Efficient Densest Subgraph Computation in Evolving Graphs

Alessandro Epasto*
Brown University, Providence
alessandro_epasto@brown.edu

Silvio Lattanzi
Google, New York
silviol@google.com

Mauro Sozio†
Institut Mines–Télécom,
Télécom ParisTech,
CNRS LTCI, Paris
sozio@telecom-paristech.fr

ABSTRACT

Densest subgraph computation has emerged as an important primitive in a wide range of data analysis tasks such as community and event detection. Social media such as Facebook and Twitter are highly dynamic with new friendship links and tweets being generated incessantly, calling for efficient algorithms that can handle very large and highly dynamic input data. While either scalable or dynamic algorithms for finding densest subgraphs have been proposed, a viable and satisfactory solution for addressing both the dynamic aspect of the input data and its large size is still missing.

We study the densest subgraph problem in the the dynamic graph model, for which we present the first scalable algorithm with provable guarantees. In our model, edges are added adversarially while they are removed uniformly at random from the current graph. We show that at any point in time we are able to maintain a $2(1+\epsilon)$-approximation of a current densest subgraph, while requiring $O(\text{poly}\log(n+r))$ amortized cost per update (with high probability), where r is the total number of update operations executed and n is the maximum number of nodes in the graph. In contrast, a naïve algorithm that recomputes a dense subgraph every time the graph changes requires $\Omega(m)$ work per update, where m is the number of edges in the current graph. Our theoretical analysis is complemented with an extensive experimental evaluation on large real-world graphs showing that (approximate) densest subgraphs can be maintained efficiently within hundred of microseconds per update.

*To appear in the International World Wide Web Conference 2015. Work partially supported by a Google Europe Ph.D. Fellowship in Algorithms, 2011; Google Focused Award; PRIN Ars TechnoMedia and NSF Award IIS-1247581. Work partially done while at Sapienza University of Rome.
†Work partially supported by a Google Faculty Research Award.

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741638.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*; E.1 [**Data Structures**]: Graphs and networks; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

Keywords

Densest Subgraph; Dynamic Graph; Approximation Algorithm

1. INTRODUCTION

Finding dense subgraphs has emerged as an important primitive in a wide range of data analysis tasks such as community detection [17, 19, 26], event detection [7], link spam detection [20], computational biology [34], distance query indexing [4, 18], etc. Densely connected users in a social network might correspond to communities, i.e., sets of users sharing similar interests or being affiliated with a same organization such as a university or a company. Entities such as city, person and company names, starting suddenly to co-occur in tweets might indicate that an interesting event involving the corresponding entities is taking place. Dense subgraphs have also been employed to give a compact representation of node distances in a graph, so as to efficiently compute distances between any two nodes given in input.

In the aforementioned application scenarios, data is large and inherently dynamic. In Facebook, users join and leave the social network frequently, with new friendship links being established or removed all the time. In Twitter, tweets are generated incessantly making older tweets less interesting. As a result, communities evolve over time, new events trigger new dense subgraphs in the corresponding entity-relationship graph, while changing distances between nodes in a graph requires frequent re-indexing.

This calls for efficient algorithms that can cope with large and highly dynamic streams of input data. While either scalable [10] or dynamic algorithms [6] for finding densest subgraphs have been proposed, a viable and satisfactory solution for addressing both the dynamic aspect of the input data and its large size is still missing.

We present the first scalable algorithm for the computation of a densest subgraph in the dynamic graph model [23] with provable guarantees. We focus on the average degree density, which is defined as the ratio between the number of edges and the number of nodes of a given undirected graph. The input of our problem consists of an undirected graph as well as a sequence of update operations on such a graph

(namely edge additions and deletions), describing its evolution over time. At each point in time in the sequence, we would like to maintain a subgraph of the current graph with near-optimal average degree density.

We consider a model similar to the one studied in [9], where edges are added adversarially while they are removed uniformly at random from the current graph. We show that at any point of the sequence of updates, we are able to maintain a $2(1+\epsilon)$-approximation of a current densest subgraph, while requiring $O(\text{poly}\log(n+r))$ amortized cost per update (with high probability), where r is the total number of update operations executed and n is the maximum number of nodes in the graph. Observe that a naïve algorithm, which recomputes the densest subgraph every time the graph changes, would require $\Omega(m)$ work per update, where m is the number of edges in the current graph.

Our theoretical analysis is complemented with an extensive experimental evaluation on real world graphs showing the effectiveness of our approach. In particular, we evaluate our algorithms in the sliding window model, where the most recent edges (e.g. generated in the last hour or day) define the current graph. This entails frequent updates with the most recent edges being added to the current graph and the less recent ones being removed. We show that approximate densest subgraphs can be maintained within hundreds of microseconds per update (on average) on large graphs requiring more than one billion of update operations, making our algorithms an effective tool for studying the evolution of communities in a social network, for finding events automatically in Twitter, as well as, for efficiently maintaining node distance indexes.

The rest of the paper is organized as follows. In Section 2 we review the relevant literature. In Section 3 we define formally the problem. Then, in Section 4 we introduce our algorithms and prove their approximation guarantees and amortized cost bounds. Later, in Section 5 we evaluate experimentally our algorithms on several large scale datasets. Finally, in Section 6 we recap and hint at possible interesting research directions stemming from our work.

2. RELATED WORK

In recent years, there have been significant efforts in devising efficient algorithms for finding densest subgraphs. Due to the large amount of work that has been done in this research area, our survey might not be comprehensive.

Densest subgraph problem. Several definitions of density have been studied in literature including cliques and quasi-cliques [13], α-β-communities [32], minimum degree density [39], k-cores [38] and the densest subgraph [21]. Among these, the average degree density stands outs as a natural definition of density. Given an undirected graph, its average degree density is defined as the ratio between the number of edges and the number of nodes in such a graph. Charikar [16] devised a linear-programming based approach as well as a linear 2-approximation algorithm for such a problem. Bahmani et al. [10] provided a $2(1+\epsilon)$ approximation algorithm that can be implemented in $O\left(\frac{\log(n)}{\epsilon}\right)$ streaming passes over the graph (for a total cost of $O\left(m\frac{\log(n)}{\epsilon}\right)$). More recently, Bahmani et al. [8] provided a near-optimal algorithm in the MapReduce model of computation. In [11], authors study the problem of finding several subgraphs with maximum total density and limited overlap. In [40], heuristics for finding quasi-cliques [40] are presented. All previously mentioned results hold for undirected graphs, while in [25] authors study the densest subgraph problem in directed graphs.

Applications. Finding dense subgraphs in large input graphs has emerged as an important subroutine used to tackle a host of real-world problems [29]. Saha et al. [35] showed how finding dense subgraphs might help in identifying interesting and novel patterns in gene annotation networks. Chen et al. [17] devise a graph partitioning algorithm based on some definition of density and applied it extensively on social and biological networks. Finding dense regions in web graphs has received large attention recently [19, 20] as those regions might represent some link-spam activity. Dourisboure et al. [19] proposed an efficient heuristic for such a problem. Similarly Kumar et al. [27] cast the problem of finding communities of web-pages as the problem of mining dense bipartite subgraphs. Densest subgraph computation is an important subroutine in graph indexing for efficient reachability and distance queries [18,24], as well as for graph compression [14]. Angel et al. [6] show how dense subgraphs represent interesting events in Twitter.

