Maintaining Densest Subsets Efficiently in Evolving Hypergraphs

Conference Paper · November 2017

DOI: 10.1145/3132847.3132907

3 authors, including:

Shuguang Hu
The University of Hong Kong
5 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Xiaowei Wu
University of Macau
53 PUBLICATIONS **419** CITATIONS
SEE PROFILE

Maintaining Densest Subsets Efficiently in Evolving Hypergraphs*

Shuguang Hu
The University of Hong Kong
Pokfulam Road, Hong Kong
sghu@cs.hku.hk

Xiaowei Wu
The University of Hong Kong
Pokfulam Road, Hong Kong
wxw0711@gmail.com

T-H. Hubert Chan
The University of Hong Kong
Pokfulam Road, Hong Kong
hubert@cs.hku.hk

ABSTRACT

In this paper we study the densest subgraph problem, which plays a key role in many graph mining applications. The goal of the problem is to find a subset of nodes that induces a graph with maximum average degree. The problem has been extensively studied in the past few decades under a variety of different settings. Several exact and approximation algorithms were proposed. However, as normal graph can only model objects with pairwise relationships, the densest subgraph problem fails in identifying communities under relationships that involve more than 2 objects, e.g., in a network connecting authors by publications.

We consider in this work the densest subgraph problem in hypergraphs, which generalizes the problem to a wider class of networks in which edges might have different cardinalities and contain more than 2 nodes. We present two exact algorithms and a near-linear time r-approximation algorithm for the problem, where r is the maximum cardinality of an edge in the hypergraph. We also consider the dynamic version of the problem, in which an adversary can insert or delete an edge from the hypergraph in each round and the goal is to maintain efficiently an approximation of the densest subgraph. We present two dynamic approximation algorithms in this paper with amortized $\text{poly}(\frac{r}{\epsilon}\log n)$ update time, for any $\epsilon > 0$. For the case when there are only insertions, the approximation ratio we maintain is $r(1+\epsilon)$, while for the fully dynamic case, the ratio is $r^2(1+\epsilon)$. Extensive experiments are performed on large real datasets to validate the effectiveness and efficiency of our algorithms.

KEYWORDS

Densest Subgraph, Graph Mining, Dynamic Data Structure

1 INTRODUCTION

In many data mining applications [4, 16, 18, 22, 26, 38], it is usually important to extract a dense subset of nodes from a large graph. In the Densest Subgraph Problem, we are given a (hyper) graph $H(V, E)$ (where $E \subseteq 2^V$), and the problem is to find a subset of nodes $S \subseteq V$ such that the *density* $\rho(S) = |E[S]|/|S|$ is maximized, where $E[S] = \{e \in E : e \subseteq S\}$ is the set of edges induced by S. In normal graphs, the problem is equivalent to finding a subgraph with maximum average degree. In the weighted setting, each $e \in E$ (resp. $u \in V$) has a non-negative integer weight w_e (resp. w_u), and the density of $S \subseteq V$ is defined as the ratio between the total edge weights $w(E[S])$ and the total node weights $w(S)$.

The Densest Subgraph Problem has enormous applications to a wide variety of problems, ranging from community detection [27, 32, 36], expert team formation [12] to computational biology [6, 21, 34]. As one of the most fundamental problems in graph mining, the problem has been extensively studied for decades. One of the first to study the problem was Goldberg [23], who provided a polynomial-time algorithm for the problem using $O(\log n)$ max-flow computations, where $n = |V|$. While Goldberg's algorithm gives an exact solution for the problem by an elegant reduction to max-flow computations, a more popular algorithm is the linear time 2-approximation algorithm by Charikar [15]. By iteratively removing nodes with minimum degree and returning the intermediate subgraph with maximum density, Charikar's algorithm computes a 2-approximation of the densest subgraph in $O(m)$ time, where $m = |E|$. An alternative exact-solution algorithm based on LP was also proposed in [15], in which the Densest Subgraph Problem is formulated as an LP with $O(m + n)$ variables and the densest subgraph is induced by the nodes with non-zero variables in the optimal LP solution.

1.1 Densest Subset in Hypergraphs

Despite the enormous results on the Densest Subgraph Problem in normal graphs, traditional model fails in extracting communities that are bound together by relationships that involve more than 2 parties. In many applications, the set of objects (nodes) under consideration are connected by relationships (hyperedges) that involve more than 2 objects. In this case it is inaccurate to describe the relationships between objects using normal edges. It is hence natural to consider the Densest Subgraph Problem in hypergraphs.

For example, in the DBLP network, each node represents an author and each edge represents a publication. Since it often happens that a publication is written by more than 2 authors, the network can only be modeled by hypergraphs. Given the hypergraph, the goal of the Densest Subgraph Problem is to identify a group of researchers S such that the average number of collaborations *within* S is maximized. It is also natural to consider the weighted case since it is obvious that the impacts, e.g., number of citations, of publications can be quite different, and one may wish to look for a group of researchers with maximum "research impact" density. Compared

*This research was partially supported by the Hong Kong RGC under the grants 17200214 and 17217716.

CIKM'17 , November 6–10, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132907

with existing solutions [19, 37], which model the problem using a "co-authorship" normal graph, our modeling using hypergraph is undoubtedly more accurate, as the co-author relationship favors publications with many authors, which is counter-intuitive.

Other applications of the problem in hypergraphs include event detection on Twitter, where each tweet can involve many tags, and collusion detection in Bitcoin transactions [31], where each transaction may involve multiple parties. Recently, the problem has found applications in the spectral analysis of hypergraphs [14, 29].

Surprisingly, while the problem is very natural and has a wide range of applications in theory and data mining, it is not well-studied before. A special case of the problem on unweighted r-uniform hypergraphs (in which we have $|e| = r$ for all $e \in E$) was recently studied by Tsourakakis [37], who introduced the r-clique Densest Subgraph Problem, for $r = O(1)$. Given an unweighted normal graph and an integer r, they considered the problem of finding a subset of nodes S such that the average number of r-cliques induced by S is maximized. It is easy to observe that their problem is a special case of the Densest Subgraph Problem in r-uniform hypergraphs by simply replacing each r-clique in the graph with a hyperedge (of cardinality r). The problem was later considered by Mitzenmacher et al. [33], who proposed randomized approximation algorithms with better space and time complexities.

1.2 Dynamic Setting

In the aforementioned applications, e.g., DBLP, Twitter and Bitcoin transactions, the hypergraphs are inherently dynamic, e.g., new publications appear in the DBLP graph almost everyday.

It is assumed in the dynamic setting that every update comes online and can either insert an edge to, or remove an existing edge from the graph [5, 11, 19, 35]. The dynamic Densest Subgraph Problem aims at maintaining an (approximate) densest subgraph under edge insertions and deletions. As the first work to study the efficient maintenance of dense subgraphs in the dynamic setting, Angel et al. [5] analyzed the magnitude of change each single edge weight update can cause in an edge-weighted graph and proposed an algorithm under streaming edge weight updates.

The Densest Subgraph Problem in the streaming model was later considered by Bahmani et al. [7]. Their algorithm makes $O(\frac{1}{\epsilon}\log n)$ passes over the input, for any $\epsilon > 0$, and outputs a subgraph whose density is guaranteed to be within a factor $(2+\epsilon)$ of the optimum. Similar to Charikar's approximation algorithm, their algorithm iteratively removes the nodes with small degrees. However, instead of removing one node at a time, their algorithm fixes a threshold β and removes nodes with degree smaller than β in each iteration. Although Bahmani et al.'s algorithm is neither faster nor simpler than Charikar's, it makes limited passes over the input and can be adapted to support updates of edges in the dynamic setting [11, 19].

The dynamic maintenance of approximate densest subgraph under edge insertions and deletions has recently drawn much attention. Epasto et al. [19] considered the problem where insertions are adversarial and deletions are random: the edge to be deleted is chosen uniformly at random from all existing edges. Based on Bahmani et al.'s $(2+\epsilon)$-approximation algorithm [7], they achieved a randomized $(2+\epsilon)$-approximation algorithm that with high probability handles each update in amortized $\text{poly}(\frac{1}{\epsilon}\log n)$ time, using $O(m+n)$ space. The case when deletions are also adversarial was considered by Bhattacharya et al. [11], who presented a dynamic algorithm that maintains a $(4+\epsilon)$-approximation with amortized $\text{poly}(\frac{1}{\epsilon}\log n)$ update time and $O(n \cdot \text{poly}(\frac{1}{\epsilon}\log n))$ space. Following [11, 19], several other dynamic algorithms for the problem were proposed. McGregor et al. [30], Esfandiari et al. [20] and Mitzenmacher et al. [33] presented semi-streaming algorithms for the problem that maintain a $(1+\epsilon)$-approximation using $O(n \cdot \text{poly}(\frac{1}{\epsilon}\log n))$ space. Their algorithms process each update also in $\text{poly}(\frac{1}{\epsilon}\log n)$ time, but the query-time can be as large as $\Omega(n \cdot \text{poly}(\frac{1}{\epsilon}\log n))$.

