

Mining Unstructured Social Streams: Cohesion, Context and Evolution

by

Pei Li

B.Eng., HuaZhong University of Science & Technology, 2007

M.Eng., Renmin University of China, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2017

© Pei Li 2017

Abstract

As social websites like Twitter greatly influence people’s digital life, unstructured social streams become prevalent, which are fast surging textual post streams without formal structure or schema between posts or inside the post content. Modeling and mining unstructured social streams in Twitter become a challenging and fundamental problem in social web analysis, which leads to numerous applications, e.g., recommending social feeds like “what’s happening right now?” or “what are related stories?”. Current social stream analysis in response to queries merely return an overwhelming list of posts, with little aggregation or semantics. The design of the next generation social stream mining algorithms faces various challenges, especially, the effective organization of meaningful information from noisy, unstructured, and streaming social content.

The goal of this dissertation is to address the most critical challenges in social stream mining using graph-based techniques. We model a social stream as a post network, and use “event” and “story” to capture a group of aggregated social posts presenting similar content in different granularities, where an event may contain a series of stories. We highlight our contributions on social stream mining from a structural perspective as follows. We first model a story as a quasi-clique, which is cohesion-persistent regardless of the story size, and propose two solutions, DIM and SUM, to search the largest story containing given query posts, by deterministic and stochastic means, respectively. To detect all stories in the time window of a social stream and support the context-aware story-telling, we propose CAST, which defines a story as a (k, d) -Core in post network and tracks the relatedness between stories. We propose Incremental Cluster Evolution Tracking (ICET), which is an incremental computation framework for event evolution on evolving post networks, with the ability to track evolution patterns of social events as time rolls on. Approaches in this dissertation are based on two hypotheses: users prefer correlated posts to individual posts in post stream modeling, and a structural approach is better than frequency/LDA-based approaches in event and story modeling. We verify these hypotheses by crowdsourcing based user studies.

Preface

The modeling of social streams discussed in Chapter 3 is primarily based on our publication at KDD 2013 (KeySee [42], a system that supports keyword search on social events). The work on cohesion-persistent story search presented in Chapter 4 is based on our publication at SDM 2016 [41]. The context-aware story-teller (CAST) discussed in Chapter 5 is mainly based on our publication at CIKM 2014 [43], and partly based on our publication at ICDE 2012 [45]. The work on incremental cluster evolution tracking (ICET) presented in Chapter 6 is based on our paper at ICDE 2014 [44]. All of the above mentioned publications [41–45] are collaborated with my supervisor Prof. Laks V. S. Lakshmanan from the University of British Columbia, Canada. Publications [42–44] are also collaborated with Prof. Evangelos Milios from Dalhousie University, Canada.

Table of Contents

Abstract	ii
Preface	iii
Table of Contents	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Unstructured Social Streams	1
1.1.1 Concept and Modeling	1
1.1.2 Mining Unstructured Social Streams	4
1.2 Opportunities and Challenges	8
1.2.1 Cohesion-Persistent Story Search	8
1.2.2 Story Context Mining	9
1.2.3 Event Evolution Tracking	10
1.2.4 Evaluation of Mining Tasks	11
1.3 Contributions and Research Plan	12
1.3.1 Query-Driven Quasi-Clique Maximization	12
1.3.2 Context-Aware Story-Telling	13
1.3.3 Incremental Event Evolution Tracking	14
1.3.4 Targeted Crowdsourcing	15
1.4 Thesis Outline	16
2 Related Work	17
2.1 Graph Mining	17
2.1.1 Graph Clustering	17
2.1.2 Community Detection and Search	18
2.1.3 Dense Subgraph Mining	18
2.1.4 Subgraph Relatedness Computation	20

Table of Contents

2.2	Social Media Analytics	20
2.2.1	Frequency Based Peak Detection	21
2.2.2	Entity Network Based Approaches	21
2.3	Topic Detection and Tracking	21
3	Modeling Unstructured Social Streams	23
3.1	Motivation	23
3.2	Social Stream Preprocessing	24
3.3	Post Network Construction	25
3.3.1	Post Similarity Computation	25
3.3.2	Post Network	25
3.3.3	Linkage Search	26
3.4	Stories and Events	27
3.4.1	Stories and Events in Social Streams	27
3.4.2	Stories and Events in Post Network	28
3.5	Context and Evolution	31
3.6	Comparing with Other Modeling Methods	33
3.7	Discussion and Conclusion	35
4	Cohesion-Persistent Story Search	37
4.1	Introduction	37
4.2	Problem Overview	40
4.3	Pre-solution on Core Tree	42
4.3.1	Core Tree: Definition and Properties	43
4.3.2	Search for Pre-Solution	47
4.4	Query-Driven Quasi-Clique Maximization	50
4.4.1	Objective and Operations	50
4.4.2	Efficient Solution Search and Rank	52
4.4.3	Iterative Maximization Algorithms	54
4.5	Experimental Study	58
4.5.1	Core Tree Construction	58
4.5.2	Performance Evaluation	59
4.6	Discussion and Conclusion	63
5	Context-Aware Story-Telling	66
5.1	Introduction	66
5.2	Story Vein	68
5.3	Transient Story Discovery	69
5.3.1	Defining a Story	69
5.3.2	Story Formation	72