Dynamic settings. Das Sarma et al. [37] studied the densest subgraph problem in a dynamic version of the congest model for distributed computation [33], which presents significantly different challenges. In this model, authors develop a $(2+\epsilon)$-approximation algorithm which requires $O\left(D\frac{\log(n)}{\epsilon}\right)$ parallel rounds where D is the diameter of the network. Contrary to our work, there is no guarantee that the amortized cost be poly-log(n). Angel et al. [6] study the problem of efficiently maintaining dense subgraphs under streaming edge weight updates, so as to find interesting events in Twitter. Authors consider a different definition of density, while they focus on finding small dense subgraphs (with at most 10 nodes). In our work, we would like not to restrict our attention to graphs with small number of nodes given that this is not the case in many cases of interest (such as communities in a social network). Moreover, in contrast to our work there is no guarantee that the amortized cost be poly-log(n) per update. Valari et al. [41] study the problem of computing the top-k densest subgraphs, when the densest subgraph is removed recursively at each time. They address such a problem in dynamic graph collections, where at each step graphs might arrive or be removed from the collection. Other authors have studied other definitions of density in dynamic graphs. Lee et al. [28] used (k,d)-cores to discover new stories from social streams and track their evolution. Several authors [5, 31, 36] addressed the related problem of maintaining a k-core decompositions in graphs subject to dynamic changes and in streaming. In [23] authors study the problem of efficiently answering queries on the properties of an evolving graph.

None of the previous approaches maintains an approximation of the densest subgraph in amortized poly-log(n) time per update and can cope with billions of edge additions or removals. Recently, independently from our work, Bhattacharya et al. [12] (unpublished at the time of preparation of this paper) studied the densest subgraph problem in the streaming model of computation achieving strong theoretical guarantees.

3. PROBLEM DEFINITION

Preliminaries. Let $G = (V, E)$ be an undirected graph. For any set of vertices $S \subseteq V$, we let $E(S)$ to be the set of edges induced by S, i.e., $E(S) = \{(u, v) : (u, v) \in E \wedge u, v \in S\}$. The average degree density $\rho_G(S)$ is defined as: $\rho_G(S) = \frac{|E(S)|}{|S|}$. In the rest of the paper we omit the subscript G in $\rho_G(S)$ when it is clear from the context, while we use the term density instead of average degree density, for simplicity. We denote by ρ_O the density of a densest subgraph. We say that an algorithm computes an α-approximation of a densest subgraph $S \subseteq V$ if such an algorithm computes a subgraph $S' \subseteq V$ such that $\rho(S) \leq \alpha\rho(S')$. We say that an algorithm has amortized time $O(\mathcal{T}(n))$ per update, if for any sequence of ℓ update operations such an algorithm completes in time $O(\ell \cdot \mathcal{T}(n))$. We denote by polylog$(n)$ a polynomial in the logarithm, such as $\log^2 n$.

Problem definition. We study the densest subgraph problem in the dynamic graph model [23], where a sequence of update operations on an input graph describes the evolution of such a graph over time. Our input consists of an undirected graph $G = (V, E)$ as well as an online sequence of update operations on the input graph, namely edge additions and deletions. Every edge in such a sequence is either added adversarially (i.e. so as to cause maximum discomfort to the algorithm) or is removed uniformly at random from the current graph. We assume that the edges to be added (adversarially) are chosen without taking into account the set of removed edges. Our graph-evolution model is similar to the model in [9]. Our goal is to maintain at any point in time in the sequence a $2(1 + \epsilon)$-approximation of a densest subgraph with amortized time of $O(\text{polylog}(|V|)$, for any $\epsilon > 0$. In particular, at any point in time in the sequence, the output of our algorithm consists of a collection of subsets of V, with the last among such subsets inducing a $2(1 + \epsilon)$-approximation of a densest subgraph in the current graph. The amortized cost must include the time required to produce such subgraphs in output.

4. ALGORITHMS

In this section, we present several algorithms to compute an approximation of a densest subgraph. We start by considering the static case to warm-up, then, we generalize our solution to a dynamic setting. For simplicity, we initially consider a model with edge addition only which is then generalized to the most general case with both edge addition and removal.

4.1 Warm-up: Static Case

We first design an approximation algorithm for the static case, which is perhaps not interesting per se but it is pivotal in dealing with the dynamic case.

Consider the following algorithm inspired by the algorithm introduced in [10]. Given rational numbers $\beta > 0, \epsilon > 0$, we start from the input graph G and remove iteratively all nodes with degree $< 2(1+\epsilon)\beta$ from the current graph, until the graph becomes empty or the total number of iterations becomes larger than $\lceil \log_{1+\epsilon}(n) \rceil$ — notice that $\log_{1+\epsilon}(n) = O\left(\log(n)\epsilon^{-1}\right)$. We then return the subgraph with maximum density among all subgraphs computed throughout the execution of the algorithm. It can be shown that if the density ρ_O of a densest subgraph is known then one can run such an algorithm with $\beta = \frac{\rho_O}{2(1+\epsilon)}$ (i.e. at each step we remove nodes with degree smaller than ρ_O) to quickly find an approximate densest subgraph. See Algorithm 1 for a pseudo-code.

Algorithm 1 Find($G(V, E)$, $\beta > 0$, $\epsilon > 0$)

```
S_0, S̄ ← V, t ← 0.
while S_t ≠ ∅ and t ≤ ⌈log_{1+ε}(n)⌉ do
    Let A(S_t) be the set of nodes with degree < 2(1+ε)β in G_t = (S_t, E(S_t)).
    S_{t+1} ← S_t \ A(S_t).
    If ρ(S_{t+1}) > ρ(S̄) then S̄ ← S_{t+1}.
    t ← t + 1.
end while
return Ḡ = (S̄, E(S̄)).
```

Unfortunately ρ_O is usually not known in advance, so in order for Algorithm 1 to be useful we need to understand its behavior when executed with an arbitrary $\beta > 0$. This is clarified by the following lemma, whose proof goes along the lines of the proofs in [10].

LEMMA 1. *If $0 < \beta \leq \frac{\rho_O}{2(1+\epsilon)}$, Algorithm 1 finds a subgraph with density at least β, while if $\beta > \rho_O$ a subgraph with density strictly less than β is found.*

PROOF. If $\beta > \rho_O$, then the claim follows by definition of ρ_O. We show that in the case when Algorithm 1 does not find a subgraph with density at least β, with $0 < \beta \leq \frac{\rho_O}{2(1+\epsilon)}$, the density of a densest subgraph is strictly less than ρ_O, leading to a contradiction.

Let $S_t \subseteq V$ be the set of nodes active at rounds t of the algorithm and $A(S_t)$ be the set of nodes with degree $< 2(1 + \epsilon)\beta$ in $G_t = (S_t, E(S_t))$. We have

$$2|E(S_t)| = \sum_{v \in S_t \setminus A(S_t)} \delta_{G_t}(v) + \sum_{v \in A(S_t)} \delta_{G_t}(v).$$

Furthermore if Algorithm 1 does not find a graph with density at least β then $\rho(S_t) < \beta$ for every t, otherwise the algorithm would have found a subgraph with density at least β. So by considering only the first summation term in the previous equation we derive

$$2|E(S_t)| \geq 2\beta(1 + \epsilon)|S_t \setminus A(S_t)| > 2\rho(S_t)(1 + \epsilon)|S_t \setminus A(S_t)|,$$

where the second inequality follows from $\beta > \rho(S_t)$. Then

$$2|E(S_t)| > 2(1 + \epsilon)|S_t \setminus A(S_t)|\frac{|E(S_t)|}{|S_t|},$$

which implies $|S_{t+1}| = |S_t \setminus A(S_t)| < \frac{1}{1+\epsilon}|S_t|$, for every step $t \geq 1$. Hence, Algorithm 1 terminates at step $t \leq \lceil \log_{1+\epsilon}(n) \rceil$ with all nodes and edges being removed from the input graph G.

We now show that if at the end of the algorithm all edges are removed and $\beta \leq \frac{\rho_O}{2(1+\epsilon)}$, then the density of a densest subgraph in G is strictly less than ρ_O leading to a contradiction.

