seed *k-core of G. Additionally, if H is* ***maximal*** *(i.e., $\nexists H'$* s.t. *$H \subset H' \wedge H'$ is a seed k-core of G), then we say that H is a k-core of G.*

OBSERVATION 1. *Let H be a k-core that contains the vertex u. Then, H is unique in the sense that there can be no other k-core that contains u.*

We denote the unique k-core that contains u as H_k^u.

DEFINITION 2. *The* maximum *k-core associated with a vertex u, denoted by H^u, is the k-core that contains u and has the largest $k = \delta(H^u)$ (i.e., $\nexists H$* s.t. *$u \in H \wedge H$ is an l-core $\wedge\ l > k$). The* maximum *k-core number of u (also called the K value of u), denoted by $K(u)$, is defined as $K(u) = \delta(H^u)$.*

OBSERVATION 2. *If H is a k-core in graph G, then there exists one and only one $(k-1)$-core $H' \supseteq H$ in G.*

OBSERVATION 3. *A vertex u with $K(u) = k$ takes part in cores $H_k^u \subseteq H_{k-1}^u \subseteq H_{k-2}^u, \ldots, \subseteq H_1^u$ by Observation 2.*

Building the core decomposition of a graph G is basically the same problem as finding the set of *maximum* k-cores of all vertices in G. The following corollary shows that given the K values of all vertices, k-core of any vertex can be found for any k.

Algorithm 1: FINDKCOREDECOMPOSITION($G(V, E)$)

```
Data: G: the graph
Compute δ_G(v) (i.e., the degree) for all vertices v ∈ V
Order the set of vertices v ∈ V in increasing order of δ_G(v)
for each v ∈ V do
    K(v) ← δ_G(v)
    for each (v, w) ∈ E do
        if δ_G(w) > δ_G(v) then
            δ_G(w) ← δ_G(w) − 1
    Reorder the rest of V accordingly
return K
```

COROLLARY 1. *Given $K(v)$ for all vertices $v \in G$ and assuming $K(u) \geq k$, the k-core of a vertex u, denoted by H_k^u, consists of u as well as any vertex w that has $K(w) \geq k$ and is reachable from u via a path P such that $\forall_{v \in P}$, $K(v) \geq k$. H_k^u can be found by traversing G starting at u and including each traversed vertex w to H_k^u if $K(w) \geq k$.*

Intuitively, in Corollary 1, all the traversed vertices are in H_k^u due to maximality property of k-cores, and all the vertices in H_k^u are traversed due to the connectivity property of k-cores, both based on Definition 1. Thus, the problem of maintaining the k-core decomposition of a graph is equivalent to the problem of maintaining its K values, by Corollary 1. The algorithm for constructing the k-core decomposition of a graph from scratch is based on the following property [26]: To find the k-cores of a graph, all vertices of degree less than k and their adjacent edges are recursively deleted. We provide its pseudo-code in Algorithm 1 for completeness.

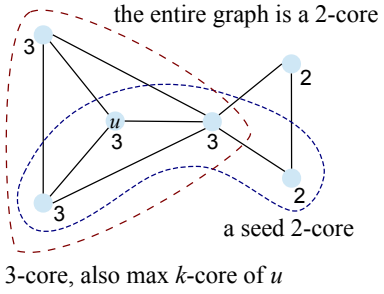

Figure 1: Illustration of k-core concepts.

Figure 1 illustrates concepts related to k-core decomposition. In the sample graph, we see the K values of the vertices printed next to them, which is simply the k-core decomposition of the graph. We see a vertex labeled u. A seed 2-core that contains u is also shown. Moreover, the entire graph is the 2-core of u, i.e., $G = H_2^u$. The figure further shows a 3-core of u, that is H_3^u, which happens to be its max-k core, that is $H_3^u = H^u$. Note that $H_3^u \subseteq H_2^u$.

3. THEORETICAL FINDINGS

In this section, we introduce our theoretical findings. These results facilitate incremental maintenance of the k-core decomposition of a graph. Since our incremental algorithms rely on finding a subgraph and processing it, we prove a number of theorems that can be used to find a small subgraph that is guaranteed to contain all the vertices whose K values change after an update.

OBSERVATION 4. *Let $G = (V, E)$ be a graph and $u, v \in V$. If there is an edge $e \in E$ between u and v and if $K(u) > K(v)$, then $e \notin H^u$ and $e \in H^v$, by Corollary 1.*

THEOREM 1. *If an edge is inserted to or removed from graph $G = (V, E)$, then the K value of vertex $u \in V$ can change by at most* 1.

PROOF. We first prove the insertion case. Assume that after the insertion of edge e, $K(u) = m$ is increased by n to $K_+(u) = m + n$, where $n > 1$. Let us denote the max k-core of u after the insertion as H^u_+, and before insertion as H^u. It must be true that $e \in H^u_+$, as otherwise H^u_+ forms a seed $m + n$-core before the insertion as well, which is a contradiction. Let $Z = H^u_+ \setminus e$. If Z is not disconnected, then it must form an $m+n-1$-core, since the degree of its vertices can decrease by at most 1 due to removal of a single edge. This leads to a contradiction since $m + n - 1 > m$ and H^u is maximal. In the disconnection case, each one of the resulting two connected components must be a seed $m + n - 1$-core as well, since the degree of a vertex can reduce by at most one in each component. Furthermore, since e is the only edge between the two disconnected components, the vertices must still have at least $m + n - 1$ neighbors in their respective components. One of these components must contain u, which is again contradiction

Next, we prove the removal case. Assume $K(u)$ is decreased by n after edge e is removed, where $n > 1$. Adding e back to the graph increases the K value of u by n, which is not possible, as shown in the first part of the proof (i.e., a contradiction). □

THEOREM 2. *If an edge (u, v) is inserted to or removed from $G = (V, E)$, where $u, v \in V$ and $K(u) < K(v)$, then $K(v)$ cannot change.*

PROOF. We first prove the insertion case. Assume that $K(v) = n$ increases and so becomes $K_+(v) = n + 1$ by Theorem 1. Then we have $e \in H^v_+$ and consequently $u \in H^v_+$. However, $K(u) < n$ before insertion and $K_+(u)$ can be at most n after insertion (Theorem 1), implying that u cannot be in a seed $n + 1$-core, i.e., a contradiction.

For the removal case, assume that $K(v) = n$ decreases and becomes $K_-(v) = n - 1$ by Theorem 1. Inserting (u, v) back to the graph should increase the K value of v to $K(v) = n$. We must also have $e \in H^v$ and thus $u \in H^v$. But this is a contradiction due to Observation 4, since $K(u) < K(v)$ and $u \notin H^v$. □

From Theorem 2, we can say that when an edge (u, v) is inserted into or removed from the graph, $K(u)$ can change by at most 1 if $K(u) \leq K(v)$, or stay the same.

THEOREM 3. *If an edge (u, v) is inserted into $G = (V, E)$, where $u, v \in V$, then all of the vertices whose K values have changed should form a connected subgraph $G' \subset G \cup (u, v)$. Similarly, if an edge (u, v) is removed from $G = (V, E)$, where $u, v \in V$, then all the vertices whose K values have changed should form a connected subgraph $G'' \subset G$.*

PROOF. We prove the insertion case first. Assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge insertion, only one of these subgraphs, say S_1, can have a vertex who gets a new neighbor in G. Then S_2 does not have any vertex that has its degree changed. This is a contradiction, because if a vertex has its K value increased, then it must have either gained a new neighbor (increased degree) or at least one of its existing neighbors must have its K value increased. Applying this recursively, we must reach a vertex whose K value is increased due to gaining a new neighbor. However, for S_2, there is no such vertex since only reachable vertices whose K values have increased are in S_2 and none of them have their degrees changed.

