Batch Dynamic Algorithm to Find k-Core Hierarchies

Kasimir Gabert*
Sandia National Laboratories
Albuquerque, New Mexico, USA
kggaber@sandia.gov

Ali Pınar
Sandia National Laboratories
Livermore, California, USA
apinar@sandia.gov

Ümit V. Çatalyürek†
Georgia Institute of Technology
Atlanta, Georgia, USA
umit@gatech.edu

ABSTRACT

Finding k-cores in graphs is a valuable and effective strategy for extracting dense regions of otherwise sparse graphs. We focus on the important problem of maintaining cores on rapidly changing dynamic graphs, where batches of edge changes need to be processed quickly. Many prior dynamic algorithms focus on the problem of maintaining a core decomposition. This finds vertices that are dense in some subgraph, but the subgraph itself is not returned. We develop a new dynamic batch algorithm to maintain cores, with their connected subgraphs, that improves efficiency over processing edge-by-edge. We implement our algorithm and experimentally show that with it core queries can be returned on rapidly changing graphs quickly enough for interactive applications. For 1 million edge batches, on many graphs we run over 100× faster than processing edge-by-edge while remaining under re-computing from scratch.

ACM Reference Format:
Kasimir Gabert, Ali Pınar, and Ümit V. Çatalyürek. 2022. Batch Dynamic Algorithm to Find k-Core Hierarchies. In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES & NDA'22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3534540.3534694

1 INTRODUCTION

An important problem in graph analysis is finding locally dense regions in globally sparse graphs. In this work we consider the problem of finding k-cores [36, 40], which are maximal connected subgraphs with minimum degree at least k. This problem has seen significant attention given its efficiency [36] and usefulness [2, 16, 20, 21, 25, 26, 44].

Many practically important graphs from web data, social networks, and related fields are both large and continuously changing. The problem of maintaining core *decompositions* on graphs has been well studied [30, 37, 47, 48]. Existing approaches run in linear time in the size of the graph, which is theoretically optimal [47], and on many real-world graphs they maintain decompositions within milliseconds after edge changes. So, is the problem solved?

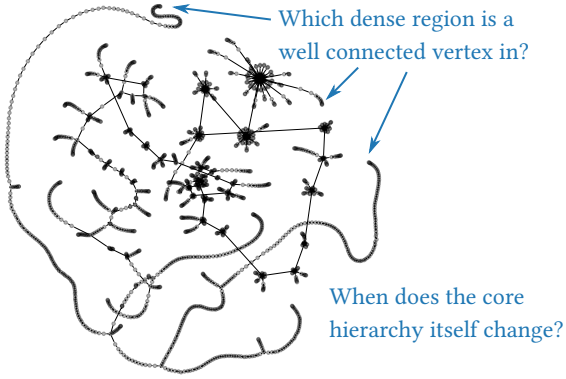

Figure 1: The core hierarchy for the LiveJournal social network graph. Tracking dense regions over time is important for understanding structural changes, and extracting the vertices within a dense region is important for almost all known k-core applications. Simply finding density levels per vertex is insufficient to answering the above questions.

Unfortunately, these approaches only address *half of the problem* of returning a k-core [38]. k-cores are originally defined as *connected* subgraphs [40]. All of the application examples referenced above rely on or use connectivity. A core decomposition, on the other hand, provides *coreness* values for every vertex: that is, the maximal value k such that the vertex is in a k-core. Computing only coreness values has limited use, with only a few known applications (e.g., [24]).

Note that k-cores are hierarchical by definition: a k-core is always fully contained in a potentially larger $(k-1)$-core, with $k > 1$. The core hierarchy is defined as all cores along with their *hierarchical relationships*. Figure 1 shows the cores of the LiveJournal graph [45] along with the hierarchical relationships and two important, example questions that cannot be answered with coreness values alone. Maintaining the hierarchy has been independently proposed several times [5, 13, 15, 31, 38] in different contexts, all of which use a Shell Tree Index (ST-Index). All of these approaches *internally use* a k-core decomposition algorithm [30, 37, 47, 48]. We provide a batch technique to maintain the ST-Index that has lower runtime variability when batch sizes grow.

Approach. The ST-Index builds on the laminar structure of cores, that is cores are either pairwise disjoint or fully contained inside one another, naturally forming a tree. Each node in this tree includes vertices in the *shell* of the given core, that is vertices which are not in any higher core. Coupled with a reverse map from vertices to tree nodes, a core can be efficiently returned by traversing the subtree staying below the desired k value. The core hierarchy is the tree, capturing both cores and relationships between cores.

*Also with Georgia Institute of Technology.
†Also with Amazon Web Services. This publication describes work performed at the Georgia Institute of Technology and is not associated with Amazon.

GRADES & NDA'22, June 12, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9384-3/22/06.
https://doi.org/10.1145/3534540.3534694

We build the tree by first identifying regions of the graph where the cores are the same, known as subcores, and then forming a directed acyclic graph (DAG) with each subcore as a node. Starting from the highest k values, we process nodes in the DAG upwards, merging them to form a tree.

In real-world graphs there is significant variance in the rate of change. As such, batch dynamic algorithms that can reduce the total work when operating on batches are desired [11, 34]. We provide a batch dynamic algorithm to maintain cores themselves, starting from core decompositions. We do this by maintaining the subcore DAG used during construction. After a batch of changes, we revisit each node in the DAG that was modified and re-compute any subcore changes. Any DAG changes are then pushed into the tree, temporarily turning the tree back into a DAG. We then traverse from the sink upwards, correcting the tree.

In [31], developed concurrently with this work, a batch maintenance algorithm for k-cores is maintained. Our presentation differs, as we explicitly describe queries and prove their efficiency, along with our algorithmic approach for the construction and batch maintenance with a subcore DAG.

Contributions. Our main contributions are:

(1) A subcore DAG based ST-Index construction
(2) A batch dynamic algorithm to maintain ST-Index that reduces the work of edge-by-edge updates
(3) An experimental evaluation on real-world graphs that show with both our edge-by-edge and batch algorithms, ST-Index is suitable for interactive use

The remainder of this paper is structured as follows. In § 2 we describe the related work. In § 3 we formally describe our model and problem. In § 4 we present ST-Index. In § 5 we provide our algorithm to compute ST-Index from scratch. In § 6 we introduce our batch algorithm. In § 7 we experimentally evaluate our implementations, and in § 8 we conclude.

