Fully Dynamic Algorithm for Top-k Densest Subgraphs

Muhammad Anis Uddin Nasir♯1, Aristides Gionis‡2, Gianmarco De Francisci Morales⋄3
Sarunas Girdzijauskas♯4

♯Royal Institute of Technology, Sweden ‡Aalto University, Finland ⋄Qatar Computing Research Institute, Qatar
[1]anisu@kth.se, [2]aristides.gionis@aalto.fi, [3]gdfm@acm.org, [4]sarunasg@kth.se

ABSTRACT

Given a large graph, the densest-subgraph problem asks to find a subgraph with maximum average degree. When considering the top-k version of this problem, a naïve solution is to iteratively find the densest subgraph and remove it in each iteration. However, such a solution is impractical due to high processing cost. The problem is further complicated when dealing with dynamic graphs, since adding or removing an edge requires re-running the algorithm. In this paper, we study the top-k densest-subgraph problem in the sliding-window model and propose an efficient fully-dynamic algorithm. The input of our algorithm consists of an edge stream, and the goal is to find the node-disjoint subgraphs that maximize the sum of their densities. In contrast to existing state-of-the-art solutions that require iterating over the entire graph upon any update, our algorithm profits from the observation that updates only affect a limited region of the graph. Therefore, the top-k densest subgraphs are maintained by only applying local updates. We provide a theoretical analysis of the proposed algorithm and show empirically that the algorithm often generates denser subgraphs than state-of-the-art competitors. Experiments show an improvement in efficiency of up to five orders of magnitude compared to state-of-the-art solutions.

1 INTRODUCTION

Finding a subgraph with maximal density in a given graph is a fundamental graph-mining problem, known as the *densest-subgraph* problem. Density is commonly defined as the ratio between number of edges and vertices, while many other definitions of density have been used in the literature [7, 28, 30, 31]. The densest-subgraph problem has many applications, for example, in community detetion [11, 14], event detection [2], link-spam detection [18], and distance query indexing [1].

In applications, we are often interested not only in one densest subgraph, but in the *top-k*. The top-k densest subgraphs can be vertex-disjoint, edge-disjoint, or overlapping [6, 17]. Different objective functions and constraints give rise to different problem formulations [6, 17, 32]. In this work, we choose to maximize the sum of the densities of the k subgraphs in the solution. In addition, we seek a solution with disjoint subgraphs. This version of the problem is known to be NP-hard [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'17, November 6–10, 2017, Singapore.
© 2017 ACM.
ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: http://dx.doi.org/10.1145/XXXXXX.XXXXXX

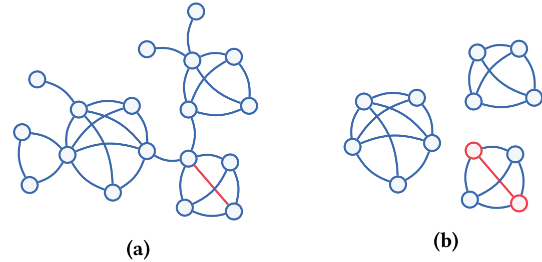

Figure 1: For the graph in Figure 1a, we are interested in extracting the top-3 densest subgraphs. Consider the arrival of an edge shown in red. Figure 1b shows the top-3 densest subgraphs after the arrival. The objective is to design an algorithm that can efficiently maintain the densest subgraphs while keeping the number of updates very low, in this case updating only the vertices in red.

To complicate the matter, most real-world graphs are dynamic and rapidly changing. For instance, Facebook users are continuously creating new connections and removing old ones, thus changing the network structure. Twitter users produce posts at a high rate, which makes old posts less relevant. Given the dynamic nature of many graphs, here we focus on a sliding-window model which gives more importance to recent events [4, 12, 13]. Finding the top-k densest subgraphs in a sliding window is of interest to several real-time applications, e.g., community tracking [33], event detection [26], story identification [2], fraud detection [8], and more. We assume the input to the system arrives as an *edge stream*, and seek to extract the k vertex-disjoint subgraphs that maximize the sum of densities [6].

A naïve solution involves executing a static algorithm for the densest-subgraph problem k times, while removing the densest subgraph in each iteration. However, such a solution is impractical as it requires to execute the algorithm k times for each update. An alternative solution to our problem is to use a dynamic densest subgraph algorithm in a pipeline manner, where the output of an algorithm instance serves as input to the following one. In this case, the graph and the instances of the algorithm are replicated independently across k instances of the algorithm, resulting in a high memory and processing cost.

In this paper, we propose a fully-dynamic algorithm that finds an approximate solution. The proposed algorithm follows a greedy approach and updates the densities of the subgraphs connected to vertices affected by edge operations (addition and removal). The

algorithm is efficiently designed based on key properties of densest subgraphs, and it is competitive against other recent dynamic algorithms [9, 15, 24].

First, our algorithm relies on the observation that only high-degree vertices are relevant for the solution. As many natural graphs have a heavy-tailed degree distribution, the number of high-degree vertices in a graph is relatively smaller than the number of low-degree ones. This simple observation enables pruning a major portion of the input stream on-the-fly. Second, the vertices that are part of a densest subgraph are connected strongly to each other and weakly to other parts of the graph. This enables independently maintaining and locally updating multiple subgraphs. Figure 1 provides an example which demonstrates this intuition.

The algorithm tracks multiple subgraphs on-the-fly with the help of a newly defined data structure called *snowball*. These subgraphs are stored in a bag, from which the k subgraphs with maximum densities are extracted. The algorithm runs in-place, and does not require multiple copies of the graph, thus making it memory-efficient. The one-pass nature of the algorithm allows extracting top-k densest subgraphs for larger values of k.

We provide a theoretical analysis of the proposed algorithm, and show that the algorithm guarantees 2-approximation for the first densest subgraph ($k = 1$) while providing a high-quality heuristic for $k > 1$ compared to other solutions. Experimental evaluation shows that our algorithm often generates denser subgraphs compared to the state-of-the-art algorithms, due to the fact that it maintains disconnected subgraphs separately. In addition, the algorithm provides improvement in runtime up to three to five orders of magnitude compared to the state-of-the-art. In summary, we make the following contributions:

- We study the top-k densest vertex-disjoint subgraphs problem in the sliding-window model.
- We provide a brief survey on adapting several algorithms for densest subgraph problem for the top-k case.
- We propose a scalable fully-dynamic algorithm for the problem, and provide a detailed analysis of it.
- The algorithm is open source and available online, together with the implementations of all the baselines.[1]
- We report a comprehensive empirical evaluation of the algorithm in which it significantly outperforms previous state-of-the-art solutions by several orders of magnitude, while producing comparable or better quality solutions.

2 PRELIMINARIES

In this section, we present our notation, revisit basic definitions, and formulate the top-k densest subgraphs problem.

Consider an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The *neighborhood* of $v \in V$ is defined as $N(v) = \{u \mid (v, u) \in E\}$, and its *degree* as $d(v) = |N(v)|$. For a subset $S \subseteq V$ we define $E(S)$ to be the set of edges whose both endpoints are in S, and $G(S) = (S, E(S))$ the subgraph *induced* by S. The *internal degree* of a vertex v with respect to $S \subseteq V$ is defined by $d_S(v) = |N(v) \cap S|$.