For the removal case, assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge removal, only one of these subgraphs, say S_1, can have a vertex who loses a neighbor in G. Then S_2 does not have any vertex that has its degree changed. This is a contradiction, because if a vertex has its K value decreased, then it must have either lost a neighbor (decreased degree) or at least one of its existing neighbors must have its K value decreased. Applying this recursively, we must reach a vertex whose K value is decreased due to losing an existing neighbor. However, for $S2$, there is no such vertex since only vertices that can be reached and whose K value has decreased are in $S2$ and none of them have their degrees changed. □

THEOREM 4. *Given a graph $G = (V, E)$, if an edge (u, v) is inserted (removed) and $K(u) \leq K(v)$, then only the vertices $w \in V$, that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with K values equal to $K(u)$, **may** have their K values incremented (decremented).*

PROOF. Before looking at the insertion and removal, we note that if the K value of any vertex in G increases (decreases) due to the insertion (removal) of (u, v), then $K(u)$ must have increased (decreased) as well. This follows from the recursive argument in Theorem 3, as otherwise none of the vertices that have their K values changed will have their degree changed.

For the insertion case, we first prove that for a vertex $w \in V$ such that $K(w) \neq K(u)$, $K(w) = m$ cannot change. We consider two cases: (i) where $K(w) < K(u)$ and (ii) where $K(w) > K(u)$.

For the $K(w) > K(u)$ case, assume $K(w)$ increases ($K_+(w) = m + 1$). We must have $(u, v) \in H^w_+$, as otherwise H^w would not be a max m-core before insertion. However, this is not possible since $K_+(w) > K_+(u)$, i.e., a contradiction.

For the $K(w) < K(u)$ case, assume $K(w)$ increases ($K_+(w) = m + 1$). Then we have $(u, v) \in H^w_+$, as otherwise H^w would not be a max m-core before insertion. We know that $m + 1 \leq K(u) \leq K(v)$, which implies $K_+(w) < K_+(u) \leq K_+(v)$. Removing (u, v) from H^w_+ decreases the degrees of u and v by one, which can reduce their K value to at least $m + 1$. This means $H^w_+ \setminus (u, v)$ is a seed $m+1$-core before the insertion, which is a contradiction.

We proved that only vertices with $K(w) = K(u)$, say $L \subseteq V$, **can** have their K values incremented. Furthermore, we know that all those vertices form a connected subgraph (Theorem 3). Since we have $u \in L$ as well, the insertion proof is complete.

We use similar arguments for the removal case. Again, we consider two cases.

For the $K(w) < K(u)$ case, assume $K(w)$ decreases ($K_-(w) = m - 1$). Say that we insert (u, v) back into the graph. The K value of w cannot increase in this case since $K_-(w) < K_-(u)$, and this is a contradiction, as shown in insertion part above.

For the $K(w) > K(u)$ case, assume $K(w)$ decreases ($K_-(w) = m-1$). We know that $(u, v) \notin H^w$ since $u \notin H^w$ due to $K(u) < K(w)$. Thus, H^w is still an m-core after the removal, creating a contradiction.

We proved that only the vertices that have $K(w) = K(u)$, say $L \subseteq V$, **may** have their K values decremented. Furthermore, by Theorem 3, we know that all those vertices form a connected subgraph. Since we have $u \in L$, the removal proof is complete. □

Summary. In this Section, we showed that if an edge (u, v) is inserted into/removed from a graph, then the K value of u can change only if $K(u) \leq K(v)$. Let us call u the root. In case $K(u) = K(v)$, then either u or v is taken as the root. In addition, we showed that any vertex that may have its K value updated must

Algorithm 2: FINDSUBCORE($G(V, E), K(), u$)

```
Data: G: the graph, K: max-k values, u: the vertex
H(V', E') ← empty graph; Q ← empty queue
cd[v] = 0; visited[v] = false, ∀v ∈ V                    ▷ Lazy init
k ← K(u)                                  ▷ Remember K value of the root
Q.push(u); visited[u] ← true
while not Q.empty() do
    v ← Q.pop(); V'.push(v)
    for each (v, w) ∈ E do
        if K(w) ≥ k then
            cd[v] ← cd[v] + 1
            if K(w) = k and not visited[w] then
                Q.push(w); E'.push((v, w))
                visited[w] ← true
return H and cd
```

Algorithm 3: SUBCORE: INSERTEDGE($G(V, E), K(), u_1, u_2$)

```
Data: G: the graph, K: max-k values, (u1, u2): inserted edge
r ← u1                                                   ▷ Set the root
if K(u2) < K(u1) then r ← u2
G ← G ∪ (u1, u2)                                ▷ Add the edge into G
H, cd ← FINDSUBCORE(G, K, r)                             ▷ Find subcore
▷ Now update the K values of the vertices in H
k ← K(r)                                  ▷ Remember K value of the root
Sort cd values in increasing order (using bucket sort)
for each v ∈ H in order do
    if cd[v] ≤ k then                  ▷ Cannot be part of a k+1-core
        for each (v, w) ∈ H do
            if cd[w] > cd[v] then
                cd[w] ← cd[w] − 1
                Reorder cd values accordingly
    else                    ▷ All remaining vertices become part of k+1-core
        for each w ∈ H do
            K(w) ← k + 1
        break
```

have a K value that is equal to that of the root, and must be connected to the root via a path that contains only the vertices that have the same K value. We rely on these results in the next section.

4. INCREMENTAL ALGORITHMS

In this section, we introduce three algorithms to incrementally maintain the K values of vertices when a single edge is inserted or removed. The subcore (Section 4.1) and purecore (Section 4.2) algorithms are basic applications of the theoretical results given in the previous section, are easy to implement, and form a baseline for evaluating the performance of the traversal algorithm (Section 4.3). The traversal algorithm relies on additional ideas that aggressively cut the search space, but is more involved than the earlier two.

4.1 The Subcore Algorithm

Our first algorithm for maintaining the K values of vertices when a single edge is inserted or removed is based on Theorem 4. We define a new subgraph as follows:

DEFINITION 3. *Given a graph $G = (V, E)$ and a vertex $u \in V$, the* subcore *of u, also denoted as S_u, is a set of vertices $w \in V$ that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with their K values equal to $K(u)$.*

Given a graph $G = (V, E)$ and the K values of all $w \in V$, if an edge (u_1, u_2) is inserted to E, Algorithm 3 updates the K values. Similarly, if an edge (u_1, u_2) is removed from E, Algorithm 4 updates the K values. Both algorithms make use of Definition 3.

The basic idea is to locate the subcore of the root vertex and apply a process very similar to Algorithm 1 on the subcore. Algorithm 2 provides the pseudo code for finding the subcore. To find the subcore, we perform a BFS traversal and collect all vertices reachable from the root through vertices having the same K value as the root. During this process, we also collect the *core degree* (`cd`) values for each vertex in the subcore. The core degree of a vertex is its degree in its max-core and determines if a vertex can change its K value or not. As a result, the `cd` of a vertex simply counts the number of its neighbors with a K value equal to or greater than the K value of the root. Core degrees help us eliminate vertices that cannot be part of a $k+1$ core, where k is the core value of the root. In particular, if the core degree is not larger than k, we can eliminate the vertex from consideration. Once it is eliminated, it results in decrementing the core degree values of its neighbors in the subcore and the process can be repeated. Similar to Algorithm 1, this has to be performed in increasing order of the core degree values.

Algorithm 3 shows how the subcore and the `cd` values are used to update the K values on an edge insertion. We order the `cd` values of the vertices in the subcore in increasing order. At each step, we pick the unprocessed vertex with the smallest `cd` value from the subcore. If it has a `cd` value less than or equal to the root's K value, say k, then it cannot be part of a $k+1$-core. Thus, for each of its neighbors in the subcore that have a higher `cd`, we decrement the neighbor's `cd` by 1, since the vertex being processed cannot be part of a higher core. We reorder the remaining vertices based on their updated `cd` values. Otherwise, that is if the current vertex has a `cd` value larger than k, all remaining vertices must also have their `cd` values larger than k, which means we can form a seed $k+1$ core with them. We increment their K values, completing the insertion.

Algorithm 4 shows how the subcore and the `cd` values are used to update the K values in the case of a removal. Unlike Algorithm 3, here we need to perform two subcore searches when the K values of the vertices incident upon the removed edge are the same, since the removal separates them. Once we locate the subcore, the process is very similar to that of the insertion. We pick the unprocessed vertex with the smallest `cd` value from the subcore and if it has a `cd` value less than the K value of the root, say k, then it cannot be part of a k-core anymore. As a result, we decrement its K value and for each of its neighbors in the subcore that have a higher `cd`, we decrement the neighbor's `cd` by one, since the vertex currently being processed cannot be part of a higher core. After this, we reorder the remaining vertices based on their `cd` values. Otherwise, if the current vertex has a `cd` value larger than or equal to k, then all remaining vertices must also have their `cd` values larger than or equal to k, which means that we can still form a seed k core with them. Thus, we stop processing and complete the removal.

