4-2014

Dense subgraph maintenance under streaming edge weight updates for real-time story identification

Google, Inc.

Nick Koudas
University of Toronto

Nikos Sarkas
University of Toronto

AT&T Labs-Research

Michael Svendsen
Iowa State University

See next page for additional authors

Dense subgraph maintenance under streaming edge weight updates for real-time story identification

Abstract

Recent years have witnessed an unprecedented proliferation of social media. People around the globe author, everyday, millions of blog posts, social network status updates, etc. This rich stream of information can be used to identify, on an ongoing basis, emerging stories, and events that capture popular attention. Stories can be identified via groups of tightly coupled real-world entities, namely the people, locations, products, etc, that are involved in the story. The sheer scale and rapid evolution of the data involved necessitate highly efficient techniques for identifying important stories at every point of time. The main challenge in real-time story identification is the maintenance of dense subgraphs (corresponding to groups of tightly coupled entities) under streaming edge weight updates (resulting from a stream of user-generated content). This is the first work to study the efficient maintenance of dense subgraphs under such streaming edge weight updates. For a wide range of definitions of density, we derive theoretical results regarding the magnitude of change that a single edge weight update can cause. Based on these, we propose a novel algorithm, DynDens, which outperforms adaptations of existing techniques to this setting and yields meaningful, intuitive results. Our approach is validated by a thorough experimental evaluation on large-scale real and synthetic datasets.

Keywords

Dense subgraphs, Story identification, Entity graph

Disciplines

Electrical and Computer Engineering | Systems and Communications

Comments

This is a manuscript of an article published as Angel, Albert, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. "Dense subgraph maintenance under streaming edge weight updates for real-time story identification." *The VLDB Journal* 23, no. 2 (2014): 175-199. The final publication is available at Springer via DOI: 10.1007/s00778-013-0340-z. Posted with permission.

Authors

Google, Inc.; Nick Koudas; Nikos Sarkas; AT&T Labs-Research; Michael Svendsen; and Srikanta Tirthapura

Noname manuscript No.
(will be inserted by the editor)

Dense Subgraph Maintenance under Streaming Edge Weight Updates for Real-time Story Identification

Michael Svendsen · Albert Angel* · Nick Koudas · Nikos Sarkas · Divesh Srivastava · Srikanta Tirthapura

the date of receipt and acceptance should be inserted later

Abstract Recent years have witnessed an unprecedented proliferation of social media. People around the globe author, every day, millions of blog posts, social network status updates, etc. This rich stream of information can be used to identify, on an ongoing basis, emerging stories, and events that capture popular attention. Stories can be identified via groups of tightly-coupled real-world entities, namely the people, locations, products, etc, that are involved in the story. The sheer scale, and rapid evolution of the data involved necessitate highly efficient techniques for identifying important stories at every point of time.

The main challenge in real-time story identification is the maintenance of dense subgraphs (corresponding to groups of tightly-coupled entities) under streaming edge weight updates (resulting from a stream of user-generated content). This is the first work to study the efficient maintenance of dense subgraphs under such streaming edge weight updates. For a wide range of definitions of density, we derive theoretical results regarding the magnitude of change that a single edge weight update can cause. Based on these, we propose a novel algorithm, DynDens, which outperforms adaptations of existing techniques to this setting, and yields meaningful, intuitive results. Our approach is validated by a thorough experimental evaluation on large-scale real and synthetic datasets.

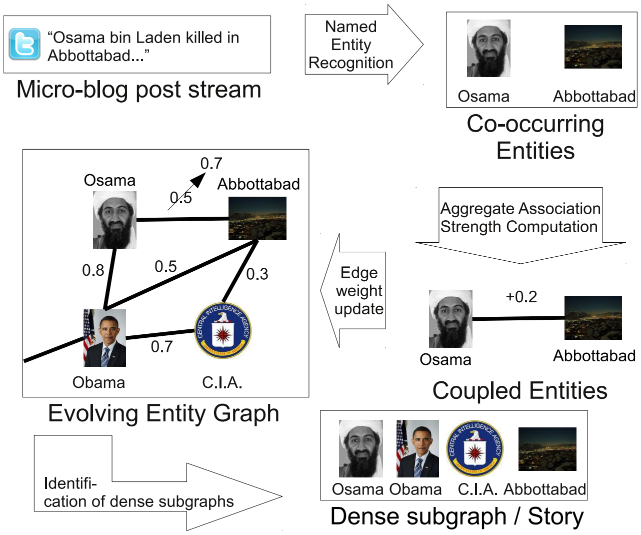


Fig. 1 Real-time identification of "bin Laden raid" story, and connection to Engagement

1 Introduction

Recent years have witnessed an unprecedented proliferation of social media. Millions of people around the

* Work in [4], and related technical report, done while at University of Toronto

Michael Svendsen
Iowa State University
E-mail: svendsen@iastate.edu

Albert Angel
Google Inc.
E-mail: albertangel@google.com

Nick Koudas
University of Toronto
E-mail: koudas@cs.toronto.edu

Nikos Sarkas
University of Toronto
E-mail: nsarkas@cs.toronto.edu

Divesh Srivastava
AT&T Labs-Research
E-mail: divesh@research.att.com

Srikanta Tirthapura
Iowa State University
E-mail: snt@iastate.edu

globe author on a daily basis millions of blog posts, micro-blog posts and social network status updates. This content offers an invaluable uncensored window into current events, and emerging stories capturing popular attention.

For instance, consider the U.S. military strike in Abbottabad, Pakistan in early May 2011, which resulted in the death of Osama bin Laden. This event was extensively covered on Twitter, the popular micro-blogging service, significantly in advance of traditional media, starting with the live coverage of the operation by an (unwitting) local witness, to millions of tweets around the world providing a multifaceted commentary on every aspect of the story. Similar, if fewer, online discussions cover important events on an everyday basis, from politics and sports, to the economy and culture (notable examples from recent years range from the concerts of Justin Bieber and the death of Michael Jackson, to revolutions in the Middle East and the economic recession). In all cases, stories have a strong temporal component, making timeliness a prime concern in their identification.

Interestingly, such stories can be identified by leveraging the real-world entities involved in them (e.g. people, politicians, products and locations) [27]. The key observation is that each post on the story will tend to mention the same set of entities, around which the story is centered. In particular, as post length restrictions or conventions typically limit the number of entities mentioned in a single post, each post will tend to mention entities corresponding to a single facet of a story. Thus, by identifying pairs of entities that are strongly associated (recurrently mentioned together), one can implicitly detect facets of the underlying event of which they are the main actors. By piecing together these aspects, the overall event of interest can be deduced.

For example, in the case of the U.S. military strike mentioned above, one facet, consisting of people discussing the raid, is centered around "Abbottabad" where the raid took place, and the involvement of the "C.I.A."; another thread, drawing connections to bin Laden is centered on "Abbottabad" and "Osama bin Laden"; a third, commenting on the presidential announcement, involves "Barack Obama" and "Osama bin Laden"; and so on. The resulting overall story at some point of time involves the union of these entities. Such sets of entities can be then used by systems such as Grapevine [3] to enable the interactive exploration of the story by users of the system.

Given a measure to quantify the strength of association between two entities (such as the Log-likelihood ratio [27], the χ^2 measure, or the correlation-coefficient [5], etc.), one can abstract the real-time stream of posts giving rise to an evolving (weighted) entity graph, denoting the pairwise entity association strength[1]. An important story can then be identified via a cohesive group of strongly associated entity pairs; i.e. a dense subgraph in the entity graph, given an appropriate definition of density. Moreover, note that, as the entities in a story need to be presented to users to facilitate navigation, story cardinality needs to be constrained to moderate sizes; after all, it would not be very interesting or helpful to present users with a story centered around 100 main entities. This process is illustrated in Figure 1.

Every post that is published, results in the weight update of one or more edges in the entity graph. The high frequency of post generation, coupled with our need for timely reporting of emerging stories, necessitates that the identification of dense structures in the entity graph be highly efficient. This work thus addresses the problem of dENse subGrAph maintenance for edGE-weight update streaMs under sizE constraiNTs, or Engagement for brevity. Besides being useful as-is for identifying stories from social media in real-time, solutions to this problem can also be used as building blocks for more complex computations; e.g. identified dense subgraphs can undergo diversification before being presented to the user [2], or they can be re-ranked taking their external sparsity into account, in order to identify (soft) clusters of associated entities.

Addressing Engagement at web scales presents several challenges. Principal among these is that, a change in the weight of a single edge, can impact the density of many subgraphs, necessitating a potentially unbounded exploration of the entity graph. Thus, any efficient solution to Engagement needs to incrementally maintain dense subgraphs at every point of time, without recomputing them from scratch. Moreover, there does not exist a single definition of graph density suitable for all scenarios; selecting the most appropriate definition for a given setting depends, for instance, on the perceived relative importance of having large, versus well-connected, dense subgraphs. Consequently, solutions to Engagement need to be applicable under very general notions of density; however, existing techniques are only applicable to very limited subsets of this problem.

In this context, in this work we propose DynDens, an efficient algorithm for Engagement. We theoretically quantify the magnitude of change in dense subgraphs that a single edge weight update can cause. Based on this, we show how maintaining some sparse

[1] Note that our techniques are equally applicable irrespective of the association measure, so we will not be focusing on any one measure in particular; in Section 5 we review two measures used in our evaluation, and discuss how arbitrary association measures can be efficiently used in our setting.

subgraphs, in addition to dense ones, enables the incremental maintenance of dense subgraphs. The resulting algorithm, DynDens, makes use of an efficient index for subgraphs, which decreases memory consumption and processing effort. It is complemented by theoretically sound heuristics, that can offer improved performance. A comprehensive experimental evaluation on real and synthetic data highlights the effectiveness of our approach.

To summarize, our main contributions in this work are:

– Motivated by the need to identify emerging stories in real-time, for a wide range of measures of entity association, we formalize the problem of dENse subGrAph maintenance for edGE-weight update streaMs under sizE constraiNTs (Engagement), for a very broad notion of graph density.
– We propose an efficient algorithm DynDens, based on a novel quantification of the maximum possible change effected by a single edge weight update. By maintaining a small number of sparse subgraphs, DynDens is able to efficiently and incrementally compute dense subgraphs.
– We design an efficient dense subgraph index, which decreases memory consumption and processing effort, and propose theoretically sound heuristics for DynDens that can offer improved performance.
– We describe an update procedure to facilitate incremental changes to DynDens parameters, allowing for parameter exploration during execution without the need to perform a full recomputation of the index.
– We validate our techniques via a thorough experimental evaluation on both real and synthetic datasets.

The remainder of this paper is organized as follows: After providing a formal problem statement in Section 2, we present our proposed algorithm DynDens in Section 3. We explore the theoretical basis for DynDens in Section 4, evaluate the proposed techniques in Section 5. In Section 6 we describe a method for dynamic threshold adjustment. We discuss some performance improvements to DynDens in Section 7. Finally, we review related work in Section 8, and conclude in Section 9.

2 Formalization

Let us now turn to defining Engagement. At a high level, let us consider a weighted graph, with a constant number of vertices. At every discrete time interval, the weights of one or more edges are adjusted (including potentially edge additions and removals). The goal is to maintain, at each point of time, all subgraphs with "density" greater than a given threshold.

Connections to real-time story identification: Before fully formalizing the problem, let us first draw some connections to its application in real-time story identification. In this context, vertices correspond to real-world entities, and edge weights to their (current) pairwise association strengths (the choice of association strength measure will depend on characteristics of the specific problem instance; in Section 5 we discuss several such choices). We assume that a procedure exists for processing streams of (entity-annotated) posts, and generating the appropriate edge weight updates at each time interval (in Section 5 we discuss such procedures for a variety of measures of interest).

Data model: We represent the problem domain as i) a complete weighted graph $G = (V, E)$ with N vertices, where w_{ij} is the weight of edge between nodes i and j; and ii) a stream of edge weight updates of the form $update_i = (a, b, \delta)$, signifying that at time instant i, the weight of the edge between vertices a and b changed from w_{ab} to $w_{ab} + \delta$.

Density: We define subgraph density as follows: for every subgraph $C \subseteq V$, its density is $dens(C) = \frac{score(C)}{S_{|C|}}$, where $score(C) = \sum_{i,j \in C \wedge i<j}(w_{ij})$. S_n is a function quantifying the relative importance of a subgraph's cardinality to its density; with the appropriate choice of S_n, virtually all quantifications of graph density can be represented.

Note that we do not consider counter-intuitive quantifications of graph density, such as (but not limited to) a definition of density where the removal of a vertex from an unweighted clique results in an increase of its density. To safeguard against such quantifications of density, we require that S_n have the following intuitive monotonicity properties: $\frac{n}{n-1} \leq \frac{S_n}{S_{n-1}} \leq \frac{n}{n-2}$. This encompasses the full spectrum of choices of density functions commonly used in the literature; typical choices include $S_n = \frac{n \cdot (n-1)}{2}$ (thus density is defined as the average edge weight, favoring small, dense subgraphs; we term this instantiation AvgWeight), and $S_n = n$ (thus density represents a generalized average node "degree", favoring large subgraphs; we term this case AvgDegree). Another commonly used measure, that lies in between the previous two, is $S_n = \sqrt{n}(n-1)$; we term this case SqrtDens.

Cardinality constraint: Finally, let N_{max} be a (user-specified) maximum cardinality for subgraphs of interest. (In the context of real-time story identification, this constraint ensures that any subgraphs identified are small enough to be used for navigation / exploration purposes - cf. Section 1).

ENGAGEMENT: Given the above, the goal of ENGAGEMENT is to maintain, at every point of time i, the subgraphs (vertex subsets) with density over a given threshold T, subject to cardinality constraints, i.e. $\{V_j | V_j \subseteq V \wedge dens(V_j) \geq T \wedge |V_j| \leq N_{max}\}$. We term these output-dense subgraphs.

Notation: Before going into the details of our proposed approach, let us introduce some notation that is used throughout this work.

We denote each vertex by a natural number, so $V = \{1, \cdots, N\}$ denotes the set of vertices in G. Let $\hat{e}_i$ be the i'th basis vector (an N-dimensional vector, with value 1 in its i'th coordinate, and 0 elsewhere). We will denote a subset $C \subseteq V$ by its corresponding vector $\mathbf{c} = \sum_{i \in C} \hat{e}_i$, and will sometimes refer to either interchangeably; we will also on occasion denote the cardinality of subset C as $|\mathbf{c}|$.

Let $\mathbf{\Gamma}_u$ be the neighborhood vector of vertex u: $\mathbf{\Gamma}_u = (w_{1u}, w_{2u}, \cdots, w_{Nu})$. For convenience, we will also make use of the following normalized version of S_n: Let $g_n = \frac{S_n}{n \cdot (n-1)}$. By the monotonicity properties of S_n, it follows that $g_n \leq g_{n-1}$.