However, all the aforementioned results focus on normal graph. As far as we know, we are the first to consider the dynamic maintenance of approximate densest subgraphs in hypergraphs.

1.3 Our Results

We use $r = \max_{e\in E}\{|e|\}$ to denote the maximum cardinality of a hyperedge and define $M := \sum_{e\in E}|e| \le rm$. We give two algorithms for computing the exact solution for the Densest Subgraph Problem in hypergraphs, which generalize the existing algorithms [15, 23, 37]. Both of our algorithms can be applied to the case when both nodes and edges have non-negative integer weights.

Theorem 1.1. *Given a weighted hypergraph $H(V,E)$ with $n = |V|$ nodes and $m = |E|$ edges, the Densest Subgraph Problem can be solved by either using $O(\log W)$ computations of max-flow in a flow network with $O(M)$ edges, where W is the total weight of nodes and edges, or solving a linear program with $O(m+n)$ variables and $O(M)$ constraints.*

Our flow-based algorithm generalizes the result of [37], which works only for unweighted r-uniform hypergraph. In addition, we provide an LP-based solution for hypergraphs that extends Charikar's LP-based solution [15] for normal graphs, and a simple r-approximation algorithm (in Section 2.3) that runs in $\tilde{O}(M)$ time.

We then consider the dynamic version of the problem, in which both the nodes and the edges have unit-weight. We first show that Bahmani et al.'s algorithm [7], which is the basis of most existing dynamic algorithms for normal graphs, can be extended to hypergraphs (Section 3.1). Built on top of that, we develop two dynamic algorithms for the maintenance of approximate densest subgraphs in hypergraphs with amortized $\text{poly}(\frac{r}{\epsilon}\log n)$ update time.

Theorem 1.2. *There exists a dynamic algorithm for the Densest Subgraph Problem in unweighted hypergraphs that maintains an $r(1+\epsilon)$-approximation under arbitrary edge insertions using $O(n)$ extra space, in amortized $\text{poly}(\frac{r}{\epsilon}\log n)$ time per update.*

Theorem 1.3. *There exists a dynamic algorithm for the Densest Subgraph Problem in unweighted hypergraphs that maintains an $r^2(1+\epsilon)$-approximation under arbitrary edge insertions and deletions using $O(rm \cdot \text{poly}(\frac{1}{\epsilon}\log n))$ extra space, in amortized $\text{poly}(\frac{r}{\epsilon}\log n)$ time per update.*

It is worth mentioning that all our algorithms and analysis do not depend on $\min_{e\in E}\{|e|\} \ge 2$, which means that self-loops, e.g., single-author publications, are allowed in the hypergraphs. Moreover, our dynamic algorithms can be applied to multi-graphs and hence both Theorems 1.2 and 1.3 hold for hypergraphs with integer edge weights that are bounded by $\text{poly}(\frac{r}{\epsilon}\log n)$. We doubt that

better results could be obtained for general edge weights since the densest subgraph could change dramatically when an edge with very large weight is inserted or deleted.

Experimental Evaluation. We evaluate our exact and approximation algorithms on several real-world networks. Our experimental results show that our exact-solution algorithm runs efficiently, and returns a subset of nodes with a much larger density than the solution returned by existing result [19], which replaces hyperedges by complete graphs in a normal graph. Moreover, our approximation algorithm runs several times faster than the exact algorithm, and returns a solution with density very close to the optimum.

We perform an extensive evaluation of our dynamic algorithms on dynamic real-world datasets. As in [19], we adopt a sliding window model on the hypergraphs, where hyperedges are added to the graph by the time they are created, e.g., by the time when publications are accepted, while least recent hyperedges are removed. Our experimental results show that we are able to maintain a dense subgraph within hundreds of microseconds per update on large graphs. Moreover, as the first to implement the fully-dynamic maintenance algorithm for densest subgraph on hypergraphs, compared to [19] (which supports only random deletions), our maintained solution has a higher density, and is more stable.

1.4 Other Related Work

When there is a size constraint on the subgraph, i.e., computing a densest subgraph on k nodes, the problem is known as the Densest-k-Subgraph Problem. In contrast to the Densest Subgraph Problem, the Densest-k-Subgraph Problem turns out to be much harder. It was proved in [25] that the problem does not admit any PTAS unless NP$\subseteq \cap_{\epsilon>0}$ BPTIME(2^{n^ϵ}). The current best approximation ratio for the problem is $O(n^{\frac{1}{4}})$ by Bhaskara et al. [9] while there is evidence showing that approximating the problem within ratio $n^{o(1)}$ might be harder than Unique Games or Small Set Expansion [10]. Following [9], the Densest-k-Subgraph Problem was extended to hypergraphs very recently by [17], in which an $O(n^{0.697831+\epsilon})$-approximation algorithm was presented. Independent from our work, they also considered the static Densest Subgraph Problem in hypergraphs and proposed similar algorithms. Other generalizations of the problem that aim at finding at most k subgraphs have also been considered [8, 39].

2 STATIC ALGORITHMS

We consider in this section the Densest Subgraph Problem in weighted hypergraphs $H(V, E)$, i.e., each $u \in V$ (resp. $e \in E$) has a non-negative integer weight w_u (resp. w_e).

Notations. We denote $n = |V|$, $m = |E|$, and $r = \max_{e\in E} |e|$. Let $M = \sum_{e\in E} |e|$. We can assume that there are no isolated nodes, and hence $n \le M \le rm$. Note that $O(M)$ is the input size. For all $u \in V$, we let $E_u = \{e \in E : u \in e\}$ be the set of adjacent edges and $E_u[S] = \{e \in E : u \in e \subseteq S\}$ be the set of edges adjacent to u that are induced by vertex subset $S \subseteq V$. For all $F \subseteq E$ (resp. $S \subseteq V$), we denote by $w(F) = \sum_{e\in F} w_e$ (resp. $w(S) = \sum_{u\in S} w_u$) the total weight of F (resp. S). From now on, we use S^* to denote a set of nodes that induces the maximum density: $\rho(S^*) = \max_{S\subseteq V} \rho(S)$. Observe that $\rho(S^*) \ge \frac{w(E)}{w(V)}$. For an integer $k \ge 1$, we use $[k]$ to denote $\{1, 2, \ldots, k\}$.

2.1 Max-Flow-Based Exact Algorithm

We use binary search to find the maximum density. For each candidate value $\frac{w(E)}{w(V)} \le \beta \le w(E)$, we define a flow network G_β, which is a directed graph whose edges have capacities, and has source s and sink t. The set of nodes in G_β is $\{s, t\} \cup V \cup E$. The set of edges in G_β contains the following (as shown in Figure 1):

For all $u \in V$, there is an edge from s to u with capacity $c(s, u) = \delta_u = \sum_{e\in E_u} \frac{w_e}{|e|}$ and an edge from u to t of capacity $c(u, t) = \beta w_u$.

For all $e \in E$ and $u \in e$, there is an edge from u to e with capacity $c(u, e) = \frac{w_e}{|e|}$ and an edge from e to u of capacity $c(e, u) = \infty$. Note

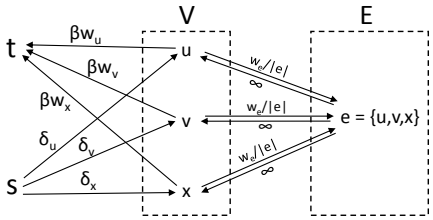


Figure 1: Auxiliary Graph G_β

that G_β has $m + n + 2$ nodes, $O(M)$ edges and can be constructed in $O(M)$ time.

Lemma 2.1. *The maximum flow from s to t in G_β is less than $w(E)$ if and only if $\rho(S^*) > \beta$.*

Proof. Note that we always have max-flow$(s, t) \le w(E)$ since there is an st-cut $(\{s\}, \{t\} \cup V \cup E)$ of capacity $\sum_{u\in V} \delta_u = w(E)$. Now suppose we compute the max-flow from s to t in G_β and find a minimum st-cut as $(\{s\} \cup V_1 \cup E_1, \{t\} \cup V_2 \cup E_2)$, where $V_2 = V \backslash V_1, E_2 = E \backslash E_1$, then we have (where cut(A, B) is the total capacities of edges from A to B):

max-flow$(s, t; G_\beta)$ = cut$(\{s\} \cup V_1 \cup E_1, \{t\} \cup V_2 \cup E_2)$
$= \sum_{u\in V_2} \delta_u + \sum_{u\in V_1} \beta w_u$ + cut(V_1, E_2) + cut(E_1, V_2).