4.2 The Purecore Algorithm

In Section 4.1, the subcore algorithm relied only on the K values of the vertices to locate a small subgraph that contains all the vertices that can have their K values changed. In this section, we look at the *purecore* algorithm that takes advantage of additional information about each vertex, so that a smaller set of candidate vertices can be located, reducing the overall cost of the algorithm. For this purpose, we define the *maximum-core degree* of a vertex.

DEFINITION 4. *The maximum-core degree of a vertex u, denoted as $MCD(u)$, is defined as the number of u's neighbors, w, such that $K(u) \leq K(w)$.*

The maximum-core degree of a vertex differs from the core degree of a vertex by the fact that it is not defined in terms of the root vertex of an insertion. If the MCD value of a vertex is not greater than its K value, and no new adjacent edge is inserted, then there is

Algorithm 4: SUBCORE:REMOVEEDGE($G(V,E), K(), u_1, u_2$)

Data: G: the graph, K: max-k values, (u_1, u_2): inserted edge
$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$
$G \leftarrow G \setminus (u_1, u_2)$ ▷ Remove the edge from G
if $K(u_1) \neq K(u_2)$ **then**
  H, cd $\leftarrow$ FINDSUBCORE(G, K, r) ▷ Find subcore
else
  H_1, cd$_1$ $\leftarrow$ FINDSUBCORE(G, K, u_1) ▷ Find subcore of u_1
  H_2, cd$_2$ $\leftarrow$ FINDSUBCORE(G, K, u_2) ▷ Find subcore of u_2
  $H \leftarrow H_1 \cup H_2$; cd $\leftarrow$ cd$_1 \cup$ cd$_2$
▷ Now update the K values of the vertices in H
$k \leftarrow K(r)$ ▷ Remember K value of the root
Sort cd values in increasing order (using bucket sort)
for each $v \in H$ *in order* **do**
  if cd$[v] < k$ **then** ▷ Cannot be part of a k-core anymore
    $K(v) \leftarrow k - 1$
    for each $(w, v) \in H$ **do**
      if cd$[w] >$ cd$[v]$ **then**
        cd$[w] \leftarrow$ cd$[w] - 1$
        Reorder cd values accordingly
  else break; ▷ All remaining vertices still in a k-core

no way for this vertex to increment its K value, because the number of neighbor vertices in a higher core will not be enough. Therefore, it is used to test whether a vertex can increment its K value or not, upon a new edge insertion.

OBSERVATION 5. *For a given graph $G = (V, E)$ and a vertex $u \in V$, $MCD(u) \geq K(u)$.*

The observation follows simply from the definition of k-core, since $MCD(u) < K(u)$ would mean u cannot participate in a k-core with $K(u) = k$, leading to a contradiction. Note that $MCD(u)$ is simply an upper bound on $K(u)$.

We reduce the *subcore*, described in Definition 3, to a *purecore* by putting an extra condition regarding MCD values. The basic idea is that, if a vertex in the subcore does not have a MCD value greater than the K value of the root, it means that the vertex does not have enough neighbors that can participate in a higher core.

DEFINITION 5. *Given a graph $G = (V, E)$ and a vertex $u \in V$, the* purecore *of u, denoted as P_u, is the set of vertices $w \in V$ that have $K(w) = K(u)$ and $MCD(w) > K(u)$, and are reachable from u via a path that consists of vertices with K values equal to $K(u)$ and MCD values greater than $K(u)$.*

Algorithm 5 finds the purecore P_u of a vertex u.

THEOREM 5. *Given a graph $G = (V, E)$, if an edge (u, v) is inserted and $K(u) \leq K(v)$, then only the vertices $w \in P_u$* ***may*** *have their K values incremented.*

PROOF. When an edge (u, v) is inserted to the graph and $K(u) \leq K(v)$, then the K value of a vertex $w \in S_u$, where $w \neq u$, cannot increment if $MCD(w) = K(w)$. Assume $K(w)$ increments, then $MCD(w)$ has to increment as well, and for this to happen either w should get a new neighbor, which is not possible since $w \neq u$, or some of its neighbors should have their K values decreased, which is not possible as no edges were removed. □

With purecore, the algorithm to update the K values of vertices, when edge (u, v) is inserted, is the same as Algorithm 3, except that Algorithm 5 (FINDPURECORE) is used in place of Algorithm 2 (FINDSUBCORE).

When an edge (u, v) is removed from the graph and $K(u) \leq K(v)$, then the K value of any vertex $w \in S_u$ can potentially

Algorithm 5: FINDPURECORE($G(V,E), K(), u$)

Data: G: the graph, K: max-k values, u: the vertex
$H(V', E') \leftarrow$ empty graph; $Q \leftarrow$ empty queue
cd$[v] = 0$; visited$[v] =$ **false**, $\forall v \in V$ ▷ Lazy init
$k \leftarrow K(u)$ ▷ Remember K value of the root
Q.push(u); visited$[u] \leftarrow$ **true**
while not Q.empty() **do**
  $v \leftarrow Q$.pop(); V'.push(v)
  for each $(v, w) \in E$ **do**
    if $K(w) > k$ **or** ($K(w) = k$ **and**
        MCDEGREE(G, K, w)$> k$) **then**
      cd$[v] \leftarrow$ cd$[v] + 1$
      if $K(w) = k$ **and not** visited$[w]$ **then**
        Q.push(w); E'.push((v, w));
        visited$[w] \leftarrow$ **true**
return H and cd

decrement. Note that $MCD(w)$ can decrease if either w loses a neighbor, which is the case for u, or K value of some neighbor of w decrements, which is the case for neighbors of u when $K(u)$ decrements. As a result, for removal, we do not rely on the purecore.

4.3 The Traversal Algorithm

We now present the traversal algorithm that visits an even smaller subgraph to update the k-value decompostion. First, we introduce an optimization to speedup the computation of the MCD values and then an additional metric to further scope the search.

4.3.1 Residential Core Degrees

In Section 4.2, we find a smaller set of candidate vertices to be updated by using more information about each vertex. Using more information, such as the MCD values, requires more computation in Algorithm 5. Thus, for a vertex u, when the size of P_u is **large** and close to the size of S_u, Algorithm 5 turns out to be more expensive than Algorithm 2. To alleviate this problem, we make two types of core degree values to constantly reside in memory (i.e., *residential*). We maintain the MCD values, introduced in Definition 4, and the PCD values of vertices defined as follows.

DEFINITION 6. *The purecore degree of a vertex u, denoted as $PCD(u)$, is the number of u's neighbors, w, such that* ***either*** *$K(u) = K(w)$ and $MCD(w) > K(u)$* ***or*** *$K(u) < K(w)$.*

For a vertex v, its purecore degree $PCD(v)$ is the number of neighbors w it has that either has a higher K value than v or has the same K value but in turn has enough neighbors to potentially increase its K value (in case an insertion was made and the K values are to be updated). The PCD value of a vertex represents its potential number of neighbors in a next max-core. It is a stronger indicator than its MCD value for showing eligibility to increase the K value and also useful, because if $PCD(v) \leq k$ where k is the K value of the root, then v cannot increment its K value.

Maintaining the MCD and PCD values of vertices after each insertion and removal should be done efficiently so that unnecessary updates of those values are avoided. In general, the MCD value of a vertex is based on the K values of its neighbors, as seen from Definition 4, and the PCD value of a vertex is based on the K and MCD values of its neighbors, as described in Definition 4. Observation 6 gives a rule of thumb for MCD and PCD maintenance.