Table of Contents

5.4	Story Context Tracking	73
5.4.1	Story Relatedness Dimensions	74
5.4.2	Story Context Search	76
5.4.3	Interpretation of Story Vein	79
5.5	Experimental Study	80
5.5.1	Tuning Post Network	80
5.5.2	Quality Evaluation	81
5.5.3	Performance Testing	85
5.6	Discussion and Conclusion	87
6	Incremental Event Evolution Tracking	89
6.1	Introduction	89
6.2	Problem Formalization	92
6.3	Incremental Tracking Framework	93
6.4	Skeletal Graph Clustering	94
6.4.1	Node Prioritization	95
6.4.2	Skeletal Cluster Identification	97
6.5	Incremental Cluster Evolution	97
6.5.1	Fading Time Window	97
6.5.2	Network Evolution Operations	98
6.5.3	Skeletal Graph Evolution Algebra	99
6.5.4	Incremental Cluster Evolution	101
6.6	Incremental Algorithms	103
6.7	Experiments	104
6.7.1	Tuning Skeletal Graph	105
6.7.2	Cluster Evolution Tracking	106
6.7.3	Running Time of Evolution Tracking	112
6.8	Discussion and Conclusion	113
7	Crowdsourcing-Based User Study	116
7.1	Introduction	116
7.2	Related Work	117
7.3	Hypotheses in Social Stream Mining	119
7.4	Quality Control for Crowdsourcing	122
7.5	Experiments	128
7.6	Discussion and Conclusion	131
8	Summary and Future Research	133
8.1	Summary	133
8.2	Future Research	134

Table of Contents

Bibliography 136

List of Tables

3.1	Event evolution patterns	33
4.1	Notation.	40
4.2	Three Maximization Operations.	51
4.3	Running time for core tree construction.	65
4.4	Aggregated quality scores for different methods from 100 tests.	65
5.1	Major Notations.	68
5.2	The number of edges in the post network, and the number of stories and relatedness links in the story vein as the changing of temporal proximity functions. We define a story as a (5, 3)-Core.	81
5.3	Top 20 stories detected by LDA from Tech-Full and described by high frequency words. We treat top 100 posts of each topic as a story.	83
5.4	The accuracy of RCS, as the number of simulations n grow. We define n as a number proportional to the neighboring post size of story S and measure the accuracy based on DCS.	88
6.1	Notation.	91
6.2	Tuning post network.	115

List of Figures

1.1	The illustration of major components in a social website. . . .	2
1.2	The similarity between posts in a time window of social streams is captured by a post network. Each story or event in social streams can be modeled as a specifically defined subgraph in post network.	3
1.3	An illustration of an event and its included stories. This event is about “MH370 Search” and has a clear evolution history, i.e., <i>emerge</i> , <i>grow</i> and <i>decay</i> , in a time span of three months. This event includes several stories which are related.	6
4.1	The query S is marked by solid nodes, with $\lambda = 0.5$. A maximal quasi-clique and the query-driven maximum quasi-clique (QMQ) are circled by a dotted ellipse and a solid ellipse, respectively.	41
4.2	Architecture of the efficient query-driven maximum quasi-clique search by DSG tree.	43
4.3	A graph G with k -Cores identified recursively in (a) and its corresponding core tree in (b).	44
4.4	(a) A small graph shown and (b) its corresponding core tree, showing deepest tree nodes containing graph nodes v_1, \dots, v_5 ; G_5 and G_6 in (b) are annotated by dotted circles in (a). . . .	49
4.5	(a) An example for the QMQ maximization. (b) Illustrating the inflection node on $\mathbb{F}(x)$, where solutions with rank $x \leq x^*$ are preferred.	57
4.6	The number of maximal cores first rapidly increases, and then slowly decreases with the coreness k on three data sets. . . .	59
4.7	(a) Running time of 1000 pre-solution queries as the increasing of query set size. (b) Portion of pre-solution chosen from G_l , out of all pre-solutions, for different query set sizes. $\lambda = 0.9$. .	60