2 RELATED WORK

k-cores were introduced independently in [36, 40]. [36] additionally provided a peeling algorithm that uses bucketing to run in $O(n+m)$. The main strategy for computing k-cores has remained roughly the same since then: iteratively peeling the graph, or excluding vertices with too low of degrees, until all degrees are k.

For maintenance, [30] and [37] independently proposed Traversal, which limits consideration of vertices around an edge change if they provably cannot update values. [37] defines the notion of subcores and purecores, variants of which are used in all known maintenance algorithms to limit considered subgraphs. [48] proposed Order, which is the current state-of-the-art and maintains a *peeling order*, instead of coreness values directly, using an order-statistic treap and a heap. Parallel approaches have relied on identifying a set of vertices that can be independently peeled [1, 3, 22, 23]. [4, 47] provide batch algorithms that reduce work as multiple edges are processed simultaneously.

All of the above focus on computing the *coreness values* for vertices. In fact, the lack of focus on connectivity has, in some cases, resulted in later work redefining cores to not include connectivity (e.g., [35]) which limits their usefulness.

Numerous other targets, similar to cores, have been proposed [35]. [12, 49] develop weighted extensions to cores, [32] uses core concepts to reinforce connections within networks, [19] proposes notions of cores for multilayer networks, and [46] ensures vertices in core-like regions are also relatively cohesive given their neighbors. In cases where the cores are used for downstream algorithms, returning the actual (connected) vertices is identified as crucial and algorithms are built to support such queries [33].

Community search [9, 42] is a more general problem for returning a connected set of vertices in a community based on a seed set. The community is commonly defined with a *minimum degree* measure [14]. In this case, if the query consists of a single vertex, community search can return exactly a core. For this reason, we pull from the field of community search to develop ST-Index. [5] proposed the first known shell tree index. It does not support efficient queries, as it creates additional vertices for each coreness level that must be addressed. [38] identifies the same problem that we address—cores require connectivity—and proposes a shell tree-like index with a static construction in the more general nuclei framework, but leaves out maintenance. [13] operates on attributed graphs and extends [38]'s approach and [5]'s index with incremental and decremental algorithms, but without batch algorithms. We use this as our baseline. Concurrent with this work, [31] provides a batch algorithm that is based on [13] and batches changes to the tree directly, without the use of a DAG.

3 PRELIMINARIES

A graph $G = (V, E)$ is a set of vertices V and set of edges E. An edge $e = \{u, v\} \in E$ represents the connection between two distinct vertices $u, v \in V$. We denote $n = |V|$ and $m = |E|$.

We use $\Gamma(v)$ to represent the neighboring edges of $v \in V$. The degree of $v \in V$ is $|\Gamma(v)|$. For directed graphs, Γ^{in} represents edges ending at the given vertex and Γ^{out} represents edges leaving a vertex. If there is more than one graph, we use Γ_G for graph G. The neighborhood of a vertex set $S \subseteq V$, $\Gamma(S)$, represents vertices and edges connected to S, that is the subgraph induced by S and all neighbors of vertices in S.

Dynamic Graph Model. We consider graphs that are changing over time, known as dynamic graphs. An *edge change* either adds a non-existent edge to the graph or removes an existing edge—we assume it is not a multigraph. Vertices are implied by the edges, and so are automatically added and removed when they have a non-zero degree. The dynamic graph then consists of a turnstile stream of edge changes, starting from the empty graph.

In this model, the timestamp of edges received is not preserved and not used by the algorithm. An algorithm that does take into consideration timestamps is called a *temporal* algorithm, and can be either dynamic or static.

Definition 3.1. Consider a dynamic graph at some point in time. A *dynamic graph algorithm* starts from the graph at a previous point in time, the output at that point in time, and potentially some additional input state. Then, given just the changes in the graph from the previous time to the current time, the algorithm will produce the same output as a graph algorithm running only on the latest graph.

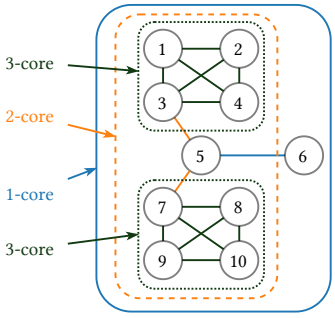

Figure 2: An example graph and its cores. Note that there are two separate 3-cores.

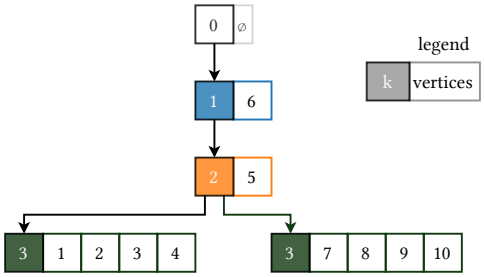

Figure 3: The shell tree for the graph shown in Figure 2. On the left side are the k-shell values, and on the right side are the contained vertices. Each directed edge indicates inclusion of the deeper cores.

We call an *incremental algorithm* a dynamic graph algorithm which can only handle edge insertions and a *decremental algorithm* one which can only handle edge deletions. A *batch dynamic* algorithm can handle $t' > t + 1$. Our batch algorithm, described in Section 6, has an additional state bound by the size of the graph.

Cores. We provide a brief background on k-cores.

Definition 3.2. Let G be a graph and $k \in \mathbb{N}$. A k-core in G is a set of vertices V' which induce a subgraph $K = (V', E')$ such that: (1) V' is maximal in G; (2) K is connected; and (3) the minimum degree is at least k, $\min_{v \in V'} |\Gamma_K(v)| \geq k$.

Figure 2 shows an example graph and its cores. There are two *separate* $k = 3$ cores, one with vertices 1 through 4 and the other with vertices 7 through 10. If all vertices with less than a degree 3 are iteratively removed, the remaining graph consists of those two separate connected components.

Definition 3.3. Let $G = (V, E)$ be a graph and $v \in V$. The coreness of v, denoted $\kappa[v]$, is the value k such that v is in a k-core but not in a $(k + 1)$-core.

Definition 3.4. Let $G = (V, E)$ be a graph. The k-core number of G, denoted ρ_G and shortened to ρ, is given by $\rho = \max_{v \in V} \kappa[v]$.

Problem Statement. We consider the problem of efficiently supporting core and coreness queries on a dynamic graph stream. Let $k \in \mathbb{N}$ and $u \in V$.