Unless explicitly stated, we will focus on the time instant where the weight of the edge between vertices a and b is updated from $w_{ab} = w$ to $w + \delta$. Whenever a quantity X can be affected by this update, we will denote its value before the update as X^- and its value after the update as X^+. We omit this superscript when it does not affect results in any way. For example, $w_{ab}{}^- = w$, $w_{ab}{}^+ = w + \delta$.

3 The DYNDENS approach

Let us now discuss how our proposed algorithm, DYNDENS, identifies, at every point of time, all output-dense subgraphs.

Dense subgraphs and growth property: Observe that there is an inherent tradeoff in the set of subgraphs that DYNDENS will maintain, which we term "dense" subgraphs. At one extreme, DYNDENS could opt to maintain only output-dense subgraphs, with the other extreme being to maintain *all* subgraphs. However, neither of these is desirable: the former because it does not enable incremental computation of output-dense subgraphs, the latter due to its prohibitive costs. We will subsequently (Section 4.1.3) formally quantify this tradeoff. For now, loosely speaking, we will say that C is a dense subgraph iff it has density greater than a given threshold $T_{|C|}$ (which is a function of the cardinality of C), and cardinality of at most N_{max} (for a complete list of density-related terms used in this work cf. Table 1). T_n is defined in a manner that ensures that every dense graph with n vertices has at least one dense subgraph with $n-1$ vertices (thus it is possible to identify all dense subgraphs by "growing" dense subgraphs of smaller cardinalities).

Table 1 Definitions of density-related properties

Subgraph C is $\cdots$	iff
Static properties	
dense	$dens(C) \geq T_{\lvert C\rvert}$
sparse	$dens(C) < T_{\lvert C\rvert}$
output-dense	$dens(C) \geq T$
too-dense	$dens(C) \geq T_{\lvert C\rvert+1}$
Dynamic properties	
stable-dense	$dens(C)^- \geq T_{\lvert C\rvert} \wedge dens(C)^+ \geq T_{\lvert C\rvert}$
newly-dense	$dens(C)^- < T_{\lvert C\rvert} \wedge dens(C)^+ \geq T_{\lvert C\rvert}$
losing-dense	$dens(C)^- \geq T_{\lvert C\rvert} \wedge dens(C)^+ < T_{\lvert C\rvert}$

Specifically, T_n is a monotonically increasing function of n with the property $T_n \cdot g_n > T_{n-1} \cdot g_{n-1}$. At a high level, this monotonicity property ensures the desired containment property mentioned earlier (see Section 4.1 for details[2]). Moreover, we require that $T_{N_{max}} = T$[3]. We discuss the concrete instantiation of T_n used by DYNDENS in Section 4.1.3.

Edge weight updates: The basic operation of DYNDENS is to maintain dense subgraphs, following the update of the weight of an edge (a, b), from w to $w + \delta$. If this impacts the set of output-dense subgraphs, the latter is updated as well. Handling updates with $\delta < 0$ (i.e. where the weight of an edge decreases) is straightforward: all dense subgraphs containing both a and b are examined, and their density is decreased by an appropriate amount. If they are no longer output-dense, this is reported; if, in addition, they are no longer dense (losing-dense), they are evicted from the index.

Positive updates: Of greater interest is the case where $\delta > 0$, i.e. the edge weight update corresponds to an increase in weight. In this case, additional subgraphs, that were not dense prior to the update, might now be dense (newly-dense subgraphs). DYNDENS leverages the growth property to compute these as follows:

[2] Another way to view dense graphs is the following: Consider the measure $normDens(C) = \frac{dens(C)}{T_{|C|}}$, consisting of a density measure, normalized by the threshold function T_n; a graph C is dense iff it has $normDens(C) \geq 1$. While $normDens(C)$ is not a suitable measure of density per se, it has the following important growth property: every graph C has a subgraph C' of cardinality $|C'| = |C| - 1$ with $normDens(C') \geq normDens(C)$. This containment/growth property additionally implies that, if there are no dense subgraphs of cardinality n, there can be no dense subgraphs of any cardinality $> n$.

[3] Recall that T_n is an increasing function of n, and the set of maintained subgraphs needs to include all output-dense subgraphs of cardinality $\leq N_{max}$ having density $\geq T$.

- **Cheap explore**: DynDens will try to augment all dense subgraphs containing either a or b, with b or a, respectively; resulting newly-dense subgraphs will be inserted into the dense subgraph index. In some cases, this step alone is sufficient and/or can be applied only to a subset of these subgraphs (cf. Section 7.1) for details).
- **Explore**: DynDens will try to augment dense subgraphs containing both a and b, with one neighboring vertex; resulting newly-dense subgraphs will be inserted into the dense subgraph index.
- **Exploration iterations**: The above procedure may need to be performed iteratively for newly-dense subgraphs discovered via cheap exploration or exploration. Interestingly, the iteration depth is upper bounded by a corollary of the growth property. Specifically, in Section 4.1.3, we define T_n parametrized by a parameter δ_{it} that indirectly controls the number of dense subgraphs maintained by DynDens. As we show in Section 4.1, we can guarantee that at most $\lceil \frac{\delta}{\delta_{it}} \rceil$ iterative exploration iterations need to be performed, in order to identify all newly-dense subgraphs, following an edge weight update of magnitude δ.
- **Explore All**: In a few cases, the above exploration may need to be performed on non-neighboring nodes as well, resulting in a very costly procedure. In most cases, DynDens avoids performing this procedure via a better, implicit representation of some dense subgraphs in the index (cf. Section 3.2.3).

In one sentence, DynDens explores the neighborhood of some materialized dense subgraphs, using pruning conditions for when to stop exploring around a subgraph. The remainder of this section aims to fill in the blanks in the preceding sentence. We discuss the workings of DynDens, and illustrate them with a practical example in Section 3.1, followed by important technical details in Section 3.2. We defer the exposition of the theoretical results on which DynDens is based till Section 4.

3.1 The DynDens algorithm

Algorithm 1 Algorithm DynDens

```
Input: Updated edge $(a, b)$, magnitude of update $\delta$
1: if $\delta < 0$ then
2:     Update the density of all dense subgraphs containing $a$ and $b$; evictlosing-dense subgraphs from the index; report any subgraphs that are no longeroutput-dense
3:     return
4: for all dense subgraphs $C$ st. $a \in C \vee b \in C$ do {// including $C = \{a, b\}$ if it is newly-dense}
5:     if $a \notin C$ or $b \notin C$ then
6:         if $C$ should be cheap-explored and $C \cup \{a, b\}$ is newly-dense then
7:             Add $C \cup \{a, b\}$ to the index, report it if it is output-dense
8:             explore($C \cup \{a, b\}, 2$)
9:     else
10:        Update the density of $C$, report it if it just became output-dense
11:        explore($C, 1$)
```

Let us now discuss DynDens in greater detail, with reference to Algorithm 1. At a high level, DynDens maintains an in-memory index of all dense subgraphs (we defer discussing index implementation details to Section 3.2); at every edge weight update, it outputs information regarding subgraphs that became, or stopped being output-dense. If the edge weight update was negative, only some index maintenance needs to be done (line 2). Otherwise, some stable-dense subgraphs containing a and/or b are examined (line 4)[4]. Note that, to ensure correctness, also the subgraph $\{a, b\}$ may be examined, even if it was not present in the index (base case in line 4). Subgraphs in the index containing only one of a, b are cheap-explored, if needed[5] (line 6).

Subgraphs in the index that contain both a and b, as well as newly-dense subgraphs previously identified, are subsequently explored (line 11) - i.e. DynDens will try to augment them with a neighboring node (we defer discussing the precise details on how this is done efficiently to Section 3.2). This will be recursively repeated on any newly-dense subgraphs discovered up to $\lceil \frac{\delta}{\delta_{it}} \rceil$ times (the theoretical results that enable this bounding are discussed in Section 4.1). A high-level description of the exploration procedure is shown in Algorithm 2.

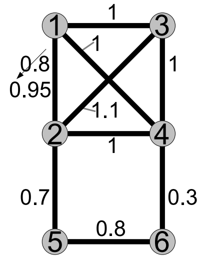
Algorithm 2 will first ensure that the subgraph should be explored. Specifically, the subgraph should not have been too-dense before the update (line 1), for otherwise its dense supergraphs would have been stable-dense, and hence already identified. Moreover, as previously mentioned, DynDens will not explore around any subgraph more times than necessary. Finally, in a few cases, explored subgraphs will need to be augmented with every other vertex, not just neighboring ones (Explore-All; line 2). As the latter is a costly procedure, in Section 3.2.3 we will present a way to mitigate the associated cost.

[4] The index provides functionality to ensure that no subgraph is examined more than once.

[5] For instance, subgraphs that were too-dense need not be explored, as, by definition, their dense supergraphs would have been stable-dense, and hence already identified. Moreover, this step can also be skipped in other circumstances, cf. Section 7.1 for details.

Algorithm 2 Procedure **explore**(C, i)

```
Input: Subgraph $C$. Iteration number $i$
1: if $C$ was not too-dense before the update and $i \leq \lceil \frac{\delta}{\delta_{it}} \rceil$ and $|C| < N_{max}$ then
2:     if $C$ is too-dense then
3:         for all $y \notin C$ do {// Explore-All}
4:             Add $C \cup \{y\}$ to the index; report it if it is output-dense
5:             explore($C \cup \{y\}, i + 1$)
6:     else
7:         for all neighbors $y$ of $C$ do
8:             if $C \cup \{y\}$ is newly-dense then
9:                 Add $C \cup \{y\}$ to the index; report it if it is output-dense
10:                explore($C \cup \{y\}, i + 1$)
```



(a) Entity graph

Subgraph	Density	output-dense?
Dense, before update		
1, 3	1.0	Y
1, 4	1.0	Y
2, 3	1.1	Y
2, 4	1.0	Y
3, 4	1.0	Y
1, 3, 4	1.0	Y
2, 3, 4	$1.0\overline{3}$	Y
newly-dense, after update		
1, 2	0.95	N
1, 2, 3	$1.01\overline{6}$	Y
1, 2, 4	$0.98\overline{3}$	N
1, 2, 3, 4	$1.008\overline{3}$	Y

(b) Dense subgraph index

Fig. 2 Execution example

*Execution Example.*To illustrate the workings of DynDens, let us examine a simple example of its execution. Consider the sample entity graph of Figure 2(a), and assume an AvgWeight definition of density (i.e. the density of a subgraph is its average edge weight), a density threshold of $T = 1$, and a maximum desired subgraph cardinality of $N_{max} = 4$. Assume that δ_{it} has been set to 0.15, so that the thresholds T_n, for subgraphs of cardinality n to be considered dense are $T_2 = 0.9, T_3 = 0.975$ and $T_4 = T = 1$ (cf. Section 4.1.3 for details). Thus, the dense subgraphs for this graph are shown in Figure 2(b) (output-dense subgraphs are emphasized). Finally, assume that the weight of edge $(1, 2)$ is updated from 0.8 to 0.95 ($\delta = \delta_{it} = 0.15$). Let us examine how DynDens will handle this update; to facilitate this discourse, the newly-dense subgraphs that are inserted into the index are shown in the bottom half of Figure 2(b).

At a high level, DynDens will examine $\{1, 2\}$, as well as all dense subgraphs containing vertex 1 and/or 2 (Algorithm 1, line 4), i.e. $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$. $\{1, 2\}$ will be added to the index (Algorithm 1, line 10), and will be explored (line 11). Its exploration will entail the addition of newly-dense subgraphs $\{1, 2, 3\}$ and $\{1, 2, 4\}$ to the index (Algorithm 2, line 8); the former will also be reported as output-dense. Since $\frac{\delta}{\delta_{it}} = 1$, these newly-dense subgraphs will not be further explored (Algorithm 2, line 10 and line 1). Moreover, during this exploration subgraph $\{1, 2, 5\}$ will be examined, but as its density is less than T_3, it will not be added to the index.

DynDens will also cheap-explore subgraphs $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$ (Algorithm 1, line 6). This will result in subgraphs $\{1, 2, 3\}$, $\{1, 2, 4\}$ being examined (twice each) (Algorithm 1, line 7); as they are already present in the index, this will not affect anything. Moreover, DynDens will attempt to explore these subgraphs (Algorithm 1, line 8); however, since $\frac{\delta}{\delta_{it}} = 1$, they will not be explored (Algorithm 2, line 1).

Finally, DynDens will cheap-explore subgraphs $\{1, 3, 4\}$ and $\{2, 3, 4\}$. The first cheap exploration will result in newly-dense subgraph $\{1, 2, 3, 4\}$ being added to the index, and reported as output-dense (Algorithm 1, line 7); the second one will revisit this subgraph, and do nothing. Moreover, in both cases, since $|\{1, 2, 3, 4\}| = 4 \geq N_{max}$, these subgraphs will not be explored (Algorithm 2, line 1).

Observation: From the simplified execution example presented above, one can observe that DynDens (as currently presented) can end up performing redundant computations; e.g. some subgraphs are examined unnecessarily many times. Subsequently, in Section 3.2.2 and Section 7.2, we discuss how to reduce such unnecessary computations.

3.2 Implementation considerations

Having presented DynDens at a high level, let us now see some important considerations that arise when implementing it in practice. We first introduce the underlying indexing structure used by DynDens in Section 3.2.1; this index also enables DynDens to avoid redundant computations (Section 3.2.2) as well as the costly operation of explore-all (Algorithm 2, line 2 cf. Section 3.2.3).

3.2.1 Index

DynDens requires an efficient index for both the evolving graph itself, as well as for dense subgraphs. For the graph index, maintaining node adjacency lists is sufficient (i.e. a mapping $\forall u \in V : u \rightarrow \mathbf{\Gamma}_u$); this

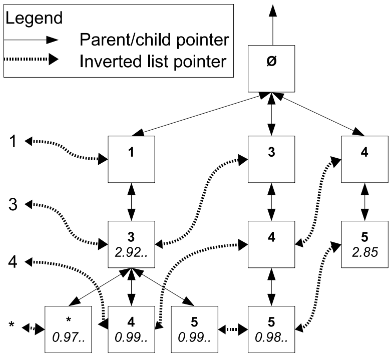


Fig. 3 Dense subgraph index

also enables the efficient exploration of a subgraph (via merging the relevant adjacency lists[6]).

The dense subgraph index is more interesting to examine, as it needs to efficiently support several functionalities. To name a few: for every dense subgraph, access to its vertices, cardinality and density; insertion, update and deletion of dense subgraphs from the index; iteration over all dense subgraphs containing vertices a or b, where each subgraph must be accessed exactly one time (needed for positive edge weight updates); and for a given dense subgraph C, and a given vertex u, access to subgraph $C \cup \{u\}$, and insertion of $C \cup \{u\}$ into the index if it is not already present (needed for exploration). Moreover, as DynDens needs to perform frequent random accesses on dense subgraphs, the index needs to be in-memory, so maintaining a low memory footprint is important. As most dense subgraphs will tend to have high overlap, the dense subgraph index should minimize the amount of redundant information stored.