List of Figures

4.8	(a) Running time of <i>Add</i> and Add-MC for different solution sizes, with query node size $ S = 3$; (b) ratio between the search spaces of <i>Add</i> and Add-MC, given current solution, as query size increases; (c) average #iterations for various methods on LiveJournal data set; (d) average running time of different methods on LiveJournal, as query size increases; $\lambda = 0.9$ by default.	65
5.1	Illustrating the workflow of StoryVein , which has three major steps: (1) post network construction, (2) transient story discovery and (3) story context search.	68
5.2	(a) A 3-Core without similarity witness between p_1 and p_2 . (b) The illustration of generating a (3, 1)-Core from a 3-Core.	71
5.3	Top 10 results of HashtagPeaks, MentionPeaks and EntityPeaks on Tech-Lite dataset. The ground truth for precision and recall is top 10 major stories selected from main stream technology news websites.	82
5.4	Top 10 transient stories generated by our proposed story-teller on Tech-Lite. Each story is represented as a (5,3)-Core, and we render a story into an entity cloud for human reading. Some transient stories may be related, as linked by lines. The curve on the bottom is the breakdown of tweet frequency on each day in Jan 2012.	84
5.5	A fragment of story vein tracked from CNN-News, which has a time span from January 1 to June 1, 2014.	85
5.6	An example to illustrate our context-aware story-teller. Each tag cloud here is a single story identified from the post network. Sets of stories with higher relatedness are grouped together in rectangles to aid readability.	86
5.7	(a) Running time of $(d + 1)$ -Core generation and $(d + 1, d)$ -Core generation (Zigzag , NodeFirst). (b) The number of connected components generated by $(d + 1)$ -Cores and $(d + 1, d)$ -Cores. (c) Running time of different context search approaches. All experiments are running on Tech-Full dataset with the time window set to one week.	87

List of Figures

6.1	Post network captures the correlation between posts in the time window at each moment, and evolves as time rolls on. The skeletal graph is shown in bold. From moment t to $t + 1$, the incremental tracking framework will maintain clusters and monitor the evolution patterns on the fly.	90
6.2	(a) The commutative diagram between dynamic networks G_t, G_{t+1} and cluster sets S_t, S_{t+1} . The “divide-and-conquer” baseline and our <i>Incremental Tracking</i> are annotated by dotted and solid lines respectively. (b) The workflow of incremental tracking module, which shows our framework tracks cluster evolution dynamics by only consuming the updating subgraph ΔG_{t+1}	93
6.3	The functional relationships between different types of objects defined in this paper, e.g., the arrow from G_t to \overline{G}_t with label <i>Ske</i> means $\overline{G}_t = Ske(G_t)$. Refer to Table 6.1 for notations.	95
6.4	An illustration of the fading time window from time t to $t + 1$, where post priority may fade w.r.t. the end of time window. G_t will be updated by deleting subgraph G_{old} and adding subgraph G_{new}	98
6.5	(a) The relationships between primitives and evolutions. Each box represents an evolution object and the arrows between them describe inputs/outputs. (b) The evolutionary behavior table for clusters when adding or deleting a core post p	101
6.6	The trends of the number of core posts, core edges and events when increasing δ from 0.3 to 0.8. We set $\delta = \varepsilon = 0.3$ as the 100% basis.	105
6.7	Examples of Google Trends peaks in January 2012. We validate the events generated by cTrack by checking the existence of volume peaks at a nearby time moment in Google Trends. Although these peaks can detect bursty events, Google Trends cannot discover the merging/splitting patterns.	107
6.8	Lists of top 10 events detected from Twitter Technology streams in January 2012 by baseline HashtagPeaks, UnigramPeaks, Louvain and our incremental tracking approach eTrack.	107
6.9	The merging and splitting of “SOPA” and “Apple”. At each moment, an event is annotated by a word cloud. Baselines 1 and 2 only works for the detection of new emerging events, and is not applicable for the tracking of merging and splitting dynamics. The evolution trajectories of and Baseline 3 are depicted by solid and hollow arrows respectively.	111

List of Figures

6.10	The running time on two datasets as the adjusting of the time window length and step length.	112
7.1	Crowdsourcing task for Hypothesis 1.	120
7.2	Crowdsourcing task for Hypothesis 2.	121
7.3	The workflow of quality control steps.	124
7.4	(a) A voting example with 3 workers and 3 questions, where each question has two options: A and B. (b) The computation of normalized quality vector q on each iteration, where quality scores on iteration 0 are obtained from the qualification test. (c) and (d): The distributions of quality weights among options for each question on iteration 1 and 15, respectively. .	127
7.5	Qualification test sample used before real crowdsourcing tasks.	129
7.6	Running EMQ for verifying Hypothesis 1 and 2. In (b), the percentage of weighted votes on options D/E increase from 74.4% (before EMQ) to 83.4% (after EMQ). In (c), the percentage of weighted votes on options D increase from 56.7% (before EMQ) to 75.4% (after EMQ).	130

Chapter 1

Introduction

As social streaming websites like Twitter become popular and gradually dominate people’s digital life, the current Web is in the *social* age. The information propagation channel in the social web age can be viewed as post streams along the timeline, where each post is a tweet in Twitter. We call these post streams as *unstructured social streams*, since there is no formal structure or schema between these posts or inside a post. Modeling and mining the unstructured social streams becomes a fundamental problem in social web analysis, which leads to numerous applications [48, 55, 66, 77], e.g., answering “what’s happening now?” on Twitter. In this chapter, we first provide an overview on the modeling of unstructured social streams. Then, we discuss the opportunities, challenges, and our contributions in various mining tasks of unstructured social streams.