[1] https://github.com/anisnasir/TopKDensestSubgraph

Finally, for a subset of vertices $S \subseteq V$ we define its *density* ρ_S by

$$\rho_S = \frac{|E(S)|}{|S|}. \quad (1)$$

Note that the density of any subgraph is equal to half of its *average internal degree*.

Definition 2.1 (Densest subgraph). Given an undirected graph $G = (V, E)$, the densest subgraph S^* is a set of vertices that maximizes the density function, i.e.,

$$S^* = \arg\max_{S \subseteq V} \rho_S. \quad (2)$$

We say that an algorithm A computes an *α-approximation* of the densest subgraph if A computes a subset $S \subseteq V$ such that $\rho_S \geq \frac{1}{\alpha} \rho_{S^*}$, where $S^* \subseteq V$ is the densest subgraph of G.

Next we introduce other concepts related to densest subgraph: *graph core*, *core decomposition*, and *induced core subgraph* of a vertex.

Definition 2.2 (j-core). Given an undirected graph $G = (V, E)$ and an integer j, a j-core of G is a subset of vertices $C \subseteq V$ so that each vertex $v \in C$ has internal degree $d_C(v) \geq j$, and C is maximal with respect to this property.

Definition 2.3 (Core decomposition). A core decomposition of a graph $G = (V, E)$ is a nested sequence $\left\{C_i\right\}$ of cores

$$V = C_0 \supseteq C_1 \supseteq \ldots \supseteq C_\ell \supseteq \varnothing, \quad (3)$$

where each C_i is a j-core for some j.

Definition 2.4 (Core number). Given a core decomposition $V = C_0 \supseteq C_1 \supseteq \ldots \supseteq C_\ell \supseteq \varnothing$ of a graph $G = (V, E)$, the core number $\kappa(v)$ of a vertex v is the largest j such that $v \in C$ and C is a j-core. By overwriting notation, the core number $\kappa(C)$ of a core C is the largest j for which C is a j-core.

Additionally, we use $\kappa_S(v)$ to denote the core number of a vertex v in the subgraph induced by S. The largest core (or *main core*) of a subgraph of $G(S) = (S, E(S))$ is denoted by $C_\ell(S)$, while the main core of G is simply denoted by C_ℓ.

Note that the density of a j-core is at least $j/2$, as each vertex in the core has degree at least j and each edge is counted twice. This observation implies that the main core of a graph is a 2-approximation of its densest subgraph.

Lemma 2.5. *Consider the core decomposition of a graph G, i.e., $C_1 \subseteq C_2 \subseteq \ldots \subseteq C_\ell$. The maximum core C_ℓ is 2-approximation of the densest subgraph of G [21].*

Proof. Let S^* be the densest subgraph of G having density ρ_{S^*}. Every vertex in $G(S^*)$ has degree at least ρ_{S^*}; otherwise a vertex with degree smaller than ρ_{S^*} can be removed to obtain an even denser subgraph. Thus, S^* is a ρ_{S^*}-core. Given the core decomposition of G, we know that $\rho_{C_\ell} \geq \frac{1}{2}\kappa(C_\ell)$. We want to show that $\rho_{C_\ell} \geq \frac{1}{2}\rho_{S^*}$. Assume otherwise, i.e., $\rho_{C_\ell} < \frac{1}{2}\rho_{S^*}$. Then $\kappa(C_\ell) < \rho_{S^*}$. It follows that S^* is a higher-order core than C_ℓ, a contradiction. □

The concept of a core subgraph $\mathcal{I}(v)$ induced by a vertex v [23, 27] is also pertinent to our analysis.

Definition 2.6 (Induced core subgraph). Given a graph $G = (V, E)$ and a vertex $v \in V$, the induced core subgraph of vertex v, denoted by $\mathcal{I}(v)$, is a maximal connected subgraph containing the vertex v s.t. the core number of all the vertices in $\mathcal{I}(v)$ is equal to $\kappa(v)$.

In other words, the induced subgraph contains all vertices that are reachable from v and have the same core number $\kappa(v)$.

All previous definitions apply to static graphs. Let us now focus on dynamic graphs. In particular, we consider processing a graph in the *sliding window edge-stream model* [13]. According to this model, the input to our problem is a stream of edges. The edge e_i is the i-th element of the stream. Equivalently, we say that edge e_i has timestamp i. A sliding window $W_t(x)$, defined at time t and of size x, is the set of all edges that arrive between e_{t-x+1} and e_t,

$$W_t(x) = \{e_i, i \in [t - x + 1, t]\}. \quad (4)$$

For each edge $e_i = (u, v)$, we consider that u and v appear at time i, and we use $V_t(x)$ to denote the set of vertices that appear in a length-x sliding window at time t. The graph in a length-x sliding window at time t is then defined to be $G_t(x) = (V_t(x), W_t(x))$.

We are now ready to formally define the problem that we consider in this paper, i.e., finding the top-k densest subgraphs in sliding window. We first define the problem in a static setting.

Definition 2.7. Given an undirected graph $G = (V, E)$ and an integer $k > 0$, the top-k densest subgraphs of G is a set of k disjoint maximal set of vertices $\mathcal{S} = \{S_1, \ldots, S_k\}$ that maximize the sum of its densities:

$$\rho_k(\mathcal{S}) = \max \sum_{i=1}^{k} \rho_{S_i}, \text{ for all } S_i \in \mathcal{S} \quad \text{subject to}$$

$$\text{there is no } S_j \supset S_i \mid \rho_{S_j} \geq \rho_{S_i}, \quad \text{for all } S_i, S_j \subseteq \mathcal{S} \quad (5)$$

$$S_i \cap S_j = \varnothing, \quad \text{for all } i, j \in \{1 \ldots k\}, i \neq j. \quad (6)$$

As already shown by Balalau et al. [6], the problem defined aboce is **NP**-hard, for any $k > 1$. The problem we consider in this paper is the following.

Problem 2.8. *Given a graph stream $\{e_i\}$ and a sliding window length x, maintain the top-k densest subgraphs $\rho_k(\mathcal{S})$ of the graph $G_t(x)$, at any given time t.*

3 BACKGROUND

In this section we present a brief review over several algorithms for finding dense subgraphs. Additionally, we discuss how these methods can be used for solving Problem 2.8.

Densest subgraph in static graphs. Finding the densest subgraph according to the density definition (1) can be solved in polynomial time. An elegant solution involving reduction to the minimum-cut problem was given by Goldberg [20]. As the fastest algorithm to solve the minimum-cut problem runs in $O(nm)$ time [25], Goldberg's algorithm is not scalable to large graphs.

Asahiro et al. [3] and Charikar [10] propose a linear-time algorithm that provides a factor-2 approximation. This algorithm iteratively removes the vertex with the lowest degree in each iteration, until left with an empty graph. Among all subgraphs considered during this vertex-removal process, the algorithm returns the densest. The time complexity of this greedy algorithm is $O(m + n)$. Bahmani et al. [5] propose a MapReduce version of the greedy algorithm, with approximation ratio $2(1+\epsilon)$, while making $O(\log_{1+\epsilon} n)$ passes over the input graph.