- The coreness query $\mathcal{K}(u)$ returns $\kappa[u]$.
- The core query $C(u, k)$ returns the vertices of the k-core subgraph that contains u.
- The hierarchy query $\mathcal{H}$ returns the hierarchical structure of the cores as a tree, with the root as the 0-core

Prior work in the context of cores has focused only on supporting $\mathcal{K}$ queries on dynamic graphs. Unfortunately, this prevents many of the applications of k-cores which rely on *extracting dense regions* of a graph.

4 SHELL TREE INDEX

In this section we present the Shell Tree Index, ST-Index, which is able to efficiently return cores for different vertices: its runtime is asymptotically the size of the result and its space is linear in the number of vertices. This index has been independently developed several times [5, 13, 15, 31, 38] in different contexts. We present the index here for completeness. We will address how to construct the index in Section 5 and how to maintain it in Section 6.

$\mathcal{K}(u)$ queries, or *coreness* queries, can be efficiently returned using an array, so we focus on C and $\mathcal{H}$ queries.

Lemma 4.1 ([37]). *Cores form a laminar family, that is every pair of cores are either disjoint or one is contained in the other.*

Definition 4.2. Let $G = (V, E)$ be a graph and $K \subseteq V$ a k-core in G for some $k \in \mathbb{N}$. Then S is a k-shell if $S = \{v \in K : \kappa[v] = k\}$.

Note that the shell is *disconnected*, however it is a subset of a *connected* core. This means that the traditional approach of using coreness values to compute the shell does not work. We address shell computation later in Section 5, using *subcores*.

A *shell tree* T is at the heart of the ST-Index. We call the vertices of T *tree nodes*, to distinguish from the vertices in G. Each node has two additional pieces of data associated with it: a k value and a set of vertices (in G). T is built as follows. A root node is made with $k = 0$ and a vertex set of isolated vertices (those with $|\Gamma_G(v)| = 0$). Next, nodes are made in T for every k-shell. Its k attribute is set to k corresponding to the shell and its vertex list is set to the vertices in the k-shell. An edge is created in T by linking k-shells, following Lemma 4.1. An example shell tree is shown in Figure 3. The ST-Index consists of T and a map M, mapping $v \in V$ to the appropriate node in T.

Lemma 4.3. *The shell tree is a directed, rooted tree.*

Proof. Suppose a tree node u, corresponding to core K_u has two in-edges. By definition 4.2, each parent corresponds to a unique k-shell. Consider the two corresponding cores, K_1 and K_2. They both include K_u, yet are distinct, and so they have non-trivial overlap contradicting Lemma 4.1. The root is defined with $k = 0$. □

Lemma 4.4. *The out-degree of a non-root tree node with no corresponding vertices in the shell tree can be at most 1.*

Proof. Let the tree node with no corresponding vertices be at level $k > 0$ with out-degree at least 2. Then, there are two distinct *cores* at $k + 1$ (not necessarily shells), and one core at k. The two cores at $k + 1$ must be disconnected by construction.

However, because the tree node has no corresponding vertices, we know that every vertex in the k-core is also in a $(k + 1)$-core.

Furthermore, the k-core is connected. Hence, it is not possible for the two cores at $k+1$ to be disconnected. □

Lemma 4.5. *Let $G = (V, E)$ be a graph with $n = |V|$. The number of nodes in the shell tree is at most $n + 1$.*

Proof. By Lemma 4.4, each node in the tree (besides the root) must have at least one vertex. As there are at most n vertices, the size of the tree is at most $n + 1$. □

Queries on ST-Index. The three queries, $\mathcal{K}(u)$, $C(u, k)$, and $\mathcal{H}$ are returned as follows.

- $\mathcal{K}(u)$ follows the map $M[u]$ to the shell tree node n, and then returns the k value for n.
- $C(u, k)$ runs a tree traversal staying above k
- $\mathcal{H}$ returns the tree nodes and attributes directly.

Efficiency. We next address the ST-Index query efficiency.

Theorem 4.6. *$C(u, k)$ queries on ST-Index run in $O(|C(u, k)|)$ and correctly return the k-core.*

Proof. First, we show correctness. Let C^* be the core for $C(u, k)$, that is C^* is a k-core and $u \in C^*$. The traversal will cover all vertices in the subtree containing u at level k and higher. By Lemma 4.1 we know all denser cores are fully contained in the desired k-core. By Lemma 4.4, we know that any split will occur in an explicit tree node with vertices in the resulting shell. So, this split will be captured by the tree traversal. As such, all vertices in the tree nodes traversed with values k or more exactly form the k-core.

Let down represent higher k values in the tree. Next, we show efficiency. Every downward link in the subtree needs to be fully explored, and there are no nodes with overlapping vertices in the tree. Once a downward traversal occurs, there is no need to check parents. When traversing upwards, all children except the previous one will be explored downwards. Every node is visited only once and all visited vertices are part of the returned core.

As ST-Index is a tree, whether to traverse to the parent can be decided based on whether the parents' value is lower than k. This will result in one additional operation. As such, the runtime is $O(|C(u, k)|)$ and efficient. □

Theorem 4.7. *The ST-Index takes $O(n)$ space.*

Proof. The ST-Index consists of a map of size n between vertices and tree nodes, along with the shell tree itself. By Lemma 4.5, the tree has at most $n + 1$ nodes and n tree edges. Each tree node may have vertices, but there are no redundant vertices. So, the size is $O(n + n + 1 + n + n) = O(n)$. □

Returning ST-Index directly efficiently resolves $\mathcal{H}$ queries.

5 COMPUTING THE ST-INDEX

Computing (and maintaining) the ST-Index hinges on building (and maintaining) the shell tree. We propose a *subcore directed acyclic graph*, that provides a link between core decompositions and the shell tree. Computing the ST-Index is broken into three parts: coreness values, subcore DAG, and the shell tree.