OBSERVATION 6. *For a graph $G = (V, E)$, when the K value of a vertex $u \in V$ changes, the MCD values of vertices u, v can change, where $(u, v) \in E$. When the MCD value of a vertex*

Algorithm 6: TRAVERSAL: INSERTEDGE($G(V,E), K(), u_1, u_2$)

Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD: purecore degrees, (u_1,u_2): inserted edge
$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$
$G \leftarrow G \cup (u_1, u_2)$ ▷ Add the edge into G
PREPARERCDS
▷ Perform a traversal over vertices that have root's K value, while evicting the ones that cannot be a part of a k+1-core
$S \leftarrow$ empty stack ▷ To perform DFS
`visited`$[v] =$ **false**, $\forall v \in V$ ▷ To perform DFS (lazy init)
`evicted`$[v] =$ **false**, $\forall v \in V$ ▷ To remember evicted vert. (lazy init)
`cd`$[v] = 0, \forall v \in V$ ▷ To find vertices to be evicted (lazy init)
$k \leftarrow K(r)$ ▷ Remember the K value of the root
`cd`$[r] \leftarrow PCD(r)$ ▷ Set `cd` of root
S.push(r); `visited`$[r] \leftarrow$ **true**
while not S.empty() **do** ▷ Do a DFS traversal
  $v \leftarrow S$.pop()
1 **if** `cd`$[v] > k$ **then** ▷ Vertex is currently part of a k+1-core
    for each $(v, w) \in E$ **do**
      ▷ Neighbouring vertex currently part of a k+1-core
      if $K(w) = k$ **and** $MCD(w) > k$ **and not** `visited`$[w]$ **then**
        S.push(w); `visited`$[w] \leftarrow$ **true**
        ▷ Use + as `cd`$[w]$ may be < 0 due to evictions
        `cd`$[w] \leftarrow$ `cd`$[w] + PCD(w)$
    else ▷ Vertex cannot be part of a k+1-core
      if not `evicted`$[v]$ **then** ▷ Recursively perform eviction
        PROPAGATEEVICTION($G, K,$ `cd`, `evicted`, k, v)
for each v s.t. `visited`$[v]$ **do** ▷ Find visited vertices
  if not `evicted`$[v]$ **then** ▷ If not evicted as well
    $K(v) \leftarrow K(v) + 1$ ▷ The vertex is part of a k+1-core
RECOMPUTERCDS

Algorithm 7: PROPAGATEEVICTION($G(V,E), K(),$ `cd`$[],$ `evicted`$[], k, v$)

Data: G: the graph, K: max-k values, `cd`: cd values, `evicted`: evicted values, k: max-k of root, v: evicted vertex
`evicted`$[v] \leftarrow$ **true**
for each $(v, w) \in E$ **do**
  if $K(w) = k$ **then**
1   `cd`$[w] \leftarrow$ `cd`$[w] - 1$
2   **if** `cd`$[w] = k$ **and not** `evicted`$[w]$ **then**
      PROPAGATEEVICTION($G, K,$ `cd`, `evicted`, k, v)

$u \in V$ changes, the PCD values of vertices v can change, where $(u, v) \in E$. As a result, when the K value of a vertex $u \in V$ changes, the PCD values of vertices u, v, w can change, where $(u, v), (v, w) \in E$.

In summary, the observation says that a K value update can result in changes in the MCD values within the 1-hop neighborhood of the vertex, whereas changes in the PCD values can happen within the 2-hop neighborhood.

Based on Observation 6, when an edge (u, v) is inserted into or removed from a graph $G = (V, E)$, we first recompute the MCD value of the root vertex u and the PCD values of its neighbors. Next, we apply the algorithm to update the K values of vertices. Last, we do the following two operations to adjust the MCD and PCD values:

- Recomputing the MCD values of vertices $w, x \in V$ for which $K(w)$ is updated and $(w, x) \in E$.
- Recomputing the PCD values of vertices $w, x, y \in V$ for which $K(w)$ is updated and $(w, x), (x, y) \in E$

Further shortcuts are possible, based on the K and MCD values of the updated vertices, to minimize the number of MCD and PCD re-computations. We skip the details for brevity.

4.3.2 Root Awareness

So far, in all our incremental algorithms, we first find a subgraph and its corresponding `cd` values by a BFS traversal (phase 1). In a second phase, we process that subgraph by reordering the vertices with respect to their `cd` values and remove the vertex with the minimum `cd` at each step. Traversing the subgraph and computing the `cd` values should be done prior to the second phase, since we need all the vertex degrees in the subgraph. However, Theorem 4 says that if the K value of some vertex changes, then the K value of at least one extremity of the inserted/removed edge, named as the root vertex (say u), must change. For the insertion algorithm, this fact suggests a *root-aware* approach, in which all vertices know whether the root still has a chance to change its K value. Additional operations are avoided once the algorithm detects that $PCD(u) \leq K(u)$, i.e., u cannot increment its K value. This condition implies that there is no chance for the root to increase its K value. We achieve this root-aware approach by applying a Depth-First Search (DFS) with an *eviction* mechanism, where the vertices $v \in V$ are evicted if $PCD(v) \leq K(v)$. By doing that, we combine phases 1 and 2. For the removal algorithm, being root-aware does not bring improvement, since a similar shortcut is already applied in Algorithm 4.

This root-aware approach does not need the `cd` values of all the vertices in the subgraph. As a result, we create the `cd` values for each vertex *on-the-fly* during DFS, avoiding the first phase of our previous algorithms completely. We leverage the *residential core degrees*, introduced in Section 4.3.1, to speed up the creation of `cd` values. On-the-fly creation of `cd` values makes the insertion algorithm more efficient. The root-aware approach does not result in any improvement to the removal algorithm. Still, creating `cd` values on-the-fly avoids the need to execute phase 1.

Algorithm 6 updates the K values of vertices by utilizing Algorithm 7, when edge (u, v) is inserted into the graph $G = (V, E)$.

THEOREM 6. *Algorithm 6 finds the vertices whose K values needs to be updated.*

PROOF. First, we prove that after an edge is inserted, if $PCD(u) \leq K(u)$ for a vertex $u \in V$, then it cannot increase its K value as shown in lines labeled 1 and 2 in Algorithms 6 and 7, respectively. Assume it does and say that $k = K(u)$. Then, after $K(u)$ increases, u must have at least $k + 1$ neighbors with greater or equal K value, by Observation 5. However, at most k neighbors of u can have their K values greater than or equal to k after $K(u)$ increases, since $PCD(u) \leq K(u)$ before $K(u)$ is increased, i.e., a contradiction.

Second, we prove that if $PCD(u) \leq K(u)$, where u is the visited vertex, then $PCD(w)$ must be decremented as shown in line labeled 1 in Algorithm 7, where w is a neighbor of u having K value of $K(u)$. Assume that $PCD(w)$ is not decremented. Then u is supposed to be in the max-core of w, if w increases its K value. However, u cannot be in the max-core of w, since it cannot increase its K value as proved in the first paragraph of proof, contradiction.

We traverse the graph starting from the root and evict the vertices as shown in above proofs. Non-evicted and traversed vertices increment their K values at the end of the algorithm. □

In Algorithm 6, we start with preparing residential core degrees as explained in Section 4.3.1. Then we do a DFS starting from the root, say r, and at each step we pop the vertex v from the top of

the stack and push some of its neighbors, say w, into the stack, if v and w are candidates to be in a $k+1$-core, where $k = K(r)$. If v cannot be in a $k+1$-core, then we mark it as evicted and initiate a recursive eviction from v. In a recursive eviction, the cd values of vertices x are decremented, for $(v, x) \in E$ and $K(x) = k$. If the cd value of x turns out to be equal to k and x is not already marked as evicted, then we start another eviction from x. When DFS finishes, we increment the K values of all vertices that were visited but not evicted. Last, we adjust the residential core degrees as discussed in Section 4.3.1.

The edge removal using the traversal algorithm employs a similar on-the-fly updating of the `cd` values. Again we start with preparing residential core degrees as explained in Section 4.3.1. If the `cd` value of v turns out to be below its K value (i.e., K needs to be decremented), we perform a recursive decrement operation starting from v. In recursive decrement, we decrement $K(v)$ and the `cd` values of vertices w, where $(v, w) \in E$ and $K(w) = k$ and k is the K value of the root. If w gets a smaller `cd` value than k and $K(w)$ has not decremented yet, then we start another recursive decrement from w. When the recursion completes, we adjust the residential core degrees as discussed in Section 4.3.1.