Core decomposition in static graphs. The core decomposition of a graph G is the process of identifying all cores of G, as defined in 2.3. Batagelj and Zaversnik [7] propose a linear-time algorithm to obtain the core decomposition. The algorithm first considers the whole graph and then repeatedly removes the vertex with the smallest degree. The core number $\kappa(v)$ of a vertex v is set equal to the degree of v at the moment that v is removed from the graph.

Densest subgraph in evolving graphs. There is a growing body of literature on finding dense subgraphs in evolving graphs [9, 15, 19, 24]. We focus mainly on the deterministic algorithm for densest subgraph in evolving graphs. For instance, Epasto et al. [15] propose an efficient algorithm for computing the densest subgraph in the dynamic graph model [16]. Their work assumes that edges are inserted into the graph adversarially but deleted randomly. Even though the algorithm can, in practice, handle arbitrary edge deletions, its approximation guarantees hold only under the random edge-deletion assumption. The algorithm is similar to the one by Bahmani et al. [5], and it provides a $2(1 + \epsilon)^6$-approximation of the densest subgraph, while requiring polylogarithmic amortized cost per update with high probability.

Core maintenance in evolving graphs. Saríyüce et al. [27] propose the *traversal algorithm*, for efficient core maintenance. This algorithm identifies a small set of vertices that are affected by edge updates and processes these vertices in linear time in order to maintain a valid core decomposition. Li et al. [23] propose an efficient three-stage algorithm for core maintenance in large dynamic graphs. The algorithm maintains a core decomposition of an evolving graph by applying updates to very few vertices in the graph. Once these few vertices have been identified, the algorithm computes the correct core numbers via a quadratic operation.

Finding top-k densest subgraphs. The problem of finding top-k densest subgraphs has been mainly studied for finding *overlapping* subgraphs in static graphs [6, 17].

Next, we discuss how the algorithms presented above (Charikar [10], Batagelj and Zaversnik [7], Bahmani et al. [5], Saríyüce et al. [27], Li et al. [23], and Epasto et al. [15]) can be used to produce top-k densest subgraphs.

Our first observation is that a set of k dense subgraphs can be obtained from any algorithm that finds the densest subgraph by k repeated invocations. The time complexity of computing a set of k dense subgraphs in this manner is simply the running-time complexity of the densest-subgraph algorithm multiplied by k. From a practical point of view, all the static algorithms mentioned are not in-place algorithms, and thus require copying the whole graph for processing. Furthermore, when a vertex or edge is added or deleted from the graph, the whole k dense subgraph computation has to be repeated.

The second observation is that, by using the algorithm of Epasto et al. [15], we can obtain a set of k dense subgraphs by running k instances of the fully-dynamic algorithm in a pipeline manner. The idea is to run k instances of the algorithm in which the output of each instance $i \in \{0 \ldots k - 1\}$ is fed into the next $(i + 1)$ instance as a removal operation. The pipeline version of the algorithm requires keeping k copies of the input graph and an additional

$O(kn)$ size space for bookkeeping. Note that the output of each instance of the pipeline might cascade, which requires updating the vertices in all the instances. In particular, vertices that cease to be part of solution in upstream instances need to be added in downstream instances. Likewise the vertices that become part of densest subgraphs in upstream instances need to be removed from a downstream instances. The modification of the algorithm, as discussed above, is expensive in terms of memory, as it requires replicating the graph and the algorithm's structures k times. In addition, running and maintaining k parallel instances makes the algorithm compute-intensive.

Finally, to maintain top-k densest subgraphs in evolving graphs, we can leverage algorithms for core decomposition maintenance [23, 27]. by leveraging Lemma 2.5, Thus, the idea is to find and maintain top-k disjoint maximum cores. In order to maintain such cores we run a single instance of the algorithm by Saríyüce et al. [27] or Li et al. [23] that maintains the core number of all the vertices in the graph. We then extract the top-k densest subgraphs by: (*i*) extracting the main core, (*ii*) removing the vertices in the main core and updating the core number for rest of the vertices, and (*iii*) repeating the steps until k subgraphs are extracted.

4 ALGORITHM

The main idea of our algorithm is to maintain and update multiple dense subgraphs online. These subgraphs are candidates for the top-k densest subgraphs. However, maintaining multiple subgraphs for fully-dynamic streams requires answering two interesting questions: (*i*) how to reduce the search space of the solution, and (*ii*) how to split the whole graph into subgraphs.

To answer the aforementioned questions, we make two observations. First, since dense subgraphs are formed by relatively high-degree vertices, one can find dense subgraphs by keeping track of these high-degree vertices only. Second, these subgraphs can be updated locally upon edge updates, without affecting the other parts of the graph.

Based on these observations, we develop an algorithm that reduces the solution space by considering only high-degree vertices, and divides the whole graph into smaller subgraphs, each representing a dense subgraph. The top-k densest subgraphs among the candidate subgraphs provide a solution for Problem 2.8.

We begin by designing an algorithm to find the densest subgraph (top-1) and then we extend it to find the top-k densest subgraphs. Our algorithm might not be the most efficient solution for the (top-1) densest-subgraph problem per se, but it provides efficient outcomes when extended to solve the top-k densest-subgraph problem.

We start by defining some properties of the densest subgraph that we leverage in our algorithm.

LEMMA 4.1. *Given an undirected graph $G = (V, E)$, the densest subgraph $S^* \subseteq V$ with density ρ_{S^*}, all the vertices $v \in S^*$ have degree $d_{S^*}(v) \geq \rho_{S^*}$.*

PROOF. This lemma holds according to the definition of optimal density. In an optimal solution, each vertex has degree larger than or equal to ρ_{S^*}. Otherwise, removing the vertex from the subgraph will increase the average degree, and thus the density, of the subgraph. □

Given Lemma 4.1, at any time t, the densest subgraph S_t^* of graph G_t contains only vertices v that have degree $d(v) \geq d_{S_t^*}(v) \geq \rho_{S_t^*}$. Then, given G_t and $\rho_{S_t^*}$, we want to compute the densest subgraph after the addition of a new vertex $u \notin V_t$ at time $t+1$.

Let $d(u)$ be the degree of vertex $u \in V_{t+1}$ and S_{t+1}^* be the densest subgraph at time $t+1$. For simplicity, assume that the graph G_{t+1} is connected. According to Lemma 4.1, for any vertex u to be the part of the densest subgraph, its internal degree satisfies $d_{S_{t+1}^*}(u) \geq \rho_{S_{t+1}^*}$. As vertex u is added to the graph the new density is always greater, i.e., $\rho_{S_{t+1}^*} \geq \rho_{S_t^*}$. Therefore, for vertex u to be the part of densest subgraph, the degree of vertex u should satisfy $d(u) \geq \rho_{S_t^*}$. Therefore, if the degree of vertex u is lower than the $\rho_{S_t^*}$, it cannot be part of the densest subgraph $\rho_{S_{t+1}^*}$ and can be ignored.