First, observe that cut$(E_1, V_2) = 0$, since otherwise cut$(E_1, V_2) = \infty$; this implies that any edge e intersecting V_2 cannot be in E_1. On the other hand, since $(\{s\} \cup V_1 \cup E_1, \{t\} \cup V_2 \cup E_2)$ is a minimum st-cut, if there is an edge $e \subseteq V_1$ such that $e \in E_2$, then we can strictly reduce the cut by moving e from E_2 to E_1. Hence, we have shown that $E_1 = E[V_1]$ and $E_2 = E \backslash E[V_1]$ and have the following:

$$\begin{aligned}
&\text{max-flow}(s, t; G_\beta)\\
=&\sum_{u\in V} \delta_u - \sum_{u\in V_1} \delta_u + \beta w(V_1) + \text{cut}(V_1, E\backslash E[V_1])\\
=&w(E) - (\text{cut}(V_1, E) - \beta w(V_1) - \text{cut}(V_1, E\backslash E[V_1]))\\
=&w(E) - (\text{cut}(V_1, E[V_1]) - \beta w(V_1))\\
=&w(E) - w(V_1)(\rho(V_1) - \beta).
\end{aligned}$$

Hence, if the max-flow is less than $w(E)$, then $\rho(S^*) \ge \rho(V_1) > \beta$.

Conversely, if there is some $V_1 \subseteq V$ such that $\rho(V_1) > \beta$, then by taking $V_2 := V \setminus V_1$, $E_1 := E[V_1]$ and $E_2 := E \setminus E_1$, the cut $(\{s\} \cup V_1 \cup E_1, \{t\} \cup V_2 \cup E_2)$ has capacity $w(E) - w(V_1) \cdot (\rho(V_1) - \beta) < w(E)$, which implies that the max-flow is strictly less than $w(E)$. □

Combing the above argument with a standard binary search on β, Algorithm 1 solves the Densest Subgraph Problem in weighted hypergraphs using $O(\log W)$ max-flow computations, where $W := w(V) + w(E)$.

Algorithm 1 Weighted-densest-subgraph($H(V, E)$):

1: lower := $\frac{w(E)}{w(V)}$, upper := $w(E)$, $S^* := V$.
2: **while** upper − lower $\geq \frac{1}{(w(V))^2}$ **do**
3: $\beta := \frac{\text{upper+lower}}{2}$.
4: **if** max-flow$(s, t; G_\beta)$ =cut$(S_\beta, T_\beta) < w(E)$ **then**
5: lower := β, $S^* := S_\beta \cap V$. ▷ S^* keeps a candidate solution: $\rho(S^*) > \beta$
6: **else**
7: upper := β. ▷ $\forall S \subseteq V, \rho(S) \leq \beta$
8: **return** S^*.

For any two subsets of nodes S_1 and S_2, if $\rho(S_1) \neq \rho(S_2)$, then we have $|\rho(S_1) - \rho(S_2)| \geq \frac{1}{w(S_1)\cdot w(S_2)} \geq \frac{1}{(w(V))^2}$. Hence, the above binary search terminates in $\log((w(V))^2 \cdot (w(E) - \frac{w(E)}{w(V)})) = O(\log W)$ iterations. In each iteration with a fixed β, by Lemma 2.1, we can either find a subgraph with density strictly larger than β, or make sure that no subgraph has density larger than β by computing max-flow$(s, t; G_\beta)$ in G_β, which contains $O(rm)$ edges. Hence, we can solve the Densest Subgraph Problem in weighted hypergraphs using $O(\log W)$ computations of max-flow on a flow network with $O(rm)$ edges. We have shown the part of Theorem 1.1 concerning the max-flow-based exact algorithm.

2.2 LP-Based Exact Algorithm

In this section, we use an LP similar to [15] to give an exact algorithm to find a densest subgraph in a weighted hypergraph. We introduce a variable $y_u \in [0, 1]$ for each $u \in V$ and variable $x_e \in [0, 1]$ for each $e \in E$ in the following LP.

$$\begin{aligned} \max \quad & \textstyle\sum_{e\in E} w_e x_e \\ \text{s.t.} \quad & x_e \leq y_u, \qquad \forall u \in e \\ & \textstyle\sum_{u\in V} w_u y_u = 1, \\ & x_e, y_u \geq 0, \qquad \forall e \in E, u \in V. \end{aligned}$$

The interpretation is that for some candidate subset S, we can define a feasible solution $z^S = (y^S, x^S)$ such that $y^S_u = \frac{1}{w(S)}$ if $u \in S$ and $y^S_u = 0$ otherwise. Moreover, $x^S_e = \frac{1}{w(S)}$ if $e \in E[S]$ and 0 otherwise. Note that z^S is a feasible solution for the above LP, and the density $\rho(S) = \sum_{e\in E} w_e x^S_e$. By adopting a similar proof strategy as in [8, Lemma 4.1], we obtain the following lemma.

Lemma 2.2. *Given any optimal solution $z^* = (y^*, x^*)$ for the above LP, $P = \{u \in V : y^*_u > 0\}$ induces a graph with maximum density.*

Proof. First notice that given variables y_u, the objective is maximized when $x_e = \min_{u\in e} y_u$ for all $e \in E$ since $w_e \geq 0$. As noted above, for any $S \subseteq V$, we can derive a feasible solution $z^S = (y^S, x^S)$, whose objective value is $\rho(S)$. Let LP* = LP(z^*) be the optimal value of the LP, then for all $S \subseteq V$ we have

$$\text{LP}^* \geq \text{LP}(z^S) = \sum_{e\in E[S]} w_e \frac{1}{w(S)} = \rho(S). \tag{1}$$

Let $P \subseteq V$ be the nodes v such that $y^*_v > 0$. Let $a = w(P)$ and $b = \min_{u\in P} y^*_u$. Note that $ab \leq \sum_{u\in P} w_u y^*_u = 1$. Then we have $z^* = abz^P + (1 - ab)\widehat{z}$, where

$$\widehat{z} = (\widehat{x}, \widehat{y}), \quad \widehat{y}_u = \max\{0, \frac{y^*_u - b}{1 - ab}\}, \quad \widehat{x}_e = \max\{0, \frac{x^*_e - b}{1 - ab}\}.$$

Note that $\widehat{z}$ is feasible since $\widehat{x}_e = \min_{u\in e} \widehat{y}_u$ and $\sum_{u\in V} w_u \widehat{y}_u = \frac{\sum_{u\in P} w_u y^*_u - ab}{1-ab} = 1$.

Because the objective value is linear and the optimal solution is a convex combination of feasible solutions z^S and $\widehat{z}$, it follows that LP* = LP(z^*) = LP$(\widehat{z})$ = LP(z^P) = $\rho(P)$, which combined with (1) implies that $\rho(P) \geq \max_{S\subseteq V} \rho(S)$. □

2.3 Near Linear-Time r-Approximation

Consider any node $u \in S^*$. Since $\rho(S^*) = \max_{S\subseteq V} \rho(S)$, we have $\frac{w(E_u[S^*])}{w_u} \geq \rho(S^*)$, as otherwise $\rho(S^*\backslash\{u\}) = \frac{w(E[S^*]) - w(E_u[S^*])}{w(S^*) - w_u} > \rho(S^*)$ is a contradiction. Hence, similar to the idea of removing the node with minimum degree in an unweighted normal graph [15], we iteratively remove the node with minimum value of $\frac{w(E_u[S])}{w_u}$, where S is the set of remaining nodes. We show that at the point when a node $u \in S^*$ is removed as the minimum degree node, the current graph must have a density within a factor r of the optimum.

Algorithm 2 Approx-densest-subgraph($H(V, E)$):

1: $S_1 := V$.
2: **for** $i = 1, 2, \ldots, n - 1$ **do**
3: $u_i := \arg\min_{u\in S_i} \frac{w(E_u[S_i])}{w_u}$.
4: $S_{i+1} := S_i\backslash\{u_i\}$.
5: **return** $\arg\max_{i\in[n]} \rho(S_i)$.

Lemma 2.3. *Algorithm 2 returns an r-approximate densest subgraph in $O(M \log n)$ time.*

Proof. Consider the iteration such that $S^* \subseteq S_i$ while $S^* \nsubseteq S_{i+1}$, which means $u_i \in S^*$. Then, by the above argument we have

$$\begin{aligned} \rho(S_i) = \frac{w(E[S_i])}{w(S_i)} &\geq \frac{\sum_{u\in S_i} w_u \frac{w(E_u[S_i])}{w_u}}{rw(S_i)} \geq \frac{\sum_{u\in S_i} w_u \frac{w(E_{u_i}[S_i])}{w_{u_i}}}{rw(S_i)} \\ &= \frac{w(E_{u_i}[S^*])}{rw_{u_i}} \geq \frac{\rho(S^*)}{r}. \end{aligned}$$

Since the algorithm returns S_i with the maximum density over all iterations, the approximation ratio follows. Note that we can organize the set of nodes as a min-heap based on values $\frac{w(E_u[S])}{w_u}$, where S is the set of nodes that remain. We also keep track of the total remaining edge weights and node weights. In iteration i we can compute $\rho(S_i)$ and find a node u_i with minimum $\frac{w(E_u[S_i])}{w_u}$ in $O(1)$ time.