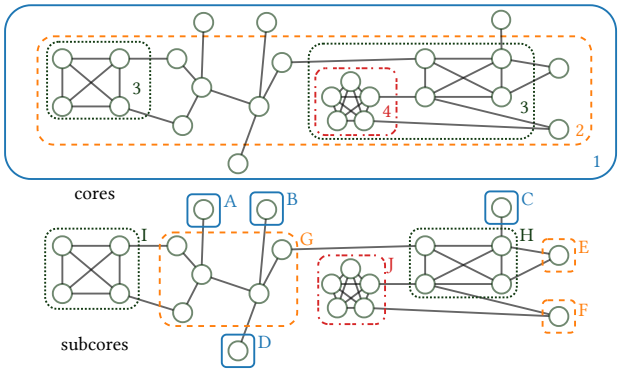

Figure 4: An example graph along with its cores (top) and subcores (bottom). Note that a core may consist of multiple subcores, and subcores are disjoint.

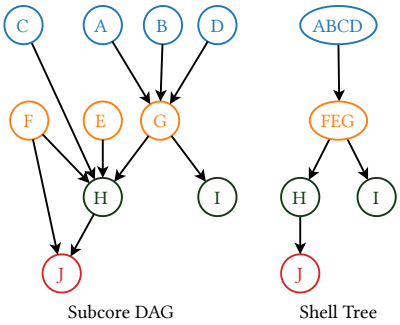

Figure 5: The corresponding subcore DAG and shell tree from the example graph in Figure 4.

Computing Coreness Values. Computing coreness values has been well studied on graphs [10, 36]. The most direct approach, known as peeling, starts by keeping track of vertex degrees. It then moves through coreness values, removing vertices with insufficient degree and recording when they are removed. This is efficient, running in $O(n + m)$, when using buckets [36]. We refer the reader to [35].

Computing the Subcore DAG. Next, we introduce the *subcore directed acyclic graph (DAG)*, which is used to bridge between coreness values and cores.

Definition 5.1. Let G be a graph. A *subcore* is a connected subgraph C such that (1) C is maximal and (2) $\forall v \in C$, $\kappa[v] = k$ for some $k \in \mathbb{N}$.

Subcores were introduced in [37] to limit the region that may have coreness values change on graph changes. Figure 4 shows an example graph with cores and subcores.

Observation 5.1. Subcores are disjoint, by maximality of cores and property (2), and so there are at most n subcores.

After breaking cores up into subcores, the glue to link them back together is saved as a *subcore DAG*. The subcore DAG is built with a directed edge from every lower k subcore to a strictly higher k subcore that it is *directly connected* to. Figure 5 shows the example DAG and shell tree from Figure 4.

Input: graph $G = (V, E)$, κ
1 $C \leftarrow \emptyset$; $D \leftarrow \emptyset$ ▷ DAG vertices and edges
2 $L \leftarrow [v : v \in V]$ ▷ Labels
▷ Compute the subcores
3 **for** $v \in V$ **do**
4 | **if** $L[v] \neq v$ **then** continue
5 | $C \leftarrow C \cup \{v\}$
| ▷ Perform a BFS that stays within κ levels from v
6 | $Q \leftarrow$ Queue(); Q.push(v)
7 | **while** $Q \neq \emptyset$ **do**
8 | | $n \leftarrow Q$.pop()
9 | | **for** $w \in \Gamma(n) : L[w] \neq v \wedge \kappa[w] = \kappa[v]$ **do**
10 | | | Q.push(w)
11 | | | $L[w] = v$
▷ Produce the DAG edges
12 **for** $v \in V$ **do**
13 | **for** $n \in \Gamma(v)$ *where* $L[v] \neq L[n]$ **do**
14 | | $D \leftarrow D \cup \{\langle L[v], L[n]\rangle\}$
15 **return** $DAG{=}(C, D)$

Algorithm 1: Building the subcore DAG.

Lemma 5.2. *The subcore DAG size is bound by G.*

Proof. Each vertex in the subcore DAG corresponds to a connected subgraph in the graph, and every edge in the DAG is a directed edge that results from contracting all vertices in each subcore. Contraction only removes edges and vertices, and no new edges or vertices are added. □

Observation 5.2. The subcore DAG is not a tree. Consider a 3-clique and a 4-clique, connected via an edge, and both connected to another vertex forming a triangle.

The process of building the subcore DAG is shown in Algorithm 1. This algorithm performs a breadth-fist search (BFS) for each vertex. The search is constrained to stay within a κ level, and DAG edges are emitted on graph edges that leave κ levels. Efficient connected components algorithms, e.g., [41], could be used instead.

Lemma 5.3. *Algorithm 1 runs in $O(n + m)$.*

Proof. From lines 6–11, inside the internal BFS, each vertex will be visited once. Inside, each edge will be visited once. Finally, the BFS only starts from unvisited vertices.

For lines 12–14, each vertex and edge will again be visited, resulting in $O(n + m)$ work. □

5.1 Building the Shell Tree

Given a subcore DAG and κ values, we can compute the shell tree. Our algorithm starts with the DAG and modifies it as it moves from the sinks upwards (towards lower k values), using a max-heap. Each processed vertex: 1) identifies neighbors that are at its κ level, and merges itself with them; 2) sets a single node that is an in-neighbor with the closest κ value as the tree parent; and 3) moves all other in-edges to the identified parent, ensure it becomes a tree. The details are presented in Algorithm 2.

Lemma 5.4. *Algorithm 2 correctly builds the shell tree.*

Input: DAG=(C, D), κ
1 $T = (N, E) \leftarrow$ DAG
2 $S \leftarrow \emptyset$
3 $H \leftarrow$ Heap() ▷ Empty Heap
4 **for** *sink* $s \in N$ **do**
5 | H.push($\kappa[s], s$)
6 **while** $H \neq \emptyset$ **do**
7 | $v \leftarrow H$.pop()
8 | **if** $v \in S$ **then continue**
9 | $S \leftarrow S \cup \{v\}$
| ▷ Merge with neighbors at same level
10 | **while** $\exists n \in \Gamma(v) : \kappa[n] = \kappa[v]$ **do**
11 | | Merge(v, n)
12 | | $S \leftarrow S \cup \{n\}$
| ▷ Move all remaining and new in neighbors
13 | $t \leftarrow \arg\max_{n \in \Gamma^{\text{in}}(v)} \kappa[n]$
14 | **for** $n \in \Gamma^{\text{in}}(v)$ **do**
15 | | **if** $n \neq t$ **then** MoveEdge($\langle n, v\rangle \rightarrow \langle n, t\rangle$)
16 | | H.push($\kappa[n], n$)
17 **return** T

Algorithm 2: Constructing the shell tree.