Now, considering the case when $d(u) \geq \rho_{S_t^*}$. Adding the vertex u to densest subgraph will update the density:

$$\rho_{S_{t+1}^*} \leftarrow \frac{|E(S_t^*)| + d_{S_t^*}(u)}{|S_t^*|+1}. \tag{7}$$

We also know that $\rho_{S_{t+1}^*} \geq \rho_{S_t^*}$, which means that

$$\frac{|E(S_t^*)| + d_{S_t^*}(u)}{|S_t^*|+1} \geq \frac{|E(S_t^*)|}{|S_t^*|}. \tag{8}$$

From this inequality it follows $d_{S_t^*}(u) \geq \rho_{S_t^*}$. Using these properties, we ignore the vertices of the new edge that have degree lower than the current estimate of the density.

Further, we are interested in finding the main core in the remaining subgraph of high-degree vertices, as it represents a 2-approximation of the densest subgraph according to Lemma 2.5. To this end, we propose a new data structure that relies on Lemma 4.1, the *snowball*.

4.1 Snowball

A snowball D is an incremental data structure that stores a strongly connected subgraph, which maintains the following invariants:

- The core number $\kappa_D(v)$ of each vertex $v \in D$ inside a snowball is equal to the main core ($C_\ell(D)$) of the snowball.
- All the vertices in the snowball are connected.

These invariants ensure that all the vertices in the snowball have the same core number, which is the main core of the snowball by definition. A snowball maintains these invariants while handling the following graph update operations: a) adding/removing a vertex, and b) adding/removing an edge.

4.2 Bag of snowballs

The high-degree vertices in the graph are assigned to a snowball. As these vertices are not strongly connected, they might end up in different snowballs. We store each of these disconnected snowballs in a data structure called the bag, denoted by B.

The bag ensures that each snowball is vertex disjoint. Further, the bag provides an additional operation: extracting the densest snowball among the set of snowballs. The density of the extracted snowball is the maximal density, which is the threshold separating the high-degree vertices from the low-degree ones. We denote this estimate of the maximal density $\tilde{\rho}_{S^*}$.

The bag is a supergraph which contains a set of snowballs and all the edges between the snowballs. We maintain all the core numbers of the nodes in the bag by leveraging a core decomposition

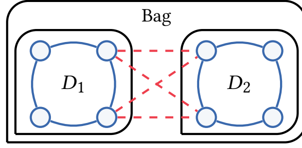

Figure 2: Example showing that the bag requires maintaining the core number of the vertices. Initially, the bag contains two snowballs with core number 2, i.e., D_1 and D_2. Consider the arrival of the edges shown in red. The greedy assignment of the edges might skip creating a new snowball with core number 3, using the four nodes in the middle.

algorithm (see Section 3). The core number of each node in the bag is used to ensure that each node has the maximum possible core number. Figure 2 provides an example explaining one of the issues that may arise. In the example, the bag contains two snowballs, however, it is possible to produce a new snowball with a larger core number. Next, we define the algorithms to update this data structure upon graph updates.

4.3 Addition operations

Vertex addition: As discussed in Section 2, the updates appear in the form of an edge stream. Here, we define the vertex addition algorithm that acts as a helper for edge addition. The algorithm is triggered when at least one of the endpoints of a new edge is a high-degree vertex. In particular, there are two cases to consider: 1) the bag already contains the high-degree vertex, and 2) the bag does not contain the high-degree vertex. In both cases, the goal is to add the new vertex to one of the snowballs (if needed).

Algorithm 1 defines the algorithm for vertex addition. For the first case, the algorithm scans the bag to find the snowball that contains the vertex and returns it. For the second case, the algorithm first identifies the candidate snowballs, then it assigns the vertex to one of the candidate snowballs. The candidate snowballs are the ones having the main core number smaller than or equal to the internal degree of the new vertex ($\kappa_u(D) \geq C_\ell(D)$). Among the candidate snowballs, the new node is assigned to the snowball with maximum internal degree $d_u(D)$, breaking ties randomly.

Once a vertex is added to a snowball, the core number of the snowball may increase. This change requires removing the vertices with core number lower than the main core of the snowball. This procedure can be implemented efficiently in linear time by sorting the vertices based on their degree similar to bin sort.[2]

Verification. Due to the greedy assignment of vertex to the snowball, it is possible that the vertex ends up not having the highest possible core number. For example, Figure 3 shows an example where the greedy assignment does not result in optimal solution.

Therefore, after addition, the algorithm ensures that the core number of the snowball, where the new vertex is added, equals the core number of the new vertex in the graph. The algorithm verifies that the core number of the added vertex by comparing it with the core number of the vertex in the bag. Note that the bag represents the supergraph containing all the snowballs and edges between the snowballs. If the core number within the bag is larger than the one in the snowball, the algorithm merges all the

[2]The MaintainInvariant method at line 12 of Algorithm 1.

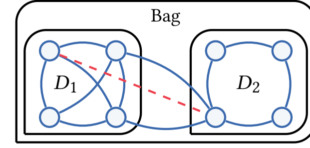

Figure 3: Example showing that arrival of a new edge allows the vertex that is part of snowball D_2 to become part of snowball D_1, which has greater core number (3).

Algorithm 1 Vertex Addition in the Bag of Snowballs

```
1:  procedure ADDTOBAG(u)
2:      S* ← ∅
3:      for D_i ∈ B do
4:          if u ∈ D_i then
5:              return D_i                                  ▷ First Case
6:          if (d_{D_i}(u) ≥ C_ℓ(D_i) and ρ_{S*} < ρ_{D_i}) then
7:              S* ← D_i
8:      if S* = ∅ then
9:          S* ← {u}
10:     else
11:         S* ← S* ∪ {u}
12:         MAINTAININVARIANT(S*)
13:     return S*                                           ▷ Second Case
```

Algorithm 2 Maintain Invariant

```
1:  procedure MAINTAININVARIANT(D_i)
2:      do
3:          repeat ← false
4:          for u ∈ D_i do
5:              if ((κ_{D_i}(u) < C_ℓ(D_i)) then
6:                  D_i ← D_i \ {u}
7:                  if (d(u) ≥ ρ̃_{S*}) then
8:                      ADDTOBAG(u)
9:                  repeat ← true
10:     while repeat
```

snowballs in the induced subgraph of newly added vertex. As all the vertices in the induced subgraph have the same core number, merging them ensures creating a larger snowball.[3] We leverage the core decomposition algorithm by Saríyüce et al. [27] for the implementation.

Algorithm 3 Fix Main Core

```
1:  procedure FIXMAINCORE
2:      for D_i ∈ B do
3:          if (D_i ∩ I(u) > 0) then
4:              D_u ← D_u ∪ D_i
5:      MAINTAININVARIANT(D_u)
```

Theorem 4.2. *Given the bag B, the algorithm ensures that B contains the main core of the graph within one of the snowballs after the vertex addition.*

Proof. Let us assume that at time t the bag contains the main core of the graph. Now, we need to show that at time $t+1$, after the node addition, the bag still contains the main core of the graph. In general, vertex addition method is called whenever there is an edge addition. The only way for the new vertex to affect the main core of the graph is that the new vertex is the part of the main core.

[3]The FixMainCore method at line 15 of Algorithm 4.