When an edge e is removed (the first time a node in e is removed), the values of at most $|e|$ nodes in the remaining set will be affected. Hence, in total, there will be at most $\sum_{e\in E} |e|$ updates to the min-heap, each of which takes $O(\log n)$ time. Therefore, the total running time of the algorithm is $O(n + \sum_{e\in E} |e| \log n) = O(M \log n)$. □

As we will see from our experimental results (Section 5), the actual approximation ratio of our algorithm on several large real-world graphs is very close to 1.

3 INCREMENTAL ALGORITHM

In this section, under the insertion-only (incremental) setting, we give a dynamic algorithm to maintain an $r(1+\epsilon)$-approximate densest subgraph, with amortized poly$(\frac{r}{\epsilon}\log n)$ time per edge insertion.

As argued in Section 1, our algorithm works for unweighted hypergraphs but can be easily extended to edge-weighted hypergraphs with edge weights upper bounded by poly$(\frac{r}{\epsilon}\log n)$. From now on, we assume that the hypergraph under consideration is unweighted, but might have multiple edges. In the following, $\tilde{O}$ hides the poly$(\frac{r}{\epsilon}\log n)$ factor. Define $\tau := \lceil \log_{1+\epsilon} n \rceil$. In the dynamic case, we use m and M to refer to the number of edges and the total degree in the **current** graph under consideration.

3.1 Static $r(1+\epsilon)$-Approximate Algorithm

We first show how to obtain a partitioning of nodes that is similar to Bahmani et al.'s algorithm [7]. Note that in the unweighted case we have $\frac{m}{n} \le \rho(S^*) \le m$. We call $|E_u[S]|$ the *degree* of node u in the graph induced by S. Note that $\sum_{u\in V}|E_u| = M$. Consider the following algorithm that fixes a threshold $\beta > 0$ and removes nodes of degree less than β.

Algorithm 3 Find$(H(V,E), \beta, \epsilon)$:

```
S_0 := A_0 := V, i := 0.
while S_i ≠ ∅, A_i ≠ ∅ and i < τ = ⌈log_{1+ε} n⌉ do
    A_i := {u ∈ S_i : |E_u[S_i]| < β}.        ▷ nodes of small degree
    S_{i+1} := S_i \ A_i.
    i := i + 1.
return Ŝ := arg max_{i≤τ} ρ(S_i).
```

Let the A_i's be constructed as above and $A_\tau = S_\tau$. Note that $(A_0,\dots,A_\tau)$ defines a partitioning of nodes into $\tau+1 = O(\log_{1+\epsilon} n)$ batches and we have $S_i = A_{\ge i} = \cup_{j=i}^{\tau} A_j$.

Lemma 3.1. *If $\beta > r(1+\epsilon)\rho(\widehat{S})$, then $S_\tau = \emptyset$; if $\beta \le \rho(S^*)$, then $S^* \subseteq S_\tau \ne \emptyset$.*

Proof. If $\rho(\widehat{S}) < \frac{\beta}{r(1+\epsilon)}$, then $\rho(S_i) < \frac{\beta}{r(1+\epsilon)}$ for all $S_i \ne \emptyset$. For all $S_i \ne \emptyset$, we have $\rho(S_i)|S_i| = |E[S_i]| \ge \frac{1}{r}\sum_{u\in S_i}|E_u[S_i]| \ge \frac{\beta}{r}|S_i\backslash A_i| > (1+\epsilon)\rho(S_i)|S_{i+1}|$, which implies $|S_{i+1}| < \frac{|S_i|}{1+\epsilon}$. Hence, we have $|S_\tau| < \frac{n}{(1+\epsilon)^\tau} \le 1$, which means $S_\tau = \emptyset$.

As argued in Section 2.3, for all $u \in S^*$, $|E_u[S^*]| \ge \rho(S^*)$. Hence if $\beta \le \rho(S^*)$, then $|E_u[S_i]| \ge \beta$ for all $i = 0,1,\dots,\tau-1$, which means that no node from S^* will be removed in any iteration. Thus $S^* \subseteq S_\tau \ne \emptyset$. □

Algorithm 4 Approx-densest$(H(V,E), \beta_0, \epsilon)$:

```
Ŝ := V, β := max{m/(rn), β_0}.        ▷ β_0 provides a lower bound
while true do                          ▷ at most O(τ) iterations
    S' := Find(H, β, ε).               ▷ Õ(M) time
    if β ≤ r(1+ε)ρ(S') then
        Ŝ := S', β := (1+ε)β.
    else
        return Ŝ.
```

Lemma 3.2. *Algorithm 4 returns an $r(1+\epsilon)^2$-approximation $\widehat{S}$ of the densest subgraph in $O(M\tau^2) = \tilde{O}(M)$ time.*

Proof. Define $B = \{\frac{m}{rn}(1+\epsilon)^i : i \in [2\tau]\}$. Let $\beta^* \in B$ be the minimum such that $S_\tau = \emptyset$ when Algorithm 3 is run with $\beta = \beta^*$. Note that when run with $\beta = \frac{\beta^*}{1+\epsilon}$ in Algorithm 3, we have $S_\tau \ne \emptyset$. Let $\widehat{S}$ be returned by Algorithm 3 when run with $\beta = \beta^*$. By Lemma 3.1, we have $\rho(\widehat{S}) \ge \frac{\beta^*}{r(1+\epsilon)^2} > \frac{\rho(S^*)}{r(1+\epsilon)^2}$, which implies a $r(1+\epsilon)^2$-approximation.

Since Algorithm 3 can be easily implemented in $O(M\tau)$ time and Algorithm 4 terminates with $O(\tau)$ calls of Algorithm 3, we immediately have the lemma. □

Note that in Algorithm 4, in the last call of Find(H,β,ϵ), a partitioning of nodes into $(A_0,A_1,\dots,A_\tau)$ is constructed such that $A_\tau = S_\tau = \emptyset$ (by Lemma 3.1).

3.2 Edge Insertion-Only Setting

We show in this section how to maintain an $r(1+\epsilon)$-approximate densest subgraph under edge insertion-only setting. Our algorithm maintains a partition $(A_0,A_1,\dots,A_\tau)$ such that $A_\tau = \emptyset$ and A_i contains the set of nodes (of degree less than β) that are removed in the i-th iteration of Algorithm 3. Given such a partition, for all $u \in V$, let $l(u)$ be the *level* of u: $u \in A_{l(u)}$ and $b(u) = |E_u[S_{l(u)}]| < \beta$ be the degree of u when it is removed. Let $l(e) = \min_{u\in e} l(u)$ for all $e \in E$. Note that $l(u)$ (resp. $l(e)$) is the time when $u \in V$ (resp. $e \in E$) is removed.

The intuition behind the update algorithm is simple: under edge insertions, the degrees of nodes could only increase and to maintain the partition, we increase the level of node u if $b(u) = |E_u[S_{l(u)}]| \ge \beta$ after edge insertions. To guarantee the approximation ratio, we rebuild the partition if $A_\tau \ne \emptyset$. We show that we do not need to rebuild the partition frequently by showing that every time it is rebuilt, β^* is increased by a $(1+\epsilon)$ factor.

Let $N(u) = \cup_{e\in E_u} e\backslash\{u\}$ be the neighbors of u.

Algorithm 5 Insertion-only-approx-densest$(H(V,E), \epsilon)$:

```
Ŝ := Approx-densest(H, 0, ε),
let A_i, S_i and β be as in the last call of Find().      ▷ S_τ = ∅
for each newly inserted edge e do
    E := E ∪ {e} and update b(u) for all u ∈ e.          ▷ O(|e|) time
    label all nodes in e "bad".
    while exists a bad node do
        pick a bad node u, label u "good" and let l'(u) := l(u).
        while b(u) ≥ β and l'(u) < τ do
            l'(u) := l'(u) + 1,
            b(u) := |{e ∈ E_u : min_{v∈e\{u}} l(v) ≥ l'(u)}|.
        if l'(u) > l(u) then
            for each v ∈ N(u) s.t. l(u) < l(v) ≤ l'(u) do
                update b(v), label v "bad".
            l(u) := l'(u).
        if l(u) = τ then
            Rebuild: Ŝ := approx-densest(H, β, ε),
            update A_i, S_i, β, l() and b().
            label all nodes "good".
```

Proof of **Theorem 1.2**: **Approximation Ratio.** As shown in Algorithm 5, we always maintain $\beta = \beta^*$ to be the minimum value

in B such that produces a partition $(A_0, A_1, \ldots, A_\tau)$ with $A_\tau = \emptyset$. Since β^* is non-decreasing under edge insertions, by Lemma 3.1, as long as $A_\tau = \emptyset$, the subset $\widehat{S}$ we maintain induces an $r(1+\epsilon)$-approximation.