Proof. We argue that after running Algorithm 2, each node will exactly contain the shell. First, a node needs to contain all connected subcore DAG nodes at the given κ value. Second, it cannot have additional nodes merged with it. We argue correctness via induction on κ. At the highest κ level, by the DAG properties, we know the tree nodes connected to the sink are shells and valid. Now, consider a tree node with κ and assume nodes at $\kappa' > \kappa$ are valid. The node is formed by merging DAG nodes at the same level, which are all connected. Any connectivity that is not at level κ will be preserved by moving edges to the node's parent. By Lemma 4.1, we know that any DAG neighbors that it is connected to will also be connected to the parent, and so the new tree node is valid. □

Lemma 5.5. *Algorithm 2 runs in $O(\rho(n + m) \log n)$.*

Proof. The heap processes each vertex once, and each vertex can potentially have all edges attached, resulting in $O(n + m)$ per iteration. However, edges may be carried upwards, and in the worst case all edges except one are carried upwards costing ρ. The heap contributes $\log n$. □

6 MAINTAINING THE ST-INDEX

In this section, we show how to maintain the ST-Index on a graph stream. The objective is to develop a batch dynamic algorithm that will output ST-Index with a small internal state, a quick runtime, and low variability.

Maintaining coreness values is an important part of our overall approach, as it is the first computational step. We provide pointers to coreness value maintenance and then we present our batch maintenance algorithm.

6.1 Maintaining Coreness

We refer the reader to [17, 30, 37, 48] for algorithms to maintain κ. These approaches (and similarly ST-Index) extend to trusses [8]

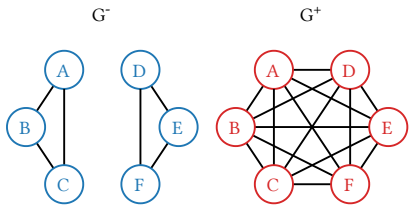

Figure 6: An example graph before a batch of insertions (G^-) and after (G^+). The coreness moves from $\kappa = 2$ to $\kappa = 5$ for each vertex.

and other nuclei [39] by use of a hypergraph [18]. For our experiments we implemented and use Order [48], the state-of-the-art decomposition maintenance algorithm.

For notational convenience, consider a time t. Let G^- denote $G^{(t)}$ and G^+ denote $G^{(t+\Delta)}$. Let κ^- denote the κ values in G^- and κ^+ denote κ values in G^+.

We take advantage of the following crucial property of coreness values on graphs: the subcore theorem.

Theorem 6.1 ([37]). *Let $\{u, v\}$ be an edge change. Suppose $\kappa_{G^-}[u] \leq \kappa_{G^-}[v]$. Then, only vertices in the subcore containing u may have κ values change in G^+, and they may only change by 1 (increase by 1 for insertion, decrease by 1 for deletion.)*

6.2 Batch Maintenance

The idea for our batch maintenance is to *keep the subcore DAG in memory* and use it to update the subcore tree. We maintain an additional pointer between every node in the tree and every node in the subcore DAG. There are two main parts to maintaining the subcore tree in the subcore batch algorithm. First, we maintain the subcore DAG by iterating over changed vertices and recomputing any subcore changes, creating and merging subcores (locally) as appropriate. Second, we need to maintain the ST-Index given the DAG changes. To do this we begin by making all of the DAG changes propagate forward to the tree. Any deleted DAG node results in deleting the reference from the subcore tree, any newly empty tree nodes are deleted, and any new DAG nodes and their connections are added to the tree. The tree is now no longer a DAG. We correct the tree with Algorithm 2.

The baseline approach is presented in Appendix A, and denoted SingleEdge. SingleEdge processes each edge update and either merges branches of the ST-Index at the approach levels or splits them. SingleEdge is not a batch algorithm, however [31] extends SingleEdge by collecting edges and making the merges and splits in batches. In this approach, edge insertions and deletions need to be processed separately. Unlike SingleEdge, our batch approach naturally handles deletions identically to insertions and so both insertions and deletions can be mixed inside of batches. The Batch algorithm is presented in Algorithm 3. Following the example in Figure 6, Figure 7 shows saved work between SingleEdge and Batch.

Our runtime is the cost of Algorithm 2 plus the cost of a BFS over each modified subcore. Correctness follows from Algorithm 2 as we maintain data structures. In the worst case this can be the

Input: ST-Index = (M, T), DAG D, batch B

```
1  C ← {v : v ∈ e ∈ B}; K ← ∅
2  I ← ∅                                              ▷ Visited set
3  for v ∈ C do
4      if v ∈ I then continue
5      I ← I ∪ {v}
6      Q ← Queue; Q.push(v)                           ▷ Change queue
7      while Q ≠ ∅ do
8          q ← Q.pop()
9          n_d, n_T ← L[q]                            ▷ DAG/Tree node of q
10         K ← K ∪ {n_D}
11         n'_d ← new DAG node
12         assign q to n'_d in D and M, T
13         S ← Queue; S.push(q)                       ▷ Subcore queue
14         while S ≠ ∅ do
15             n ← S.pop()
               /* Check if n is in the subcore                    */
16             if κ⁺[n] ≠ κ⁺[q] then
                   /* If n changed, process it separately         */
17                 if n ∉ I and κ⁻[n] ≠ κ⁺[n] then
18                     I ← I ∪ {n}
19                     Q.push(n)
20                 continue
21             if n ∉ I then
22                 I ← I ∪ {n}
23                 Q.push(n)
24             assign n to n'_d in D and M, T
25 remove newly isolated nodes in D
26 copy DAG edges from DAG nodes in K to T
27 remove newly empty tree nodes in T
28 run Algorithm 2
```

Algorithm 3: The Batch algorithm.

runtime of Algorithm 2, but we next show empirically we run faster than re-computing from scratch.

7 EMPIRICAL ANALYSIS

In this section we perform an experimental evaluation of our approach to demonstrate that it is able to provide core queries on rapidly changing real-world graphs.

Environment. We implemented our algorithm in C++ and compiled with GCC 10.2.0 at O3. We ran on Intel Xeon E5-2683 v4 CPUs at 2.1 GHz with 256 GB of RAM and CentOS 7. To perform coreness maintenance, we implemented Order [48]. Any coreness maintenance approach can be used in its place. We include all memory allocation costs in our runtimes. We use a hash map of vectors to store the graph, and store both in- and out-edges. We ran five trials for each experiment and show the results from all trials.