1.1 Unstructured Social Streams

In this section, we first scope out the definition and position of unstructured social streams in the framework architecture of a social website, and then, we briefly introduce the modeling of a social stream as a post network. Following that, we discuss the major tasks and challenges in social stream mining, and provide a reasoning on why we adopt structural approach rather than other social stream mining approaches, like content or frequency based approaches.

1.1.1 Concept and Modeling

Social Website Components. While a precise and widely recognized definition of social streams is still missing in the literature, we try to refine its scope in the context of social website. Here, a *social website* generally refers to a website that supports the interaction among people in a social network, in which they create and share information. Twitter and Weibo are two typical social websites, where users are connected in virtual online communities, and short textual posts are created and shared among them.

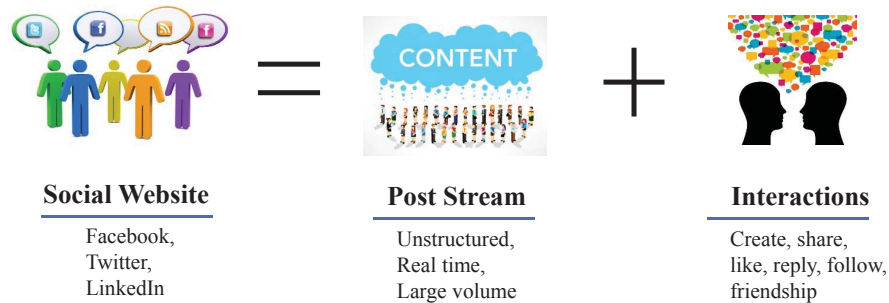


Figure 1.1: The illustration of major components in a social website.

In Figure 1.1, we decompose a social website into two major components: post stream and interactions. Each of them is explained below.

- *Post Stream.* In social websites, posts are the atomic form of the content created by users and propagated across social networks. The length of each post is usually very short, e.g., a tweet only has at most 140 characters. Post streams are also unstructured, since the majority of posts do not have any relationships between them and there is no schema for the content of a post. In most popular social websites, the volume of posts surges very quickly, e.g., Twitter reached a peak of 143,199 tweets per second¹ on August 3, 2013.
- *Interactions.* The interactions in a social website may happen between different types of objects. The typical interaction happens between the user and a post, e.g., a user creates a post. Interactions may also happen between a post and a post. For example, the “reply” refers to the interaction between two posts, and two posts without “reply” relationship may be semantically interacted if they tell the same story.

Unstructured Social Streams. Conceptually, we call the textual post streams, e.g., daily updates, on a social website as social streams. We describe a social post by three types of information: author, text content and creation time. We define a social stream as a first-in-first-out queue of posts ordered by time of creation, in which each post is associated with the text content and a timestamp. An illustration of the social stream in a time window can be found in Figure 1.2. Mining social streams is highly challenging, with the difficulties originating from two important properties of social streams:

¹<https://blog.twitter.com/2013/new-tweets-per-second-record-and-how>

1.1. Unstructured Social Streams

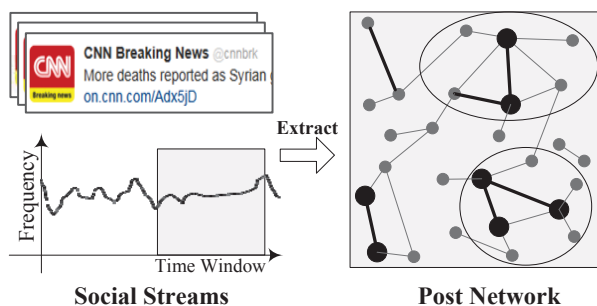


Figure 1.2: The similarity between posts in a time window of social streams is captured by a post network. Each story or event in social streams can be modeled as a specifically defined subgraph in post network.

- *Unstructured.* The unstructured nature of social streams has two aspects. First, the text content inside a post is unstructured. Social posts such as tweets are usually written in an informal way with lots of grammatical errors, and even worse, a correctly written post may have no significance and be just noise. The design of a processing strategy that can quickly judge what a post talks about is a challenging problem. Second, the relationship between posts is usually unstructured. Our statistics on Twitter Tech-Full data set (described in Chapter 5.5) shows that only 5% tweets are retweets or replies, making the majority of tweets independent with each other, even though they may talk about similar events or stories in the social website. The effective organization of these unstructured posts in social streams is a very challenging problem.
- *Streaming.* The streaming nature of social streams also poses two challenges. First, in popular social websites, new posts emerge quickly in every second, e.g., Twitter generates thousands of tweets per second. The design of social stream mining algorithms should be efficient enough to handle the quick surge of new posts in a timely manner, e.g., it should use a single-scan algorithm with linear scalability. Second, since it is impossible and unnecessary to process all historical posts, old posts should be naturally outdated as the time rolls on. The design of a time window with proper time-decay effect is essential to handle the streaming data. This time window can be regarded as the window of observation.