To address these requirements posed by DynDens, we propose the following in-memory index. Each subgraph has a unique id corresponding to its location in memory; it is also represented by its (sorted) set of vertices. DynDens will maintain a prefix tree of dense subgraphs, illustrated in Figure 3. Each node in the prefix tree contains pointers to its children, indexed by vertex id, a pointer to its parent, as well as information (such as cardinality and density) on the dense subgraph it represents, if applicable. Figure 3 shows a view of the index when subgraphs $\{1, 3\}, \{1, 3, 4\}, \{1, 3, 5\}, \{3, 4, 5\}$, $\{4, 5\}$ are dense (ignore the node labeled $*$ for now), along with the density of each subgraph.

Additionally, to enable effective iteration over dense subgraphs containing one or two given vertices, DynDens will also maintain inverted lists, i.e. a mapping from vertices to (pointers to) all subgraphs containing a vertex. To decrease the size of inverted lists, the inverted list of a vertex u will only contain tree nodes where the lexicographically largest vertex is u. Thus, in order to iterate over all subgraphs containing u, DynDens will iterate over all subgraphs in its inverted list, and their tree descendants. Furthermore, to facilitate inverted list maintenance, inverted lists are implemented as linked lists of prefix tree nodes (shown in Figure 3 as dashed arrows)[7]. Inverted lists are updated whenever a new node is created in the tree, or when a leaf node is deleted. Moreover, if the deletion of a leaf node results in its parent having no children, and representing no dense subgraph, the parent will be recursively deleted.

Our proposed dense subgraph index efficiently addresses the requirements of DynDens. Specifically, the prefix tree enables DynDens to reduce its memory footprint, by not storing redundantly many overlapping dense subgraphs. Moreover, looking up $C \cup \{u\}$ is $O(|C|+1)$ in all cases (and $O(1)$ if vertex u is lexicographically greater than any other vertex in C); after a look-up, update or insertion into the index is $O(1)$. Enumerating the vertices in a subgraph C is $O(|C|)$, via parent pointer traversal. Deleting a subgraph C from the index is $O($ number of leaf nodes deleted); this is typically $O(1)$ and at worst $O(|C|)$, due to the design of the prefix tree with embedded inverted lists.

3.2.2 Avoiding redundant computation

Besides efficiently providing the requisite functionality for DynDens, our proposed dense subgraph index can also be used (i) to ensure that subgraphs that were dense before the update are examined exactly once (this is required for the correctness of DynDens), and (ii) to greatly reduce the number of newly-dense subgraphs that are examined more than once (without sacrificing correctness).

The former (i) can be guaranteed by fixing the order in which dense subgraphs are examined. Specifically, if subgraphs containing vertices a and/or b need to be examined, and assuming $a < b$ (lexicographically), DynDens will traverse the subtrees of all index nodes on the inverted list corresponding to b. Subsequently, it will traverse the subtrees of index nodes on the inverted list corresponding to a, stopping the traversal whenever a b node is encountered. This procedure is aided by flags that are set on a per-index node basis, to help DynDens distinguish newly-dense subgraphs in the index.

For the latter (ii), we leverage the theoretical result that all newly-dense subgraphs can be identified in at

[6] Specifically, when exploring subgraph C, DynDens will compute $\mathbf{\Gamma}_C = \sum_{v \in C} \mathbf{\Gamma}_v$; for every vertex $u \notin C$, the score of $C \cup \{u\}$ can be computed as $score(C \cup \{u\}) = score(C) + \mathbf{\Gamma}_C \cdot \hat{e}_u$.

[7] To further reduce the memory footprint of the index, circularly singly-linked lists can be used; however, using doubly-linked lists results in simpler index maintenance, esp. wrt. deletions.

most $\lceil \frac{\delta}{\delta_{it}} \rceil$ exploration iterations (Section 4). Upon insertion into the index, dense subgraphs are annotated with the exploration iteration at which they were identified (i in Algorithm 2); these annotations persist until the end of Algorithm 1. Algorithm 2 will operate as above for subgraphs not annotated with an iteration number, or annotated with an iteration number greater than the current i. Otherwise, the subgraph does not need to be further examined.

3.2.3 Implicit representation of too-dense subgraphs

Having introduced the dense subgraph index used by DynDens, let us now revisit a challenge posed by the presence of too-dense subgraphs, and show how the index can be leveraged to overcome it.

Recall that a subgraph is too-dense iff, after adding any other vertex to it, it is still dense. Thus, when exploring a too-dense subgraph, DynDens needs to consider its cartesian product with the entire set of vertices V, resulting in $|V|$ dense subgraph insertions into the index (explore-all, Algorithm 2, line 2). This is a very costly procedure; unsurprisingly, it was experimentally found to dominate all other processing costs, in cases where too-dense subgraphs existed (cf. Section 5.1).

To avoid this cost, we propose a modification to the dense subgraph index, which we term ImplicitTooDense. At a high level, it entails the implicit representation of supergraphs of too-dense subgraphs, so that explore-all will only examine/insert into the index a small number of dense subgraphs.

Specifically, we introduce a fictitious vertex named $*$, which is lexicographically larger than all other vertices. For every too-dense subgraph C, the index will store a subgraph $C \cup \{*\}$, representing all $C \cup \{y\}$ where y is a vertex disconnected from C; these supergraphs of C will not be explicitly inserted in the index. Given this convention, DynDens will handle the explore-all procedure of a subgraph C that just became too-dense by normally exploring all neighbors of C (as in Algorithm 2, line 7), and inserting the subgraph $C \cup \{*\}$ into the index. For instance, revisiting Figure 3, assume subgraph $\{1, 3\}$ is too-dense. Rather than exploring, and inserting into the index all its disconnected supergraphs $\{1, 3, 6\}, \{1, 3, 7\}, \cdots, \{1, 3, |V|\}$, DynDens has only inserted a node representing $\{1, 3, *\}$.

In the unlikely event $C \cup \{*\}$ needs to be explored at any time (this would correspond to the exploration of all supergraphs of C augmented with one disconnected vertex), DynDens will try instead to augment C with all edges in the graph that are not incident on C [8].

Because every vertex a is potentially contained in $C \cup \{*\}$, whenever an iteration is performed on the index (Algorithm 1, line 4), the inverted list corresponding to $*$ needs to be examined as well. This inverted list also needs to be maintained during negative edge weight updates, if a subgraph stops being too-dense. Finally, note that whenever dealing with a subgraph represented by a $*$ index entry, DynDens also needs to ensure that the subgraph is not explicitly represented elsewhere in the index, which is, however, a very efficient operation.

As we verify experimentally (Section 5.1), the above ImplicitTooDense modification to the index offers significant performance benefits to DynDens.

4 Theoretical results

Having introduced our proposed DynDens algorithm, in this section we elaborate on its theoretical underpinnings. We first prove its correctness, by deriving a bound on the number of exploration iterations that are required, as a function of the magnitude of the edge weight update performed (this is the basis of DynDens, cf. Algorithm 2, line 1). Specifically, Section 4.1.2 presents a general result, on when a single exploration iteration per stable-dense subgraph is sufficient. Section 4.1.3 provides a concrete instantiation for T_n (recall that T_n determines the relationship between dense and output-dense subgraphs), based on which the desired bound is then obtained in Section 4.1.4. We conclude this section by discussing results pertaining to the complexity of DynDens (Section 4.2).

4.1 Exploration iterations

4.1.1 Formalization

The notion of exploration iterations performed by DynDens has been used throughout its description; before presenting theoretical results on them, this would be a good opportunity to formalize this notion.

Let $C_A = \{C \cup \{b\} | C \subseteq V \wedge a \in C \wedge b \notin C \wedge C \text{ is stable-dense}\}$ be the set of graphs consisting of a stable-dense subgraph containing a, augmented with b (similarly, let $C_B = \{C \cup \{a\} | C \subseteq V \wedge b \in C \wedge a \notin C \wedge C \text{ is stable-dense}\}$). Let $C_0 = C_A \cup C_B$; this is the set of all subgraphs that will be examined via cheap-exploration only.

Let $C_{AB} = \{C \cup \{y\} | C \subseteq V \wedge a, b \in C \wedge \text{C is stable-dense} \wedge y \text{ is a neighbor of some node in } C\}$ be the set of graphs consisting of a stable-dense subgraph containing a and b, augmented with some other node; this is the set of

[8] Moreover, DynDens does not need to examine all such edges, but only those having high enough weight.

all subgraphs that will be examined via a single exploration iteration.

Let $C_1 = C_0 \cup C_{AB}$; this is the set of graphs containing a and b that consist of a stable-dense subgraph, augmented with one node.

For $i > 1$, let $C_i = \{C \cup \{y\} | C \in C_{i-1} \wedge \text{C is newly-dense} \wedge y \text{ is a neighbor of some node in } C\}$. C_i is the set of graphs containing a newly-dense subgraph that contains a, b, and is discoverable after i exploration iterations.

4.1.2 When is a single exploration sufficient

Let us now provide a sufficient condition for all newly-dense subgraphs C of cardinality $|C| = n \geq 3$ to contain a stable-dense subgraph of cardinality $n-1$. Specifically, it is sufficient that:

$$\delta \leq (n-2)(n-1)(g_n \cdot T_n - g_{n-1} \cdot T_{n-1}) \tag{1}$$

Note that $g_n = \frac{S_n}{n \cdot (n-1)}$, and the properties of T_n guarantee that the above bound on δ is strictly positive.

Proof :

Outline: If all $n-1$ subgraphs of C were sparse before the update, then the contributions of each vertex in C to $dens^{-}(C)$ should be large. Hence, C must be very dense. However, C was sparse before the update. Thus, the update must have been very large. If the update is not very large, then there will exist an $n-1$ subgraph that was dense before the update.

Detailed proof:

Let $D^{-}_i = \sum_{j \in C} w^{-}_{ij}$

Observe that $\sum_{i \in C} D^{-}_i = 2 \cdot score^{-}(C)$, and, since $score^{+}(C) = score^{-}(C) + \delta$, $score^{+}(C) = \frac{1}{2} \cdot \sum_{i \in C} D^{-}_i + \delta$.

Since C is newly-dense, it is the case that

$$\frac{1}{2} \cdot \sum_{i \in C} D^{-}_i < S_n \cdot T_n \tag{2}$$

$$\frac{1}{2} \cdot \sum_{i \in C} D^{-}_i + \delta \geq S_n \cdot T_n \tag{3}$$

Observe that for any $j \in C$, it is the case that $\sum_{i,k \in C/\{j\}} w^{-}_{ik} = (\frac{1}{2} \sum_{i \in C} D^{-}_i) - D^{-}_j$.

Assume all $n-1$ subgraphs of C were sparse before the update. Then $\forall j \in C$ it is the case that:

$$(\frac{1}{2} \cdot \sum_{i \in C} D^{-}_i) - D^{-}_j < S_{n-1} \cdot T_{n-1} \tag{4}$$

From Equation 3 and Equation 4, it follows that

$$D^{-}_j > S_n \cdot T_n - S_{n-1} \cdot T_{n-1} - \delta \tag{5}$$

Summing the above over all $j \in C$, we obtain:

$$\sum_{j \in C} D^{-}_j > n \cdot (S_n \cdot T_n - S_{n-1} \cdot T_{n-1}) - n \cdot \delta \tag{6}$$

From the above and Equation 2, we obtain that

$$2 \cdot S_n \cdot T_n > n \cdot (S_n \cdot T_n - S_{n-1} \cdot T_{n-1}) - n \cdot \delta \iff$$
$$\delta > (n-2)(n-1)(g_n \cdot T_n - g_{n-1} \cdot T_{n-1}) \tag{7}$$

Thus, if the above does not hold (i.e. Equation 1 holds), C must have a $n-1$ subgraph that was dense before the update.

Corollary: The $n-1$ subgraph of C that was dense before the update will either not contain one of a or b (so augmenting it with that vertex will yield C), or it will contain both a and b. Consequently, for values of n where Equation 1 holds, all newly-dense subgraphs of cardinality n will be contained in $C_A \cup C_B \cup C_{AB} = C_1$.

4.1.3 Instantiating T_n

Based on the form of Equation 1, let us now propose a convenient instantiation for T_n, that will satisfy the requisite monotonicity properties, while greatly simplifying the bounds we subsequently derive, thus providing additional intuitions. Specifically, the instantiation of T_n that will be used throughout this work is:

$$T_n = \frac{1}{g_n} \left(g_{N_{max}} \cdot T + \delta_{it} \cdot \left(\frac{n-2}{n-1} - \frac{N_{max}-2}{N_{max}-1} \right) \right) \tag{8}$$

where δ_{it} is a tunable parameter. Note that this is a reasonable value for T_n from a maintenance perspective ; for instance,

- if $S_n = n$, $T_n = (n-1)T_2 + (n-2)\delta_{it} = \frac{n-1}{N_{max}-1}(T + \delta_{it}) - \delta_{it} = O(n)$
- if $S_n = n(n-1)$, $T_n = T_2 + (1 - \frac{1}{n-1})\delta_{it} = T - \delta_{it}(\frac{1}{n-1} - \frac{1}{N_{max}-1}) = O(1)$

Importantly, this instantiation results in a much simplified form of Equation 1, specifically $\delta < \delta_{it}$. In the following, we will leverage this fact, to obtain a bound on the number of exploration iterations that DynDens needs to perform.

Moreover, for our proposed techniques to be meaningful, it must be the case that $T_n >> 0 \; \forall n \in \{2, \cdots, N_{max}\}$. This, along with the above simplified form of Equation 1, leads to the following validity range for δ_{it}: $\delta_{it} \in (0, \frac{S_{N_{max}} T}{N_{max}(N_{max}-2)})$; the lower bound would correspond to maintaining only output-dense subgraphs, and the upper bound to maintaining most[9] subgraphs.

[9] Specifically, all subgraphs of cardinality N_{max}, and most subgraphs of lower cardinalities.

Note that the implication that $\delta_{it} << \frac{S_{N_{max}}T}{N_{max}(N_{max}-2)}$ is not overly restrictive. For instance, for $S_n = n(n-1)$ δ_{it} needs to be significantly smaller than $T(1+\frac{1}{N_{max}-2})$ (i.e. about half the average edge weight of any output-dense subgraph); for $S_n = n$ δ_{it} needs to be significantly smaller than $\frac{T}{N_{max}-2}$ and so on.

Proof: From $T_n > 0$ and Equation 8 we obtain:

$$g_{N_{max}}T + \delta_{it} \cdot \left(\frac{n-2}{n-1} - \frac{N_{max}-2}{N_{max}-1}\right) > 0 \iff$$

$$\delta_{it} \cdot \left(\frac{1}{n-1} - \frac{1}{N_{max}-1}\right) < g_{N_{max}}T \iff$$

$$\delta_{it}\frac{N_{max}-n}{(N_{max}-1)(n-1)} < g_{N_{max}}T \iff$$

(The above holds for $n = N_{max}$ focus on $n < N_{max}$)

$$\delta_{it} < \frac{(N_{max}-1)(n-1)}{N_{max}-n}g_{N_{max}}T \iff$$

since $2 \leq n < N_{max}$

$$\delta_{it} < \frac{N_{max}-1}{N_{max}-2}g_{N_{max}}T$$

4.1.4 Bounding the number of iterations

We are now able to extend Equation 1, to cases where $\delta > \delta_{it}$.