We define a directed graph H, obtained by finding an orientation of the edges in the input graph G as follows. At every step $t \geq 1$ of Algorithm 1, for every node v in $A(S_t)$ we assign all edges of v in G_t, to v (breaking ties arbitrarily). Let $\delta_H(v)$ be the in-degree of v in the directed graph H. From the fact that all nodes and edges have been removed at the end of the algorithm, it follows that all edges are assigned to some node. We can then write:

$$\rho(S_O) = \frac{|E(S_O)|}{|S_O|} \leq \frac{\sum_{v \in S_O} \delta_H(v)}{|S_O|} < 2(1 + \epsilon)\beta \leq \rho_O,$$

which leads to a contradiction. Therefore, our algorithm would have found a subgraph with density at least β, when executed with $\beta \leq \frac{\rho_O}{2(1+\epsilon)}$. □

The following lemma can be proved in a very similar way to Lemma 1, its proof is omitted for brevity and deferred to the full version of the paper.

LEMMA 2. *Let $\beta, \epsilon > 0$. Let $\bar{G}$ and S_k be, respectively, the subgraph and the k-th set found by Algorithm 1 with input β, ϵ, where $k = \lceil \log_{1+\epsilon}(n) \rceil$. If $\rho(\bar{G}) < \beta$, then $S_k = \emptyset$. Moreover, if $S_k = \emptyset$ then $\rho_O < 2(1+\epsilon)\beta$.*

Lemma 1 suggests the following natural algorithm for finding an approximate densest subgraph in the static case: Starting from a "sufficiently small" value for β, we run Algorithm 1 with different values of β (namely all integral powers of $(1+\epsilon)$) outputting the subgraph with maximum density among those we found. Algorithm 2 shows a pseudo-code for such an algorithm, where $\bar{\rho}$ is a lower bound for the density of a densest subgraph (this parameter will be used in the next sections).

Algorithm 2 *FindDensest*$(G(V,E), \bar{\rho}, \epsilon > 0)$

```
Ḡ = ∅.
β ← max( 1/(4(1+ε)), (1+ε)ρ̄ ).
while (true) do
    G' ← Find(G, β, ε).
    if ρ(G') ≥ β then
        Ḡ ← G'.
        β ← (1+ε)ρ(Ḡ).
    else
        return (β, Ḡ).
    end if
end while
```

While Algorithm 2 might perform worse than other known algorithms for the static case, it will be a key ingredient for our dynamic algorithms. We conclude this section by proving its performance guarantees.[1]

THEOREM 1. *For any $\epsilon > 0$, FindDensest$(G, 0, \epsilon)$ computes a $2(1+\epsilon)^2$-approximation of a densest subgraph.*

PROOF. If the input graph is not empty then $\rho_O \geq \frac{1}{2}$. Therefore, it follows from Lemma 1 that Algorithm 2 executed with $\beta = \frac{1}{4(1+\epsilon)}$ finds in the first step a set of density at least β.

The loop in Algorithm 2 terminates as soon as a subgraph with density β is not found, while outputting a subgraph with density $\frac{\beta}{1+\epsilon}$. From Lemma 1 it follows that $\rho_O \leq 2(1+\epsilon)\beta$, which proves the approximation guarantee. □

THEOREM 2. *For any $\epsilon > 0$, FindDensest$(G, 0, \epsilon)$ requires $O(m \log(n) \epsilon^{-1})$ operations.*

PROOF. From the fact that $\rho_O \leq n$ and from Lemma 1, it follows that the while loop of Algorithm 2 is executed at most $\lceil \log_{1+\epsilon} n \rceil$ times. Since the input graph in undirected, Algorithm 1 can be implemented in $O(n+m)$ using the same method described in [25] for computing densest subgraphs in an undirected graph. □

[1]For simplicity, we express the approximation guarantee of our algorithms as $2(1+\epsilon)^2$, for any $\epsilon > 0$. Observe that this is equivalent to say that its approximation guarantee is $2(1+\epsilon)$, for any $\epsilon > 0$. The asymptotic amortized cost does not change.

4.2 Dynamic Case: Edge Insertion Only

In this section, we describe how to maintain a $2(1+\epsilon)^2$-approximation of a densest subgraph with amortized cost of $O\left(\frac{\log(n)^2}{\epsilon^2}\right)$ per edge insertion. We first assume that the set V of all the nodes appearing in the evolving graph is known in advance, then we show how to remove such assumption.

The core idea is to maintain the main properties of the *Find* and *FindDensest* algorithms, without having to rerun the algorithm from scratch every time the graph is updated.

Consider the first call to Algorithm 2. Let $S_0, \ldots, S_k$, $k = \lceil \log_{1+\epsilon}(n) \rceil$ be the sets computed by the last call to *Find* in Algorithm 2. Our algorithm will maintain throughout the execution the following invariant:

INVARIANT 1. *Given $\beta, \epsilon > 0$ and the current graph $G = (V, E)$, the nodes in V are organized as a collection of sets $S_0, \ldots, S_k$, $k = \lceil \log_{1+\epsilon}(n) \rceil$, where*

- *$S_0 = V$;*
- *S_{t+1} is obtained from S_t by removing all nodes with degree less than $2(1+\epsilon)\beta$ in $G_t = (S_t, E(S_t))$, $t = 0, \ldots, k-1$;*
- *$S_k = \emptyset$.*

Note that Invariant 1 is satisfied at the end of any call to Algorithm 2 (it follows from Lemma 2), while we maintain it as follows. Given an edge (u, v) being added to the current graph $G = (V, E)$ and a collection of sets $S_0, \ldots, S_k$ satisfying Invariant 1, we first check whether adding (u, v) to G does not violate the invariant. If this is the case we add (u, v) to G while leaving the sets S_t's unchanged. Otherwise, let $u \in V$, and let t be such that $u \in S_t \setminus S_{t+1}$ and $\delta_{G_t}(u) \geq 2(1+\epsilon)\beta$, while let $\hat{t} > t$ be the smallest index such that $\delta_{G_{\hat{t}}}(u) < 2(1+\epsilon)\beta$. If $\hat{t} = \lceil \log_{1+\epsilon}(|V|) \rceil$ then we rebuild the sets S_t's from scratch, which is done by running Algorithm 2 and using the sets S_t's computed in the last call to *Find*.

Otherwise, we add u to the sets $S_{t+1}, \ldots, S_{\hat{t}}$. In turn, we might need to move the neighbors of u to other sets and so on. This is iterated for u and v until the invariant becomes satisfied or we need to move some node to a set $S_{\hat{t}}$, with $\hat{t} = \lceil \log_{1+\epsilon}(|V|) \rceil$. In the latter case, we rebuild the sets S_t's from scratch.

A pseudo-code of the algorithm is shown in Algorithm 3. The algorithm returns *true* if the sets S_t's need to be rebuilt, which is done in Algorithm 4. Note that maintaining the invariant might be costly sometimes, as moving nodes from one set to the other might trigger a chain effect involving a large part of the nodes in the current graph. However, we are able to prove that this does not happen "often" (see Theorem 4).

Armed with an algorithm for maintaining our invariant, we now present the main algorithm which is executed continuously. Algorithm *Main* (Algorithm 4) receives an initial graph $G = (V, E)$ (potentially with no edges) in input and runs *FindDensest* to build a collection of sets $S_0, \ldots, S_k$ satisfying Invariant 1. Then the algorithm outputs our approximation of the densest subgraph $\bar{G}$. When a new edge (u, v) is added to the current graph G, algorithm *Add* is executed and the sets S_t's are updated so as to maintain the invariant. If this can be done, then the density of the densest

Algorithm 3 $Add((u,v),(S_0,\ldots,S_k),G=(V,E),\beta,\epsilon)$

Require: $S_0,\ldots,S_k$ must satisfy Invariant 1
Ensure: $S_0,\ldots,S_k$ are updated so to satisfy Invariant 1 after adding (u,v), if this is not possible return a flag $Rebuild = true$ signaling that such sets must be rebuilt.