Baseline. As our baseline, we implemented the non-batch maintenance approach from [13], which we ported to the case of computing cores on graphs (see Appendix A). We refer to this as SingleEdge. When operating on a batch, SingleEdge runs independently for each edge change. Insertions and deletions can therefore easily

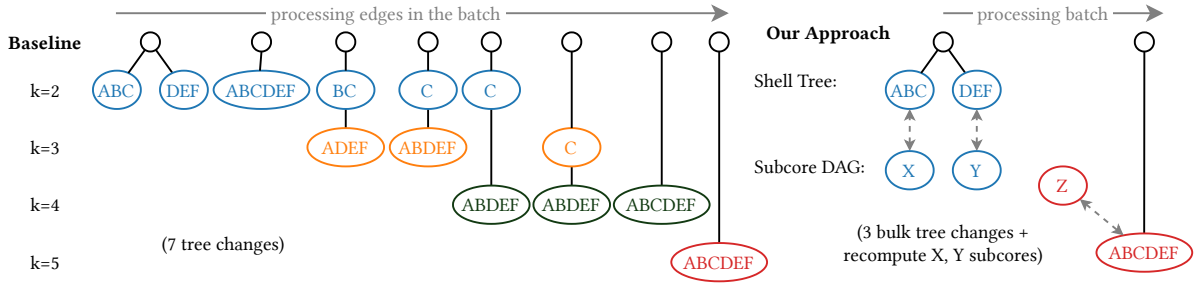

Figure 7: Following the example in Figure 6, we compare processing with the baseline SingleEdge (Appendix A) and our batch approach. The main cost is an increase in memory to store the subcore DAG.

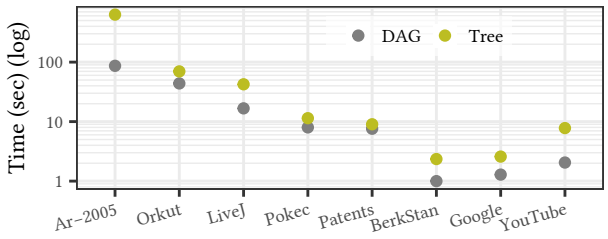

Figure 8: The ST-Index construction time, broken down into DAG construction and Tree construction.

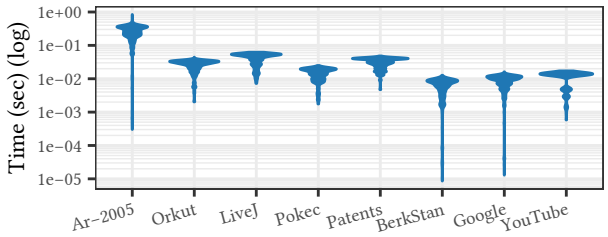

Figure 9: The runtime to return C queries, which are low enough for interactive use.

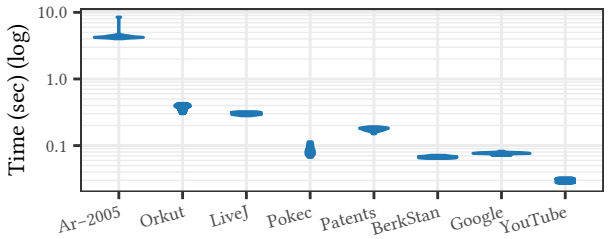

Figure 10: The runtime to return $\mathcal{H}$ queries.

be mixed. We only show results with insertions as they are the harder case [13] and there are few known benchmark datasets with frequent deletions.

Datasets. The graphs that we evaluate with are benchmark graphs that are representative of real-world graphs from a variety of domains and with different properties. We downloaded them from SNAP [28] (excluding Ar-2005, downloaded from [7]). Table 1 contains statistics on the graphs used, the size of the subcore DAG, and the size of the tree. We removed self loops, duplicate edges, and treated graphs as undirected. We randomized the edge order, simulating a graph stream, and performed our experiments by first removing random edges and next inserting them.

Table 1: Graphs used with n, m in millions.

Name	n, m	DAG n, m	$\lvert T\rvert$
Ar-2005 [6, 7]	22, 640	12, 47	28 K
Orkut [45]	3, 117	1, 22	254
LiveJ [45]	4, 35	2, 12	2 K
Pokec [43]	2, 22	1, 5	54
Patents [27]	4, 17	2, 4	4 K
BerkStan [29]	0.7, 7	0.2, 0.8	2 K
Google [29]	1, 4	0.4, 1.2	5 K
YouTube [45]	1, 3	1, 2.5	140

Experiments. Our main experimental goal is to evaluate the real-world feasibility of our approach on modern graphs and systems with highly variable and large batch sizes.

First, we show the index construction time for Batch. The results are shown in Figure 8. In all cases building the tree is more expensive than building the DAG. The overall runtime reinforces the need for dynamic algorithms as for large graphs, such as Orkut, the DAG construction takes around 90 seconds and the tree construction takes around 330 seconds.

Next, we want to show that ST-Index is a useful index for cores. We report the query times for C in Figure 9 and $\mathcal{H}$ in Figure 10 on ST-Index. For C, we performed queries from 1000 randomly sampled vertices with uniformly random k-values such that the vertex is in a k-core. We plot the distribution of the runtime results. For all graphs, all cores are returned in under one second with many in the tens of milliseconds. It is important to note that, except for Ar-2005, there is no long tail. In all cases, the distributions are relatively focused in one time range. Given that our query is efficient the runtime largely consists of copying memory. The denser the core the faster the return tends to be, as there are fewer vertices to copy out. In many cases, the runtimes are fast enough to be used for interactive applications, e.g., in web page content. For $\mathcal{H}$, we report