Algorithm 4 Edge Addition

```
1: procedure addEdge((u,v))
2:   if ((d(u) < ρ̃_{S*})) and (d(v) < ρ̃_{S*})) then
3:     return
4:   else if ((d(u) ≥ ρ̃_{S*}) and (d(v) < ρ̃_{S*})) then
5:     D_u ← addtoBag(u)
6:   else if ((d(u) < ρ̃_{S*}) and (d(v) ≥ ρ̃_{S*}) then
7:     D_v ← addtoBag(v)
8:   else
9:     D_u ← addtoBag(u)
10:    D_v ← addtoBag(v)
11:    if (D_u = D_v) then
12:      D_u ← D_u ∪ (u, v)
13:      MaintainInvariant(D_u)
14:    else if (v ∈ I(u)) then
15:      fixMainCore(u)
```

Algorithm 5 Edge Deletion

```
1: procedure removeEdge((u,v))
2:   if ((d(u) < ρ̃_{S*}) and (d(v) < ρ̃_{S*})) then
3:     return
4:   if ((d(u) < ρ̃_{S*}) and (d(u) + 1 ≥ ρ̃_{S*})) then
5:     removeVertex(u)
6:     return
7:   if ((d(v) < ρ̃_{S*}) and (d(v) + 1 ≥ ρ̃_{S*})) then
8:     removeVertex(v)
9:     return
10:  for D_i ∈ B do
11:    if (D_i ∩ (u, v) ≠ ∅) then
12:      D_i ← D_i \ (u, v)
13:      for x ∈ D_i do
14:        if (κ_B(x) > κ_{D_i}(x)) then
15:          fixMainCore(x)
16:      MaintainInvariant(D_i)
```

After the addition of the vertex in the bag, the algorithm verifies the core number by comparing the core number of the vertex in the bag and the snowball. If the core number of the vertex in the bag is greater, the algorithm merges the snowballs containing the vertices in the induced graph of the new node in the bag. This creates a new snowball with a greater core number. ◻

Edge addition: In this case, the state of the bag is only affected if at least one of the vertices in the new edge is a high-degree vertex. In particular, there are two cases to consider: a) only one of the vertices is a high-degree vertex and b) both the vertices are high-degree vertices. For the first case, the algorithm leverages the vertex addition method to add the vertex to the bag of snowballs. For the second case, when both vertices are added to the bag of snowballs, the algorithm verifies that the main core exists in the bag. When both vertices are added to the same snowball, the algorithm adds the new edge to the same snowball and ensures that the invariant holds. Conversely, when the two vertices are added to two different snowballs, the algorithm verifies if the vertices exist in each others' induced subgraphs and fixes the main core for both the vertices. Algorithm 4 describes the algorithm for edge addition.

Theorem 4.3. *Algorithm 4 maintains the main core of the graph in one of the snowballs inside the bag.*

Proof. The proof for all the cases, other than the case when both the vertices of the new edge are assigned to two different snowballs, is similar to the vertex addition algorithm. Therefore, we consider the case when both end vertices of the added edge are added to two different snowballs. For this case, we rely on the graph in the bag. We check if both vertices are in the same core graph in the bag, and fix the core number of the vertices if they belong to the same induced subgraph. This ensures creating the graph with the highest core number. ◻

4.4 Removal operations

Vertex removal: Similarly to the addition operations, we first define the procedure for removing a vertex from the bag of snowballs. The vertex removal method is used as a subroutine for the edge removal process. A vertex is only removed from the bag when its degree becomes lower than the maximal density. Therefore, according to Lemma 2.5, the removed vertex cannot be part of the main core. The algorithm removes the vertex from the snowball within a bag without doing any other operation.

Edge removal: Now we turn our attention to edge deletion, which follows the same pattern as edge addition. Again, we leverage the bag to ensure that there exist a snowball with a core number equal to the main core of the graph. Algorithm 5 shows the algorithm for edge deletion. The bag does not require any update when either one of the vertices is low-degree or if both the vertices belong to two different snowballs. Therefore, we consider the case when one of the vertices lie at the boundary of high-degree vertices. That is, the edge deletion moves the vertex from the high-degree to low-degree. In this case, the algorithm only requires removing the vertex from the bag, without performing any other operations.

The interesting case is where both vertices of the deleted edge are high-degree and belong to the same snowball. In this case, the algorithm removes the edge from the snowball. Further, it verifies and updates (if needed) the core number of the vertices affected by the update in the snowball. Lastly, edge deletion might reduce the maximal density, and thus require adding to the bag new vertices whose degree is now greater than the new maximal density.

Theorem 4.4. *Given the bag containing a snowball with the same core number as the main core of the bag, after the edge deletion, Algorithm 5 maintains the main core of the graph in one of the snowballs inside the bag.*

Proof. An edge removal affects the bag of snowballs only when both the vertices corresponding to the removed edge belong to the same snowball. In this case, edge removal might reduce the core number of all the vertices in the snowball. The algorithm ensures that the vertices in the snowball have their maximum possible core number by comparing their core number in the snowball with the core number in the bag. Therefore, by verifying and fixing the core numbers, the algorithm maintains the main core of the graph in one of the snowballs inside the bag. ◻

4.5 Fully-dynamic top-k densest subgraphs

Now that we have a fully dynamic algorithm for finding the densest subgraph in sliding windows, we move our attention to the top-k densest-subgraph problem.

Let $\tilde{\rho}_{S^*} \geq \rho_1 \geq \ldots \geq \rho_{z-1}$ represent the densities of the z subgraphs in the bag, where ρ_{S^*} is the density of the densest subgraph. The bag contains the vertices that have a degree greater than the density $\tilde{\rho}_{S^*}$. To ensure that the bag contains at least k subgraphs,

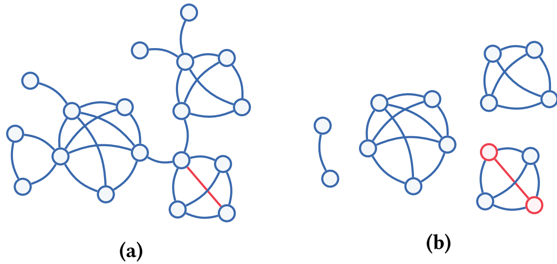

Figure 4: Example showing top-k version of the algorithm. The densities of top-3 subgraphs, before the arrival of the new edge in the bag are 1.8, 1.5 and 1.25. The algorithm stores all the vertices in the bag with degree greater than equal to 1.25, as shown in Figure 4b. After the arrival of the new edge, the update only affects one of the subgraphs in the bag.

we modify the algorithm to keep all the vertices with degree greater than ρ_{k-1}. The only modification required is to replace ρ_{S^*} by ρ_{k-1} in Algorithm 2, Algorithm 4 and Algorithm 5. Further, we leverage the priority queue to extract ρ_{k-1} from the bag of snowballs. This simple modification ensures that the bag contains at least k subgraphs and enables accessing the top-k densest subgraphs in the sliding window model. Note that the new definition of high-degree vertices is related to the density of the k-th top densest subgraph.

The modified algorithm guarantees that the bag contains the vertices with a degree greater than the ρ_{k-1} after any graph update. These graph updates include both edge additions and removals. It is necessary for the algorithm to consider edge additions, as they might affect the value of ρ_{k-1} by merging multiple subgraphs. Similarly, edge deletions have to be considered, as they might affect the value of ρ_{k-1} by reducing the density of any of the top-k densest subgraphs, merging multiple subgraphs, or splitting a subgraph.