$E \leftarrow E \cup \{(u,v)\}$.
Update degrees of u and v.
$Stack \leftarrow \emptyset$.
$Push(u,v,Stack)$.
while $Stack$ is not empty **do**
  $w \leftarrow Pop(Stack)$.
  Let S_t be such that $w \in S_t \setminus S_{t+1}$.
  If $\delta_{G_t}(w) < 2(1+\epsilon)\beta$ **continue**.
  Let $\hat{t} > t$ be smallest such that $\delta_{G_{\hat{t}}}(w) < 2(1+\epsilon)\beta$.
  If $\hat{t} = \lceil \log_{1+\epsilon}(|V|) \rceil$ **return** $true$.
  Add w to the sets $S_{t+1},\ldots,S_{\hat{t}}$.
  Push all neighbors of w to the $Stack$.
end while
return $false$.

subgraph has not increased significantly (this follows from Lemma 2). In this case the previously computed subgraph $\bar{G}$ is still a good approximation of the densest subgraph.

Otherwise, we rebuild the sets from scratch by running $FindDensest$ as in this case the density of the densest subgraph might have increased. After this operation $Main$ outputs a new dense subgraph $\bar{G}$. Notice that although rebuilding the S_t's is an expensive operation (as it requires to process the whole input graph), this is performed only a poly-log number of times (as β is multiplied each time by a factor of $(1+\epsilon)$ and never decreases).

Algorithm 4 $Main(G=(V,E),\epsilon)$

$(\beta,\bar{G}) \leftarrow FindDensest\ (G,0,\epsilon)$ and let $S_0,\ldots,S_k$ be the sets computed by $Find$.
Output $\bar{G}$.
while (true) **do**
  Wait for a new edge (u,v).
  $Rebuild \leftarrow Add((u,v),S_0,\ldots,S_k,G,\beta,\epsilon)$.
  if ($Rebuild$) **then**
    $(\beta,\bar{G}) \leftarrow FindDensest\ (G,\beta,\epsilon)$ (update $S_0 \ldots S_k$).
    Output $\bar{G}$.
  end if
end while

We now study the approximation guarantee of our algorithm.

THEOREM 3. *The algorithm always maintains a $2(1+\epsilon)^2$ approximation of the densest subgraph.*

PROOF. From Lemma 2 it follows that Invariant 1 holds at the end of $FindDensest$. Moreover, it either holds at the end of Add or $FindDensest$ is executed right after that, so as to maintain such an invariant. Therefore, at any point in time the subgraph $\bar{G}$ found by $Main$ has density at least $\frac{\beta}{1+\epsilon}$, while from Lemma 2 it follows that $\rho_O < 2(1+\epsilon)\beta$. This proves the approximation guarantee. □

The following lemma is useful for analyzing the amortized cost of the algorithm. Its proof is omitted for brevity and deferred to the full version of the paper.

LEMMA 3. *At any point in time, let n be the number nodes in the current graph $G=(V,E)$. The total number of $FindDensest$ calls in the algorithm, up to this point is $O(\log(n)\epsilon^{-1})$.*

The following theorem shows that the algorithm has amortized cost $O(\log^2(n)\epsilon^{-2})$ per edges in the graph.

THEOREM 4. *At any point in time, let n and m be the number of nodes and edges in the current graph $G=(V,E)$, respectively. The total number of operations of the algorithm, up to this point, is $O(m\log^2(n)\epsilon^{-2})$ while the algorithm requires $O(m+n)$ space.*

PROOF. Notice that to implement the algorithm, it is sufficient to store the current graph and some additional bookkeeping information of cost $O(n)$ (we do not need to store the sets S_t's explicitly but only which is the last set in which each node is present).

We now bound the total number of operations performed by $Main$. We start by proving that the total number of operations executed between two consecutive calls to $FindDensest$ is $O(m\log(n)\epsilon^{-1})$. Consider any call to $FindDensest$ and any node v. Let t and $\hat{t}$ be such that v belongs to some set $S_t \setminus S_{t+1}$ right after such a call and v belongs to $S_{\hat{t}} \setminus S_{\hat{t}+1}$ right before the next call to $FindDensest$. By construction, it must be $\hat{t} \geq t$. Moving a node from S_t to $S_{\hat{t}}$ requires at most $\delta_G(u)\cdot(\hat{t}-t)$ operations, while the number of sets S_t's is at most $\lceil \log_{1+\epsilon}(n) \rceil$. Therefore, the total number of operations between two consecutive calls to the procedure $FindDensest$ is at most $\sum_{v\in V}\delta_G(v)\log_{1+\epsilon}(n) = O(m\log(n)\epsilon^{-1})$. From Lemma 3, it follows that there are $O(\log(n)\epsilon^{-1})$ calls to $FindDensest$, with each of them requiring $O(m\log(n)\epsilon^{-1})$ operations.

This concludes the proof. □

Handling node insertions. In this section we have assumed for simplicity to know the set V of all nodes that will appear in the graph. This is not necessary as simple changes are sufficient to adapt the algorithm to the general case. Each time a new node v is introduced, we first assign it to the set S_0 (a node of degree 0 is always removed at the first iteration), then the algorithm proceed as usual. Notice as well that we use the cardinality of V only to determine the threshold $k = \lceil \log_{1+\epsilon}(|V|) \rceil$ for the maximum number of sets S_t's. It is easy to see that as $|V|$ can only grow we can simply keep track of the current cardinality of V and always use the current value when necessary. The monotonicity of $|V|$ ensure that the algorithm remains consistent and that the theorems continue to hold.

4.3 Fully dynamic case

In this section, we show how to adapt the previous algorithm to the fully dynamic model with both adversarial edge insertions and random removals as described in Section 3. For this case we provide an approximation algorithm that handles both operations in amortized poly-log time, in the size of the graph and the number of operations, with high probability. We stress that the approximation guarantees of our algorithm holds for adversarially chosen edge insertions and removals. We rely on the randomness of removals only for showing the amortized complexity bounds. In the experimental section (Section 5.2), we conduct an extensive empirical evaluation showing the effectiveness of our algorithm in the sliding window model as well.

We start from an initial graph $G=(V,E)$. We first assume that the set V of all nodes appearing in the graph is

known in advance, while we show later on in this section how to relax this requirement. Let A be the total number of (possibly repeated) edges that are inserted in the graph, i.e. A counts both the edges in the initial graph and the edge addition operations. Let R be the number of edge deletions executed. We have that $A + R = O(A)$. We will bound the total cost of this sequence of operations in the probability space defined by the random choices of the edge removals. We say that an event happens with high probability (w.h.p.) if it happens with probability $\geq 1 - A^{-c}$ for some constant $c > 1$.

The main idea of our algorithm is to lazily maintain a data structure (the sets S_t's presented in the previous section) from which we can extract a dense subgraph, while requiring a small number of operations (on average). To this purpose we maintain a slightly weaker invariant (Invariant 2) which is defined as follows:

INVARIANT 2. *Given $\beta, \epsilon > 0$ and the current graph $G = (V, E)$, the nodes in V are organized as a collection of sets $S_0, \ldots, S_k$, $k = \lceil \log_{1+\epsilon}(n) \rceil$, where*

- $S_0 = V$;
- *For $t = 0, \ldots, k-1$, $S_{t+1} \subseteq S_t$ and if $v \in S_t$ has degree larger than or equal to $2(1+\epsilon)\beta$ in $G_t = (S_t, E(S_t))$ then $v \in S_{t+1}$;*
- $S_k = \emptyset$.

Notice that this invariant is weaker than the previous one, in that, we only require that high degree nodes in S_t are also part of set S_{t+1} (no condition is placed on the removal of low degree nodes). Similar to the previous section, the sets S_t's are first defined as the ones obtained in the last call to *Find* in *FindDensest* and then are later updated as described in the algorithm.

Unfortunately, after many update operations such data structure might become corrupted and must be reconstructed from scratch by means of a costly procedure (*FindDensest*). Our goal is to minimize the number of times that such a costly procedure is executed. To this purpose we divide the execution of the algorithm in rounds, with each round containing all the operations required to maintain Invariant 2 between two consecutive calls to the costly procedure *FindDensest*. We say that a round is *good* if the total number of edge removal operations in the round is larger than or equal to a threshold R^*, while it is *bad* otherwise. R^* is defined as follows:

$$R^* := \frac{m_0 \epsilon}{6(1+\epsilon)^2 \log_{1+\epsilon}(|V|)}, \tag{1}$$

where m_0 is the number of edges in the graph at the beginning of the round. Notice that R^* might not be an integer.