Update Time. At any moment, consider the total update time we have spent so far. First we upper bound the total running time due to the rebuild procedure. Since we only rebuild when $A_\tau \neq \emptyset$, which implies $\beta < \beta^*$, after each rebuild, β must be increased by a factor of $(1+\epsilon)$. Since $\beta < m$, the number of rebuilds we have performed so far is at most $O(\log_{1+\epsilon} m) = O(\frac{r}{\epsilon}\log n)$. Hence, the update time due to the rebuild procedures is $\tilde{O}(m)$, by Lemma 3.2. It is easy to check that the update time is $O(|e|)$ when an edge is inserted and $O(|N(u)|)$ when $l(u)$ is increased by one. Since $l(u) < \tau = O(\log_{1+\epsilon} n)$ and $\sum_{u\in V}|N(u)| \leq r^2 m$, the total update time excluding the rebuild procedure is $O(M + \tau r^2 m) = \tilde{O}(m)$. Hence, the overall update time is upper bounded by $\tilde{O}(m)$ and the update time charged to each update is $\text{poly}(\frac{r}{\epsilon}\log n)$.

Space Complexity. It is easy to check that in addition to the input hypergraph, it suffices to maintain $l(u)$ and $b(u)$ for each node u (which induces A_i and S_i) together with constant number of variables. Hence, the extra space needed is $O(n)$. □

Remark. We remark that our algorithm can also be extended to support random deletions as in [19]. The algorithm is the same as they used: in addition to maintaining the sets S_i (and rebuilding when $S_\tau \neq \emptyset$), we further maintain $\rho(\widehat{S}) \geq \frac{\beta}{(1+\epsilon)^2}$ (and rebuild it if this is not true), where $\widehat{S}$ is our candidate solution that has the maximum density among S_i. To bound the amortized update time, as argued in [19], we consider two cases depending on the number of deletions R between two consecutive rebuilds:

- if $R \geq \frac{m}{\text{poly}(\frac{r}{\epsilon}\log n)}$, then we charge the total update time $\tilde{O}(m)$ to the deletions, yielding an amortized $\text{poly}(\frac{r}{\epsilon}\log n)$ update time;
- otherwise we can show that the density of $\widehat{S}$ is not decreased a lot (since the edges to be deleted are chosen uniformly at random), i.e., after R deletions, $\rho'(\widehat{S}) > \frac{\rho(\widehat{S})}{1+\epsilon}$, which guarantees that $\widehat{S}$ is still an $r(1+\epsilon)^4$-approximation.

Since our argument follows exactly as shown in [19], we omit the details in this paper and interested readers can refer to their original analysis.

4 FULLY DYNAMIC APPROXIMATION

We show in this section how to further extend our algorithms in Section 3 to support arbitrary edge deletions. The dynamic algorithm we use in this section also maintains a partitioning of nodes into $\tau+1$ batches based on their degrees. However, unlike the insertion-only case, under arbitrary edge insertions and deletions, $l(u)$ can either increase or decrease and it becomes difficult to maintain the partitions exactly. To introduce "flexibility" on maintaining partitions, we apply the idea of "lazy update", as used by [11] to the partitions: for a fixed threshold β, we remove nodes with degree less than β while keeping nodes with degree at least $\alpha\beta$, for some $\alpha > 1$. Since nodes with degree in $[\beta, \alpha\beta)$ are not guaranteed to be removed or kept, we obtain a flexibility of partitions. However, as a consequence of the flexibility, the approximation ratio is enlarged by a factor of α.

Definition 4.1 ((α,β)-decomposition). An (α,β)-decomposition (for some $\alpha \geq 1$) of $H(V,E)$ is a sequence of subsets of V such that $S_\tau \subseteq S_{\tau-1} \subseteq \ldots \subseteq S_1 \subseteq S_0 = V$ and for all $i \in [\tau]$,

(1) $\{u \in S_{i-1} : |E_u[S_{i-1}]| \geq \alpha\beta\} \subseteq S_i$,
(2) $\{u \in S_{i-1} : |E_u[S_{i-1}]| < \beta\} \cap S_i = \emptyset$.

Let $A_i = S_i \backslash S_{i+1}$ for all $0 \leq i \leq \tau-1$ and $A_\tau = S_\tau$. Let $\widehat{S} = \arg\max_{i\leq\tau}\rho(S_i)$. The following lemma is a generalization of Lemma 3.1, in which $\alpha = 1$. The first statements comes directly from the proof of Lemma 3.1 since nodes with degree less than β are removed and the second statement follows easily since for all $u \in S^*$, u would always have degree at least $\alpha\beta$ and never be removed.

Lemma 4.2. *If $\beta > r(1+\epsilon)\rho(\widehat{S})$, then $S_\tau = \emptyset$; if $\beta \leq \frac{\rho(S^*)}{\alpha}$, then $S^* \subseteq S_\tau \neq \emptyset$.*

As before, let $\beta^* \in B = \{\frac{m}{\alpha r n}(1+\epsilon)^t : t \in [2\tau]\}$ be the minimum such that $S_\tau = \emptyset$ in an (α,β^*)-decomposition. By Lemma 4.2, in the (α,β^*)-decomposition we have $\rho(\widehat{S}) \geq \frac{\beta^*}{r(1+\epsilon)^2} > \frac{\rho(S^*)}{\alpha r(1+\epsilon)^2}$.

Since $|B| = O(\log_{1+\epsilon} n)$, we maintain for every $\beta \in B$ an (α,β)-decomposition. Suppose the (α,β)-decomposition (and the densities $\rho(S_0), \rho(S_1), \ldots, \rho(S_\tau)$) are maintained for every $\beta \in B$, then in $O(\tau^2)$ time we can find β^* together with $\widehat{S}$ in the (α,β^*)-decomposition, which gives an $\alpha r(1+\epsilon)$-approximation. Hence, to prove Theorem 1.3, it suffices to show how to maintain each (α,β)-decomposition using $\tilde{O}(M)$ extra space in $\text{poly}(\frac{r}{\epsilon}\log n)$ time, for $\alpha = r(1+3\epsilon)$.

4.1 Maintaining an (α,β)-Decomposition

Fix $\beta \in B$, define (as before) $l(u)$ and $l(e)$ as the levels of nodes and edges in the partitioning $(A_0, A_1, \ldots, A_\tau)$ defined by the (α,β)-decomposition. For all $i \leq l(u)$, let $E_u^{(i)} = E_u[S_i] - E_u[S_{i+1}]$ be the hyperedges adjacent to u that are removed at level i. Note that $(E_u^{(0)}, E_u^{(1)}, \ldots, E_u^{(l(u))})$ defines a partition of E_u, based on the $l(e)$. For all $i \leq l(u)$, let $b_i(u) = |E_u[S_i]|$. Note that $b_i(u)$ is non-increasing when i increases and we have $b_{l(u)}(u) = |E_u[S_{l(u)}]| < \alpha\beta$ for all $u \notin S_\tau$ and $b_{l(u)-1}(u) \geq \beta$ for all $u \notin A_0$.

We maintain for each $u \in V$ its level $l(u)$, the partitioning $(E_u^{(0)}, \ldots, E_u^{(l(u))})$ of E_u and the degree of u at each level: $b_0(u), \ldots, b_{l(u)}(u)$. We further maintain $l(e)$ for every $e \in E$ and $\rho(S_i)$ for each $i = 0, 1, \ldots, \tau$. Since $l(e)$, $b_i(u)$, S_i and $\rho(S_i)$ can be updated in constant time whenever constant number of elements in $E_u^{(j)}$ (for some $j \leq i$) are changed or $l(u)$ is changed by 1, from now on we only discuss how $l(u)$ and $E_u^{(j)}$ are maintained while assuming that other data structures are maintained automatically. Note that for each node u, the data structure we maintain takes $O(|E_u| + l(u))$ space, by keeping only identities of the adjacent hyperedges. Hence in total, it takes $\sum_{u\in V} O(|E_u| + l(u)) + O(m) + O(\tau) = \tilde{O}(M)$-space to maintain one (α,β)-decomposition.

For each update, Algorithm 6 maintains the pre-described data structures. Algorithm 6 updates the partitioning of each E_u and guarantees $b_{l(u)}(u) < \alpha\beta$ (otherwise increase $l(u)$) and $b_{l(u)-1}(u) > \beta$ (otherwise decrease $l(u)$). Note that the algorithm executes in $O(|e|)$ time if neither Promote(u) nor Demote(u) is triggered.