As the algorithm ensures keeping the main core in the bag, it guarantees 2-approximation for the first densest subgraph ($k = 1$) while providing a high-quality heuristic for $k > 1$ compared to other solutions. We provide an example in Figure 4 for the top-k densest subgraph algorithm by expanding on the Figure 1a. We conclude this section with the theorem that provides a bound for our proposed algorithm. These bounds can be generalized for any algorithm that adapts to a solution for densest subgraph problem for the top-k case (see section 3 for examples).

Theorem 4.5. *Given an integer k, the bag contains a set of subgraphs that provides $2k$-approximation of the top-k densest-subgraph problem.*

Proof. We know that the graph does not contain any subgraph with density greater than the optimal density (ρ_{S^*}). Therefore, we know that the sum of densities for the top-k densest-subgraph problem is upper bounded by $k \times \rho_{S^*}$.

Further, the bag contains $\tilde{\rho}_{S^*}$, which provides a 2-approximation of the densest subgraph. This implies that the sum of densities of top-k densest subgraph in the bag $\geq \frac{1}{2}\rho_{S^*}$. Putting above two observations together, we can clearly see that the bag provides a solution that is a $2k$-approximation for the problem. □

4.6 Data structure

The proposed algorithm requires accessing the neighborhood information of every node. Specifically, we are interested in performing three queries on a given vertex: a) extract its degree, b) extract all its neighbors, and c) given density $\tilde{\rho}_{S^*}$, extract all the vertices with degrees greater than the density $\tilde{\rho}_{S^*}$.

Vertex map: To answer the first two queries, we need to store the neighborhood information for all vertices. We store the information in a hashmap with keys being the vertex identifiers and values being the neighbors of each vertex. Vertex map allows performing search and update operations in amortized constant time.

Degree table: For the third query, we need to order the vertices by their degrees. We use bin sort to order the vertices by their degrees, which enables extracting the vertices with degrees greater than the density $\tilde{\rho}_{S^*}$ in constant time.

5 DISCUSSION

Most of the solutions that leverage a static algorithms [5, 7, 10] require iterating over the entire graph k times upon any update. Comparatively, our algorithm mostly touches a limited region in the graph for any updates, which makes our algorithm perform significantly faster.

The top-k densest subgraph algorithm that adapts to Epasto et al. [15] requires replicating the graph across the k instances. In addition, updates across k instances might cascade and require running the algorithm k times, similarly to the static top-k solutions. Our algorithm outperforms the pipeline version of the top-k algorithm both in terms of memory and computation.

Finally, incremental algorithms for k-core maintenance requires removing top-k densest subgraph upon every update in the graph. Each removal of a densest subgraph requires updating the core number of the remaining vertices. Our algorithm efficiently avoids this removal step by maintaining the subgraphs during execution.

Performance. The proposed algorithm leverages the skew in real-world graphs and ignores a major portion of the input stream on the fly. In addition, high-degree vertices are stored as small subgraphs so that each update is often applied locally on these subgraphs. This design allows our algorithm to operate on just small portions of the graph for each update, rather than iterating on the whole graph. Finally, we use the k-core algorithm [23, 27], which allows each high-degree node to update their core number with a complexity independent of the graph size. These improvements enable our algorithm to perform significantly better than the other state-of-the-art streaming algorithms.

Memory. Our algorithm requires only $O(n^2 \text{polylog} n)$ memory, compared to $O(kn^2 \text{polylog} n)$ for the top-k version of [15] and $O(n^2 \text{polylog} n)$ for the other algorithms discussed in Section 3.

Tight Bounds. We show via an example that any algorithm for densest subgraph problem can only produce a k-approximation for top-k densest subgraph problem. Consider a graph G that contains $n \cdot m$ vertices, for any $n \geq k$. The n nodes connect to form a circle. In the circle, there is one edge connecting two non-adjacent vertices. Additionally, each of the n nodes connects to exactly m neighbors. The inner circle of the graph G is the densest subgraph. Removing the densest subgraph leaves the rest of the vertices completely disconnected. Hence, the sum of density will be equal to one. Now,

Table 1: Datasets used in the experiments.

Dataset	Symbol	n	m	$\overline{d(v)}$
Amazon [22]	AM	334 863	925 872	5.52
DBLP 1 [22]	DB_1	317 080	1 049 866	6.62
Youtube [22]	YT	1 134 890	2 987 624	5.26
DBLP 2[4]	DB_2	1 314 050	18 986 618	28.88
Live Journal[5]	LJ	5 204 176	49 174 620	18.90
Orkut [22]	OT	3 072 441	117 185 083	76.28
Friendster [22]	FR	65 608 366	1 806 067 135	55.04

consider the case when the densest subgraph is not removed first. In this case, each node in the circle along with its m neighbors create a subgraph of density almost equal to one, for higher values of m. Therefore, one can return k such subgraphs with the sum of densities equal to k, which is k times better than the previous case.

6 EVALUATION

We conduct an extensive empirical evaluation of the proposed algorithm, and provide comparisons with the existing solutions. In particular, we answer the following questions:

Q1: What is the impact on the quality of subgraphs?

Q2: What are the gains in performance?

Q3: How does the algorithm perform in terms of different input parameters?

6.1 Experimental setup

Datasets. Table 1 shows the datasets used in the experiments. The datasets are selected due to their public availability. We evaluate all the algorithms in the sliding window model. The number of edges in the sliding window is an input parameter.

Metrics. We evaluate the quality and efficiency of the algorithms. We assess the quality by the objective function, i.e., the sum of densities of the subgraphs produced by an algorithm. We evaluate the efficiency of an algorithm by reporting the average update time and the memory usage. The average update time is the average time it takes to move the sliding window. This includes adding the new edge, removing the oldest edge, and updating the top-k densest subgraphs. We report the memory usage as the average percentage of occupied memory.

Algorithms. Table 2 shows the notations used for different algorithms. The algorithms by Bahmani et al. [5] and Epasto et al. [15] require an additional epsilon parameter for execution, which provides a trade-off between quality and execution time. Here we use the defaults proposed by the authors.

Stream ordering. We consider two commonly used stream ordering schemes [29]:

- BFS: The ordering is a result of a breadth-first search starting from a random vertex.
- DFS: The ordering is a result of a depth-first search starting from a random vertex.

Experimental environment. We conduct our experiments on a machine with 2 Intel Xeon Processors E5-2698 and 128GiB of memory. All the algorithms are implemented in Java and executed on JRE 7 running on Linux. The source code is available online.[6]

Table 2: Notation for the top-k algorithms.

Symbol	Reference Algorithm	Top-1 Approx. Guarantees	In-place
CH	Charikar [10]	2	No
VB	Batagelj and Zaversnik [7]	2	No
BB_ϵ	Bahmani et al. [5]	$2(1+\epsilon)$	No
RL	Li et al. [23]	2	Yes
TR	Saríyüce et al. [27]	2	Yes
AE_ϵ	Epasto et al. [15]	$2(1+\epsilon)$[6]	No
GR	this paper	2	Yes

6.2 Experimental results

Q1: In this experiment, we compare the quality of the results produced by the different algorithm as measured by the objective function. In order to be able to run most of the algorithms, we use the smaller datasets, i.e., AM, DB_1, YT, and DB_2. As RL and TR produce same results in terms of quality, we only report the results for one of them. Due to space constraints, we show the results only with the DFS ordering. However, we achieve similar results also with the BFS one. We set the size of the sliding window $x = 100$k.