The main intuition is that in good rounds the number of maintenance operations is "justified" by a large number of update operations, causing no issues in our amortized cost analysis. We will then show that the bad rounds are not too many with respect to the good ones.

Given a parameter $\epsilon > 0$, our algorithm works with a parameter $\epsilon' = 2\epsilon + \epsilon^2$. The algorithm starts with round 1 where *FindDensest* is executed so to construct the sets S_t's. At round s, the execution of such a procedure starts a new round $s + 1$. A counter R' keeps track of the number of removals that have been been executed in the current round so to be able to determine whether the current round is good or bad. When a new edge is added to the current graph, our algorithm executes the same algorithm *Add* described in the previous section, while when an edge is removed our algorithm executes the algorithm *Remove* (see Algorithm 6 for a pseudo-code). In either case, Invariant 2 is maintained. Both algorithms return a flag *Rebuild* signaling whether the S_t's must be rebuilt, in which case, *FindDensest* is executed. See Algorithm 5 for a pseudo-code of *MainFullyDyn*. For efficiency reasons we maintain a graph $\tilde{G}$ which is a super graph of the current graph.

Algorithm 5 *MainFullyDyn* $(G = (V, E), \epsilon)$

```
ε' ← 2ε + ε², R' ← 0.
s ← 1 // Begin round 1.
(β_1, G̃) ← FindDensest (G, 0, ε').
Let S_0, ..., S_k be the sets computed by Find.
G̃ = (V, Ẽ) ← G. //supergraph of G maint. for efficiency.
Let R* as in Equation 1, where m_0 = |E|.
Output Ḡ.
while (true) do
    Wait for update (Op, u, v), Op ∈ {Add, Remove}.
    if Op = Add then
        if R' < R* then Ẽ ← Ẽ ∪ (u, v).
        Rebuild ← Add((u, v), (S_0, ..., S_k), G, β_s, ε').
    else
        Rebuild ← Remove((u, v), G, Ḡ, β_s, ε').
        R' ← R' + 1.
    end if
    if (Rebuild) then
        if R' < R* then
            Let H ← Find (G̃, β_s, ε')
            if ρ_G(H) ≥ ρ_G(Ḡ) then Ḡ ← H
        end if
        s ← s + 1. //begin round s + 1.
        (β_{s+1}, H) ← FindDensest(G, ρ_G(Ḡ), ε') (update
S_0 ... S_k as the sets computed by Find).
        G̃ = (V, Ẽ) ← G.
        if ρ_G(H) ≥ ρ_G(Ḡ) then Ḡ ← H.
        Output Ḡ.
        Let R* as in Equation 1, where m_0 = |E|.
        R' ← 0.
    end if
end while
```

We now prove the approximation guarantees of our algorithm.

THEOREM 5. *The algorithm always maintains a $2(1+\epsilon)^6$ approximation of a densest subgraph.*

PROOF. Notice that algorithm *Remove* ensures that any time we have a set S of density $\rho(S) \geq \frac{\beta}{(1+\epsilon')^2}$. Furthermore by Invariant 2 we know that if we execute *Find* (G, β, ϵ') on the current graph all nodes are removed in at most $\log_{1+\epsilon'}(|V|)$ iterations. Hence, by Lemma 2 it follows that $\rho_O < 2(1+\epsilon')\beta$ and this proves the approximation guarantee $2(1+\epsilon')^3 = 2(1+\epsilon)^6$. □

We now state the following theorem that proves that our algorithm has amortized cost $O(\log(A)\log^2(n)\epsilon^{-2})$ for edge addition and $O(\log(A)\log^3(n)\epsilon^{-4})$ for edge removal, w.h.p.

THEOREM 6. *For any $\epsilon > 0$, at any point in time, let A and R be the number of edge insertion and deletion operations. The total number of operations of the algorithm is w.h.p $O\left(A\log(A)\log^2(|V|)\epsilon^{-2} + R\log(A)\log^3(|V|)\epsilon^{-4}\right)$, while the algorithm requires $O(n+m)$ space.*

Algorithm 6 *Remove*$(u, v, G(V, E), \bar{G}, \beta, \epsilon > 0)$

Ensure: $\rho(\bar{G}) \geq \frac{\beta}{(1+\epsilon)^2}$ after removing (u, v), if this is not true return a flag *Rebuild* = *true* signaling that the densest subgraph must be recomputed.

$E \leftarrow E \setminus \{(u, v)\}$.
Update degrees of u and v.
if $\rho(\bar{G}) < \frac{\beta}{(1+\epsilon')^2}$ **then**
  return *true*
else
  return *false*
end if

Proof of Theorem 6

Our proof strategy for Theorem 6 is the following. We analyze separately the case of bad ($R' < R^*$) and good ($R' \geq R^*$) rounds. In the former case we show that the density of the densest subgraph is unlikely to be affected significantly, so we can use similar arguments to the ones used in the previous subsection. In the latter case, we amortize the cost of the re-computation of the densest subgraph using the large number of edge update operations.

The next lemma shows that many random edge removals are required to decrease significantly the density of any given subgraph. Such a lemma is used to prove that, we cannot have many (more than poly-log) consecutive bad rounds, w.h.p. Finally, we will use this result to bound the average cost of each operation.

LEMMA 4. *Given an undirected graph $G = (V, E)$, let H be a subgraph of G while let G' be a graph obtained by removing R' edges uniformly at random from G, with $0 \leq R' < R^*$. If $\rho_G(V(H)) \geq \beta$ then, $\rho_{G'}(V(H)) \geq \frac{\beta}{(1+\epsilon)}$, with probability at least $p = 1 - (6 \log_{1+\epsilon}(|V|))^{-1}$.*

PROOF. The probability of removing an edge from H is smaller than $\frac{|E(H)|}{|E(V)|-R^*}$. Let X be the random variable that counts the number of edges removed from H. We can bound $E[X]$ as follows:

$$E[X] \leq R^* \frac{|E(H)|}{|E(V)| - R^*} \leq \frac{|E(H)|\epsilon}{6(1+\epsilon)\log_{1+\epsilon}(|V|)}$$

Furthermore, note that $|E(H)| \geq \beta|V(H)|$ and that in order to decrease the density of H to $\frac{\beta}{(1+\epsilon)}$ at least $\beta|V(H)|(1-\frac{1}{1+\epsilon}) = \frac{\beta|V(H)|\epsilon}{1+\epsilon}$ edges must be removed.
Hence, from Markov inequality it follows:

$$\begin{aligned} P\left(X > \frac{\beta|V(H)|\epsilon}{1+\epsilon}\right) &\leq \frac{E[X]}{\frac{\beta|V(H)|\epsilon}{1+\epsilon}} \\ &\leq \frac{1}{6\log_{1+\epsilon}(|V|)}. \end{aligned}$$

Hence, with probability at least $1 - \frac{1}{6\log_{1+\epsilon}(|V|)}$ the graph H has density $\geq \frac{\beta}{1+\epsilon}$. □

We now state a lemma is deferred to the full version of the paper

LEMMA 5. *For any bad round s of $MainFullyDyn$, $\beta_{s+1} \geq \beta_s(1+\epsilon)$ with probability at least $1 - \frac{1}{3\log_{1+\epsilon}(|V|)}$, independently of the previous rounds.*

We now prove the following lemma that can be used to show that we cannot have many consecutive bad rounds.