Modeling Social Stream as Post Network. In this thesis, we propose an innovative approach to quickly transform an unstructured social stream into

a well-defined structure, called post network. As illustrated in Figure 1.2, we monitor social streams using a sliding time window with length Len . At any moment t , all posts generated in the time window $[\max\{t - Len, 0\}, t]$ constitute the snapshot of current social streams. We transform a social stream into a post network by the following rule: a post network at moment t can be defined as a graph $\mathcal{G}_t(\mathcal{V}_t, \mathcal{E}_t)$, where each node $p \in \mathcal{V}_t$ is a post in the snapshot, and an edge $(p_i, p_j) \in \mathcal{E}_t$ is constructed if the similarity $S(p_i, p_j)$ between p_i and p_j is higher than a given threshold ε . As the time window moves forward, new posts flow in and old posts fade out, and then $\mathcal{G}_t(\mathcal{V}_t, \mathcal{E}_t)$ will be dynamically updated at each moment, with new nodes/edges added and old nodes/edges removed. As we can see, this transformation will make $\mathcal{G}_t(\mathcal{V}_t, \mathcal{E}_t)$ an evolving network as time rolls on.

The key challenge to construct the post network is how to compute $S(p_i, p_j)$ effectively and efficiently. Traditional similarity measures such as TF-IDF based cosine similarity, Jaccard Coefficient and Pearson Correlation [53] only consider the post content. However, time stamps should play an important role in determining post similarity, since posts created closer together in time are more likely to discuss the same story than posts created at very different moments. We define the similarity between a pair of posts p_i and p_j by combing both content similarity and temporal proximity. The exact form of the similarity function will be discussed in Chapter 3. The similarity score $S(p_i, p_j)$ will fall into the interval $[0, 1]$. The similarity threshold ε will be empirically set, where $0 < \varepsilon < 1$.

1.1.2 Mining Unstructured Social Streams

Major Tasks. People easily feel overwhelmed by the information deluge coming from social streams which flow in from channels like Twitter, Facebook/LinkedIn, forums, blog websites and email-lists. There is thus an urgent need for tools which can automatically extract and summarize significant information from highly dynamic social streams, e.g., report emerging bursty events, or track the evolution of one or more specific events in a given time span. Imagine that a user called Bob follows a few news media (e.g., the CNN Breaking News channel), which feed him a social stream consisting of thousands of tweets per day. Bob does not have time to digest these tweets one-by-one, but he wants to keep synced up with the new emerging stories diffused on these information channels. The major tasks of social stream mining, can be viewed as answering the following typical queries on social streams:

- *What's trending now?* Every morning at breakfast, Bob wants to keep updated on the new events or stories emerging in his social streams. He got thousands of new tweets pushed from his followees during last night and he does not have the time or interest to read so many short and informally-written posts.
- *Tell me related news.* One day morning in 2014, Bob is reading a breaking news about "Annexation of Crimea by the Russian". Bob is unclear about the context of this story and wants to check more related stories.
- *How're things going?* Bob has an interest in the event "Ukraine Crisis" and follows this event for several months. Every week, there are some new stories happening related to the Ukraine crisis. In January 2014, Bob is on vocation and does not follow this event any more. When he is back from his vocation, Bob wants to know how the Ukraine crisis event evolved in the past month.

The answering of these key queries needs to scope out two questions: (1) how to effectively organize these meaningful information in posts? (2) How to capture the behaviors of these meaningful information in social streams? For the first question, we define story and event as two structures to organize meaningful information in posts. For the second question, we introduce cohesion, context and evolution to capture the evolution behaviors of events and stories. They are elaborated below.

Story and Event. Before the development of some highly intelligent algorithms to answer these queries, we need to define several basic concepts that support the query. In this dissertation, story and event are two basic concepts we use to aggregate and organize posts in social streams with similar meaningful information together. In related work [55, 66, 67], there are many different definitions for story and event, and some studies even use these two terms interchangeably, without proper distinction. In the following, we clearly distinguish them and define story and event from the conceptual level.