In the Promote(u) sub-routine (Algorithm 7), $l(u)$ is increased by one. Assume $l(u) = t$, to maintain the partitioning of E_u, the partition $E_u^{(t)}$ if split into two. Since $l(e)$ is possibly also increased

Algorithm 6 Maintain-decomposition($H(V,E)$):

if insert(e) **then** ▷ initialize $l(e) := \min_{u\in e} l(u)$
  for each $u \in e$, $E_u^{(l(e))} := E_u^{(l(e))} \cup \{e\}$.
else if delete(e) **then**
  for each $u \in e$, $E_u^{(l(e))} := E_u^{(l(e))} \backslash \{e\}$.
for each $u \in e$ s.t. $l(u) = l(e)$, label u "bad".
while exists a bad node u **do**
  if $l(u) < \tau$ and $b_{l(u)}(u) \geq \alpha\beta$ **then**
    Promote(u).
  else if $l(u) > 0$ and $b_{l(u)-1}(u) < \beta$ **then**
    Demote(u).
  else
    label u "good".

Algorithm 7 Promote(u):

$t := l(u)$, $l(u) := t+1$, $E_u^{(t+1)} := \emptyset$. ▷ $|E_u^{(t)}| \geq \alpha\beta$
for each $e \in E_u^{(t)}$ **do** ▷ $O(|E_u^{(t)}|)$-iterations
  if $\min_{v\in e\backslash\{u\}}\{l(v)\} \geq t+1$ **then** ▷ $O(|e|)$-time
    for each $v \in e$ **do**
      $E_v^{(t)} := E_v^{(t)}\backslash\{e\}$, $E_v^{(t+1)} := E_v^{(t+1)} \cup \{e\}$.
      if $l(v) = t+1$ and $v \neq u$ **then**
        label v "bad".

Algorithm 8 Demote(u):

$t := l(u)$, $l(u) := t-1$. ▷ $|E_u^{(t)}| < \beta$
for each $v \in e \in E_u^{(t)}$ **do** ▷ $O(\sum_{e\in E_u^{(t)}} |e|)$-time
  $E_v^{(t)} := E_v^{(t)}\backslash\{e\}$, $E_v^{(t-1)} := E_v^{(t-1)} \cup \{e\}$.
  if $l(v) = t$ **then**
    label v "bad".

for $e \in E_u^{(t)}$, we also need to update the partitioning of E_v, for each $v \in e$. Note that the whole procedure executes in $O(\sum_{e\in E_u^{(t)}} |e|) = O(r|E_u^{(t)}|)$ time. Similarly, Demote(u) (Algorithm 8) decreases $l(u)$ by one and updates the maintained data structure in $O(\sum_{e\in E_u^{(t)}} |e|) = O(r|E_u^{(t)}|)$ time.

As we have argued, as long as an (α,β)-decomposition is maintained for each $\beta \in B$, we are able to maintain an $\alpha r(1+\epsilon)$-approximation of densest subgraph in $O(\tau^2)$ time. Hence, to prove Theorem 1.3, it suffices to show that Algorithm 6 executes in amortized poly$(\frac{r}{\epsilon}\log n)$ time for $\alpha = r(1+3\epsilon)$.

Potential Function Analysis. To bound the amortized update time, we define the potential function depending on $l(u)$ and $b_i(u)$ as $\mathsf{P} := \sum_{u\in V} P(u) + \sum_{e\in E} P(e) \geq 0$, where $P(u) := \sum_{i=1}^{l(u)-1} \max\{0, \alpha\beta - \epsilon b_i(u)\}$ and $P(e) := r(\tau - l(e) + \frac{|\{u\in e: l(u)=l(e)\}|}{|e|})$. The potential function increases when edges are inserted or deleted; decreases when the data structures we maintain are updated. The following lemma implies Theorem 1.3.

Lemma 4.3. *For each computation cost in the update procedure (Algorithm 6), the potential decreases by at least $\Omega(\frac{\epsilon}{r})$ while each edge update increases the potential by at most $O(r\tau)$.*

Proof. For notational convenience, in the following, for any set and variable, we use $'$ to denote the new one after Promote(u)/Demote(u). We first upper bound the increase in potential after each update:

- **Insert**(e): $\mathsf{P}' - \mathsf{P} \leq P'(e) \leq r\tau$.
- **Delete**(e): $\mathsf{P}' - \mathsf{P} \leq \sum_{u\in e}(P'(u) - P(u)) \leq \epsilon|e|\tau \leq \epsilon r\tau$.

We next lower bound the decrease in potential after each promotion/demotion.

Promote(u): assume $l(u) = t$, then $b_t(u) \geq \alpha\beta$, $l'(u) = t+1$ and $S'_{t+1} = S_{t+1} \cup \{u\}$. The potential of nodes and edges are changed as follows.

- Since $S'_i = S_i$ for all $i \leq t$, we have
$$P(u) - P'(u) = -\max\{0, \alpha\beta - \epsilon b_t(u)\} \geq \epsilon b_t(u) - \alpha\beta.$$
- For all $v \in e \in E_u[S_t]$ s.t. $l(v) \geq t+2$,
$$P(v) - P'(v) = \max\{0, \alpha\beta - \epsilon b_{t+1}(v)\} - \max\{0, \alpha\beta - \epsilon b'_{t+1}(v)\} \geq 0.$$
- For all other nodes v, $P(v) - P'(v) = 0$.
- For all $e \in E_u[S_t]$ s.t. $\min_{v\in e\backslash\{u\}}\{l(v)\} \geq t+1$,
$$P(e) - P'(e) \geq r(l'(e) - l(e) + \frac{1}{|e|} - 1) = \frac{r}{|e|} \geq 1.$$
- For all $e \in E_u[S_t]$ s.t. $\min_{v\in e\backslash\{u\}}\{l(v)\} = t$,
$$P(e) - P'(e) \geq \frac{r}{|e|} \geq 1.$$
- For all other edges e, $P(e) - P'(e) = 0$.

Hence, overall the total potential is decreased by at least $\mathsf{P} - \mathsf{P}' \geq \epsilon b_t(u) - \alpha\beta + |E_u[S_t]| \geq \epsilon|E_u[S_t]|$. Since each promotion executes in $O(r|E_u^{(t)}|) = O(r|E_u[S_t]|)$ time, for each computation cost, the potential is decreased by $\Omega(\frac{\epsilon}{r})$.

Demote(u): assume $l(u) = t$, then $b_{t-1}(u) < \beta$, $l'(u) = t-1$ and $S'_t = S_t\backslash\{u\}$. The potential of nodes and edges are changed as follows.

- Since $S'_i = S_i$ for all $i \leq t$, we have $P(u) - P'(u) = \max\{0, \alpha\beta - \epsilon b_{t-1}(u)\} = \alpha\beta - \epsilon b_{t-1}(u)$.
- For all $v \in e \in E_u[S_t]$ s.t. $l(v) \geq t+1$, $P(v) - P'(v) = \max\{0, \alpha\beta - \epsilon b_t(v)\} - \max\{0, \alpha\beta - \epsilon b'_t(v)\} \geq -\epsilon(b_t(v) - b'_t(v))$, which means that the increase in potential of each such node v is at most ϵ fraction of the number of hyperedges adjacent to v at level t that are removed due to the demotion of u. Hence, the total decrease of potential of those nodes is $\sum_{v\in e\in E_u[S_t] \text{ s.t. } l(v)\geq t+1} P(v) - P'(v) \geq -\epsilon\sum_{e\in E_u[S_t]}|e| \geq -\epsilon r|E_u[S_t]|$.
- For all other nodes v, $P(v) - P'(v) = 0$.
- For all $e \in E_u[S_t]$, $P(e) - P'(e) \geq r(l'(e) - l(e) + \frac{1}{|e|} - \frac{1}{|e|}) = -r$.
- For all $e \in E_u^{(t-1)}$, $P(e) - P'(e) \geq -\frac{r}{|e|} \geq -r$.
- For all other edges e, $P(e) - P'(e) = 0$.

Hence, the total potential decrease by (when $\alpha = r(1+3\epsilon)$)
$$\begin{aligned}\mathsf{P} - \mathsf{P}' &\geq \alpha\beta - \epsilon b_{t-1}(u) - \epsilon r b_t(u) - r|E_u(S_t)| - r|E_u^{(t-1)}| \\ &\geq \alpha\beta - (\epsilon + \epsilon r + r)b_{t-1}(u) \geq \epsilon|E_u[S_{t-1}]|.\end{aligned}$$

Since each demotion executes in $O(r|E_u^{(t)}|) = O(r|E_u[S_{t-1}]|)$ time, for each computation cost, the potential is decreased by $\Omega(\frac{\epsilon}{r})$, which completes the analysis. □

5 EXPERIMENTS

We conduct an extensive evaluation on real-world and synthetic graphs to show the effectiveness and efficiency of our algorithms. Recall that for hypergraphs with bounded edge weight, e.g., $w_e = O(1)$, we can modify our unweighted algorithm to support finding and maintaining densest subgraph on weighted scenarios by inserting and deleting multiple edges simultaneously.