For the static algorithms, we execute them in micro-batches in which the top-k densest subgraph is recomputed after 1k edge updates, which gives them a substantial advantage. For BB_ϵ and AE_ϵ, we use $\epsilon = 0.01$. We set the maximum execution time to 7 days, due to which the AE_ϵ is not able to finish on DB_2 and YT. Finally, RL and TR contain an update phase due to the iterative extraction of the top-k densest subgraph. This phase is expensive, as the main core consists of high-degree vertices. To alleviate this issue, we execute the extraction phase every 10k update operations.

Figure 5 shows the quality results. Our algorithm GR achieves competitive quality, often generating denser subgraphs compared to all the other algorithms. For example, for the AM dataset, GR produces subgraphs that are 1.5 times better than the best algorithm among the state-of-the-art solutions. All the other algorithms produce consistent results across all the datasets. For instance, VB, RL, and TR produce top-k dense subgraphs of lowest quality compared to the other algorithms. This result is caused by the removal of the main core (backbone) of the graph in each of the k iterations. In addition, it shows how the problem considered in this paper, while related, is different from a simple k-core decomposition. The results also validate that BB_ϵ and AE_ϵ provide weaker guarantees on the quality compared to CH.

Q2: In this experiment, we turn our attention to evaluate the efficiency of the proposed algorithm, as measured by the average update time and memory usage. We again select several datasets, i.e., AM, DB_1, YT, DB_2, and LJ, and execute CH, VB, BB_ϵ, RL, TR, AE_ϵ and GR. We exclude the largest datasets as only a few algorithms are able to handle them. The algorithms are executed with both BFS and DFS ordering of the edge stream. The sliding window size is set to $x = 100$k edges, and $k = 10$ unless otherwise specified.

[4]http://konect.uni-koblenz.de/networks/dblp_coauthor
[5]http://konect.uni-koblenz.de/networks/livejournal-links
[6]https://github.com/anisnasir/TopKDensestSubgraph

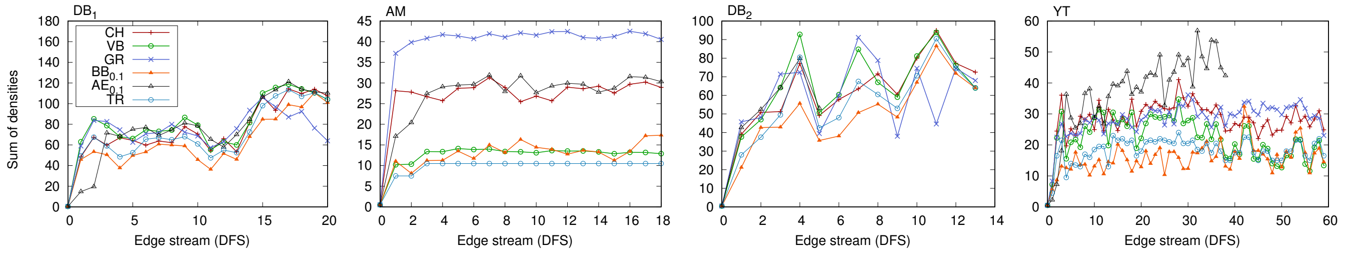

Figure 5: Sum of densities for top-10 densest subgraphs on the DB_1, AM, DB_2 and YT datasets, sliding window size 100k.

Table 3: Memory consumption as a percentage of total memory for algorithms with DB_2 and LJ dataset, sliding window size 100k.

	CH	VB	BB_ϵ	RL	TR	AE_ϵ	GR
DB_2	2.10	2.10	2.10	2.1	2.8	5	2.7
LJ	1.80	1.80	1.80	1.8	2.1	5.5	2.4

Figure 6 shows the efficiency results (note the logarithmic scale). The proposed algorithm, *GR*, outperforms all other ones in terms of update time for all the datasets and both ordering schemes. *RL* and *TR* are the slowest algorithms. In particular, for DFS, our algorithm achieves a performance gain of five orders of magnitude. This result is noteworthy as even though our algorithm is dependent on core decomposition, it is still able to beat a naïve application of those algorithms by a wide margin. This difference is due to the fact that both algorithms require maintaining the core number for all the vertices. Additionally, the extraction of top-k densest subgraph further hampers their efficiency.

Algorithms *CH*, *VB*, and $BB_{0.01}$ perform unfavorably due to their static nature. In this case, our algorithm is able to achieve more than three orders of magnitude improvement in efficiency.

Algorithm $AE_{0.01}$ is best performing among the baselines, and even outperforms *GR* in one dataset for one specific ordering (AM with BFS). However, *GR* still outperforms $AE_{0.01}$ by nearly three orders of magnitude in most other cases.

Finally, we observe the overall memory consumption of all the algorithms (reported in Table 3). The memory requirement of our algorithm lies between the static and the dynamic algorithms, i.e., $AE_{0.01}$ requires the largest amount of memory, while the static algorithms require the least. These results are in line with our expectations from the discussion in Section 3.

Q3: In this experiment, we evaluate the scalability of our algorithm. First, we execute *GR* with different values of k. Alongside, we report the average update time for BB_ϵ algorithm. We choose BB_ϵ, rather than AE_ϵ, as it can execute in mirco-batches, i.e., top-k densest subgraph every 100 edges. We set the sliding window size $x = 1$M, and use the YT and DB_2 dataset with DFS ordering. Figure 7 reports the average update time for both algorithms. The average update time of *GR* remains consistent even for higher values of k, whereas the execution time of $BB_{0.01}$ increases with the parameter.

Second, we execute *GR* with different sizes of sliding window, i.e., $x = 10$k, 100k, 1M, and 10M. We set $k = 10$ for this experiment. Again, we use the YT and DB_2 datasets with DFS ordering.

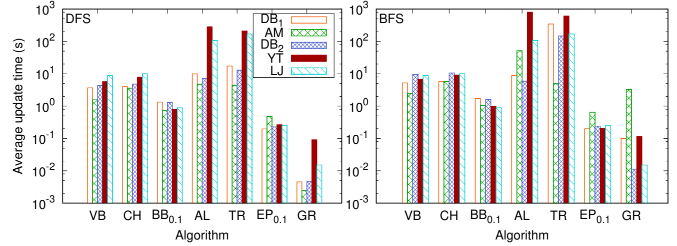

Figure 6: Update time for the algorithms on the DB_1, AM, DB_2, YT, and LJ datasets, when using both DFS and BFS ordering and $k = 10$.