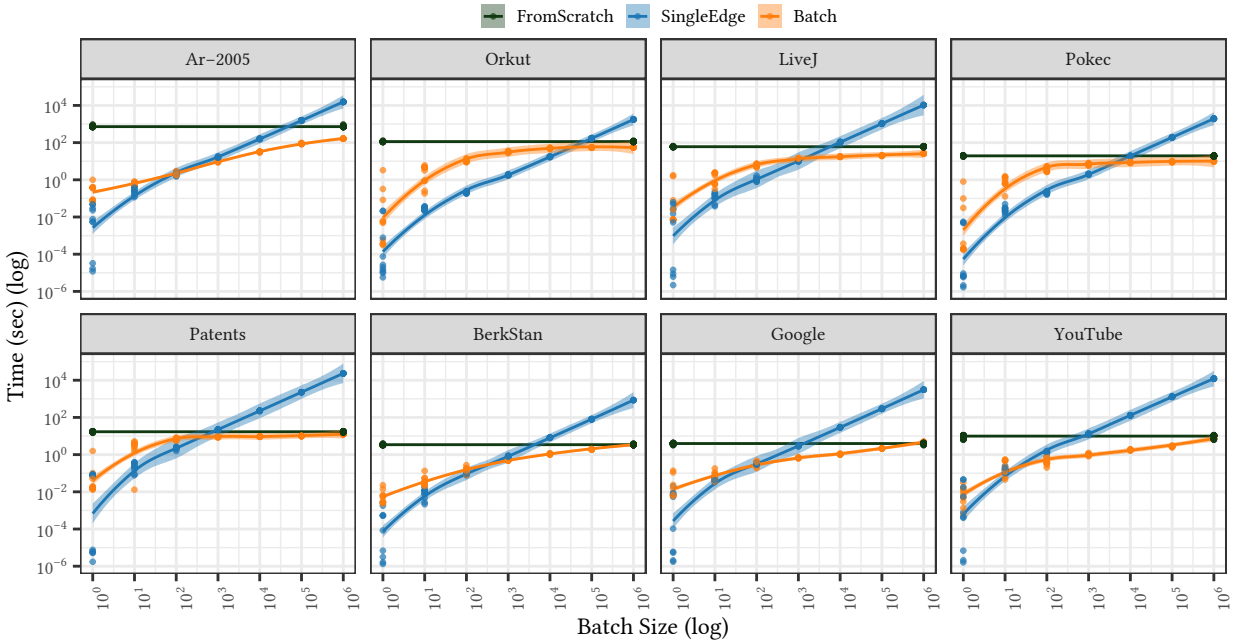

Figure 11: Varying the batch size and running Batch, SingleEdge, and FromScratch. Batch is orders of magnitude faster than SingleEdge for batches above 10^5 and remains below re-computing from scratch up to 10^6. The data points and LOESS smoothing lines with 95% confidence intervals are shown.

the time to build and return the full hierarchy, including each node at each level. This is under 10 seconds for all graphs.

Finally, we maintained cores for 100 batches of different batch sizes for each graph. The results are shown in Figure 11. In all cases, when batch sizes are large Batch remains below both FromScratch and SingleEdge. For a batch dynamic algorithm, we are looking for the region below re-computing from scratch and below single-edge algorithsm. In some graphs, such as Pokec and Patents, it is not a large region, however in all graphs it exists and provides significant improvements. Future work includes combining the DAG construction and maintenance with the direct tree maintenance to achieve an effective hybrid approach, achieving the lower of the all of the curves. Note that these are log-log plots, and so even for Patents our batch approach is 2× faster than re-computing from scratch at batch sizes of one million.

An important take-away of this approach is that, by staying below re-computing from scratch with batches that have millions of edge changes, but at the same time running quickly on batches with only a few edge changes, the algorithm can be used in environments with highly variable change rates in graphs. In many environments, there are typically periods of high activity, for example if content becomes viral, major world events occur, or during shopping holidays, and periods of relative calm. Our approach remains useful across the spectrum.

8 CONCLUSION

We focus on the important but overlooked problem of returning *cores*, as opposed to *coreness* values. We consider both core queries, which return a k-core, and hierarchy queries, which return the full core hierarchy. Our approach applies beyond k-cores to other arbitrary nuclei, such as trusses.

We develop algorithms around a tree-based index, the ST-Index, that is efficient and takes linear space in the number of graph vertices. We provide an algorithm to construct the ST-Index using a new approach based on a subcore DAG. We design and implement a batch maintenance algorithm for ST-Index that uses the same subcore DAG and can handle variable and large batch sizes. We show that our approach is able to run faster than edge-by-edge approaches on rapidly changing graphs and can return cores and hierarchies fast enough for interactive use.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was funded in part by the NSF under Grant CCF-1919021 and in part by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy. Distributed k-core view materialization and maintenance for large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2439–2452, 2014.

[2] I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: a tool for the analysis of large scale internet graphs. *arXiv preprint cs.NI/0511007*, 2005.

[3] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 161–168, 2016.
[4] W. Bai, Y. Zhang, X. Liu, M. Chen, and D. Wu. Efficient core maintenance of dynamic graphs. In *International Conference on Database Systems for Advanced Applications*, pages 658–665. Springer, 2020.
[5] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *Data mining and knowledge discovery*, 29(5):1406–1433, 2015.
[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.
[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.
[8] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 16:3–1, 2008.
[9] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002, 2014.
[10] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304, 2017.
[11] L. Dhulipala, D. Durfee, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1300–1319. SIAM, 2020.
[12] M. Eidsaa and E. Almaas. S-core network decomposition: A generalization of k-core analysis to weighted networks. *Physical Review E*, 88(6):062819, 2013.
[13] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu. Effective and efficient attributed community search. *The VLDB Journal*, 26(6):803–828, 2017.
[14] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, 29(1):353–392, 2020.
[15] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao. Effective and efficient community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment*, 13(6):854–867, 2020.
[16] H. A. Filho, J. Machicao, and O. M. Bruno. A hierarchical model of metabolic machinery based on the k core decomposition of plant metabolic networks. *PloS one*, 13(5):e0195843, 2018.
[17] K. Gabert, A. Pınar, and U. V. Çatalyürek. Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*. IEEE, May 2021.
[18] K. Gabert, A. Pınar, and U. V. Çatalyürek. A unifying framework to identify dense subgraphs on streams: Graph nuclei to hypergraph cores. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining (WSDM)*, WSDM '21, page 689–697. ACM, Mar 2021.
[19] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano. Core decomposition in multilayer networks: theory, algorithms, and applications. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(1):1–40, 2020.
[20] J. García-Algarra, J. M. Pastor, J. M. Iriondo, and J. Galeano. Ranking of critical species to preserve the functionality of mutualistic networks using the k-core decomposition. *PeerJ*, 5:e3321, 2017.
[21] P. Hagmann, L. Cammoun, X. Gigandet, R. Meuli, C. J. Honey, V. J. Wedeen, and O. Sporns. Mapping the structural core of human cerebral cortex. *PLoS Biol*, 6(7):e159, 2008.
[22] Q.-S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1287–1300, 2019.
[23] H. Jin, N. Wang, D. Yu, Q.-S. Hua, X. Shi, and X. Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2416–2428, 2018.
[24] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse. Identification of influential spreaders in complex networks. *Nature physics*, 6(11):888–893, 2010.
[25] Y.-X. Kong, G.-Y. Shi, R.-J. Wu, and Y.-C. Zhang. K-core: Theories and applications. *Physics Reports*, 832:1–32, 2019.
[26] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal. The web as a graph. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–10, 2000.
[27] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.
[28] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[29] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.
[30] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2013.
[31] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian. Hierarchical core maintenance on large dynamic graphs. *Proceedings of the VLDB Endowment*, 14(5):757–770, 2021.
[32] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang. Global reinforcement of social networks: The anchored coreness problem. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2211–2226, 2020.
[33] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*, pages 1130–1141, 2019.
[34] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv. Batch processing for truss maintenance in large dynamic graphs. *IEEE Transactions on Computational Social Systems*, 2020.
[35] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.
[36] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, 30(3):417–427, 1983.
[37] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.
[38] A. E. Sariyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *Proceedings of the VLDB Endowment*, 10(3):97–108, 2016.
[39] A. E. Sariyuce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 927–937, 2015.
[40] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
[41] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 143–153, 2014.
[42] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948, 2010.
[43] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, volume 1, 2012.
[44] M. P. Van Den Heuvel and O. Sporns. Rich-club organization of the human connectome. *Journal of Neuroscience*, 31(44):15775–15786, 2011.
[45] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
[46] C. Zhang, F. Zhang, W. Zhang, B. Liu, Y. Zhang, L. Qin, and X. Lin. Exploring finer granularity within the cores: Efficient (k, p)-core computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 181–192. IEEE, 2020.
[47] Y. Zhang and J. X. Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1024–1041. ACM, 2019.
[48] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 337–348. IEEE, 2017.
[49] W. Zhou, H. Huang, Q.-S. Hua, D. Yu, H. Jin, and X. Fu. Core decomposition and maintenance in weighted graph. *World Wide Web*, pages 1–21, 2020.