Specifically, we will show that all newly-dense subgraphs of cardinality n are contained in $C_0 \cup C_1 \cdots \cup C_{\lceil \frac{\delta}{\delta_{it}} \rceil}$, thus in order to compute all newly-dense subgraphs, it is sufficient to explore around stable-dense and newly-dense subgraphs contained in $C_0 \cup C_1 \cup \cdots \cup C_{\lceil \frac{\delta}{\delta_{it}} \rceil}$.

Proof: Firstly, observe that an edge weight update of magnitude δ is equivalent to $i = \lceil \frac{\delta}{\delta_{it}} \rceil$ updates, whose magnitudes sum up to δ. Consider, thus, the following sequence of updates:

At step 1, we perform an update of magnitude $\delta - (i-1)\delta_{it}$.

At every subsequent step $step = 2, \cdots, i$ we perform an update of magnitude δ_{it}.

Now, from Section 4.1.2, at every step $step$, we have correctly computed all dense subgraphs C of cardinality $|C| \geq n - step$.

Thus, after all i steps, we have correctly computed all dense subgraphs of cardinality n.

Finally, observe that the above procedure is equivalent to

- performing a single update of magnitude δ on all dense subgraphs, followed by
- updating newly-dense subgraphs discovered in the previous step, and
- repeating the previous step another $i-2$ times.

To further clarify the above, observe that exploring around dense subgraphs of cardinality n will result in newly-dense subgraphs of cardinality $n+1$. Thus, DynDens only needs to explore once around stable-dense subgraphs of cardinality n, and continue exploring only around newly-dense subgraphs of cardinality n.

Discussion: As witnessed from the above result, the magnitude of δ is directly correlated with the impact on dense subgraphs. A useful analogy is that of an edge weight update as a perturbation: the greater its magnitude δ, the further away in the graph its effects can be potentially felt (i.e. the further away dense subgraphs will need to be explored).

In this context, parameter δ_{it} offers a tunable space-time tradeoff; by setting it to higher values, more dense subgraphs will be maintained, but fewer exploration iterations will be required per edge update. Consequently, selecting an optimal good value for δ_{it} is data-dependent; in practice, we observe that DynDens performs well for a wide range of δ_{it} values.

4.2 Complexity results

Let us now conclude the exposition of theoretical results, with a discussion on complexity. In particular, in this section we will discuss the complexity of DynDens, followed by the complexity of the offline variant of Engagement. Our main results can be summarized as follows. Firstly, under a wide range of assumptions, the complexity of DynDens ranges from sublinear to linear in the number of output-dense subgraphs. Moreover, offline variants of Engagement are essentially tractable, and our proposed techniques will generally not offer any additional benefit for the offline case (i.e. their only benefit is enabling efficient streaming computations).

4.2.1 Complexity of DynDens

Let us examine the complexity of DynDens under a set of reasonable assumptions. Specifically, assume the following:

1. Positive edge weight updates are uniformly distributed to edges.
2. The average node degree in the graph is upper bounded by a constant (independent of the graph cardinality N)

3. The average magnitude of positive edge weight updates δ is upper bounded by a constant (independent of N)
4. The maximum cardinality of a dense subgraph of interest (N_{max}) is constant (independent of N)
5. The ratio (number of subgraphs of cardinality n with density $\geq T'$) over (number of subgraphs of cardinality n with density $\geq T$"), for $T' < T''$, is upper bounded by a function of n, T', T'' (independent of N)

Let $OD = |\{C \subseteq V | dens(C) \geq T\}|$ be the number of output-dense subgraphs.

Then, the expected cost of a positive edge weight update (computed over all positive edge weight updates) will be upper bounded by

$$E[cost] \leq O(\frac{OD}{N^2}) \tag{9}$$

Proof: For now, let us ignore the assumptions stated above (we will gradually introduce them, as needed; in this way, it will be easier to see what happens when one or more assumptions are relaxed).

Let $OD_n = |\{C \subseteq V | dens(C) \geq T \wedge |C| = n\}|$ be the number of output-dense graphs of cardinality n that our algorithm will return. $OD = \sum OD_n$ is the number of output-dense subgraphs.

Let $ND_{A,n}(T) = |\{C \subseteq V | A \subseteq C \wedge dens(C) \geq T \wedge |C| = n\}|$. For convenience, let $ND_n(T) = ND_{\{\},n}(T)$, $ND_n = ND_n(T_n)$ and $ND_{A,n} = ND_{A,n}(T_n)$. Observe that $OD_n = ND_n(T)$

Recall the quantity of "normalized density" $normDens(C) = \frac{dens(C)}{T_{|C|}}$; DynDens essentially maintains all subgraphs with $normDens \geq 1$. Observe that the number of subgraphs of cardinality n that our algorithm maintains is none other than ND_n (similarly $ND_{A,n}$ is the number of subgraphs maintained that contain a set of nodes A).

Cost of cheap-explore: The cost of cheap-exploring a subgraph of cardinality n is at most $n+|\Gamma_a|$ or $n+|\Gamma_b|$, depending on whether b or a is contained in the subgraph, respectively (Recall that cheap exploration entails list-merging Γ_a with the subgraph). The number of subgraphs of cardinality n that will be cheap-explored is $ND_{\{a\},n} + ND_{\{b\},n} - 2ND_{\{a,b\},n}$. Thus, the number of operations for cheap explorations is at most

$$\sum_{n=2}^{N_{max}} (ND_{\{a\},n} - ND_{\{a,b\},n}) \cdot (n + |\Gamma_a|) + \\ + (ND_{\{b\},n} - ND_{\{a,b\},n}) \cdot (n + |\Gamma_b|) \tag{10}$$

Let $\overline{degree}$ be the maximum degree of a node in the entire graph (hence $|\Gamma_a|$ and $|\Gamma_b|$ are upper-bounded by $\overline{degree}$). The number of operations spent on cheap explorations will thus less than[10]

$$\sum_{n=2}^{N_{max}} (ND_{\{a\},n} + ND_{\{b\},n} - 2ND_{\{a,b\},n}) \cdot (n + \overline{degree}) \tag{11}$$

Cost of explore: There are $ND_{\{a,b\},n}$ dense subgraphs of cardinality n, that will each be explored $\lceil \frac{\delta}{\delta_{it}} \rceil$ times. Exploration of a subgraph C will incur the cost of computing the neighborhood of C, for a maximum of $\sum_{i \in C} |\Gamma_i|$ operations. This is upper-bounded by $n \cdot \overline{degree}$ operations. Thus, the number of operations spent on explorations will be less than

$$\sum_{n=2}^{N_{max}} ND_{\{a,b\},n} \cdot \lceil \frac{\delta}{\delta_{it}} \rceil \cdot n \cdot \overline{degree} \tag{12}$$

Thus, the total cost of a positive edge weight update will be at most

$$\sum_{n=2}^{N_{max}} (ND_{\{a\},n} + ND_{\{b\},n}) \cdot (n + \overline{degree}) + \\ + ND_{\{a,b\},n} \cdot (\lceil \frac{\delta}{\delta_{it}} \rceil \cdot n \cdot \overline{degree} - 2(n + \overline{degree})) \tag{13}$$

Parametrize wrt. edge weight update probability: Now, let us assume that every edge (u, v) receives a positive edge weight update with probability P_{uv}, and let us compute the expected cost of a positive edge weight update, over all such updates. It is the case that

$$E[cost] \leq \sum_{(u,v) \in E} P_{uv} \cdot \\ \cdot \sum_{n=2}^{N_{max}} (ND_{\{u\},n} + ND_{\{v\},n}) \cdot (n + \overline{degree}) + \\ + ND_{\{u,v\},n} \cdot (\lceil \frac{\delta}{\delta_{it}} \rceil \cdot n \cdot \overline{degree} - 2(n + \overline{degree})) \tag{14}$$

Let $Dens_n = \{C \subseteq V || C| = n \wedge normDens(C) \geq 1\}$, and let X_n be any quantity that only depends on n, and global properties of the graph. Now, observe that

[10] Since in the following we discuss the expected cost of an update, $\overline{degree}$ could also stand for the expected degree of a node in a dense subgraph, although this would get unwieldy to formally define in the general case.

$$\sum_{(u,v)\in E} P_{uv} \cdot \sum_{n=2}^{N_{max}} ND_{\{u,v\},n} \cdot X_n =$$
$$= \sum_{(u,v)\in E} \sum_{n=2}^{N_{max}} \sum_{C\in Dens_n} P_{uv} \cdot X_n \cdot [u, v \in C]$$
$$= \sum_{n=2}^{N_{max}} X_n \cdot \sum_{C\in Dens_n} \sum_{(u,v)\in E} P_{uv} \cdot [u, v \in C] =$$
$$= \sum_{n=2}^{N_{max}} X_n \cdot \sum_{C\in Dens_n} \sum_{u,v\in C} P_{uv} \tag{15}$$

(where [*condition*] is an indicator variable, equal to 1 when *condition* holds, and 0 otherwise).

Finally, observe that $\sum_{u,v\in C} P_{uv}$ is the probability that an edge in C received a positive edge weight update.

Similarly, we derive that

$$\sum_{(u,v)\in E} P_{uv} \cdot \sum_{n=2}^{N_{max}} (ND_{\{u\},n} - ND_{\{u,v\},n}) \cdot X_n =$$
$$= \sum_{n=2}^{N_{max}} X_n \cdot \sum_{C\in Dens_n} \sum_{u\in C, v\notin C} P_{uv} \tag{16}$$

where we observe that $\sum_{u\in C, v\notin C} P_{uv}$ is the probability that an edge cutting C received a positive edge weight update.

Substituting Equation 15 with $X_n = \lceil \frac{\delta}{\delta_{it}} \rceil \cdot n \cdot \overline{degree}$, and Equation 16 with $X_n = (n + \overline{degree})$ in Equation 14, we obtain:

$$E\,[cost] \leq \sum_{n=2}^{N_{max}} (n + \overline{degree}) \cdot$$
$$\cdot \sum_{C\in Dens_n} P(\text{edge cutting } C \text{ was updated}) +$$
$$+ \lceil \frac{\delta}{\delta_{it}} \rceil \cdot n \cdot \overline{degree} \cdot \sum_{C\in Dens_n} P(\text{edge in } C \text{ was updated}) \tag{17}$$

Updates uniformly distributed (Assumption 1): To take this a little further, let us assume that edge updates are uniformly distributed, i.e.

$P(\text{edge cutting } C \text{ was updated}) \leq \frac{2|C|\overline{degree}}{N(N-1)}$
and
$P(\text{edge in } C \text{ was updated}) = \frac{|C|(|C|-1)}{N(N-1)}$

(Note that this assumption also enables us to define $\overline{degree}$ as the average node degree.)

The expected cost of a positive edge weight update is now bounded by:

$$E\,[cost] \leq \sum_{n=2}^{N_{max}} \frac{ND_n}{N(N-1)} \cdot n \cdot \overline{degree} \cdot$$
$$\cdot \left(2(n + \overline{degree}) + \lceil \frac{\delta}{\delta_{it}} \rceil n(n-1) \right) =$$
$$\frac{\overline{degree}}{N(N-1)} \cdot \sum_{n=2}^{N_{max}} ND_n \cdot n \cdot (2(n + \overline{degree}) + \lceil \frac{\delta}{\delta_{it}} \rceil n(n-1)) \tag{18}$$

Ratio of dense over output-dense is bounded (Assumption 5): We will proceed to show something stronger; in cases where the number of dense subgraphs (subgraphs with $normDens \geq 1$) is reasonably close to the number of output-dense subgraphs (subgraphs with $dens \geq T$), our techniques are tractable and efficient in the output-sensitive sense.

Let us assume that the distribution of dense subgraphs is such that the following holds, for some $a(n, T', T'')$, independent of N: $\frac{ND_n(T')}{ND_n(T")} \leq a(n, T', T")$. Moreover, let $a(T', T") = \max_{n=2}^{N_{max}} a(n, T', T")$, and let $a(T") = \max_{T' \leq T"} a(T', T")$.

Consequently,

$$ND_n = OD_n \cdot a(n, T_n, T) \leq OD_n \cdot a(T) \tag{19}$$

Substituting this into Equation 18, we obtain:

$$E\,[cost] \leq \frac{\overline{degree}}{N(N-1)} \cdot a(T) \cdot \sum_{n=2}^{N_{max}} OD_n \cdot n \cdot$$
$$\cdot \left(2(n + \overline{degree}) + \lceil \frac{\delta}{\delta_{it}} \rceil n(n-1) \right) \Longrightarrow$$
$$\Longrightarrow E\,[cost] < \frac{\overline{degree}}{N(N-1)} \cdot a(T) \cdot$$
$$\cdot \left(\lceil \frac{\delta}{\delta_{it}} \rceil \cdot N_{max}^3 + (2 - \lceil \frac{\delta}{\delta_{it}} \rceil) N_{max}^2 + 2\overline{degree} N_{max} \right) OD \tag{20}$$

High-level intuition (All other assumptions): For a better intuition, let us assume that the average node degree in our graph ($\overline{degree}$), an upper bound on the magnitude of the update δ, and the maximum cardinality of a dense subgraph of interest N_{max}, are constant (independent of N). Then, the expected cost of a positive edge weight is upper bounded by:

$$E[cost] \leq O(\frac{OD}{N^2}) \tag{21}$$

Interestingly, this is a sublinear function of the number of dense graphs. Moreover, since our algorithm produces output-dense subgraphs incrementally (in a non-blocking fashion), it is tractable, and efficient, in an output-sensitive sense.

too-dense subgraphs: Note that the proof above does not take into account the existence of too-dense subgraphs. Specifically, in the case of too-dense subgraphs, up to N explorations, instead of $\overline{degree}$, may need to be performed per subgraph. However, Assumption 5 guarantees the absence of too-dense subgraphs (the existence of a too-dense subgraph would cause the ratio to be a function of N).

Moreover, note that Assumption 5 can be relaxed by excluding from the assumption dense graphs that are implicitly represented by ImplicitTooDense. Since this is typically the case, our results will typically hold also in the presence of too-dense subgraphs.

4.2.2 Offline complexity

In concluding our examination of complexity results, let us present a few results regarding the complexity of offline variants of Engagement. Specifically, we show that the following variants are tractable:

- *Top-1* : In a given graph, find the subgraph with the highest density
- *Top-k* : In a given graph, find the k subgraphs with the highest density
- *Threshold* : In a given graph, find all subgraphs with density greater than a given threshold. Note that, for this variant to be tractable, the threshold has to be set in a manner such that there exist only a polynomial number of output-dense subgraphs.

It is worth noting that the tractability of the above problems should not be surprising, as they do not correspond to the (NP-hard) max-clique problem, but rather to the (easy) “any-clique” problem.