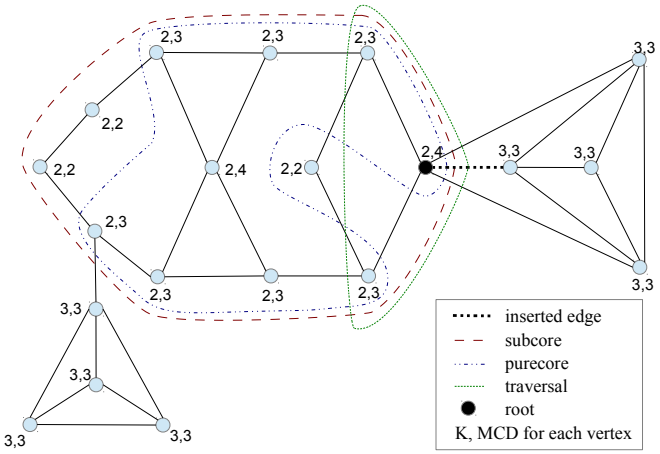

Figure 2: Illustration of the vertices visited by the subcore, purecore, and the traversal algorithms.

4.4 Illustrative Example

Figure 2 illustrates the subcore, purecore, and traversal algorithms using a sample graph. The edge drawn using a dashed bold line is the one that is being inserted into the graph. The vertex shown in black is the root vertex. The graph shows the K values and the MCD values for each vertex before the insertion. The set of vertices visited by each one of the subcore, purecore, and the traversal algorithms, for the purpose of updating the K values, is shown in the figure. The subcore algorithm visits the vertices with K value of 2, which are reachable from the root. The purecore algorithm visits the vertices with K value of 2 and MCD value of greater than 2 that are reachable from the root.

The traversal algorithm starts by updating the MCD value of the root to 5, due to the new edge. Then, DFS starts and pushes the root to the stack. When the root is popped from the stack, its two neighbors with (K, MCD) values of (2, 3) are pushed to the stack (MCD values greater than K value of the root, indicating that they can potentially be part of a larger core). Say that those vertices are x at the top and y at the bottom in Figure 2. Based on Definition 6, the `cd` values of x and y are updated to 2 since their PCD values are 2. After that, we move to the next iteration, and pop vertex x from the stack. The `cd` value of x is 2, which is not greater than the K value of the root. This means that it cannot participate in a higher core. As a result, no neighbors of x are visited and PROPAGATEEVICTION is initiated for x. In PROPAGATEEVICTION, x is evicted and the `cd` values of all neighbors of x are decremented, since all neighbors have a K value of 2 (same as root). Furthermore, PROPAGATEEVICTION is not initiated for any neighbor of x, since the `cd` value of the root (one of x's neighbors) becomes 4, and the `cd` value of other two neighbors of x become -1, all of which are different than the K value of the root.

In the next step, the DFS pops vertex y from the stack. Similar to x, the `cd` value of y is 2, which is not greater than the K value of the root. As a result, no neighbors of y are visited and PROPAGATEEVICTION is initiated for y. In PROPAGATEEVICTION, y is evicted and the `cd` values of all neighbors of y are decremented, since they have a K value of 2 (same as root). Furthermore, PROPAGATEEVICTION is not initiated for any neighbor of y, since the `cd` value of y's neighbors differ from the K value of the root. After these operations, the stack is empty, and the only vertex that is visited but not evicted is the root. As a result, the K value of the root is incremented. As the last step, the MCD and PCD values of vertices are updated as explained in Section 4.3.1.

We can easily see that the set of vertices visited by the subcore algorithm is larger than that of the purecore algorithm, whereas the traversal algorithm visits the smallest number of vertices compared to the other two.

5. IMPLEMENTATION

In this section we provide details about efficient implementations of the incremental algorithms presented. In particular, we discuss two main issues: the lazy initialization of arrays used in the algorithms, and the repeated sorting of the `cd` arrays.

5.1 Lazy arrays

The non-incremental algorithms for computing the k-core decomposition perform work that is proportional to the size of the graph. As a result, our incremental algorithms should avoid any operation that requires work in the order of the size of the graph. However, several of our algorithms include arrays like `visited`, `evicted`, `cd`, etc., that are initialized to a default value and accessed using vertex indices. For these, we use lazy arrays to avoid allocations and initializations in the order of the graph size.

A lazy array employs a hash map based data structure to implement a sparse array. For a given vertex, if its value is not currently being stored in the hash map, it is assumed to have the designated default value. When a different value for the vertex needs to be stored, the entry for it is created in the hash map.

Since hash maps provide constant lookup time, using lazy arrays achieves significant speedup when the number of vertices visited by the incremental algorithms is smaller than the graph size. On the other hand, when the number of vertices visited gets large, relative to the graph size, lazy arrays start performing worse, since the constant overhead of accessing a data item in a hash map is significantly higher than that of regular arrays.

Given that our algorithms locate a small subset of vertices for updating the k-core decomposition of a graph, the use of lazy arrays is almost always beneficial. For graphs that have very large max-k cores, relative to the graph size, (which we show to be an uncommon occurrence in practice) an implementation of lazy arrays that switches to a dense representation when the occupation percentage of the array gets larger can be an effective solution, even though we do not implement that variation in this study.

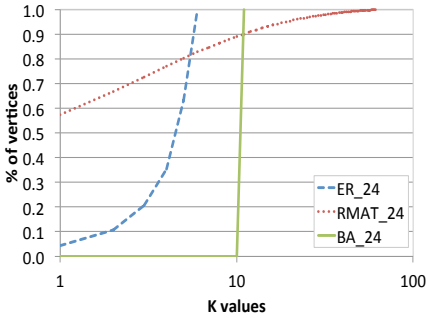

Figure 3: Cumulative K value distribution for synthetic graphs.

Graph file	Number of vertices	Number of edges	Maximum degree	Average degree	Max k
caidaRouterLevel	192,244	609,066	1,071	6.336	32
eu-2005	862,664	16,138,468	68,963	37.415	388
citationCiteseer	268,495	1,156,647	1,318	8.616	15
coAuthorsCiteseer	227,320	814,134	1,372	7.163	86
coAuthorsDBLP	299,067	977,676	336	6.538	114
coPapersCiteseer	434,102	16,036,720	1,188	73.885	844
cond-mat	16,726	47,594	107	5.691	17
power	4,941	6,594	19	2.669	5
protein-interaction-1	9,673	37,081	270	7.667	14

Table 1: Real-world graph datasets and their properties.

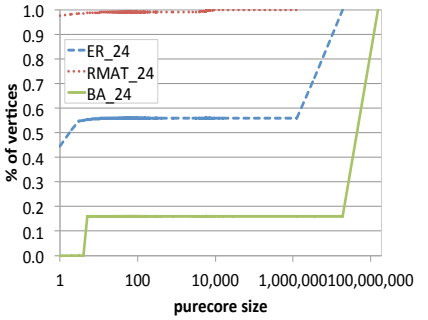

Figure 4: Cumulative purecore size distribution for synthetic graphs.

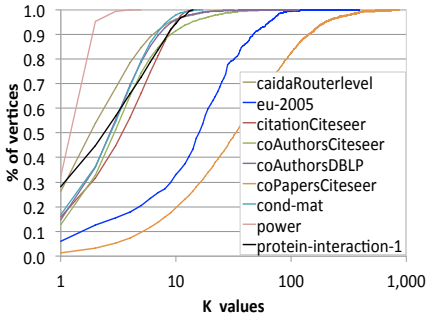

Figure 5: Cumulative K value distribution for real-world graphs.

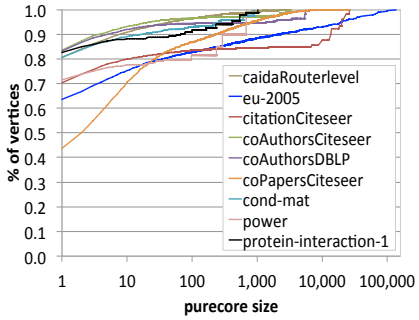

Figure 6: Cumulative purecore size distribution for real-world graphs.