A SINGLE EDGE MAINTENANCE ALGORITHM

The main idea for maintaining the ST-Index edge-by-edge is to first break apart any core or shell that was increased and then repair the tree by merging together the paths from the endpoints. For deletions, a map is made that determines where, after a core is split, it could return to in the tree. Then, the path from the core to the root is traversed and any potential split is determined. Our algorithm shares many similarities to the community search algorithm of [13]. Our algorithm addresses cores instead of the more general community search problem on attributed graphs. Specifically, it does not need to support queries involving subsets

Input: graph $G = (V, E)$, $e = \{u, v\}$, κ^-, κ^+, ST-Index $= (M, T)$

if $\kappa^-[u] > \kappa^-[v]$ **then** swap u, v
$K \leftarrow M[u]$ ▷ find the tree node for u
$S \leftarrow \{w \in V : \kappa^-[w] \neq \kappa^+\}$
if $M[u]$.vertices $= S$ **then**
  ▷ The entire shell moves as one subcore
  for $c \in K$.children **do**
    if c.k $= k + 1$ **then** Merge(K, c)
  K.k $\leftarrow k + 1$
  return T
▷ We need to merge or create a new sink
K.vertices $\leftarrow K$.vertices $\setminus S$
$X \leftarrow \langle K, k + 1, S\rangle$ ▷ new tree node with parent K, level $k + 1$, vertices S
for $w \in S$ **do**
  for $n \in \Gamma_{G^-}(w) \setminus S$ **do**
    if $\kappa^+[n] \geq k + 1$ **then** MergeOrConnect$(X, M[n])$
▷ Merge the path with v
$c \leftarrow M[v], l \leftarrow$ SINK
while $\kappa[c] \geq \kappa^+[u]$ **do**
  $l \leftarrow c; c \leftarrow c$.parent
MergePaths(X, c)
return T

Algorithm 4: SingleEdge (incremental case).

Input: ST-Index $= (M, T)$, U, V

if $U = V$ **then return**
if $\kappa[U] > \kappa[V]$ **then** swap U, V
$c \leftarrow V; l \leftarrow$ SINK
while $\kappa[c] \geq \kappa[U]$ **do**
  $l \leftarrow c; c \leftarrow c$.parent
if $\kappa[U] = \kappa[c]$ **then**
  Merge(U, c)
  return MergePaths$(c, U$.parent$)$
else
  MakeChild(U, c)
  return MergePaths(c, U)

Algorithm 5: MergePaths, which merges two paths starting from tree nodes U and V until the root.

of vertices. We refer to this approach as SingleEdge. We describe insertions in detail—deletions are similar but split nodes [13].

Let K be the tree node that has a *lower κ value* given an edge insertion. We first check if all of K's vertices leave. If so, we move K down and merge its children with connected subcores. Next, we iterate through the moved vertices and identify if they are connected to a shell tree node at level $k + 1$. If so, we merge those shell tree nodes together. If not, we create a new tree node for the moved vertices. Then, we walk up the tree from both endpoints and, starting at level $k + 1$, begin merging all visited vertices. The algorithm is presented in Algorithm 4, with merge paths presented in Algorithm 5. A visual depiction is given in Figure 12.

Lemma A.1. *The runtime for Algorithm 4 is $O(|\Gamma(S)| + \rho n)$, where S is the subcore that increases κ.*

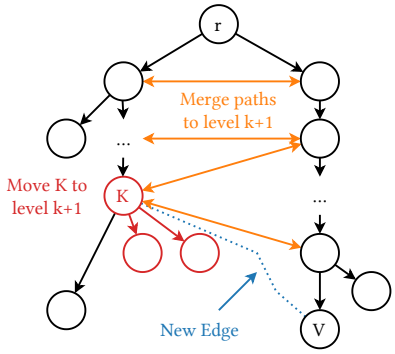

Figure 12: The incremental algorithm process. First, the tree node corresponding to the smaller κ level vertex, K, is processed. Next, the paths to K and to the tree node being connected are merged to level $k + 1$.

Proof. In the first part, the modified subcore and all of its immediate neighbors are accessed, resulting in $O(\Gamma(S))$ work. After that, in the worst case, the height of the tree will be accessed to find the closest neighbor to merge in, resulting in $O(\rho n)$ work. □