We first evaluate the effectiveness and efficiency of the solutions returned by our exact and approximate algorithms. As it is time-consuming to compute the exact densest subgraph, the comparison is run on some small size graphs. We also study how different values of ϵ affect the density of the solution, and the running time of the algorithm. Then we consider dynamic maintenance of the approximate densest subgraphs on large graphs, for which recomputing the approximate solutions for each update is time-consuming.

5.1 Datasets & Experimental setup

Real world Datasets. The datasets are publicly available. We summarize their features in table1.

- **DBLP.** The DBLP dataset is obtained from [2]. Nodes in the graph represent authors while hyperedges represent publications. We use a sliding window spanning five years, i.e., publications are removed five years after their insertions. To avoid trivial solutions[1], we regard publications with the exact same group of authors as a single hyperedge.
- **CiteULike.** The Tag-publication network dataset is obtained from [1]. We construct a hypergraph in which nodes represent publications and hyperedges represent tags. We insert a hyperedge $e \subseteq V$ if in the past three days, the publications in e are all labeled with the same tag. We remove a hyperedge if it is inserted three days ago.
- **YouTube.** This is a social network in which nodes represent YouTube users [3]. We insert a hyperedge for a group of users if they all make new friends to the same user in the last three days. We remove a hyperedge if it is inserted three days ago.

Datasets	$\|V\|$	$\|E\|$	Time
DBLP	1,159,694	1,778,467	1959-2016
CiteULike	1,038,323	2,411,819	2005-2008
YouTube	3,223,589	9,375,374	2004

Table 1: Properties of the dynamic datasets analyzed, where Time indicates the period when those edges were inserted.

Synthetic Datasets. As introduced in [24], we generate synthetic non-uniform evolving hypergraphs with attractiveness. The hypergraph is constructed while nodes are inserted. When a constant number of nodes are inserted, a random hyperedge is formed by picking a constant number of existing nodes (and all the newly-inserted nodes), where the probability that an existing node is chosen is proportional to some power of its current degree.

Observe that the maximum density of the above hypergraph is always at most 1, which means that trivially the whole graph is a close-optimal solution. To fix this (and make it denser), at each round, we introduce new nodes with probability p, and with probability $1 - p$ we add a random hyperedge on existing nodes.

Experimental Setup. The experiments were performed on a single machine, with Intel(R) Core(TM) i7-6700 CPU at 3.40GHz, 8192 KB cache size and 64 GB of main memory, running on Ubuntu 14.04 LTS. We run our max-flow based exact-solution algorithm based on the sub-routine MAXFLOW [13, 40]. All our experiments were implemented using C++ compiled with g++ and -O4 optimization. Each run employs a single core of machine while using at most 20% of the main memory.

5.2 Exact vs Approximation

DBLP Dataset. We include papers from conferences belonging to the following three research areas to conduct our experiments:

- **TCS**: STOC, FOCS, SODA, ICALP, ESA, STACS
- **ML**: ICML, NIPS, IJCAI, AAAI
- **DB**: SIGMOD, VLDB, ICDE, CIKM

The properties of the resulting data are summarized in Table 2.

Catagory	# Author	# Paper	Avg. Authors	Max. Authors
TCS	9074	11991	2.56	15
ML	25526	20606	2.78	25
DB	18863	13420	3.27	36

Table 2: Properties of publications, where Avg. Authors denotes the average number of authors per paper and Max. Author denotes the maximum number of authors in a paper.

The results are shown in Table 3. As it can be observed, the approximate algorithm runs several times faster than the exact algorithm, and returns solutions with a close-optimal density (much better than the theoretical guarantee). Moreover, sometimes the approximate solution has a smaller cardinality than the exact solution.

Method	Measure	TCS	ML	DB
Exact	$\|S\|/\|V\|(\%)$	2.56	0.17	0.38
	Density	3.96	2.95	2.66
	Time(ms)	196.12	314.59	198.90
$\epsilon = 0.1$	$\|S\|/\|V\|(\%)$	7.76	0.10	0.25
	Density	3.64	2.16	1.60
	Time(ms)	53.57	123.96	82.24
$\epsilon = 0.5$	$\|S\|/\|V\|(\%)$	7.76	0.10	0.25
	Density	3.64	2.16	1.60
	Time(ms)	54.91	121.08	83.05

Table 3: Performance on real datasets

Method	Measure	TCS	ML	DB
Ours	$\|S\|$	232	43	71
	$\|E[S]\|$	919	127	189
Existing work [19]	$\|S\|$	288	25	48
	$\|E[S]\|$	983	4	2

Table 4: Comparison of hyperedge density

Comparison of Effectiveness. In existing work [19][2], to solve the problem on finding a subset of authors with maximum collaboration density, the underlining graph is a normal graph in which

[1] We observe that in the dataset, Sudhakar M. Reddy and Irith Pomeranz co-authored 173 papers, which trivially induced a densest subgraph with density 86.5

[2] We use their code from https://github.com/aepasto/densest-subgraph

edges represent co-authorship. We compare in Table 4 our solution with theirs, and show that our solution induces a much larger number of publications per author. As we can see from Table 4, in some cases, their solutions (which have the highest densities in the co-authorship network) actually induce very few number of publications. The reason is, in their model, publications that are not induced by the returned solution have contributes to the density.

Synthetic Datasets. To generate evolving hypergraph, we start from an empty graph with ten nodes. We fix an integer parameter c and let $p := \frac{c+1}{4c}$. In each round, the number of selected existing nodes and the number of newly-inserted nodes are chosen uniformly at random from $[c]$, independently. We generate datasets for $c \in \{2, 4\}$ and for $n \in \{1k, 10k\}$. The results are shown in Table 5, where each number in the table is the average of the corresponding data from 10 independent random experiments.

Method	Measure	(1k, 2)	(1k, 4)	(10k, 2)	(10k, 4)
Exact	$\|S\|/\|V\|$(%)	1.25	1.19	0.16	0.13
	Density	12.50	25.93	21.50	74.40
	Time(ms)	15.06	32.93	279.34	543.12
$\epsilon = 0.1$	$\|S\|/\|V\|$(%)	1.50	0.98	0.13	0.13
	Density	9.70	23.51	20.83	74.39
	Time(ms)	5.65	6.79	66.23	66.11
$\epsilon = 0.5$	$\|S\|/\|V\|$(%)	7.56	2.07	0.09	0.11
	Density	6.31	17.36	17.53	73.26
	Time(ms)	4.43	6.21	67.65	66.25

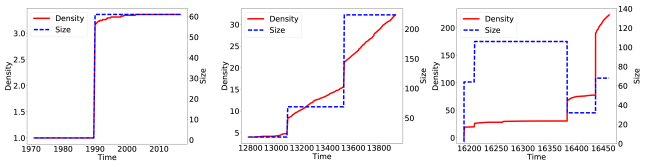
Table 5: Performance on synthetic datasets

As shown in the table, where (10k, 4) means $n = 10,000$ and $c = 4$, the approximate algorithm gives high quality results compared to the optimal solution, and runs much faster than the exact algorithm (especially when the graph becomes larger).

5.3 Incremental Case

Recall from our theoretical analysis, in Algorithm 4, the number r is used as a parameter and hence may affect the quality of the result. While an r-approximation ratio is guaranteed, in practice, a large r, e.g., $r = 20$, would lead to trivial solutions, e.g., a single hyperedge, especially in the dynamic case. We resolve this by passing in smaller values of r, e.g., the average degree of the hypergraph, into Algorithm 4, and evaluate the resulting solutions.

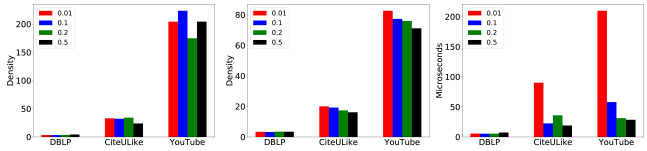
Evolution of the Densest Subgraph. The results are shown in Figure 2, where we fix $\epsilon = 0.1$. As expected (and similar to the normal graph case [19]), the density of the approximate solution increases continuously while its size changes in a stepwise fashion.



(a) DBLP **(b) CiteULike** **(c) YouTube**

Figure 2: Evolution of densest subgraph: insertion only.

Efficiency Accuracy Trade-offs. We evaluate how different values of ϵ affect our approximate solution. In Figure 3, we compare the maximum density (the density of the approximate solution after all insertions), average density (of the approximate solution after each insertion), and the update time, for $\epsilon \in \{0.01, 0.1, 0.2, 0.5\}$.



(a) max. density **(b) avg. density** **(c) avg. update time**

Figure 3: Effect of ϵ in the incremental case.