5.2 Bucket sort

Several of our algorithms require reordering the set of unprocessed vertices in a subgraph (such as a subcore or a purecore) based on their `cd` values. In the worst case this subgraph could be as large as the graph itself (again, this is uncommon in real-world graphs). To perform this re-sorting efficiently, we use bucket sort. Note that the `cd` values have a very small range, and thus bucket sort not only provides $O(N)$ sort time for the initial sort (where N is the subcore or purecore size), but it also enables $O(1)$ updates when a vertex changes its `cd` value (in our case the values only decrease). We use a bucket data structure that relies on linked lists for storing its bucket contents and on a hash map to quickly locate the link list entry of any given vertex.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate how the proposed algorithms behave under different scenarios. The first set of experiments shows the scalability of our best performing algorithm by studying its runtime performance as the size of the synthetic datasets increases. The second set of experiments compare the performance of our incremental algorithms with respect to each other on real datasets. The third experiment investigates the performance variation depending on the K values of u and v, when an edge (u, v) is inserted/removed.

Our algorithms are implemented in C++ and compiled with `gcc 4.4.4` at -O2 optimization level. All experiments are executed sequentially on a Linux operating system running on a machine with two Intel Xeon E5520 2.27GHz CPUs, with 48GB of RAM.

6.1 Datasets

Our dataset includes synthetic and real graphs. For synthetic graphs, we use the SNAP library [27] to generate networks following three different models. The first is the Erdös-Renyi model, which generates random graphs [13]. We used $p = 0.1$ to put an edge among two specified vertices and we specify $|E|/|V|$ as 8. The second is the Barabasi-Albert preferential attachment model [5], which follows a power law for the vertex degree distributions. We configure it such that each new vertex added by the generation algorithm creates 11 edges. The third model, generated with SNAP's R-MAT generator [8], follows a power law vertex degree distribution and also exhibits small world properties. We set the partition probabilities as $[0.45, 0.25, 0.20, 0.10]$, to approximate the k-core distribution of real citation graphs in our dataset.

Figures 3 and 4 show the cumulative distribution of K values and purecore sizes (i.e., number of edges of the purecore subgraph of each vertex in the graph) for the synthetic datasets with 2^{24} vertices. For a graph $G = (V, E)$, we calculate the purecore of each vertex $u \in V$ by using Algorithm 5. These figures reveal the structure of the generated graphs and how it impacts the incremental k-core decomposition performance. The K value distribution is an indication of the connectivity of the graph, while the purecore size is an indication of the potential runtime of our incremental algorithms when an edge incident upon a given vertex is inserted/removed.

As shown in Figure 3, the graph based on the Barabasi-Albert model (BA_24) has 100% of its vertices with K = 11. In addition, over 80% of its vertices result in a purecore size of over 100 million vertices. These properties of the BA graphs are due to the graph generation algorithm of the BA model, where newly inserted edges are likely to connect high degree vertices. As we will see shortly, real-world graphs do not follow such properties and the figure shows that the BA model is very poor in approximating real world graphs in terms of the K value distribution. The RMAT generated graph (RMAT_24) has nearly 60% of its vertices with very low K values. As the K value increases, the percentage of vertices with that K value decreases. Furthermore, 98% of its vertices have very small purecore sizes. The ER generated graph (ER_24) has K values up to 6, and as the K value increases, the percentage of vertices with that K value also increases. The latter behavior is unlike the RMAT generated graph. As we will see shortly, most

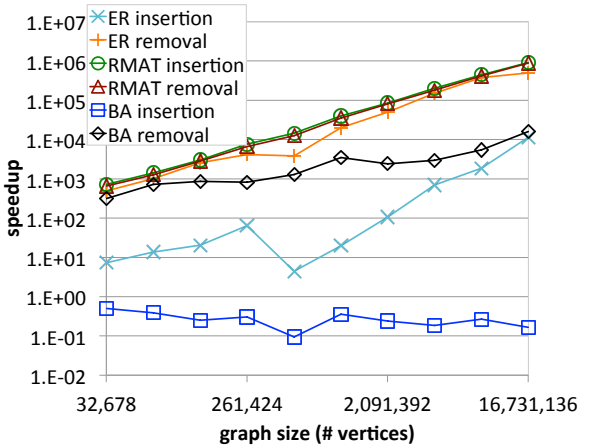

Figure 7: Speedup of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}. Removal scales better than insertion, reaching around 10^6 speedup.

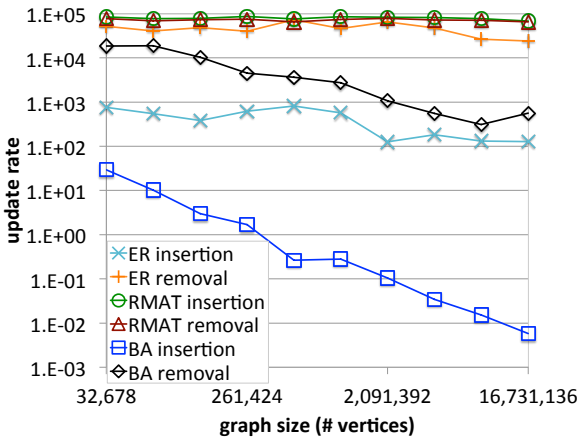

Figure 8: Update rates of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}.

real-world graphs of interest behave more closely to the RMAT generated graphs with respect to their K value distribution.

The real graphs we use are from the 10th DIMACS Graph Partitioning and Graph Clustering Implementation Challenge repository [10] and include internet router level and European domain computer network graphs (caidaRouterLevel and eu-2005), co-author and citation network graphs (citationCiteseer, coAuthorsCiteseer, coAuthorsDBLP, coPapersCiteseer), condensed matter collaboration network graphs (cond-mat), power grid network graphs (power), and protein interaction network graphs (protein-interaction-1). Table 1 provides the details about each used graph, including their vertex and edge set size, maximum and average degrees, and their maximum k value. All graphs are undirected.

Figure 5 shows the K value distribution for all graphs in Table 1. The figure shows that the vertices of both coPapersCiteseer and eu-2005 have highly variant K values. Figure 6 shows the purecore size distribution for our real datasets. The data indicates that all of the graphs have at least 80% of their vertices with corresponding purecore sizes of less than 100. This is an indication that our incremental algorithms are expected to perform well on these graphs.

As all our graphs are originally static, we emulate a streaming algorithm by considering that the whole set of edges and vertices constitute a *sliding window* snapshot. For evaluating algorithm execution, we first evict a random edge from the current graph in the window. This emulates the behavior of a full sliding window, which must open space for inserting a new data item. We then insert a new edge between two random vertices. We also evaluate worst case execution times by inserting and removing edges from vertices that have top *purecore* sizes. Such results are similar to the random insertion case, and are omitted for brevity. Note that we do not assume any specific data distribution with respect to which edges get inserted or deleted. In addition, we make no assumptions regarding edge arrival rates. Instead, we evaluate the performance of our algorithms' processing updates as fast as possible.

6.2 Scalability

In this experiment, we evaluate the performance of the traversal algorithm (Section 4.3) as the size of the synthetic graphs increase. We first report speedup numbers, which are obtained by comparing the traversal runtimes with our baseline — the non-incremental version of k-core decomposition (Algorithm 1), then present the update rates, which show the number of edge removals/insertions processed per second. Testing the algorithm under different graph sizes emulates the scenario where a streaming algorithm uses different sliding window sizes.

Figure 7 shows the speedup of our incremental insertion and removal algorithms when the number of vertices from the graph range from 2^{15} to 2^{24}. For the insertion algorithm, the RMAT graph shows the best scalability, with speedups ranging from $717\times$ to $920,000\times$ (almost 6 order of magnitude). This drastic speedup is because the K values of the vertices in the graph have high variability and majority of the vertices have very small purecore sizes, as shown in Figures 3 and 4 for the RMAT graph with size 2^{24}. Such factors result in very fast insertions. The insertion of edges into the graph following the Erdös-Renyi model (ER) show speedup ranging from $4.43\times$ to $11,500\times$. Although it also scales well with the size of the graph, the speedups are not as high as the ones observed for the RMAT graph. This behavior can be explained by the fact that the ER graph has a more uniform K value distribution when compared to the RMAT graph. Furthermore, when the graph has size of 2^{24}, over 40% of its insertions may result in touching purecores of over 1 million edges. When inserting edges into graph based on the Barabasi-Albert model (BA), our incremental algorithm is worse than the non-incremental one. As we discussed earlier, in these graphs all vertices have the same K value initially, resulting in subcore sizes that are almost equal to the graph size. In this case, the incremental algorithm does not provide any benefit on top of the base one, yet brings additional computation overheads (such as due to lazy arrays). As we will show shortly, this nature of the BA graphs are not found in real world graphs.