LEMMA 6. *Let $\epsilon > 0$. At any point in time during the execution of MainFullyDyn, let A be the total number of edge additions. There are less than $O(\log(|V|)\log(A)\epsilon^{-1})$ consecutive bad rounds, with high probability.*

PROOF. Consider an arbitrary sequence of k consecutive bad rounds $s_1, \ldots, s_k$, with $k \geq 6\lceil \log_{1+\epsilon}(|V|)\log_{3/2}(A)\rceil$. Let $p = \frac{1}{3\log_{1+\epsilon}(|V|)}$ and $t = 2\lceil \log_{1+\epsilon}(|V|)\rceil$. Let β_i be the value of β at round s_i. From Lemma 5 and union bound, it follows that the probability that there is $i < t$ such that $\beta_{i+1} < (1+\epsilon)\beta_i$ is at most $p \cdot t < \frac{2}{3}$. Therefore, with probability at least $1 - \frac{2}{3}^{3\log_{3/2}(A)} = 1 - O(A^{-3})$ there are t consecutive bad rounds $s_{i+1}, s_{i+2}, \ldots, s_{i+t}$, with $i + t < k$ such that $\beta_{j+1} \geq (1+\epsilon)\beta_j$ for any $j \in [i+1, i+t-1]$. However, by definition the β_j's are bounded by $|V|$ and therefore cannot increase by a factor of $(1+\epsilon)$ more than $\log_{1+\epsilon}(|V|) < t$ times. From the above observations, it follows that if there are more than k consecutive bad rounds then the latter condition is not satisfied with probability at least $1 - O(A^{-3})$. From the fact that there are at most $A + R = O(A)$ rounds in total and from a union-bound argument it follows that throughout the execution of the algorithm there are at most $6\lceil \log_{1+\epsilon}(|V|)\log_{3/2}(A)\rceil = O(\log(|V|)\log(A)\epsilon^{-1})$ consecutive bad rounds, w.h.p. □

We can now provide a proof sketch of the main theorem. The proof is omitted for brevity and deferred to the full version of the paper.

PROOF SKETCH OF THEOREM 6. Similarly to the algorithm in the previous section, it is easy to see that the algorithm can be implemented in space $O(n+m)$ (for storing the nodes and edges in the current graph) plus some additional space for bookkeeping information of cost $O(n)$.

We now bound the amortized cost of our algorithm. From Lemma 6, it follows that at any point in time we can partition (w.h.p.) all rounds of $MainFullyDyn$ in *super-rounds*, such that every super-round, except possibly the last one, contains $O(\log_{1+\epsilon}(|V|)\log(A))$ consecutive bad rounds followed by one final good round, while the last super-round contains $O(\log_{1+\epsilon}(|V|)\log(A))$ consecutive bad rounds which might or might not be followed by one final good round. We prove our bound on the amortized cost for any such super-round, which will prove the desired claim.

We first consider any super-round $\mathcal{S}$ containing both good and bad rounds. Let a and r be the total number of edge additions and deletions in $\mathcal{S}$. By using similar arguments to the ones in the proof of Theorem 4 and using the fact that we have at most $O(\log_{1+\epsilon}(|V|)\log(A))$ rounds in a given super-round, it follows that the total number of operations performed throughout $\mathcal{S}$ is $O\left((m_0 + a + r)\log(A)\log^2(|V|)\epsilon^{-2}\right)$, where m_0 is the number of edges at the beginning of $\mathcal{S}$.

Let a' and r' be, respectively, the total number of additions and removals in the bad rounds of $\mathcal{S}$, while let a'', r'' be the total number of additions and removals in the final good round of $\mathcal{S}$, respectively. We now use the fact that in any good round a large number of removal operations is performed.

By definition of good round, $r'' \geq R^* \in \Theta\left(\frac{m'\epsilon}{\log_{1+\epsilon}(|V|)}\right)$, where m' is the number of edges at the beginning of the good round. Using the fact that $m' = m_0 + a' - r'$ and after some

Graph	$\|V\|$	$\|E\|$	Time
DBLP	938,609	4,541,961	1959-2014
Patent (Co-Aut.)	1,162,227	3,660,945	1975-1999
Patent (Cit.)	2,745,762	13,965,410	1975-1999
LastFm	681,387	43,518,693	2005-2009
Yahoo! Answer	2,432,573	1.21×10^9	2006

Table 1: Properties of the dynamic graphs analysed. $|V|$, $|E|$, refer to the number nodes appearing in the graph and number of edge additions, respectively. Time indicates when evolution took place.

simple algebra, we derive $m_0 \in O\left(r\frac{\log(|V|)}{\epsilon^2}\right)$. Therefore, we can bound the total number of operations $C_{\mathcal{S}}$ performed in the super-round $\mathcal{S}$ as follows

$$\begin{aligned} C_{\mathcal{S}} &= O\left((m_0 + a + r)\log(A)\log^2(|V|)\epsilon^{-2}\right) \\ &= O\left(a\log(A)\log^2(|V|)\epsilon^{-2} + r\log(A)\log^3(|V|)\epsilon^{-4}\right). \end{aligned}$$

By summing up over all super-rounds we prove the desired claim. Notice that if $\mathcal{S}$ does not contain any good round then there is an additional term $O\left((A+R)\log(A)\log^2(|V|)\epsilon^{-2}\right)$. As this might happen only for the last super-round this does not change the asymptotic cost. □

Handling node insertions. Similar consideration of the previous section holds in this context. When a node is added we can assign it to the first set S_0 only. Moreover, if we define V as the set of all nodes ever inserted in the graph, then $|V|$ is again monotone increasing. So the algorithm remains consistent.

5. EXPERIMENTS

In this section, we provide an extensive experimental evaluation of our algorithms on large real-world graphs some of which contain more than one billion edges. From each dataset, we extracted an undirected graph with each edge being associated with a timestamp.

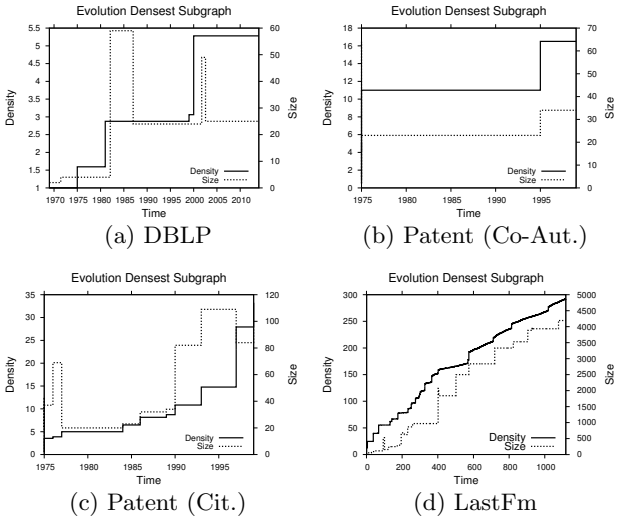
Datasets & Experimental setup.

We start by describing our publicly available datasets. See Table 1 for a summary of their features.

DBLP. The *DBLP* graph is obtained from a snapshot [1] of a coauthorship graph, constructed from 2.5 million scientific publications in computer science published between 1959 and 2014. Nodes in this graph represent authors, while an edge with timestamp t between two nodes indicates that the corresponding authors published an article in year t. To take into account only strong scientific collaborations, we restrict our attention to papers with at most 10 authors.

Patent (Co-Aut.) & Patent (Cit.). The *Patent (Co-Aut.) & Patent (Cit.)* graphs are obtained from a dataset [22] containing about 2 million U.S. patents granted between '75 and '99. In the *Patent (Co-Aut.)* graph, nodes represent inventors while an edge with timestamp t connects two inventors of the same patent granted in year t. In the *Patent (Cit.)* graph, nodes represent patents while an undirected edge with timestamp t connects two patents if one received a citation from the other in year t.

LastFm. The LastFm graph is obtained from a dataset [2, 15] of song listenings of `last.fm` users. The dataset contains about ≈ 20 million listenings of ≈ 1 million songs from ≈ 1000 users. In this graph, nodes represent songs while an edge with timestamp t connects two songs if they where *both* listed by the same 3 or more users in the same day t.



(a) DBLP (b) Patent (Co-Aut.)

(c) Patent (Cit.) (d) LastFm

Figure 1: Evolution of the densest subgraph — edge addition only.