- *Story.* A story is a set of posts that talk about very similar information in a short time span. Usually, the information carried by a story can be described by a few sentences or a small set of keywords. Since posts in social streams are usually very short, we consider a post describes at most a story in most cases. However, a story can be described by many posts, e.g., two posts "Crimea was annexed by the Russian Federation" and "Crimea voted on 6 March to formally accede as part of the Russian Federation" are talking about the same story about the annexation of

1.1. Unstructured Social Streams

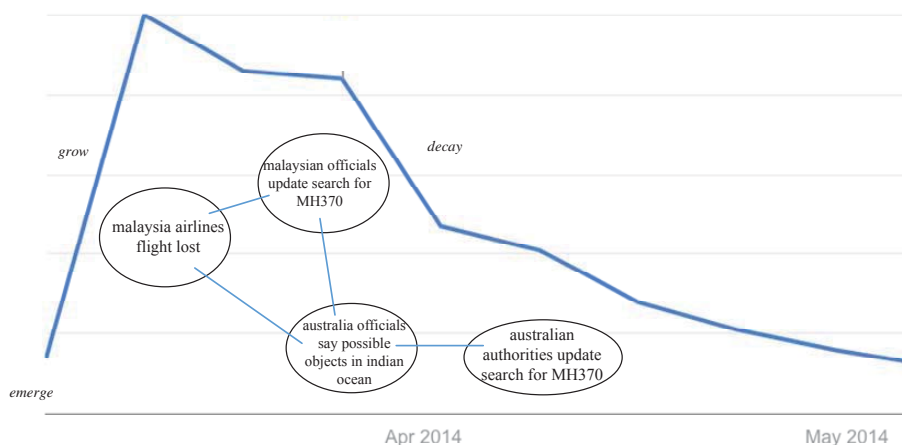


Figure 1.3: An illustration of an event and its included stories. This event is about “MH370 Search” and has a clear evolution history, i.e., *emerge*, *grow* and *decay*, in a time span of three months. This event includes several stories which are related.

Crimea by the Russian Federation in March 2014.

- *Event*. An event is a post cluster which contains many highly related stories, where the relatedness between stories is measured on both the content similarity and the time closeness. Typically, an event has a relatively long time span and follows clear evolution patterns, e.g., emerging, growing and decaying. A typical event contains lots of details and cannot be described by a short post. For example, “Ukrainian crisis” is an event, which consists of a series of related stories, e.g., “Crimean Crisis”, “War in Donbass”, “Ukraine President Elections”, etc.

Since a social stream can be modeled as a post network, stories and events correspond to subgraphs in the post network. As illustrated in Figure 1.3, we show the relationship between an event and its included stories, where stories are related with each other. Notice that in this section, stories and events are defined on the conceptual level. The detailed technical definitions for events and stories on the post network will be given in Chapter 3.

Cohesion, Context and Evolution. To capture the behaviors of events and stories, we briefly introduce *story cohesion*, *story context* and *event evolution*, as explained below. The detailed studies on them will be scattered in Chapter 4, Chapter 5 and Chapter 6.

- *Story Cohesion*. Given a set of posts which may correspond to a story,

the cohesion of this post set measures the likelihood of this post set telling the same story. Since we can measure the similarity between any two posts, the cohesion of a post set can be computed by aggregating the pairwise post similarity inside this post set. For example, a common way to define the cohesion is the ratio between the minimum degree of a post in the story and the story size. We only treat a post set with enough high cohesion as a story. Obviously, cohesion is an intrinsic feature of a story.

- *Story Context.* In contrast to the cohesion, story context is an extrinsic feature that is *between* stories. Story context tries to measure how strongly two given stories are related. Since we can compute the similarity between two posts in different stories, a natural way to measure the relatedness between two stories is by assessing the post similarity between them, potentially normalized by the sizes of stories.
- *Event Evolution.* An event usually consists of a series of stories, which allow a relatively-long time span (e.g., weeks or even months) with evolution patterns. Event evolution tries to capture the evolution path of an event, and typical evolution patterns include *emerge*, *grow*, *decay*, *disappear*, etc.

Difficulties in Social Stream Mining. Social stream mining is a category of difficult data mining problems. The aim of social stream mining is to fulfill people's information seeking needs on social streams. Typical social stream mining tasks include detecting new stories, searching for related stories and tracking the evolution of trending events. The design of social stream mining algorithms faces the following difficulties:

- *Dynamics.* The quick surge of social post streams makes the data updated very frequently, so the social stream mining algorithms should be able to handle the dynamic nature of the social streams effectively and efficiently.
- *Quality.* A large number of posts in social streams are noise or redundant. An effective social stream mining algorithm should be able to combat noise, condense redundant content and aggregate small pieces of information conveyed by each short post together.
- *Scalability.* The huge amount of posts generated by users every day raises huge challenges on the scalability of the social stream mining algorithms.

1.2 Opportunities and Challenges

Unstructured social streams are the primary data source for information exchange in social websites. It is noisy and surges quickly, with meaningful information hidden deeply inside the informally writing text content. In this section, we discuss the opportunities and challenges in the mining of unstructured social streams. First, we define a story as a cohesion-persistent subgraph in post network, which makes sure all social posts in the same story are highly related with each other. We then propose the story context search to find related stories of a given story, which allows us to build the relatedness between stories in social streams. An event is defined as a post cluster which contains many highly-related stories. The tracking of event evolution patterns is an interesting problem and we discuss the challenges to perform event evolution tracking in social streams. These approaches are supported by two hypotheses: users prefer correlated results to individual results in social stream modeling, and structural approach is better than frequency/LDA-based approaches in event and story modeling. We discuss the challenges in user studies to verify these hypotheses.