The removal algorithm scales for all three synthetic graphs, where the speedup ranges from $315\times$ to $885,000\times$. For the ER and BA graphs, the removal algorithm scales better than the insertion one because it has much lower cost (see Section 4.3). At large scales, we notice that the use of incremental algorithms becomes even more critical, since the cost of the baseline is linear in the size of the graph.

The scalability experiments indicate how good our incremental algorithm can perform for different graph sizes when there are k-core decomposition queries (*read queries*) interspersed with edge insertion and removal (*write queries*). Taking the RMAT graph with size of 2^{24} vertices as an example, we can see that if the write/read ratio is less than $900,000$ (the average speedup of one removal and one insertion), it is better to use the incremental algorithm than to compute the k-core decomposition from scratch after inserting new edges and removing the oldest ones from the graph

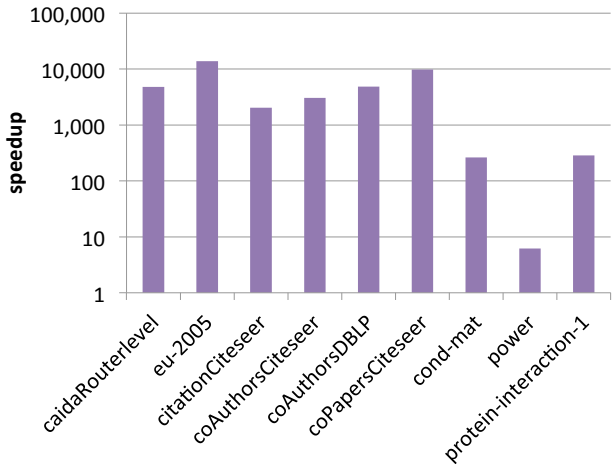

Figure 9: Subcore algorithm speedups for real datasets when compared to the baseline. Our incremental algorithm runs up to $14,000\times$ faster than the non-incremental algorithm.

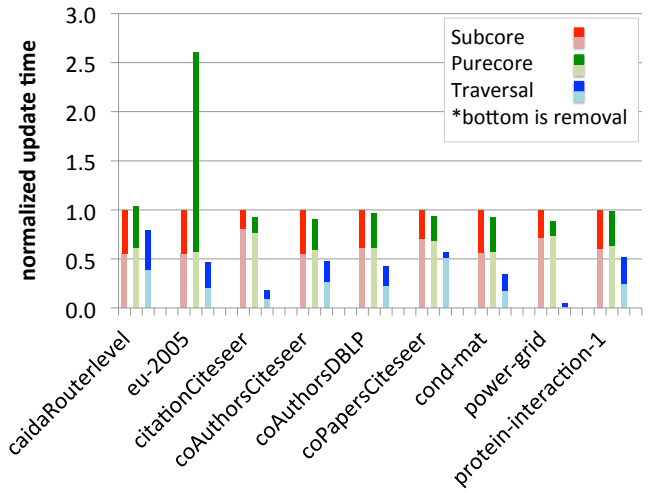

Figure 10: Average update time comparison of incremental algorithms when processing real datasets. Times are normalized by the average update time of the subcore algorithm. Traversal algorithm shows the best performance for all datasets.

(sliding window scenario).

Figure 8 shows the update rates, i.e., number of edges processed per second, for our incremental insertion and removal algorithms when the number of vertices in the graph ranges from 2^{15} to 2^{24}. For RMAT graphs, both insertion and removal rates reach up to 80,000 updates/sec and, more importantly, update rates do not change when the graph size increases. ER graphs have lower update rates for both insertion and removal. Removal rates for ER graphs stay stable as the graph size increases and insertion rates only decrease by a factor of 6 (from 761 to 127) when the graph size increases from 2^{15} to 2^{24}. For BA graphs, update rates for removal decreases from 18,500 to 310 when the graph size increases. Insertion rate has a similar decreasing behaviour with the graph size. However, the rates are much lower — starting from 28 and decreasing to 0.005 when the graph size gets bigger. The decreasing trend for the BA graphs is due to the large subcore sizes that are proportional to the graph size. Again, we will show that real world graphs do not exhibit this behavior.

6.3 Performance comparison

In this experiment, we analyze how our three incremental algorithms perform when processing one edge removal and one edge insertion (i.e., one sliding window operation) on the real datasets described in Table 1. This helps us to see whether the algorithm that is expected to give the best results, Traversal, shows the best performance for all the real datasets we have.

Figure 9 shows the performance of the subcore algorithm (Section 4.1) considering the average time taken by one graph update. The performance is shown in terms of the speedup provided by the incremental algorithm compared to the non-incremental one. The speedups vary from $6.2\times$ to $14,000\times$. The datasets in which the incremental algorithm performs the best are the eu-2005 and coPapersCiteseer. Similar to the results obtained in the synthetic graphs, the performance of the subcore algorithm benefits from the high variability in the K value distribution of the graph. The dataset in which the subcore algorithm performs the worst is power. This is because 63.19% of the vertices in the power graph have the same K value, yielding large subcore sizes.

Figure 10 shows the average update time of each algorithm normalized by the update time of the subcore algorithm. Each group of 3 columns shows the results for a given dataset. For each group, the results are displayed in the following order: subcore (Section 4.1), purecore (Section 4.2), and traversal (with residential core degrees) (Section 4.3). The stacked columns represent the update time attributed to the removal (bottom) and insertion operations (top).

The results show that the purecore algorithm can perform worse than the subcore one for some datasets (caidaRouterlevel and eu-2005) even though the purecore of a vertex is always smaller than or equal to the subcore of a vertex. This is due to the additional work performed to locate a smaller subgraph. This additional work is not always worth it if the purecore is not sufficiently small compared to the subcore. The figure also displays that the traversal algorithm shows the best performance for all datasets, being up to $20\times$ better than the subcore algorithm. The traversal algorithm shows dramatic improvement compared to subcore when processing citationCiteseer and power graphs. Our results also show that the traversal algorithm has the most efficient removal for all datasets.

Graph scale	with RCD (ratio)	without RCD
16	0.032 (%48)	0.067
18	0.175 (%52)	0.335
20	1.041 (%50)	2.047
22	4.218 (%49)	8.600
24	6.098 (%67)	8.991

Table 2: Average runtimes (secs) for one edge removal plus one edge insertion with traversal algorithm on Erdös-Renyi graphs. Ratio shows with RCD runtimes relative to without.

We also investigate the impact of Residential Core Degrees on synthetic graphs generated using the Erdös-Renyi model. Table 2 shows the average time in seconds spent for one edge removal plus one edge insertion with the traversal algorithm. For each graph, we ran the traversal algorithm with and without the Residential Core Degrees. The results show that using Residential Core Degrees provides up to %48 less runtime. The results for RMAT, not included here for brevity, show less improvement.

6.4 Performance variation

In this section, we evaluate the performance of the traversal algorithm when inserting and removing random edges into vertices with varying K values. The objective is to understand how the execution time varies as edge insertions and removals are performed on different parts of the graph with different connectivity characteristics.

For instance, performance implications of adding an edge between a vertex that has a high K value and one with a low K value versus between two vertices having close K values.

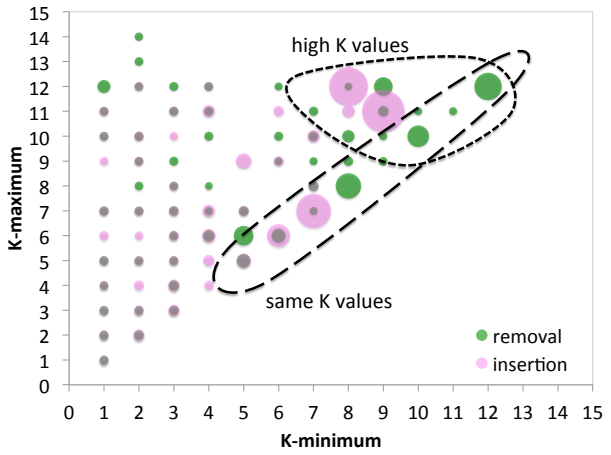

Figure 11: Edge insertion and removal execution times of the traversal algorithm for different K values. Runtime shows low variability when changing parts of the graph with different connectivity characteristics.