Yahoo! Answer. This graph is obtained from a sample of ≈ 160 million answers to ≈ 25 millions questions in Yahoo! Answer [3]. From this dataset, we derived an undirected graph where nodes represent users and an edge connects two users at time $max(t_1, t_2)$ if they both answered the same question at times t_1, t_2 respectively. We removed 6 outliers questions with more than > 5000 answers.

Experimental setup. All our experiments have been run using in-house developed C++ code compiled with `g++` and `-O4` optimization option. We used a machine equipped with Intel(R) Xeon(R) E7-4870 CPUs with 2.40GHz clock and approximately 50GB of main memory. Each run employs a single core of the machine while using less than 10% of the available main memory.

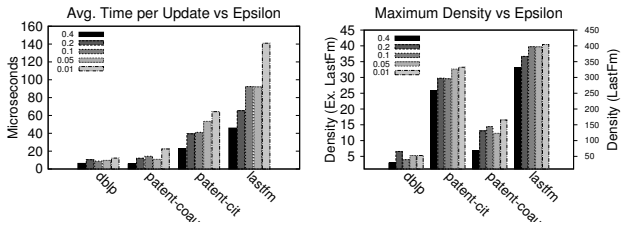
5.1 Edge insertion only

In this subsection, we show our results for the case with edge additions only. First, we show the evolution over time of the properties of the densest subgraph for some of our datasets, when edges are introduced according to their timestamps. Then, we study the tradeoff between running time and accuracy of our algorithm as a function of our accuracy parameter ϵ. Finally, we consider the natural variant of state-of-the-art approaches for static densest subgraph computation, where a densest subgraph is recomputed every time k update operations are performed. Our experimental evaluation shows that our dynamic algorithm is up to 3 orders of magnitude faster than such a natural variant.

Evolution of the densest subgraph.

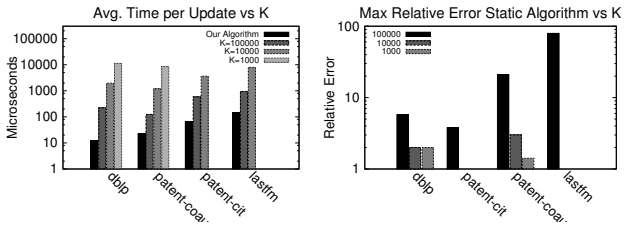
In our first experiment, edges are first sorted according to their timestamps in a non-decreasing order. Then, starting from an empty graph they are added one at a time[2] while our algorithm maintains an approximate densest subgraph

[2]Notice that in our experiments we ignore trivial update operations (e.g. inserting an edge already present).



(a) ϵ vs Update Time (b) ϵ vs Density

Figure 2: Trade-offs between the average updated time in microseconds and the density of the densest subgraph found— edge addition only. Notice how in all the graph the average update times are in the hundreds of microseconds at most. Smaller the ϵ values results in higher update time and densities.



(a) Update Time (b) Relative Error

Figure 3: Comparison between our approach and recomputing the densest subgraph every K updates with a static graph algorithm. Notice that both plots are in log scale. Our algorithm outperforms static graph algorithms by orders of magnitude even when allowed to run very infrequently and output up to 80 times more dense clusters.

of the current graph. Figure 1, illustrates the evolution of the density as well as the size of the densest subgraph found by our algorithm over time. For the DBLP and Patent graph the time is in years, while for LastFm is in days.

For this experiment we used parameter $\epsilon = 0.01$ which provides a ≈ 2.04-approximation. Notice how the density of the densest subgraph grows in a stepwise fashion, which is expected as the algorithm does not recompute the densest subgraph unless a sufficiently larger one emerges. The largest densest subgraphs found in our datasets range in size between ≈ 60 and ≈ 4000 nodes, while having density between ≈ 5 and ≈ 400. In particular we observe very dense subgraphs for our LastFm dataset where node represents songs.

Efficiency accuracy trade-offs. We now turn our attention to the tradeoff between running time and density of the densest subgraph as a function of our accuracy parameter ϵ. Figure 2(a) shows average time per update in microseconds for our algorithm. We can observe that on average a few hundreds of microseconds are sufficient for handling any update. This implies that our algorithm can handle between ≈ 7000 and ≈ 80000 edge update operations per second with very small error ($\epsilon = 0.01$). These results highlight that our algorithm can be employed effectively in a very dynamic context. Figure 2(b) illustrates how the density of the densest subgraph is affected by our accuracy parameter ϵ. Notice that, as expected, smaller values of ϵ results in denser subgraphs. However, our algorithm seems to find very dense subgraphs even for relatively large values of ϵ suggesting that our algorithm finds denser subgraphs than those predicted by our worst case analysis.

Comparison with static algorithms. To the best of our knowledge, no approximation algorithm has been designed to efficiently maintain a densest subgraph in a dynamic graph, while requiring time $o(|V| + |E|)$ per update (i.e. without recomputing the solution from scratch every time). Therefore, we study a natural variant of the algorithm proposed by Bahmani et al. [10]), where the densest subgraph is recomputed every time K update operations are performed. As it is not clear how to fix K a priori we consider several different values for K. Notice that, even though the algorithm presented by Bahmani et al. [10] provides a $2(1 + \epsilon)$ approximation, recomputing the densest subgraph every K update operations does not give any meaningful approximation guarantee. This follows from the fact that densest subgraphs might emerge very quickly, before the densest subgraph is recomputed. Indeed, this is what we observe in our experimental evaluation.

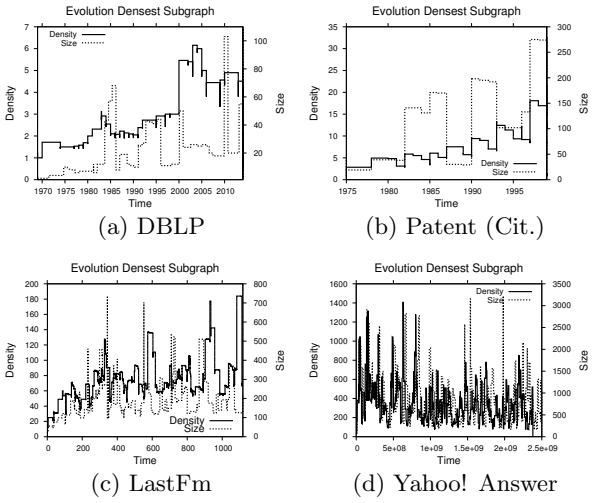
Figure 3(a) shows the comparison of the running time of our algorithm vs that of Bahmani et al. [10]. Both algorithms are set to provide an approximation guarantees of $2(1 + \epsilon)^2$ with $\epsilon = 0.01$. It is possible to see that our algorithm is up to 3 orders of magnitude faster than the one that recomputes the densest subgraph periodically, even when this is done every 100000 updates. Times in Figure 3(a) are in microseconds per update. For the dataset LastFm and Patent (Cit.) with $K = 10000$ we stopped the computation of the algorithm after few hours so the result is a lower-bound of the total time required.

We remark that an average update time in the order of tens of milliseconds is far from being practical. As an illustrative example, if each update requires 10 milliseconds, the analysis of datasets with one billion updates would require at least 100 days. Hence, it is crucial to design algorithms with an update time in the order of tens to hundreds of microseconds or less to scale to such datasets.

Moreover, recomputing the subgraph periodically may also lead to poor results, because, dense subgraphs that emerge and disappear quickly might be missed. This is illustrated in Figure 3(b), showing that the naive approach might find subgraphs that are up to ≈ 80 times less dense than the one found by our algorithm (essentially providing no guarantees in terms of accuracy). In comparison our algorithm, is guaranteed to find a subgraph that is always within 2.04 factor from the optimum. In Figure 3(b), we report only the results for the runs that completed in the allotted time.