1.2.1 Cohesion-Persistent Story Search

Since a post in social streams can be modeled as a node and two posts with high similarity will be connected by an edge, a social stream can be transformed into a post network. As a result, a story in social streams corresponds to a connected subgraph in the post network. In the post network, we define a story as a connected subgraph $G_i(V_i, E_i)$ with enough high cohesion, where the cohesion $\mathcal{C}(G_i)$ is defined as the ratio between the minimum degree and the maximum possible degree in G_i . Supposing $deg(v, G_i)$ is the degree of node v in G_i , we have $\mathcal{C}(G_i) = \frac{\min deg(v, G_i)}{|V_i|-1}$ for any node $v \in V_i$.

There are several reasons why we use cohesion to define a story. The root cause is similar posts are connected together in the post network. If two posts are similar with each other in both the text content and the creation time, it is likely that these two posts are telling the same story, and they will be connected by an edge in the post network. A story in a social stream is a subgraph in the post network. If nodes in a subgraph are highly connected with each other, it is very likely that the posts corresponding to this subgraph are telling the same story. We argue that, *cohesion* is an effective way to guarantee every post in the story is similar to the majority of posts in the same story, regardless of the story size.

Clearly, a story is a special kind of dense subgraph. In related work, there

are many definitions for dense subgraphs, e.g., quasi-clique, k -Plex, k -Core [47] and k -Truss [34]. Supposing $N(p)$ is the neighbor set of node $p \in V_i$ in $G_i(V_i, E_i)$, these dense subgraphs are defined as:

- Quasi-clique: $|N(p)| \geq \lambda(|V_i| - 1)$ for any post $p \in V_i$ and $0 < \lambda \leq 1$;
- k -Plex: $|N(p)| \geq |V_i| - k$ for every post $p \in V_i$;
- k -Core: $|N(p)| \geq k$ for every post $p \in V_i$;
- k -Truss: for every edge $(p_i, p_j) \in E_i$, $|N(p_i) \cap N(p_j)| \geq k$.

The cohesion of a k -Plex can be very low if its size is very close to k . The cohesion of a k -Core can be also very low if its size is very larger than k , similarly for k -Truss. Out of them, only for λ -quasi-cliques, the cohesion is always at least λ as the subgraph size changes. The intuition of a cohesion-persistent story is best captured by a λ -quasi-clique, as the degree of nodes grows with the node size in a quasi-clique. Given several posts already read by the user as the input, the finding of the cohesion-persistent story containing these posts is called the cohesion-persistent story search problem. On the post network, the cohesion-persistent story search problem is actually the query-driven quasi-clique search problem.

To the best of our knowledge, there is no existing studies on the query-driven quasi-clique search problem. existing studies on quasi-clique maximization (without any query) are mainly based on local search [15, 33, 62], in which a solution moves to the best neighboring solution iteratively, updated node-by-node. To apply local search on the query-driven quasi-clique search problem, we face two challenges: (1) the efficient finding of an initial solution, (2) efficient iterative maximization approaches for searching better neighboring solutions. In addition, existing local search methods are usually based on deterministic heuristics, which may easily trap the optimization process into a local maximum. Thus, we face new challenges of developing randomized algorithms to find the largest query-driven quasi-cliques.

1.2.2 Story Context Mining

There are many previous studies [48, 55, 66, 77] on detecting new emerging stories from social streams. Since these stories detected by previous studies are not cohesion-persistent, we proposed cohesion-persistent story in Section 1.2.1. However, all these story detection approaches only serve the need for answering “*what’s happening now?*” in social streams, and are not able to find the relatedness between stories. In reality, stories usually do not happen in isolation, and the recommendation of related stories will greatly enhance the user’s experience and improve the participation. For example, “Crimea votes to join Russia” (on May 6, 2014) and “President Yanukovich signs

compromise” (on February 21, 2014) are two separate stories, but they are actually highly related under the same event “Ukraine Crisis”. The *context-aware* story-telling for streaming social content not merely detects trending stories in a given time window of observation, but also builds the “context” of each story by measuring its relatedness with other stories on the fly. As a result, context-aware story-telling has advantages on answering advanced user queries like “*tell me related stories*”, which is crucial to help digest large volume social streams. Context-aware story-telling on social streams raises the following challenges:

- *Identification of transient stories from time window.* Story detection should be robust to noisy posts and efficient enough to support single-pass tracking, which is essential in the streaming environment.
- *Story context search on the fly.* Story relatedness computation should be efficient to find related stories of a given story, and interpretable to build a story graph that supports the story-telling to users.