Figure 11 shows the performance results for the citationCiteseer graph, which serves as a good representative for our real dataset. This graph has vertices with K values varying from 1 to 15. A bubble in the graph indicates the time taken to insert or remove an edge between two random vertices u and v. If $K(u) \leq K(v)$, $K(u)$ is displayed on the x-axis, while $K(v)$ is displayed on the y-axis. The size of the bubble indicates the average execution time for the insertion (pink) and removal (green) of an edge. The larger the bubble is, the greater the execution time is.

The graph shows that the runtime of the traversal algorithm has low variability. This is a good property, as it means that the algorithm is able to locate a small subgraph to traverse irrespective of the properties of the neighborhoods of the two vertices u and v. Our algorithm shows low runtime variability, as we consistently traverse subgraphs using the vertex with the lowest K value as root.

Execution times vary more when the K values of the different vertices are the same (diagonal). The reason is that the traversal algorithm visits the subgraphs associated with both vertices affected by the new edge, resulting in longer execution times. We also see that insertions between vertices with large K values have large execution times. In general, the execution times we see are proportional to the sizes of the subcores and **not** to the max-cores. In other words, what affects the execution time are the sizes of the subgraphs with the same K value. For small K values, such subgraphs are small, because they are bounded by higher K valued vertices, which in turn belong to their max-core. For large K values, subcores are bigger, because large K valued vertices tend to be close to each other due to the definition of k-core. Although their max-core sizes are small relative to that of small K valued vertices, their subcore sizes turn out to be larger.

7. RELATED WORK

The definition of k-core is first introduced by Seidman [26] to characterize the cohesive regions of graphs. Batagelj et al. [6] developed an efficient algorithm to find the k-core decomposition of a graph. In our work, we build upon these works to develop k-core decomposition algorithms that are incremental in nature, making it possible to apply these algorithms in streaming settings where edge insertions and removals happen frequently, such as maintaining a recent history of a dynamic graph.

There are many application areas of k-core decomposition including but not limited to social networks [17, 29], visualization of large networks [1, 31, 15], and protein interaction networks analysis [3, 30]. In social network analysis, k-cores has been used for community detection [17], clustering [29], and criminal network detection [23].

Thanks to its well-defined structure, k-cores has been used extensively to analyze the structure of certain types of networks [11, 21] and to generate graphs with specific properties [7]. Many graph problems like maximal clique finding [4], dense subgraph discovery [2], and betweenness approximation [18] use k-core decomposition as a subroutine.

In terms of algorithms specific to finding k-core decompositions, an external-memory algorithm for k-core decomposition is introduced in [9]. There are also studies about k-core decomposition on directed [16] and weighted [17] networks. To the best of our knowledge, there is no study on incremental algorithms for k-core decomposition, which makes our work unique in that respect.

Concurrently with our work, Li et al. [20] published a report on incremental algorithms for core decomposition. Our algorithms differ from theirs in two important aspects: (1) They propose quadratic complexity incremental algorithms, whereas our algorithms have linear complexity. (2) The speedup results achieved by our algorithm outperform theirs. For instance, their best algorithm has $6.3\times$ speedup on the cond-mat graph, while our best algorithm (Traversal) achieves a speedup of $776.4\times$.

8. FUTURE WORK

We plan to extend our work along several directions. First, we plan to improve the Traversal algorithm by storing more information at each vertex so that the set of traversed vertices is reduced. Currently, we make use of 2-hop neighborhood information (PCD values) in the Traversal algorithm. Using more than 2-hop neighborhood information, i.e., 3-hop, 4-hop, might result in better running times. Studying the trade-off between further limiting the search space versus reducing the maintenance cost of neighborhood information is an interesting direction for future work. Second, we plan to work on batch update algorithms where multiple edges are inserted to or removed from the graph. Our current incremental algorithms can be used to handle batch updates such that each edge in batch is updated separately. However, we aim to process multiple edges at once so that the traversals of overlapping subgraphs of different edges are shared and the total execution time is reduced. Our initial findings show that many of the theorems and algorithms we designed for single edge insertion and removal cases need fundamental modifications in order to handle batch edge updates. Another interesting direction of future work is the asymptotic bounds of the k-core decomposition on certain kinds of graphs. All of the algorithms we presented provide heuristics for fast updates, but there is no fast update guarantee for a given type of graph. We believe that introducing theoretical bounds for the complexity of the problem will be useful for certain domain of graphs, like Erdös-Renyi graphs and different types of real-world graphs (friendship, protein-interaction, etc.).

9. CONCLUSION

In this paper we have introduced streaming algorithms for k-core decomposition of graphs. The key feature of these algorithms is their incremental nature — the ability to update the k-core decomposition quickly when a new edge is inserted or removed, without

having to traverse the entire graph. Our experimental evaluation shows that these incremental algorithms can perform significantly better than their batch alternatives, where the speedup in execution time increases with the increasing graph size. Given the importance of k-core decomposition in detection of dense regions and communities, max. clique finding, and graph visualization, we believe these incremental algorithms will serve as a fundamental building block for future incremental solutions for other graph problems.

Acknowledgment

We would like to thank the anonymous referees for their valuable comments. This work is partially sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program (Agreement Number: W911NF-12-C-0028). The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

10. REFERENCES

[1] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: A tool for the visualization of large scale networks. *The Computing Research Repository (CoRR)*, abs/cs/0504107, 2005.

[2] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 25–37, 2009.

[3] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4, 2003.

[4] B. Balasundaram, S. Butenko, and I. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research*, 59:133–142, 2011.

[5] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *The Computing Research Repository (CoRR)*, cs.DS/0310049, 2003.

[7] M. Baur, M. Gaertler, R. Görke, M. Krug, and D. Wagner. Augmenting k-core generation with preferential attachment. *Networks and Heterogeneous Media*, 3(2):277–294, 2008.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, 2004.

[9] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 51–62, 2011.

[10] DIMACS. 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10.

[11] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core organization of complex networks. *Physical Review Letters*, 96, 2006.

[12] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *World Wide Web Conference (WWW)*, pages 461–470, 2007.

[13] P. Erdös and A. Rényi. On the evolution of random graphs. In *Institute of Mathematics, Hungarian Academy of Sciences*, pages 17–61, 1960.

[14] S. Fortunato. Community detection in graphs. *Physics Reports*, 483(3-5):75–174, 2009.

[15] M. Gaertler. Dynamic analysis of the autonomous system graph. In *International Workshop on Inter-domain Performance and Simulation (IPS)*, pages 13–24, 2004.

[16] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *IEEE International Conference on Data Mining (ICDM)*, pages 201–210, 2011.

[17] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *International Conference on Advances in Social Network Analysis and Mining (ASONAM)*, pages 87–93, 2011.

[18] J. Healy, J. Janssen, E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 137–148, 2006.

[19] G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.

[20] R.-H. Li and J. X. Yu. Efficient core maintenance in large dynamic graphs. *CoRR*, abs/1207.4567, 2012.

[21] T. Luczak. Size and connectivity of the k-core of a random graph. *Discrete Math*, 91(1):61–68, 1991.

[22] A. A. Nanavati, G. Siva, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi. On the structural properties of massive telecom call graphs: findings and implications. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 435–444, 2006.

[23] F. Ozgul, Z. Erdem, C. Bowerman, and C. Atzenbeck. Comparison of feature-based criminal network detection models with k-core and n-clique. In *International Conference on Advances in Social Network Analysis and Mining (ASONAM)*, pages 400–401, 2010.

[24] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, pages 45–48, 2007.

[25] R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287–302, 1998.

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

[27] SNAP. Stanford network analysis package. http://snap.stanford.edu/snap.

[28] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. D. Harris, J. Cox, W. Szewczyk, and P. Jones. Design principles for developing stream processing applications. *Software: Practice & Experience*, 40(12):1073–1104, 2010.

[29] A. Verma and S. Butenko. Network clustering via clique relaxations: A community based approach. *10th DIMACS Implementation Challenge*, 2011.

[30] S. Wuchty and E. Almaas. Peeling the yeast protein network. *PROTEOMICS*, 5(2):444–449, 2005.

[31] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1049–1060, 2012.