Proof outline: For $S_n = n$, the top-1 variant is tractable [14]. Using Lawler’s procedure [23], the top-k variant is tractable as well (with a $k \cdot |V|$ slow down factor). As a result, the threshold variant is also tractable, if only a polynomial number of subgraphs have density over the threshold.

Using these results, we can show similar results for any other choice of S_n, with a slow-down factor of $|V|$, as follows. Conceptually, we construct $|V|$ different graphs, $G_1, G_2, \cdots, G_{|V|}$, where the weight of edge (u, v) in graph G_i is $w^i_{uv} = \frac{w_{uv} \cdot |V|}{S_i}$. We use [14] to compute the top-1 answer for all these graphs; the best of these corresponds to the overall top-1 answer. We proceed similarly for the top-k and threshold variants.

Offline complexity implications of DynDens In very special cases, DynDens could be used to efficiently solve the threshold offline variant of Engagement, with cost linear in the number of output-dense subgraphs ($O(OD)$). The cases where this could happen are special in the following sense: The final offline graph needs to be incrementally built, by inserting its edges one-by-one, in a manner that does not significantly affect the distribution of dense subgraphs in the evolving graph. Other than that, our techniques do not offer any benefit for the offline case of the problem.

5 Evaluation

Let us now discuss the experimental validation of our techniques. We will first briefly go over the experimental setup. In Section 5.1 we will present experimental evidence for the feasibility of real-time story identification via Engagement, as well as the scalability of our proposed approach. We will also examine the main factors that contribute to the efficiency of DynDens.

As we have seen throughout this work, there is a lack of existing techniques for efficiently addressing Engagement. Nevertheless, in Section 5.2 we evaluate adaptations of relevant techniques to this problem, so as to have a basis for comparison.

Finally, although efficiency has been our main focus in this work, in Section 5.3 we present some qualitative results that highlight the effectiveness of our approach.

Experimental setup: All algorithms evaluated were implemented in Java, and executed on 64-bit Hotspot VM, on a machine with 8 Intel(R) Xeon(R) CPU E5540 cores clocked at 2.53GHz. In our experiments, only one core was used, and the memory usage of the JVM was capped at 25G of RAM (the actual memory consumption was typically lower). Finally, in all performance experiments, the time reported is the median time of 3 identical runs.

Datasets: Unless otherwise noted, all our experiments were run using real-world datasets, based on a sample of all tweets for May 1st, 2011 (Our dataset consisted of 13.8M tweets. The sampling was performed by Twitter itself, as part of the restricted access provided to its data stream; for details cf. `tinyurl.com/twsam`). From these, we removed non-English tweets, and tweets that were labeled as spam (using an in-house tweet spam filter [25]), resulting in 3.8M tweets. Subsequently, we used an in-house entity extractor [3] to identify mentions of real-world entities (such as people, politicians, products, etc). 76.5% of the tweets did not mention any entity of interest; 18.3% mentioned one; 4.3% mentioned two, and under 1% mentioned three or more entities. The entire procedure took under 1h 20' (under 350 μsec per tweet on average).

Measuring correlation: Given these sets of co-occurring entities, there are many ways in which entity association can be measured; our techniques are equally applicable, irrespective of the measure used. For our evaluation, we selected two measures from the literature that we found to yield meaningful results under diverse circumstances: a combination of the χ^2 measure and the correlation coefficient inspired by [5] (weighted dataset), that has been found to be highly effective in identifying stories in the blogosphere, as well as a thresholded variant of the log-likelihood ratio [27] (unweighted dataset) that has been successfully used to identify stories in Grapevine over an extended period of time. In general, we note that any measure that measures strength of pairwise association, based on entity occurrences and pairwise co-occurrences can equally be used by our techniques.

Identifying emerging stories: Since the goal of our techniques is to identify stories in real-time, i.e. "stories happening now", a mechanism for discounting older stories is required. To achieve this, we modify our measures of correlation, by applying exponential decay to all entity occurrences and co-occurrences; for instance, in our experiments we used a mean life for a tweet of 2 hours. Note that our techniques are equally applicable without applying any decay, but the stories identified would then correspond to "cumulative stories to date" (cf. Table 3 showing stories for the entire day) as opposed to "current emerging stories".

Approximating complex association measures: Finally, for many measures of association (e.g. statistical measures, such as the log-likelihood ratio), the appearance of a document with just a single entity, can influence the weight of all edges in the graph (e.g. the log-likelihood ratio of a pair of entities is a function of the number of documents that have appeared to date). This would pose a significant challenge to incremental computations; to overcome it, we make use of the following approximation, that is applicable to any measure: the weight of an edge connecting entities e_1, e_2 is computed by ignoring all documents that have appeared after the latest time that either e_1 or e_2 appeared in some document.

Intuitively, this will not significantly affect edges connecting popular entities; indeed we observed that in practice the resulting drop in precision entailed by this approximation was fairly low[11]. Importantly, this approximation enables us, after observing a document that mentions entities $e_1, \cdots, e_j$, to only update the weights of edges that are incident to at least one of these entities are updated, i.e. only the weights of edges $\{(e_i, X) | i \in \{1, \cdots, j\}, X \in V\}$ will be updated.

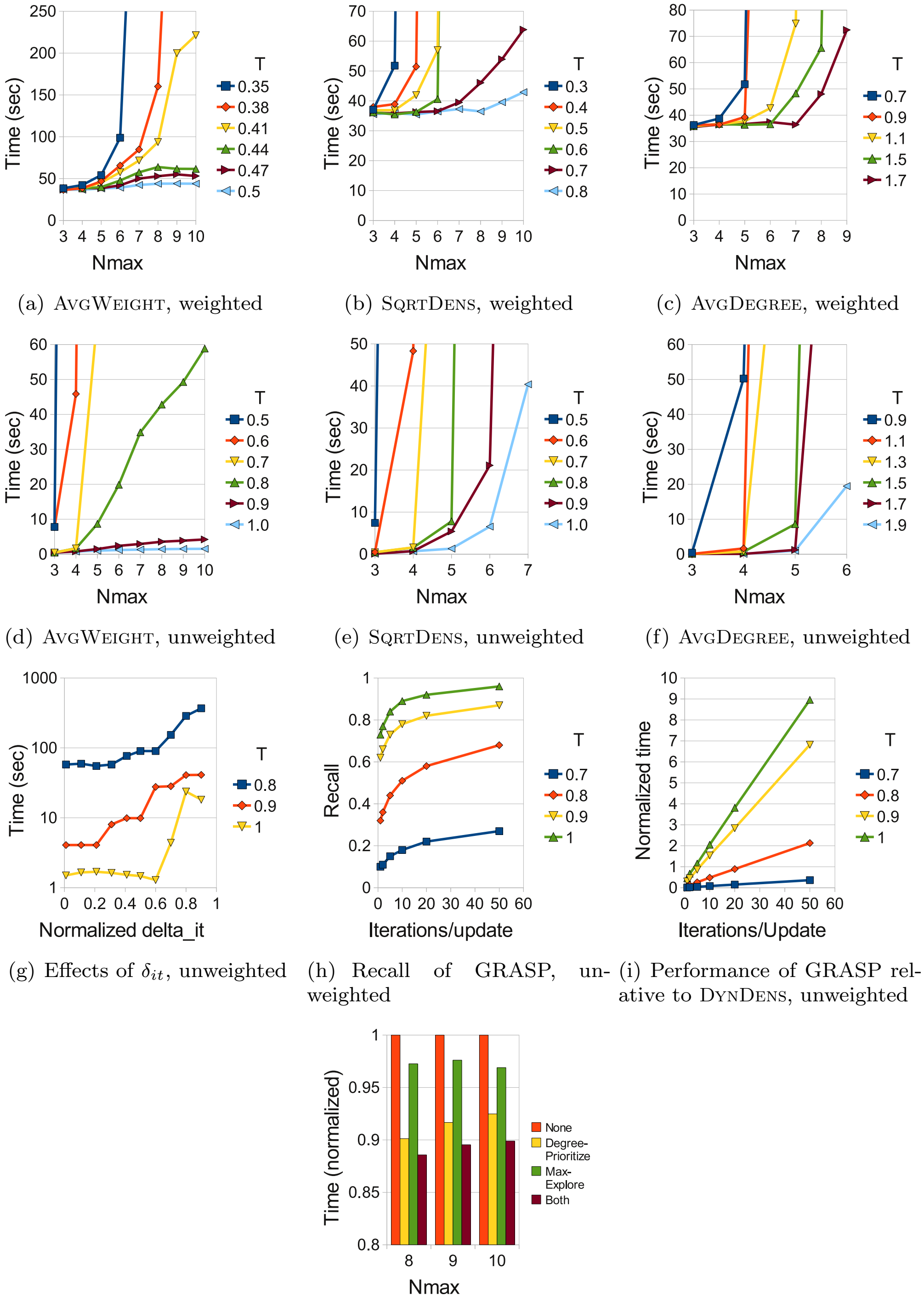
Taking the above into account, the precise manner in which our experimental datasets were created is as follows.

For every tweet where at least one entity was identified, entity occurrences and co-occurrences were updated (taking exponential decay into account, with a mean tweet life of two hours). Thereafter, in the case of the weighted dataset, the χ^2 and correlation coefficient of salient entity pairs was updated; the updated edge weight was computed as $max($ correlation coefficient $, 0)$ if χ^2 showed significant correlation ($p < 5\%$), and 0 otherwise. This procedure resulted in 952K positive edge weight updates, and 40.5M negative ones (recall that the latter are very cheap to process).

In the case of the unweighted dataset, the log-likelihood ratio of salient entity pairs was updated. Two entities were connected with an edge iff each entity appeared in at least 5 tweets, and log-likelihood showed significant correlation ($p < 1\%$). This procedure resulted in 43K positive edge weight updates (edge additions), and 41K negative ones (edge removals).

In either case, this step took under 90 seconds for the entire day. The streams of edge weight updates were loaded to memory before initiating our experiments, and the updates were provided to DynDens sequentially, and in-memory. This reflects the expected usage of DynDens, as the edge weight updates that constitute its input will typically be generated by another process in real-time. All times reported correspond to the time required to process all edge weight updates resulting from a dataset, while maintaining output-dense subgraphs after each update. Specifically, they do not include the time required to preprocess the dataset (e.g. entity extraction, correlation computation), nor do they include the fixed initialization costs of DynDens (such as JVM initialization and initialization of necessary indexing structures).

[11] Specifically, we measured the error entailed by this approximation, i.e. the absolute difference of the approximated value of each edge weight, minus the actual value of the correlation measure, for all edges, at 100 uniformly distributed time instants. The median error over all edges was invariably 0; the average absolute error over all edges and all time instants was 0.0003 for the weighted dataset, and 0.002 for the unweighted one, and the average relative error was 10% and 6% respectively.



(a) AvgWeight, weighted

(b) SqrtDens, weighted

(c) AvgDegree, weighted

(d) AvgWeight, unweighted

(e) SqrtDens, unweighted

(f) AvgDegree, unweighted

(g) Effects of δ_{it}, unweighted

(h) Recall of GRASP, unweighted

(i) Performance of GRASP relative to DynDens, unweighted

(j) Effect of heuristics, synthetic

Fig. 4 Experimental evaluation

5.1 Efficiency and Scalability

Let us now examine some of our experimental findings. Figures 4(a)-4(f) show the time required to process all updates from either dataset, for a variety of definitions of density , and for a wide range of values of density threshold T, maximum dense subgraph cardinality N_{max}. In these figures, δ_{it} has been set to 1% of its maximum value, given the values of the other parameters (thus the number of maintained dense subgraphs is typically close to the number of output-dense subgraphs). All runs were capped at 10 minutes (runs that took longer than that were terminated); all figures are cropped to exclude such time-outs.

We observe that DynDens is able to very efficiently process large datasets, across a wide range of useful operating parameters, validating its applicability for efficiently addressing Engagement. The chosen parameters range from instances with none, or only a few output-dense subgraphs, to instances with too many output-dense subgraphs (in the thousands); i.e. the extremal parameter values correspond to instances of less practical interest. Interestingly, one can observe a sharp increase in performance beyond certain values of parameters T and N_{max}. This is due to the ensuing sharp drop in the average number of output-dense subgraphs. For instance, with reference to Figure 4(d), the average[12] number of output-dense subgraphs of cardinality at most 6, for $T = 1$ is 3.4K; for $T = 0.8$ it is 13.4K; while for $T = 0.7$ it is over 52K. Similar trends can be observed in the other figures as well; for details cf. Table 2. Finally, another interesting trend is that DynDens exhibits higher performance for definitions of density that are closer to *normDens*, such as AvgWeight; the reason for this is that it needs to maintain fewer dense subgraphs per output-dense subgraph for such measures.

Table 2 Statistics on output-dense subgraphs

Figure	T	output-dense subgraphs	T	output-dense subgraphs	T	output-dense subgraphs
4(a)	0.5	53	0.41	211	0.35	1.8K
4(b)	0.8	<1	0.6	32	0.5	128
4(c)	1.7	<1	1.1	13	0.9	1.1K
4(e)	1	2.2K	0.9	5.3K	0.8	215K
4(d)	1	3.4K	0.8	13.4K	0.7	52.7K
4(f)	1.9	4.3K	1.7	19K		

Having discussed the scalability and efficiency of DynDens, let us now turn to evaluating its inner workings. Firstly, let us examine the effects of the δ_{it} parameter. Recall that, low values of δ_{it} correspond to DynDens materializing fewer dense subgraphs, and, correspondingly, having to perform potentially more explorations. In our experiments, we found our techniques to perform equally well for a wide range of values of δ_{it}; however, selecting a value for it, based on characteristics of the dataset can be beneficial to performance. In Figure 4(g), we show the time taken by DynDens to process the unweighted dataset (note the semilog scale), for $N_{max} = 10$ and AvgWeight, across all possible values for δ_{it} (shown normalized to its maximum value for each threshold). We observe an interesting local optimum wrt. δ_{it}, arising from the tradeoff of having to materialize more subgraphs, while enabling faster updates; i.e. increasing δ_{it} improves performance, up to a point where the additional dense subgraphs that need to be maintained make this a performance drain. For instance, this point is around 0.2 for $T = 0.8$, around 0.1 for $T = 0.9$, and around 0.6 for $T = 1$. It is also interesting to note that this tradeoff comes into play again for $T = 1$ and high δ_{it}.

As we previously saw in Section 3.2.3, ImplicitTooDense is crucially important for DynDens to operate efficiently, in the presence of too-dense subgraphs. We validated this intuition experimentally, by executing a variant of DynDens that did not make use of ImplicitTooDense, on the weighted dataset, and comparing its runtime to that of DynDens. We experimented with execution parameters ($N_{max} \in \{9, 10\}, T \in [0.44, 0.5]$ and with δ_{it} between 1% and 50% of its maximum value, given the values of the other parameters. Invariably, the variant without ImplicitTooDense took longer than 20 minutes to complete (and was killed after 20 minutes, in the interests of brevity), while DynDens took 40-85 seconds to complete.