As expected, in general, the update time increases when ϵ gets small. For $\epsilon = 0.1$, which we choose to run most of our dynamic algorithms, the update time per update is within a hundred microseconds for all datasets, which is much faster than running the static approximation algorithm. Surprisingly, our experimental results show that the improvement of density is very small for smaller value of ϵ, which means that it is unnecessary to fix a very small ϵ.

5.4 Fully Dynamic Case

Recall that in the fully-dynamic algorithm, we maintain an (α, β)-decomposition for each $\beta \in B$, where $|B| = 2\tau$. Hence the space and time complexity of the algorithm is much higher than the incremental algorithm. However, in general we do not need to maintain that many copies as we know that $\rho(S^*) \leq \max_{u \in V} |E_u|$, which is usually small in real-world graphs.

Improved Maintenance on Normal Graphs. We first evaluate our dynamic algorithm on normal graphs under *arbitrary* insertions and deletions. To the best of our knowledge, we are the first to implement a fully dynamic algorithm for maintaining densest subgraphs in evolving graphs. We use the DBLP dataset, which is also used by Epasto et al. [19]. We compare the density curve of our approximate solution with theirs (which we denote by "ELS") in the following table. Recall that our algorithm supports arbitrary edge deletions, while theirs supports only random deletions.

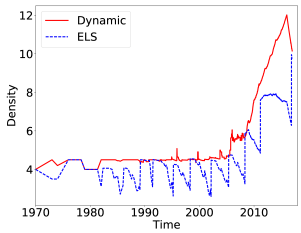
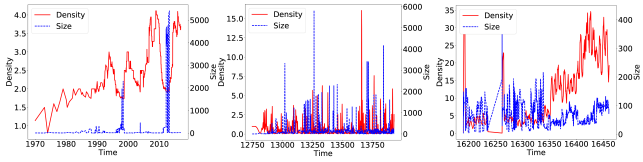


Figure 4: Evolution of the densest subgraph: ours vs ELS

As shown in Figure 4, our approximate solution is almost always better than the solution maintained by ELS[3]. Moreover, compared to ELS, which rebuilds the solution periodically (when "rebuilds" are triggered as the maintained solution becomes unacceptably sparse), our solution is more "stable", i.e., the density of our solution changes gradually over the updates of edges. As we can see from the figure, there is a clear trend of increase in the density, which is consistent with previous observations [19, 28].

[3]Here we plot the density curve with a finer granularity than theirs [19].

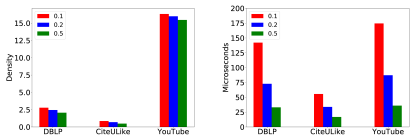
Evolution of the Densest Sub-hypergraph. We then evaluate our algorithm performance on dynamic hypergraphs. We first report the evolution of the densest subgraph in the following figure. As shown in Figure 5, the approximate solutions change continuously while edges are inserted and deleted from the data sets.



(a) DBLP (b) CiteULike (c) YouTube

Figure 5: Evolution of densest subgraph: fully Dynamic.

Efficiency Accuracy Trade-offs. At last, we evaluate the effect of ϵ on the density and update time. As the space complexity is high, we do not perform the experiment with $\epsilon = 0.01$, which has been shown not very helpful, in the previous experimental results.



(a) avg. density (b) avg. update time

Figure 6: Trade-off between the average update time (in microseconds) and the density of the subgraph.

As shown in Figure 6, even when the hypergraphs change dramatically, our dynamic algorithm maintains the approximate solution efficiently (within 200 microseconds). Unsurprisingly, the density and update time decrease while ϵ increases. However, similar to the incremental case, it turns out that compared to the update time, the density of the solution is less sensitive to the change of ϵ.

REFERENCES

[1] 2016. CiteULike tag-publication network dataset – KONECT. (Oct. 2016). http://konect.uni-koblenz.de/networks/citeulike-ti

[2] 2016. DBLP dataset. (2016). http://dblp.uni-trier.de/xml/

[3] 2016. YouTube network dataset – KONECT. (Oct. 2016). http://konect.uni-koblenz.de/networks/youtube-u-growth

[4] Albert Angel, Nick Koudas, Nikos Sarkas, and Divesh Srivastava. 2012. Dense Subgraph Maintenance under Streaming Edge Weight Updates for Real-time Story Identification. *PVLDB* 5, 6 (2012), 574–585.

[5] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.* 23, 2 (2014), 175–199.

[6] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4 (2003), 2.

[7] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest Subgraph in Streaming and MapReduce. *PVLDB* 5, 5 (2012), 454–465.

[8] Oana Denisa Balalau, Francesco Bonchi, T.-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. 2015. Finding Subgraphs with Maximum Total Density and Limited Overlap. In *WSDM*. ACM, 379–388.

[9] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. 2010. Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph. In *STOC*. ACM, 201–210.

[10] Aditya Bhaskara, Moses Charikar, Aravindan Vijayaraghavan, Venkatesan Guruswami, and Yuan Zhou. 2012. Polynomial integrality gaps for strong SDP relaxations of Densest k-subgraph. In *SODA*. SIAM, 388–405.

[11] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. 2015. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *STOC*. ACM, 173–182.

[12] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *KDD*. ACM, 1316–1325.

[13] Yuri Boykov and Vladimir Kolmogorov. 2004. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence* 26, 9 (2004), 1124–1137.

[14] TH Chan, Anand Louis, Zhihao Gavin Tang, and Chenzi Zhang. 2016. Spectral Properties of Hypergraph Laplacian and Approximation Algorithms. *arXiv preprint arXiv:1605.01483* (2016).

[15] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX (Lecture Notes in Computer Science)*, Vol. 1913. Springer, 84–95.

[16] Jie Chen and Yousef Saad. 2012. Dense Subgraph Extraction with Application to Community Detection. *IEEE Trans. Knowl. Data Eng.* 24, 7 (2012), 1216–1230.

[17] Eden Chlamtác, Michael Dinitz, Christian Konrad, Guy Kortsarz, and George Rabanca. 2016. The Densest k-Subhypergraph Problem. In *APPROX-RANDOM (LIPIcs)*, Vol. 60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:19.

[18] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and classification of dense implicit communities in the Web graph. *TWEB* 3, 2 (2009), 7:1–7:36.

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

[20] Hossein Esfandiari, MohammadTaghi Hajiaghayi, and David P. Woodruff. 2016. Brief Announcement: Applications of Uniform Sampling: Densest Subgraph and Beyond. In *SPAA*. ACM, 397–399.

[21] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. In *ISMB (Supplement of Bioinformatics)*. 156–157.

[22] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB*. ACM, 721–732.

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

[24] Jin-Li Guo, Xin-Yun Zhu, Qi Suo, and Jeffrey Forrest. 2016. Non-uniform Evolving Hypergraphs and Weighted Evolving Hypergraphs. *Scientific Reports* 6 (2016).

[25] Subhash Khot. 2006. Ruling Out PTAS for Graph Min-Bisection, Dense k-Subgraph, and Bipartite Clique. *SIAM J. Comput.* 36, 4 (2006), 1025–1071.

[26] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2006. Structure and evolution of online social networks. In *KDD*. ACM, 611–617.

[27] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. 1999. Trawling the Web for Emerging Cyber-Communities. *Computer Networks* 31, 11-16 (1999), 1481–1493.

[28] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007), 2.

[29] Anand Louis. 2015. Hypergraph markov operators, eigenvalues and approximation algorithms. In *STOC*. ACM, 713–722.

[30] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. 2015. Densest Subgraph in Dynamic Graph Streams. In *MFCS (2) (Lecture Notes in Computer Science)*, Vol. 9235. Springer, 472–482.

[31] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. 2016. A fistful of Bitcoins: characterizing payments among men with no names. *Commun. ACM* 59, 4 (2016), 86–93.

[32] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert Endre Tarjan. 2008. Finding Strongly Knit Clusters in Social Networks. *Internet Mathematics* 5, 1 (2008), 155–174.

[33] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos E. Tsourakakis, and Shen Chen Xu. 2015. Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling. In *KDD*. ACM, 815–824.

[34] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense Subgraphs with Restrictions and Applications to Gene Annotation Graphs. In *RECOMB (Lecture Notes in Computer Science)*, Vol. 6044. Springer, 456–472.

[35] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Amitabh Trehan. 2012. Dense Subgraphs on Dynamic Networks. In *DISC (Lecture Notes in Computer Science)*, Vol. 7611. Springer, 151–165.

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

[37] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*. ACM, 1122–1132.

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

[39] Elena Valari, Maria Kontaki, and Apostolos N. Papadopoulos. 2012. Discovery of Top-k Dense Subgraphs in Dynamic Graph Collections. In *SSDBM (Lecture Notes in Computer Science)*, Vol. 7338. Springer, 213–230.

[40] Vladimir Kolmogorov Yuri Boykov. 2015. MAXFLOW - software for computing mincut/maxflow in a graph. (2015). https://github.com/gerddie/maxflow