5.2 Fully dynamic case

In this section, we evaluate our fully dynamic algorithm in datasets containing both edges additions and removals. To show the viability and effectiveness of our approach, we evaluate our algorithm in the *sliding window model*, where a sliding window of the most recent edges (say in the last hour or day) defines the current graph. Over time new recent edges might be added to the current graph, while old ones are being removed. This might results in high rate update operations. The dynamic graphs in this section are obtained in the following way. For DBLP, and Patent (Co-Aut.) & (Cit.) we keep in the sliding window the edges generated



(a) DBLP (b) Patent (Cit.)

(c) LastFm (d) Yahoo! Answer

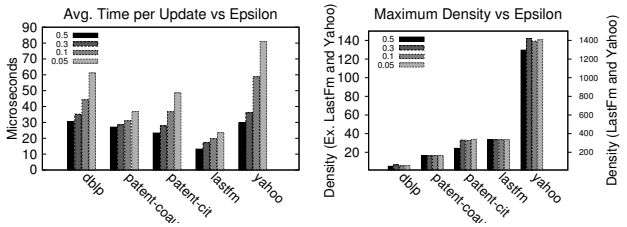
Figure 4: Evolution of the densest subgraph — fully dynamic case.

in the last 5 years, while for LastFm we keep the edges generated in the last 30 days. For our larger graph Yahoo! Answer, instead we keep in the sliding window the last 10 millions edges. In case of a tie the corresponding edges are sorted randomly. We experimented with different sizes for the sliding windows, leading to very similar results (which are omitted for space issues).

Evolution of the densest subgraph. Figure 4 shows the evolution of the densest subgraph in the sliding window model. Notice that in the case of LastFm and Yahoo! Answer both size and density varies very quickly, while for the other graphs we observe less pronounced variations. This is expected as, for instance in Yahoo! Answer, the appearance of new popular topic can trigger the rapid emergence of very large (albeit shortly lived) dense communities of users. In the context of co-authorships graphs, instead, we clearly expect slower variations in the community structure (introducing an edge requires the authorship of a research paper). In our co-authorhsip graphs (DBLP and Patent) we observe a clear trend of densification of the communities over time, which is consistent with previous observations [30]. However, notice that in our work we are able to study for the first time the evolution of the density of the densest subgraphs (as opposed to the density of the whole graph), which is possible thanks to the ability of our algorithm to process efficiently very large dynamic graphs. This pattern is less clear in LastFm and Yahoo! Answer but our analysis of these graphs is limited to significantly smaller periods of times, with other transient phenomena possibly prevailing.

These results highlight that our algorithm can be a valuable tool in studying the evolution of communities in a social network over time.

Efficiency accuracy trade-offs. Figure 5(a) shows the average time per update in microseconds of our algorithm as a function on the parameter ϵ for some our datasets. We can consistently observe that a densest subgraph can be maintained within hundreds of microseconds per update on average. This is the case also for our larger datasets requiring billions of updates, allowing the computation of densest subgraphs in evolving graphs at these scales.



(a) ϵ vs Update Time (b) ϵ vs Density

Figure 5: Trade-offs between the average updated time in microseconds and the density of the densest subgraph found— fully dynamic case.

Figure 5(b), instead, shows how the density of the densest subgraph found is affected by our accuracy parameter ϵ. Similar considerations of the insertion only graph section holds in this case.

By comparing Figure 4 and Figure 5(a), we observe higher update times in the most highly dynamic datasets, such as LastFm and Yahoo!, Answer) as opposed to DBLP, Patent. This is expected as frequent changes in the density of the densest subgraph requires the frequent executions of the costly *FindDensest* procedure. Notice, however, that our algorithm scales very well in very large and highly dynamic datasets like Yahoo! Answer.

6. CONCLUSIONS AND FUTURE WORKS

We studied the problem of maintaining an approximate densest subgraph problem in dynamic graphs. We give the first algorithm that maintains a constant factor approximation when edges are added adversarially and removed randomly, while ensuring amortized cost being poly-log per update on average. Our extensive experimental evaluation shows that our algorithm can cope with more than one billion of update operations within a few hundreds of microseconds per update, on average.

An interesting direction for future work is to adapt into a dynamic environment the algorithm developed in [11] for finding several dense subgraphs with limited overlap. It would also be interesting to improve the approximation factor of our algorithm by using a technique similar to the one presented in [8], as well as, to design algorithms that work under adversarial edge deletion.

Acknowledgments.

We thank Flavio Chierichetti, Ravi Kumar and Alessandro Panconesi for their many insightful comments and suggestions.

7. REFERENCES

[1] DBLP dataset (October 2014). `http://dblp.uni-trier.de/xml/`.

[2] Last.fm song network dataset - KONECT. `http://konect.uni-koblenz.de/networks/lastfm_song`.

[3] Yahoo! Answers browsing behavior version 1.0. - Yahoo! Research Webscope Datasets. `http://webscope.sandbox.yahoo.com`.

[4] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, 2013.

[5] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy. Distributed k -core view materialization and maintenance for large dynamic graphs. *IEEE Trans. Knowledge and Data Eng.*, 2014.

[6] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, and S. Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *The VLDB Journal*, 2014.

[7] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *PVLDB*, 2012.

[8] B. Bahmani, A. Goel, and K. Munagala. Efficient primal-dual algorithms for mapreduce. *Manuscript*, 2012.

[9] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *KDD*, 2012.

[10] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 2012.

[11] O. D. Balalau, F. Bonchi, T.-H. Chan, F. Gullo, and M. Sozio. Finding subgraphs with maximum total density and limited overlap. In *WSDM*, 2015.

[12] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. E. Tsourakakis. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *STOC*, (to appear), 2015.

[13] M. Brunato, H. H. Hoos, and R. Battiti. On effectively finding maximal quasi-cliques in graphs. In *Learning and Intelligent Optimization*. 2008.

[14] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, pages 95–106. ACM, 2008.

[15] Ò. Celma Herrada. Music recommendation and discovery in the long tail. Technical report, 2009.

[16] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Approximation Algorithms for Combinatorial Optimization*. 2000.

[17] J. Chen and Y. Saad. Dense subgraph extraction with application to community detection. *IEEE Trans. on Knowledge and Data Eng.*, 2012.

[18] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 2003.

[19] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *WWW*, 2007.

[20] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLBD*, 2005.

[21] A. V. Goldberg. *Finding a maximum density subgraph.* UC Berkeley, 1984.

[22] B. H. Hall, A. B. Jaffe, and M. Trajtenberg. The NBER patent citation data file: Lessons, insights and methodological tools. Technical report, National Bureau of Economic Research, 2001.

[23] G. F. Italiano, D. Eppstein, and Z. Galil. Dynamic graph algorithms. *Algorithms and Theory of Computation Handbook*, 1999.

[24] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.

[25] S. Khuller and B. Saha. On finding dense subgraphs. In *Automata, Languages and Programming*. 2009.

[26] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[27] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. *Computer networks*, 1999.

[28] P. Lee, L. V. Lakshmanan, and E. Milios. CAST: A Context-Aware Story-Teller for Streaming Social Content. In *CIKM*, 2014.

[29] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*. Springer, 2010.

[30] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. KDD*, 2007.

[31] R.-H. Li, J. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowledge and Data Engineering*, 2014.

[32] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. Finding strongly knit clusters in social networks. *Internet Mathematics*, 2008.

[33] D. Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 2000.

[34] B. Saha, A. Hoch, S. Khuller, L. Raschid, and X. Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *RECOMB*, 2010.

[35] B. Saha, A. Hoch, S. Khuller, L. Raschid, and X.-N. Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Research in Computational Molecular Biology*, 2010.

[36] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 2013.

[37] A. D. Sarma, A. Lall, D. Nanongkai, and A. Trehan. Dense subgraphs on dynamic networks. In *Distributed Computing*. 2012.

[38] S. B. Seidman. Network structure and minimum degree. *Social networks*, 1983.

[39] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD, 2010*, pages 939–948, 2010.

[40] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In *KDD*, 2013.

[41] E. Valari, M. Kontaki, and A. N. Papadopoulos. Discovery of top-k dense subgraphs in dynamic graph collections. In *Scientific and Statistical Database Management*, 2012.