5.2 Comparison with other techniques

As we have already discussed throughout this work, to the best of our knowledge, prior to DynDens, no techniques have been proposed for efficiently addressing Engagement in its general form. Thus, in order to have a basis for comparison, in this section we evaluate adaptations of relevant techniques to subsets of Engagement, namely the dynamic maximal clique algorithm proposed in [28] (Stix), and the Greedy Randomized Adaptive Search Procedure used to identify large quasi-cliques in [1] (Grasp). We wish to stress that, by its very nature, this comparison is not fair,

[12] Averaged over all updates, and excluding output-dense subgraphs that are not represented in the index, e.g. most too-dense subgraphs, augmented with a non-neighboring node (cf. Section 3.2.3).

as the goals of the aforementioned techniques are entirely different from those of ENGAGEMENT, while said techniques are not as general as DYNDENS.

Let us review each comparison in detail. The STIX algorithm proposed in [28] identifies all maximal cliques in dynamic unweighted graphs. This is similar to ENGAGEMENT for $T = 1$, AVGWEIGHT and unweighted graphs, but subtly different, in that ENGAGEMENT requires the identification of *all* cliques. Recall that the output of ENGAGEMENT will be used to present stories to a human user, thus the subgraphs produced cannot be too large. If STIX were used to address ENGAGEMENT, and a maximal clique of cardinality e.g. 20 were identified, all its subgraphs of cardinality e.g. 5 or less would need to be enumerated, and provided as output. (As an aside, the converse is entirely possible in other application domains with different requirements. E.g. if DYNDENS were used for community detection in unweighted graphs, where identifying maximal cliques might be preferable to identifying all cliques, an additional step to filter non-maximal cliques might be needed.)

Keeping in mind the caveats above, we implemented STIX using an efficient in-memory hash-based index[13], and executed it on the unweighted dataset, measuring its execution time, and ignoring the time that would be needed for enumerating all subgraphs of maximal cliques. We compared this runtime to DYNDENS with AVGWEIGHT, $T = 1$ (so as to have a basis for comparison), $N_{max} = 5$ [14], and set δ_{it} to half its maximum value, given the values of the other parameters.

Even though a comparison of STIX and DYNDENS is entirely artificial, the runtime of STIX and DYNDENS were roughly equal: STIX took 958 seconds to process the dataset, compared to 936 sec for DYNDENS. DYNDENS performed even better for lower N_{max}, and took more time for higher N_{max}. Thus, we conclude that DYNDENS is best suited to applications of ENGAGEMENT, while STIX is preferable for applications that require identifying maximal cliques in unweighted subgraphs.

Let us now review the comparison to GRASP, proposed in [1]. This is an approximate randomized algorithm for identifying large dense subgraphs in unweighted graphs. While [1] has significantly more general contributions, for the purposes of this discussion, the algorithm proposed therein can be used to identify subgraphs with density over a given threshold T, under AVGWEIGHT, in unweighted graphs. GRASP will not necessarily identify all dense subgraphs, but can be executed multiple times per update, to identify an increasingly larger number of such subgraphs. It is important to note that, again, the comparison with DYNDENS is not straightforward, as GRASP is geared towards identifying a few large dense subgraphs, as opposed to all dense subgraphs.

Nevertheless, we implemented GRASP, using an efficient in-memory hash-based index [15]. We set the parameter α that controls its greediness vs. randomness tradeoff to 0.5, after ensuring that this did not result in any significant differences in performance[16]. We executed GRASP on the unweighted dataset, for a varying number of iterations per edge weight update (more iterations mean higher runtime, and a higher likelihood of identifying more dense subgraphs), and measured its runtime, and recall (fraction of output-dense subgraphs that it identified, excluding disconnected subgraphs, as GRASP does not produce these). We limited GRASP to searching for subgraphs of cardinalities up to $N_{max} = 5$, and normalized the runtime of GRASP to the runtime of DYNDENS for the same parameters[17] (i.e. the normalized runtime of DYNDENS is 1). The normalized runtime of GRASP is reported in Figure 4(i), and its recall in Figure 4(h). As we can see, GRASP offers a runtime/recall tradeoff, and can thus be at times more efficient than DYNDENS (however, in such cases, it offers recall of under 80%). Moreover, GRASP offers diminishing returns wrt. recall (i.e. it takes increasingly many iterations to achieve arbitrarily high recall; even though the increase in runtime is linear wrt. the number of iterations, the increase in recall is decidedly sublinear). Thus, in this context, GRASP is best suited to identifying a sample of all dense subgraphs. However, since high recall is of crucial importance in story identification (missing 20% of important stories would not generally be acceptable), DYNDENS is best suited to addressing ENGAGEMENT in this setting.

5.3 Qualitative results

Whereas the focus of this work is to efficiently identify dense subgraphs in an incremental manner, we also

[13] [28] does not provide indexing details, so we opted for an efficient solution, albeit with high memory consumption. We also experimented with an adaptation of STIX that used our proposed index, which has much lower memory requirements, but this invariably resulted in increased runtime for STIX.

[14] Since the goal is story identification, we set N_{max} to a low value, corresponding to story cardinalities suitable for consumption by a human.

[15] The index used in [1] is optimized for secondary storage, hence not very useful for the purposes of our comparison.

[16] The average (over the values of all other parameters tested) standard deviation of varying $\alpha \in (0, 1)$ was 4%, and the median standard deviation was 1%.

[17] For DYNDENS we selected a reasonable value of δ_{it}, given the values of the rest of the parameters.

provide evidence of the effectiveness of our approach. Evaluating the quality of our results for realtime story identification is both inherently challenging, due to the lack of a ground truth for what constitutes an important story for a given medium (e.g. a micro-blogging site vs. a news agency), as well as beyond the scope of this work. We will thus present some sample results of utilizing dense subgraphs for story identification.

In order to present sample results, we chose to focus on stories at the granularity of a single day (since presenting stories that were heavily discussed at a specific date and time would be hard to process out of context). We used a dataset similar to the "unweighted" one from our performance experiments, with the following two modifications: entity correlations were computed over the entire dataset, as opposed to using exponential decay; and edge weights were retained for pairs of entities with log likelihood of over 5% significance, as opposed to being thresholded and restricted to $\{0, 1\}$. We computed dense subgraphs of cardinality up to $N_{max} = 5$, using AvgDegree to quantify density, so as to favor larger dense subgraphs; for presentation purposes these were subsequently re-ranked in a diversity-aware manner [2] (subgraph overlap was penalized by multiplying subgraph density by $1 - 0.8 \cdot ($ fraction of story entities covered by previous stories$)$).

The first half of Table 3 presents the resulting top stories. We observe that discussions on Osama bin Laden's death feature prominently in the list; moreover, given the typical tone of conversation on Twitter, distinct discussions involved comparing Pres. Obama's announcement to famous athletes[18], and even on the rapid propagation of the news on Twitter. Other stories cover the evolving crisis in Libya, as well as lighter, ongoing issues, such as Harry Potter, and the antics of Lady Gaga.

For comparative purposes, we also performed the same procedure on a dataset consisting of all blog posts made on major blog hosting platforms during the same day, and report the resulting stories in the second half of Table 3.

6 A Method for Dynamic Threshold Adjustment at Runtime

In our discussions so far, we have assumed that the threshold T does not change during execution. Importantly, we tacitly assumed that T is set to a value such that the number of resulting dense subgraphs are meaningful and of practical interest. However, in many practical applications the characteristics of the data stream may deviate from what had been expected; and there is a need for an efficient procedure for updating the threshold. In particular, it is useful to set a lower and upper limit on the number of desired output-dense subgraphs. Then, when the number of output-dense subgraphs leaves the specified bounds, an incremental threshold update can b performed to bring the number back in range.

The update could be performed via a brute force algorithm as described in Section 5.2, or by reprocessing all edge updates using DynDens with the threshold set to the new value. However, these methods make no use of the current information stored in the index and consequently waste time in recalculation. We will now examine how to efficiently update the threshold at runtime without performing a full recomputation.

6.1 Update Procedure

The update procedure can be split into two cases: (1)when the threshold T is increased, and (2)when the threshold T is decreased. The case of increasing T is easy, and can be handled by scanning the index to remove any dense and output-dense subgraphs that no longer meet the updated criteria.

The case of decreasing T is not as straightforward. When the threshold T is decreased, previously non-dense subgraphs may now be considered dense under this new threshold (note that no subgraph can be evicted from the index). However, unlike the positive edge update case, where we could limit our search to a subset of dense subgraphs all of which contain the updated edge, we have no restrictions on which subgraphs may become newly-dense. Consequently, an exploration must be performed on each dense subgraph currently in the index. In addition, each non-dense edge in the graph must be examined to see if it has become newly-dense.

Algorithm 3 shows the steps necessary for an update of T. Line 1 updates the δ_{it} parameter proportionally to the change in the threshold. This ensures that δ_{it} does not become too large or too small relative to the updated threshold. Lines 2-4 describe the threshold increasing case, while lines 5-9 the threshold decreasing case. In the decreasing case, the first for loop (lines 6

[18] A Cleveland blogger compared Osama bin Laden to athlete LeBron James; the discussion continued on Twitter, resulting in a sports-related meme around the death of bin Laden.

[19] Clint Eastwood was mentioned in conjunction with this story as part of a meme started by comedian Steve Martin, who jokingly tweeted that "[...] President Obama will announce that Clint Eastwood shot Osama Bin Laden".

Table 3 Top stories, May 1st 2011

Description	Entities
From tweets	
Pres. Obama announces killing of Osama bin Laden	Barack Obama,United States House Permanent Select Committee on Intelligence,Osama bin Laden,NBC News
Commentary on death of bin Laden, comparison to famous athletes	Barack Obama,LeBron James[18],Delonte West[18],Osama bin Laden
Discussions on Lady Gaga's activities	Lady Gaga,Galeria
Libya crisis:NATO Airstrike results in death of 3 grandchildren of Gaddafi	NATO,Libya
Discussions on Harry Potter	Hermione Granger,Draco Malfoy,Bella Swan
News on Osama Bin Laden's Death Spreads On Twitter	Clint Eastwood[19],Barack Obama,United States House Permanent Select Committee on Intelligence,Osama bin Laden,CBS News
From blog posts	
Libya crisis:NATO Airstrike results in death of 3 grandchildren of Gaddafi	NATO,Muammar al-Gaddafi,Libya
Barack Obama mocks Donald Trump at White House dinner	White House,Barack Obama,Donald Trump,Seth Meyers,Osama bin Laden
Sony PlayStation Network Hacked	Sony,PlayStation,Kazuo Hirai
U.S. Banks probed over credit default swaps	Wells Fargo,Bank of America,Citigroup,JPMorgan Chase
The royal wedding	Royal Wedding,Prince William of Wales,Kate Middleton
Pres. Obama announces killing of Osama bin Laden	White House,Barack Obama,United States,Pakistan,Osama bin Laden

Algorithm 3 ThresholdUpdate(T_{old}, T_{new})

```
Input: Previous threshold $T_{old}$. New threshold $T_{new}$
1: $\delta_{it} = \delta_{it} * \frac{T_{new}}{T_{old}}$
2: if $T_{old} < T_{new}$ then
3:     for all dense subgraphs $C$ do
4:         evict $C$ from the index if $C$ is losing-dense; report if no longer output-dense (based on $T_{new}$)
5: else if $T_{old} > T_{new}$ then
6:     for all edges $e$ in the graph do
7:         add $e$ to the index if newly-dense; report if output-dense (based on $T_{new}$)
8:     for all dense subgraphs $C$ do
9:         UpdateExplore($C$)
```

Algorithm 4 Procedure **UpdateExplore**(C)

```
Input: Subgraph $C$
if $C$ is not too-dense before the update procedure and $|C| < N_{max}$ then
    if $C$ is too-dense then
        for all $y \notin C$ do {// Explore-All}
            Add $C \cup \{y\}$ to the index; report it if it is output-dense (based on $T_{new}$)
            UpdateExplore($C \cup \{y\}$)
    else
        for all neighbors $y$ of $C$ do
            if $C \cup \{y\}$ is newly-dense (not stable-dense) then
                Add $C \cup \{y\}$ to the index; report it if it is output-dense (based on $T_{new}$)
                UpdateExplore($C \cup \{y\}$)
```

and 7) is the base case. This is necessary to ensure newly dense edges make it into the index to then be further explored in the second for loop. The explore procedure called in line 9 is given by Algorithm 4 and is similar to explore procedure given in Algorithm 2. In this case, we can stop any time a stable-dense subgraph is found, because this subgraph would have already been present in the index and therefore will be explored via the for loop in line 8 of the ThresholdUpdate. Similar to the DynDens explore procedure we are able to stop anytime a subgraph that became newly-dense through the update procedure is found, as it has already been processed.

6.2 Evaluation

Experimental Setup: The threshold update algorithm was implemented in Java, and executed on a 64-bit machine with an Intel(R) Core(TM) i7 CPU clocked at 2.67GHz. In our experiments, only one core was used with the JVM memory capped at 1.5 GB.

Datasets: The data for the threshold update experiments was a set of 8 synthetically generated graphs of 2 different sizes (249K nodes with 750K edge updates and 500K nodes with 1.5M edge updates) and 4 different generation strategies - random, edgePreferential, nodePreferential, nodePreferentialBoolean.

For the random, edgePreferential, and nodePreferential strategies, each update had a weight uniformly distributed between 0.0 and 1.0, while for nodePreferentialBoolean the updates took weights of either 0 or 1. In all cases, the update was negative with probability 0.1. The strategies mainly differ in how they select which edge to update. The random strategy selects the edge uniformly at random, while the edgePreferential strategy selects an edge with probability 0.2 from a set of predefined edge bins, otherwise it selects randomly from the remainder of the graph. Similarly, the nodePreferential and nodePreferentialBoolean strategies select edges such that the edge falls within bins of predefined nodes with probability 0.2.

Results: To evaluate the efficiency of the threshold update procedure, we ran threshold updates of varying magnitudes for both the increasing and decreasing case. We compare the time taken by the update procedure to that of a full recomputation using DynDens by treating each edge weight as an update, with the threshold set to the final updated value and the MaxExplore, DegreePrioritize, and ImplicitTooDense optimizations turned on. We will call this procedure DynDensRecompute to distinguish it from DynDens. We compare against this procedure rather than the brute force algorithm described in section 5.2, since the brute force algorithm took over 15 minutes to complete, while DynDensRecompute finished in several seconds.

For the case of increasing threshold, DynDensRecompute was initially run on the input with a threshold set to 0.8. The update procedure was then run to update T. This process was repeated to update T to values ranging from 0.8 to 1.0 for each dataset; the results for each graph are shown in figure 6(a) with times normalized to the run time of DynDensRecompute.

For the decreasing case, DynDensRecompute was initially run on the input with a threshold set to 1.0, followed by an update procedure updating to values of T between 0.8 and 1.0. The normalized results for the decreasing case are recorded in figure 6(b).

In both cases, only the time for the update procedure to run was recorded (not the time for the initial DynDensRecompute computation) and the median of 3 identical runs was used. Also note, that all threshold updates were run with only the ImplicitTooDense optimization turned on.