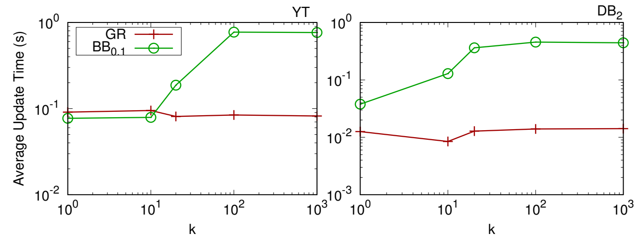

Figure 7: Update time for *GR* and $BB_{0.01}$ on the YT and DB_2 datasets as a function of k. Note that $BB_{0.01}$ is executed in micro-batches of size 100.

Table 4: Average update time for the *GR* algorithm with sliding windows of different sizes x.

	$x = 10$k	$x = 100$k	$x = 1$M	$x = 10$M
YT	0.80ms	91.14ms	90.33ms	95.49ms
DB_2	4.97ms	7.58ms	8.45ms	32.21ms

Table 4 reports the average update time for different configurations. Increasing the size of the sliding window does not affect the average update time significantly. This result validates our claim that most of the updates are local to the subgraphs, and do not require iterating through the whole graph to extract the top-k densest subgraphs.

Finally, we study the performance of our algorithm in on the largest datasets. We select the OT and FR datasets with DFS ordering, and execute the algorithm with a sliding window of $x = 100$k, with $k = 10$. Figure 8 shows the result of the experiment. The plot is generated by taking the moving average of the update time and the sum of densities. The average update time of the algorithm

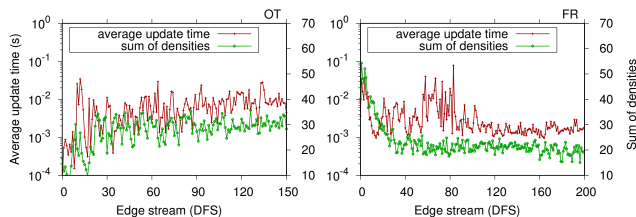

Figure 8: Quality and efficiency for *GR* on the OT and FR datasets over the stream.

mostly remains constant throughout the execution, and our algorithm provides steady efficiency over the stream. This behavior remains consistent even when the densities are fluctuating, as in the case for the OT dataset.

7 RELATED WORK

Valari et al. [32] were the first one to study the top-k densest subgraph problem for a stream consisting of a dynamic collection of graphs. They proposed both an exact and an approximation algorithm for top-k densest subgraph discovery. Similar to our algorithm, the proposed algorithm relies on the core decomposition to provide the approximation guarantees. The top-k densest subgraphs produced by the algorithm are edge-disjoint. Balalau et al. [6] studied the problem of finding the top-k densest subgraph with limited overlap. They defined the top-k densest subgraphs as a set of k subgraphs that maximizes the sum of densities, while satisfying an upper bound on the pairwise Jaccard coefficient between the set of vertices of the subgraphs. The problem of finding the top-k densest subgraph as sum of densities was shown to be NP-hard [6] and efficient heuristic was proposed to solve the problem. Further, Galbrun et al. [17] studied the problem of finding the top-k overlapping densest subgraphs and provided constant-factor approximation guarantees.

8 CONCLUSION

We studied the top-k densest subgraphs problem for graph streams, and proposed an efficient one-pass fully-dynamic algorithm. In contrast to the existing state-of-the-art solutions that require iterating over the entire graph upon update, our algorithm maintains the solution in one-pass. Additionally, the memory requirement of the algorithm is independent of k. The algorithm is designed by leveraging the observation that graph updates only affect a limited region. Therefore, the top-k densest subgraphs are maintained by simply applying local updates to small subgraphs, rather than the complete graph. We provided a theoretical analysis of the proposed algorithm and showed empirically that the algorithm often generates denser subgraphs than the state-of-the-art solutions. Further, we observed an improvement in performance of up to five orders of magnitude when compared to the baselines.

This work gives rise to further interesting research questions: Is it necessary to leverage k-core decomposition algorithm as a backbone? Is it possible to achieve stronger bounds on the threshold for high-degree vertices? Can we design an algorithm with a space bound on the size of the bag? Is it possible to achieve stronger approximation guarantees for the problem? We believe that solving these questions will further enhance the proposed algorithm, making it a useful tool for numerous practical applications.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 349–360.

[2] Albert Angel, Nikos Sarkas, Nick Koudas, and Divesh Srivastava. 2012. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB* 5, 6 (2012), 574–585.

[3] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 1996. Greedily finding a dense subgraph. In *SWAT*. 136–148.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and issues in data stream systems. In *PODS*. ACM, 1–16.

[5] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *VLDB* 5, 5 (2012), 454–465.

[6] Oana Denisa Balalau, Francesco Bonchi, TH Chan, Francesco Gullo, and Mauro Sozio. 2015. Finding subgraphs with maximum total density and limited overlap. In *WSDM*. ACM, 379–388.

[7] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[8] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*. 119–130.

[9] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *STOC*. ACM, 173–182.

[10] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *Approx. Algo. for Comb. Opt.* 84–95.

[11] Jie Chen and Yousef Saad. 2012. Dense subgraph extraction with application to community detection. *TKDE* 24, 7 (2012), 1216–1230.

[12] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. 2013. Dynamic graphs in the sliding-window model. In *ESA*. Springer, 337–348.

[13] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813.

[14] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2007. Extraction and classification of dense communities in the web. In *WWW*. ACM, 461–470.

[15] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient Densest Subgraph Computation in Evolving Graphs. In *WWW*. 300–310.

[16] David Eppstein, Zvi Galil, and Giuseppe F Italiano. 1998. *Dynamic graph algorithms*. Springer.

[17] Esther Galbrun, Aristides Gionis, and Nikolaj Tatti. 2016. Top-k overlapping densest subgraphs. *Data Mining and Knowledge Discovery* (2016), 1–32.

[18] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *VLDB*. VLDB Endowment, 721–732.

[19] Aristides Gionis and Charalampos E Tsourakakis. 2015. Dense subgraph discovery: Kdd 2015 tutorial. In *SIGKDD*. ACM, 2313–2314.

[20] Andrew V Goldberg. 1984. *Finding a maximum density subgraph*. University of California Berkeley, CA.

[21] Guy Kortsarz and David Peleg. 1994. Generating sparse 2-spanners. *Journal of Algorithms* 17, 2 (1994), 222–236.

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[23] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient core maintenance in large dynamic graphs. *TKDE* 26, 10 (2014), 2453–2465.

[24] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T Vu. 2015. Densest Subgraph in Dynamic Graph Streams. In *MFCS*. Springer, 472–482.

[25] James Orlin. 2013. Max flows in $O(nm)$ time, or better. In *STOC*. 765–774.

[26] Polina Rozenshtein, Aris Anagnostopoulos, Aristides Gionis, and Nikolaj Tatti. 2014. Event detection in activity networks. In *SIGKDD*. 1176–1185.

[27] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *VLDB* 6, 6 (2013), 433–444.

[28] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *KDD*. ACM, 939–948.

[29] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *KDD*. ACM, 1222–1230.

[30] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly graph decomposition. In *WWW*. ACM, 1089–1099.

[31] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*. ACM, 104–112.

[32] Elena Valari, Maria Kontaki, and Apostolos N Papadopoulos. 2012. Discovery of top-k dense subgraphs in dynamic graph collections. In *SSDBM*. 213–230.

[33] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *ICML* 42, 1 (2015), 181–213.