As can be seen in the results, the update procedure is much more efficient than performing a full recomputation. These experiments show a decrease of about 5 to 10 times for the decreasing case, and even larger for the increasing case.

Figures 6(c) and 6(d) give the raw runtimes in milliseconds for the increasing and decreasing updates, respectively. As expected for both the increasing and decreasing cases, a monotonic increase in runtime is seen as the magnitude of the threshold change increases. More accurately, the increase in runtime is caused by the change in the number of stored subgraphs (which is loosely correlated to the threshold change). Table 4 shows the number of stored subgraphs for each graph at the various threshold levels. If the number of dense subgraphs does not change, for example see the NodePrefBool249 and NodePrefBool500 graphs, the runtime stays constant. The reason that the number of subgraphs remains unchanged in this case can be attributed to the fact that all edge updates are of magnitude either 0 or 1 in the NodePrefBool249/500 graphs. Thus, each update is already larger than the threshold, resulting in a large number of dense subgraphs that do not vary as the threshold changes from 0.8 to 1.0. We also note that the runtime to process these particular graphs is significantly higher than the other graphs of equal size. This can be attributed to the number of dense subgraphs these datasets contain, which corresponds to about 3 times more than the other graphs of the same size.

7 Heuristics

We examine two additional heuristics that can offer modest performance improvements to DynDens without affecting the quality of results. Both are related to limiting the number of explorations, and cheap explorations performed.

7.1 MaxExplore

Whereas it serves to prove the correctness of DynDens, the bound on exploration iterations that need to be performed on a subgraph C is overly pessimistic, as it is based on several worst-case assumptions. In this section, we develop MaxExplore, an improvement over the previous bound, that takes the graph neighborhood of the updated edge, as well as the cardinality of the subgraph being explored, into account. As it is a fairly cheap bound to compute, we expect MaxExplore to lead to performance improvements in the case of dense subgraphs on which multiple exploration iterations would have otherwise been performed.

Specifically, we subsequently define three functions $maxExplore$, $maxExplore_a$, $maxExplore_b$, that only depend on the graph neighborhood of a and b, such that:

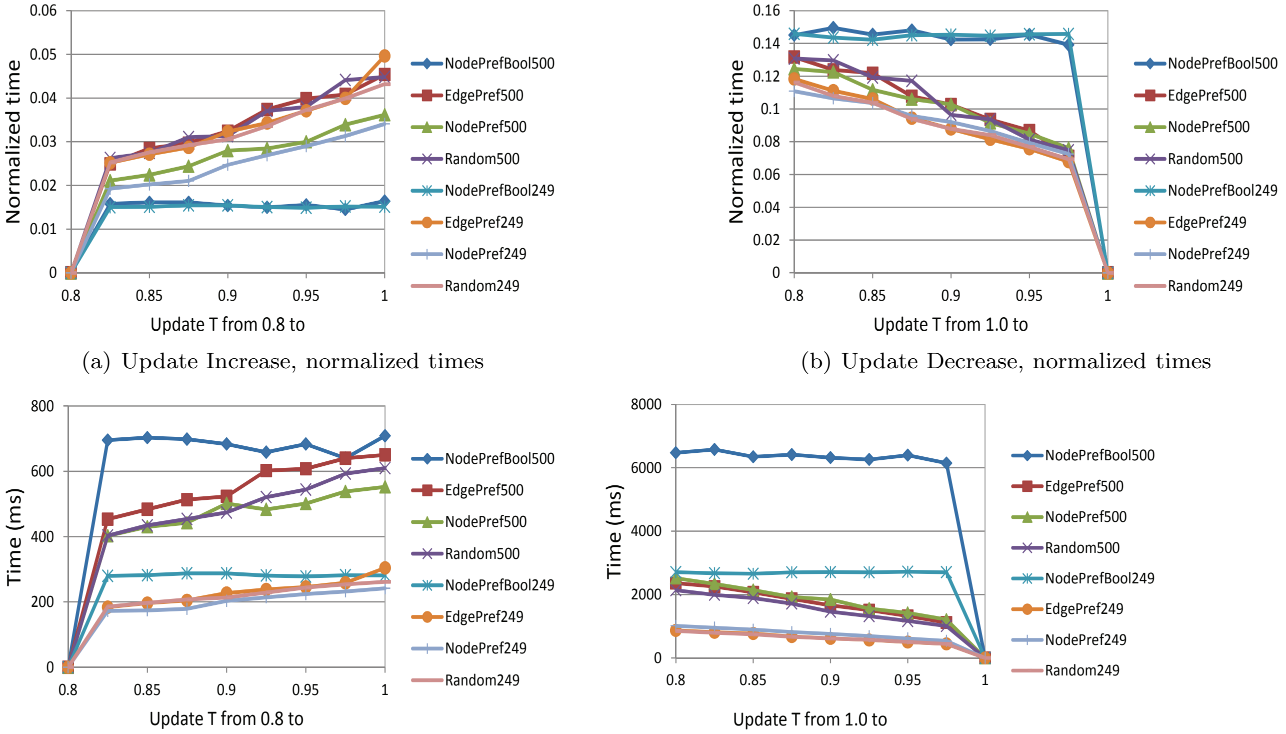
Exploration: If $maxExplore = 3$, DynDens does not need to perform any explorations (all newly-dense subgraphs can be identified via cheap exploration; DynDens only needs to examine whether edge ab is newly-dense, and perform cheap explorations). Otherwise, the number of exploration iterations that need to be performed on a subgraph C are $\min(maxExplore - |C|, \lceil \frac{\delta}{\delta_{it}} \rceil)$ (instead of $\lceil \frac{\delta}{\delta_{it}} \rceil$).

Cheap exploration: Furthermore, if $maxExplore_a \geq maxExplore_b$ (similarly for the converse), DynDens only needs to i) cheap-explore subgraphs containing only b,

Table 4 The number of subgraphs stored in the index for each threshold.

T	NodePref-Bool500	EdgePref-500	NodePref-500	Random-500	NodePref-Bool249	EdgePref-249	NodePref-249	Random-249
0.800	1501543	419342	419666	419531	753105	210078	210963	210065
0.825	1501543	385670	385818	385726	753105	193108	193787	193043
0.850	1501525	351833	351934	351974	752940	176123	176809	176253
0.875	1501525	318458	318088	318445	752940	159286	159615	159177
0.900	1501525	284487	284642	284669	752940	142288	142465	142145
0.925	1501525	250763	251143	250816	752940	125404	125460	125402
0.950	1501525	217202	217329	217308	752940	108550	108535	108508
0.975	1501525	183568	183617	183619	752940	91827	91523	91537
1.000	1501525	150129	149651	149994	752940	74948	74861	74881

Fig. 5 Threshold update evaluation



(a) Update Increase, normalized times

(b) Update Decrease, normalized times

(c) Update Increase

(d) Update Decrease

and ii) cheap-explore subgraphs of cardinality $\leq maxExplore_a - 1$ containing only a.

Our precise theoretical result, from which the above follow directly is that: All newly-dense subgraphs C of cardinality $|C| \geq maxExplore_a$ belong to C_A (i.e. consist of a stable-dense subgraph containing a, augmented with b; similarly for b).

The definitions of the three *maxExplore* functions are as follows: Let $best_b(i)$ be the ith largest weight ($i \in \{1, \cdots, |\mathbf{\Gamma}_b| - 1\}$) of any node in $\mathbf{\Gamma}_b$ except a, and let $best_b(0) = w + \delta$; for $i \geq |\mathbf{\Gamma}_b|$ define $best_b(i) = 0$. Let $top_b(i) = \sum_{j=0}^{i} best_b(i)$. Let $Z = 2(g_{N_{max}} T + \frac{\delta_{it}}{N_{max}-1})$. Let $maxExplore_a = \min(i \in \{3 \cdots N_{max}\} | top_b(i-1) \leq Z(i-1) - \delta_{it} \wedge best_b(i) < Z)$ (similarly $maxExplore_b$). Let $maxExplore = \min(maxExplore_a, maxExplore_b)$.

Proof sketch: If a newly-dense subgraph of cardinality n does not contain a stable-dense subgraph containing a and not b (similarly, a stable-dense subgraph containing b and not a), then the contribution of b (sim. of a) to the subgraph's score should be "large". Thus, if the maximum possible contribution of b (sim. of a) to the score of a subgraph of cardinality n containing a and b is "small", then all newly-dense subgraphs of cardinality n must contain a stable-dense subgraph containing a (sim. b). Before proving the correctness of this procedure, we prove a related lemma.

Lemma: Denoting $\mathbf{c} = \mathbf{x} + \hat{e}_a + \hat{e}_b$ (where $\mathbf{x} \cdot \hat{e}_a = \mathbf{x} \cdot \hat{e}_b = 0$), a sufficient condition for all newly-dense subgraphs C of cardinality $|C| = n \geq 3$ to belong in C_A (similarly C_B) (i.e. consist of a stable-dense subgraph containing a, augmented with b (similarly vice-versa)) is:

$$\max_{|\mathbf{x}|=n-2 \wedge \mathbf{x}\cdot\hat{e}_a=\mathbf{x}\cdot\hat{e}_b=0} \mathbf{x}\cdot\mathbf{\Gamma}_b - S_n \cdot T_n + S_{n-1}\cdot T_{n-1} + w + \delta \leq 0 \tag{22}$$

Proof of lemma: The intuition behind this lemma is as follows. If a newly-dense subgraph of cardinality n does not contain a stable-dense subgraph containing a and not b (similarly, a stable-dense subgraph containing b and not a), then the contribution of b (sim. of a) to the subgraph's score should be "large". Thus, if the maximum possible contribution of b (sim. of a) to the score of a subgraph of cardinality n containing a and b is "small", then all newly-dense subgraphs of cardinality n must contain a stable-dense subgraph containing a (sim. b). Specifically, let $x = \sum_{i,j\in\mathbf{x}\wedge i<j} w_{ij}$. Assume C is a newly-dense graph of cardinality n, then

$$x + \mathbf{x}\cdot(\mathbf{\Gamma}_a + \mathbf{\Gamma}_b) + w + \delta \geq S_n \cdot T_n \tag{23}$$

For C to not consist of a stable-dense subgraph containing a augmented with b (similarly of a stable-dense subgraph containing b augmented with a), it is necessary that $X \cup \{a\}$ (sim. $X \cup \{b\}$) is sparse:

$$x + \mathbf{x}\cdot\mathbf{\Gamma}_a < S_{n-1}\cdot T_{n-1} \tag{24}$$

From Equation 23, it is thus necessary that

$$\mathbf{x}\cdot\mathbf{\Gamma}_b + w + \delta > S_n\cdot T_n - S_{n-1}\cdot T_{n-1} \tag{25}$$

So, for n where the above does not hold for any C of cardinality n, i.e. Equation 22 holds, all newly-dense subgraphs C of cardinality n must consist of a stable-dense subgraph containing a, augmented with b.

Proof: Returning to the core of our proof, from Equation 8 we obtain:

$$\begin{aligned}
&S_nT_n = g_nT_nn(n-1) = \\
&= \left(g_{N_{max}}T + \delta_{it}\cdot(\frac{n-2}{n-1} - \frac{N_{max}-2}{N_{max}-1})\right)n(n-1) = \\
&= g_{N_{max}}Tn(n-1) + \delta_{it}n(n-2) + \\
&+ \delta_{it}(1 - \frac{1}{N_{max}-1})n(n-1) = \\
&= g_{N_{max}}Tn(n-1) + \delta_{it}n\frac{n-N_{max}}{N_{max}-1}
\end{aligned} \tag{26}$$

$$\begin{aligned}
&S_{n-1}T_{n-1} - S_nT_n = g_{N_{max}}T((n-1)(n-2) - n(n-1)) + \\
&+ \frac{\delta_{it}}{N_{max}-1}(n(n-1-N_{max}) - n + 1 \\
&+ N_{max} - n(n - N_{max})) \\
&= \delta_{it} - 2(n-1)(g_{N_{max}}T + \frac{\delta_{it}}{N_{max}-1}) = \delta_{it} - Z(n-1)
\end{aligned}$$

Substituting the above into Equation 22, and using the definition of $top_b(i)$, we obtain:

$$top_b(n-2) \leq Z(n-1) - \delta_{it}$$

Now, by the definition of $maxExplore_a$, all dense subgraphs C of cardinality $|C| = maxExplore_a$ satisfy Equation 7.1, and hence belong in C_A. For $|C| = n > maxExplore_a$, by the definition of $maxExplore_a$, we have:

$$\begin{aligned}
&top_b(n-1) = top_b(maxExplore_a - 1) + \sum_{i=maxExplore_a}^{n-1} best_b(i) \leq \\
&Z(maxExplore_a - 1) - \delta_{it} + (n - maxExplore_a)\cdot \\
&\cdot best_b(maxExplore_a) \\
&(\text{because for } i > 0\ ,\ best_b(i) \geq best_b(i+1)\ ) \\
&< Z(maxExplore_a - 1) - \delta_{it} + (n - maxExplore_a)\cdot Z = \\
&= Z(n-1) - \delta_{it}
\end{aligned}$$

Thus all subgraphs C with $|C| \geq maxExplore_a$ satisfy Equation 7.1, and hence belong in C_A.

7.2 DegreePrioritize

Another challenge in the basic form of DynDens previously discussed, is that a single graph might be explored multiple times, by exploration procedures originating from each of its dense subgraphs. In this section we develop DegreePrioritize, a way to organize the search space, and thus often avoid performing redundant explorations. DegreePrioritize is inspired by the degree-based criterion proposed in [29]. At a high level, it guarantees that DynDens does not need to explore (or cheap-explore) a subgraph with vertices having dense connections to the subgraph. We thus expect DegreePrioritize to offer the greatest benefit to performance in cases of dense subgraphs on which redundant, multiple-iteration explorations would have otherwise been performed. Specifically, we show that for all newly-dense subgraphs to be discovered, it is sufficient:

- when exploring a subgraph C, to not consider nodes u s.t. $\mathbf{\Gamma}^{-}{}_u \cdot \mathbf{c} > \frac{2}{|C|-1}\cdot score^{+}(C)$, and
- when cheap-exploring a subgraph C with node a (sim. b), to not perform the cheap-exploration if $\mathbf{\Gamma}^{+}{}_u \cdot \mathbf{c} > \frac{2}{|C|-1}\cdot score^{-}(C)$

Proof: Let $D_u = \mathbf{\Gamma}_u \cdot \mathbf{c}$ be a generalized "degree" of vertex u wrt. subgraph C. We will first show a stronger version of the growth property of Section 4.1.2, that characterizes the node that augments a stable-dense subgraph to obtain a newly-dense one (it is, in a sense, the minimum ‘degree’ node in the newly-dense subgraph, as quantified by D_u). We then use this property to obtain a bound on D_u for single-iteration explorations (i.e. when $\delta \leq \delta_{it}$). Finally, we extend our result to multiple-iteration explorations.

Part 1: If $\delta \leq \delta_{it}$, for every newly-dense subgraph C of cardinality $|C| = n$, let $u = \mathit{arg}\min_{v \in C} D^{-}_{v}$ (if two nodes have equal D^{-}_{v}, select the smallest one lexicographically). Then $C \setminus \{u\}$ is stable-dense .

Proof of part 1: From the theorem of Section 4.1.2, $\exists v \in C : C \setminus \{v\}$ is stable-dense, i.e. $\mathit{score}^{-}(C \setminus \{v\}) \geq S_{n-1}T_{n-1}$. However, $\forall u' \in C : D^{-}_{u'} \leq D^{-}_{v}$ it is the case that $\mathit{score}^{-}(C \setminus \{u'\}) = \mathit{score}^{-}(C) - D^{-}_{u'} \geq \mathit{score}^{-}(C) - D^{-}_{v} = \mathit{score}^{-}(C \setminus \{v\}) \geq S_{n-1}T_{n-1}$, i.e. $C \setminus \{u'\}$ is stable-dense. Thus, also $C \setminus \{u\}$ is stable-dense.

Part 2: Given the case outlined in part 1, it is the case that $D^{-}_{u} \leq \frac{2}{|C|-2} \cdot \mathit{score}^{-}(C \setminus \{u\})$.

Proof of part 2: Since $\forall i \in C\ D^{-}_{u} \leq D^{-}_{i}$, summing over every i, we obtain: $n \cdot D^{-}_{u} \leq \sum_{i \in C} D^{-}_{i}$. However, it is the case that $\forall j \in C\ \mathit{score}^{-}(C \setminus \{j\}) = \left(\frac{1}{2}\sum_{i \in C} D^{-}_{i}\right) - D^{-}_{j}$, thus, substituting u for j, the previous equation can be written as: $n \cdot D^{-}_{u} \leq 2\mathit{score}^{-}(C \setminus \{u\}) + 2D^{-}_{u}$, i.e. $D^{-}_{u} \leq \frac{2}{|C|-2} \cdot \mathit{score}^{-}(C \setminus \{u\})$.

Part 3: (Extension to multiple-iteration explorations) In the same manner as Section 4.1.4 (i.e. decomposing one large δ update, into multiple δ_{it}-sized ones), in order to cover multiple-iteration explorations, we slightly relax the above conditions; they should reflect the weakest inequality that a node u would have to satisfy during some (cheap) exploration iteration, in order to be a candidate for exploration. This results in the two inequalities initially stated.

7.3 Evaluation

In our evaluation of DynDens, the above heuristics were enabled. Thus, to evaluate their performance benefits, we also evaluated variants of DynDens where either DegreePrioritize and/or MaxExplore were disabled, on both our weighted and unweighted datasets. We observed that these heuristics were responsible for very modest performance improvements of up to 4%, and sometimes even resulted in worse performance.

By design, we expect the proposed heuristics to offer performance benefits in cases where many explorations would have otherwise been performed in their absence. To validate this, and further investigate their potential to improve performance, we evaluated them on a synthetic dataset that consisted of near-cliques, mixed with random edges, that was generated as follows: In an initially empty graph with 100K vertices, 250K updates were generated, each of magnitude $(0, 0.1]$ (with probability 0.3 the update was negative). With probability 0.9, the update occurred within one of 100 predefined sets of 10 vertices each; otherwise, it was uniformly randomly distributed to the remainder of the graph. Finally, in order to evaluate the proposed heuristics in the absence of too-dense subgraphs, updates that would result in too-dense subgraphs for $T = 0.7$ and δ_{it} at 40% of its maximum value, were rejected.

Figure 4(j) shows the time taken by each DynDens variant (no heuristics enabled, only DegreePrioritize enable, only MaxExplore enabled, both heuristics enabled), normalized by the time taken by the first variant; the operating parameters were $T = 0.7$, $N_{max} \in \{8, 9, 10\}$, and δ_{it} at 40% of its maximum value (note that the Y axis does not start at 0). The proposed heuristics are seen to offer performance improvements of up to over 10%; thus, while not as crucial as ImplicitTooDense to performance, we believe that the low effort required to implement these heuristics make them worthwhile for inclusion in DynDens.

8 Related work

While we are not aware of any work that addresses the maintenance of dense subgraphs in weighted graphs, under streaming edge weight updates, for a broad definition of density, there exists a rich literature of works dealing with related problems.

[28] addresses incremental maximal clique maintenance, from a mostly theoretical perspective, and using a growth property. This is very closely related to a special case of Engagement (namely, for unweighted graphs, AvgWeight, and $T = 1$). An important difference from our work is that, this specific instantiation of Engagement deals with *all* cliques, with cardinality constraints, as opposed to maximal cliques of unconstrained cardinality. As discussed in Section 5.2, while the former is better suited to real-time story identification, the latter may be preferable in other scenarios.

[29] addresses near-clique identification, in an offline setting, again from a mostly theoretical perspective, and using a growth property; this corresponds to the offline version of Engagement for unweighted graphs, and AvgWeight. The techniques proposed therein cannot be efficiently dynamized in a straight-forward fashion, as the information they rely upon cannot be efficiently maintained across updates. Our DegreePrioritize pruning condition is inspired by the parent degree-based criterion proposed in this work. [24] addresses the same problem, using a similar growth property, and with a focus on a parallel implementation. As with the other works, the techniques developed therein are not straightforward to efficiently dynamize.

Max (quasi-) clique: Related problems occur in the maximum clique [26] and quasi-clique literature. To overcome the intractability and inapproximability of this problem, heuristics (typically randomized) have been used to discover large (quasi-) cliques. A crucial difference from Engagement is that it requires the enumeration of *all* dense subgraphs (recall that, from an application perspective, each subgraph corresponds to a story of interest). In contrast, works in the maximum (quasi-) clique domain are geared towards identifying one "good" subgraph per execution iteration. Moreover, most such heuristic techniques are not straightforward to efficiently dynamize.

Perhaps most closely related is the state-of-the-art Greedy Randomized Adaptive Search Procedure used in [1] to identify large dense subgraphs (quasi-cliques). Although this work is more focused towards developing techniques for limited main-memory scenarios, their techniques can be dynamized in an efficient manner to address Engagement for unweighted graphs and AvgWeight (cf. Section 5.2).

Local density: Other works have dealt with edge-weight update semantics, albeit with much simpler definitions of density. For instance [31] and others maintain dense subgraphs over sliding windows using *neighbor-based patterns* (i.e. whether a dense subgraph should be augmented with an additional node is decided based on local information only). As the problem being addressed therein is very different from Engagement, the proposed techniques are inapplicable in the latter domain.

Max-flow: [12], [20] and others use (primarily) max-flow based algorithms to identify dense subgraphs. While max-flow algorithms can be dynamized [22], [18], these algorithms can only identify and maintain clusters containing user-specified nodes. In a related vein, [14] uses max-flow to find the top-1 dense subgraph (for AvgDegree); however their techniques cannot be efficiently applied to a top-k or threshold variant, nor can they be efficiently dynamized (the core of the proposed algorithm consists of a binary search where each comparison is a max-flow computation).

Dynamic graphs: Other works (e.g. [10], [6]) have dealt with dynamic graph algorithms under edge weight updates, but do not deal with density problems, focusing instead on properties such as planarity, connectivity, triangle counting, etc. A notable exception is [17], which discusses approximation algorithms to general maximization problems in dynamic graphs. It is, however, theoretical in nature, and its focus is on the approximation ratio of the resulting algorithm, not on efficiency.

Clustering: Related problems are also dealt with in the incremental clustering literature (e.g. [11], [15], [8]); however, these deal with graph node insertion and deletion, and the proposed techniques cannot directly accommodate streaming edge weight updates. A tangentially related problem is evolutionary clustering ([7], [21]) which identifies clusters based on both density, and historical data; the goal is to introduce temporal smoothing, so that clusters behave in a stable fashion over time.

Communities of interest: [9], and its extension [19], address the problem of supporting efficient retrieval of important 2-neighbors of any node, where the importance of a neighbor is related to local and global edge thresholds. The focus is on better representation of actual interactions, and removal of spurious information, and the provided insights are invaluable for any applications that involve dynamic graphs. However, the problem examined in these works, is substantially different from Engagement, hence techniques proposed in these works do not apply in Engagement.

Shingling: [13] proposes techniques to identify large dense subgraphs in an offline fashion via *recursive shingling*. While this could potentially be dynamized, it is geared towards large subgraphs (100-10K nodes), and would not be effective on smaller subgraphs. [30] also uses LSH to identify cliques of moderate size in large graphs; it is however not easily amenable to dynamization, as it has a significant preprocessing phase.

Data structures: Finally, the index structure used by DynDens resembles the FP-tree [16], in that both store overlapping subsets in a prefix tree, with inverted lists embedded into the tree structure. However, the FP-tree is optimized for static data, and assumes that tree nodes can be statically ordered in a way that heuristically decreases tree size; this makes it unsuitable for Engagement, where tree nodes dynamically change. Moreover, other improvements of the FP-tree over a plain prefix tree are not applicable to Engagement, as the problems solved are different.

9 Conclusions

Motivated by the need to mine important stories and events from the social media collective, as they emerge, in this work we examine the problem of maintaining dense subgraphs under streaming edge weight updates. For a broad definition of graph density, we propose the first efficient algorithm, DynDens, which is based on novel theoretical results regarding the magnitude of change that a single edge weight update can have. DynDens is highly efficient, and able to gracefully scale to rapidly evolving datasets, and we validate the efficiency and effectiveness of our approach via a thorough evaluation on real and synthetic datasets. In addition, we examine the dynamic adjustment of the density threshold T during execution, demonstrating that our incremental update procedure, provides great improvements over a full recomputation strategy.

Moreover, there are many exciting new directions stemming from this work. For example, an important problem in the social media space is the timely identification of online communities. While it is easy to see how ENGAGEMENT can be applied to this domain, its characteristics are somewhat different from those of real-time story identification (e.g. social graphs are frequently directed, communities are typically subgraphs of larger cardinality than stories, etc.), and it would be interesting to explore how to adapt DYNDENS to the diverse challenges this domain imposes.

References

1. Abello, J., Resende, M.G.C., Sudarsky, S.: Massive quasi-clique detection. In: Proc. 5th Latin American Symposium on Theoretical Informatics, pp. 598–612 (2002)
2. Angel, A., Koudas, N.: Efficient diversity-aware search. In: SIGMOD Conference, pp. 781–792 (2011)
3. Angel, A., Koudas, N., Sarkas, N., Srivastava, D.: What's on the grapevine? In: Proc. SIGMOD Conference, pp. 1047–1050 (2009)
4. Angel, A., Sarkas, N., Koudas, N., Srivastava, D.: Dense subgraph maintenance under streaming edge weight updates for real-time story identification. Proc. VLDB **5**(6), 574–585 (2012)
5. Bansal, N., Chiang, F., Koudas, N., Tompa, F.W.: Seeking stable clusters in the blogosphere. In: Proc. 33rd international conference on Very large data bases (VLDB), pp. 806–817 (2007)
6. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proc. 13th annual ACM-SIAM symposium on Discrete algorithms (SODA), pp. 623–632 (2002)
7. Chakrabarti, D., Kumar, R., Tomkins, A.: Evolutionary clustering. In: Proc. ACM KDD Conference, pp. 554–560 (2006)
8. Charikar, M., Chekuri, C., Feder, T., Motwani, R.: Incremental clustering and dynamic information retrieval. In: Proc. 29th annual ACM symposium on Theory of computing (STOC), pp. 626–635 (1997)
9. Cortes, C., Pregibon, D., Volinsky, C.: Computational methods for dynamic graphs. Journal of Computational and Graphical Statistics (2003)
10. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: M.J. Atallah (ed.) Algorithms and Theory of Computation Handbook, chap. 8. CRC Press (1999). URL `http://www.info.uniroma2.it/~italiano/Papers/dyn-survey.ps.Z`
11. Ester, M., Kriegel, H.P., Sander, J., Wimmer, M., Xu, X.: Incremental clustering for mining in a data warehousing environment. In: Proc. of the 24rd International Conference on Very Large Data Bases, pp. 323–333 (1998)
12. Flake, G.W., Lawrence, S., Giles, C.L.: Efficient identification of web communities. In: Proc. Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 150–160 (2000)
13. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: Proc. of the 31st international conference on Very large data bases (VLDB), pp. 721–732 (2005)
14. Goldberg, A.: Finding a maximum density subgraph. Tech. rep., University of California at Berkeley (1984). URL `http://nma.berkeley.edu/ark:/28722/bk000570k8g`
15. Guha, S., Meyerson, A., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering data streams: Theory and practice. IEEE Transactions on Knowledge and Data Engineering **15**, 515–528 (2003)
16. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proc. ACM SIGMOD international conference on Management of data, pp. 1–12 (2000)
17. Hartline, J., Sharp, A.: An incremental model for combinatorial maximization problems. In: Proc. 5th International Workshop on Experimental Algorithms, pp. 36–48 (2006)
18. Hartline, J., Sharp, A.: Incremental flow. Networks **50**(1), 77–85 (2007)
19. Hill, S., Agarwal, D.K., Bell, R., Volinsky, C.: Building an effective representation for dynamic networks. Journal of Computational and Graphical Statistics **15**(3), 584–608 (2006)
20. Khuller, S., Saha, B.: On finding dense subgraphs. In: Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP), pp. 597–608 (2009)
21. Kim, M.S., Han, J.: Chronicle: A two-stage density-based clustering algorithm for dynamic networks. In: Discovery Science, pp. 152–167 (2009)
22. Kumar, S., Gupta, P.: An incremental algorithm for the maximum flow problem. J. Math. Model. Algorithms **2**(1), 1–16 (2003)
23. Lawler, E.L.: A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Management Science **18**(7), 401–405 (1972)
24. Long, J., Hartman, C.: ODES: an overlapping dense sub-graph algorithm. Bioinformatics **26**(21), 2788–2789 (2010)
25. Mathioudakis, M., Koudas, N.: Twittermonitor: trend detection over the twitter stream. In: SIGMOD Conference, pp. 1155–1158 (2010)
26. Pardalos, P.M., Xue, J.: The maximum clique problem. Journal of Global Optimization **4**, 301–328 (1994)
27. Sarkas, N., Angel, A., Koudas, N., Srivastava, D.: Efficient identification of coupled entities in document collections. In: Proc. ICDE Conference, pp. 769–772 (2010)
28. Stix, V.: Finding all maximal cliques in dynamic graphs. Computational Optimization and Applications **27**, 173–186 (2004)
29. Uno, T.: An efficient algorithm for solving pseudo clique enumeration problem. Algorithmica **56**, 3–16 (2010)
30. Wang, N., Parthasarathy, S., Tan, K.L., Tung, A.K.H.: Csv: visualizing and mining cohesive subgraphs. In: Proc. of the 2008 ACM SIGMOD international conference on Management of data, pp. 445–458 (2008)
31. Yang, D., Rundensteiner, E.A., Ward, M.O.: Neighbor-based pattern detection for windows over streaming data. In: Proc. EDBT Conference, pp. 529–540 (2009)