To the best of our knowledge, there is no publicly available training data set for the context-aware story-telling on social streams, which makes the existing studies [59, 68] on Story Link Detection (SLD) not applicable, because SLD is trained on well-written news articles. Furthermore, we cannot apply topic tracking techniques (e.g., [32]) to story context search, because topic tracking is usually formulated as a classification problem [4], with an assumption that topics are predefined before tracking, which is unrealistic for story context search on social streams. All these constraints make the mining of story contexts on social streams an extremely challenging problem.

1.2.3 Event Evolution Tracking

People easily feel overwhelmed by the information deluge coming from highly dynamic social streams. There is thus an urgent need to provide users with tools which can automatically extract and summarize significant information from social streams, e.g., report emerging bursty events, or track the evolution of one or more specific events in a given time span. In this thesis, an event is defined as a group of posts that contains many related stories. There are several previous studies [10, 26, 48, 55, 66, 67] on detecting new emerging events from text streams, designed to answer simple queries like “*what’s trending now?*”. However, in many scenarios, users are dissatisfied with only providing them new emerging events. Instead, users may want to know the evolution history of an event and like to issue advanced queries like “*how’re things going?*”. The ideal answer of such a query would be a “panoramic view”

of the event, which improves user experience greatly. Here, the panoramic view of an event means the whole evolution life cycle of an event, including primitive operations like *emerge*, *grow*, *decay* and *disappear* and composite operations like *merge* and *split*. Technically, we can model social streams as dynamically evolving post networks and model events as clusters in these networks, obtained by means of a clustering approach that is robust to the large amount of noise present in social streams. Accordingly, we consider the above kind of queries as an instance of the *event evolution tracking* problem, which aims to track the evolution patterns of events at each moment in such dynamic post networks.

In many scenarios, social streams are of large scale and evolve quickly. There are several major challenges in event evolution tracking:

- The first challenge is the effective design of incremental computation framework for event evolution tracking. Traditional approaches (e.g., [38]) based on decomposing a dynamic network into snapshots and processing each snapshot independently from scratch are prohibitively expensive. An efficient single-pass incremental computation framework is essential for event evolution tracking over social streams that exhibit very large throughput rates. To our knowledge, the event evolution problem has not yet been studied.
- The second challenge is the formalization and tracking of event evolution operations under an incremental computation framework, as the network evolves. Most related work reports event activity by volume over the time dimension [48, 55]. While certainly useful, this is just not capable of showing the composite evolution behaviors about how events split or merge, for instance.
- The third challenge is the handling of bulk updates. Since dynamic post networks may change rapidly, a node-by-node approach to incremental updating will lead to poor performance. A subgraph-by-subgraph approach to incremental updating is critical for achieving good performance over very large, fast-evolving dynamic networks such as post networks. But this in turn brings the challenge of incremental cluster maintenance against bulk updates on dynamic networks.

1.2.4 Evaluation of Mining Tasks

The approaches proposed in this dissertation are based on two hypotheses: users prefer related results to individual results in social stream modeling, and structural approach is better than frequency/LDA-based approaches in

event and story modeling. User study is an effective way to verify a hypothesis about user preferences and satisfaction. Traditional user study involves lots of human labor work. With the rising of the Internet, crowdsourcing has recently become a popular mechanism behind user studies. Amazon Mechanical Turk (MTurk) is the most popular crowdsourcing marketplace. As recorded in 2014², there are over 500,000 workers from over 190 countries. Besides the normal MTurk workers, it is a well-known fact that there are lots of spammers and bots among Amazon workers.

In this user study, we try to perform hypotheses verification for the social stream mining through crowdsourcing tasks on Amazon Mechanical Turk (MTurk). Given the fact that there are a large number of bots and spammers on MTurk, the most critical challenge is the quality control in crowdsourcing. Existing techniques on quality control includes majority voting, minimum time constraint, etc. However, none of them can solve the “smart spammer” problem, in which workers pass the qualification test but perform like a spammer simply to get the reward with minimal work. Especially, since these smart spammers are still qualified workers for crowdsourcing tasks, none of existing approaches can detect them effectively. All these facts make the user quality control in crowdsourcing a very challenging problem.

1.3 Contributions and Research Plan

Section 1.2 explained the challenges in cohesion-persistent story search, story context mining, event evolution tracking and quality control in user studies. In this section, we briefly introduce the main ideas behind the solutions we propose to conquer these challenges, and summarize the contributions we made in this dissertation.

1.3.1 Query-Driven Quasi-Clique Maximization

As discussed in Section 1.2.2, the cohesion of a k -Plex can be very low if its size is very close to k , and the cohesion of a k -Core or k -Truss can be also very low if its size is very larger than k . Thus, λ -quasi-clique is the best definition for cohesion-persistent stories in the post network, since every post in a λ -quasi-clique is similar to at least a portion λ (e.g., $\lambda = 0.9$) of other posts. Recall that two posts are similar if they are talking about similar content and generated in close time moments. Given a query S which is a set of posts, the problem of finding the largest cohesion-persistent story that

²<http://docs.aws.amazon.com/AWSMechTurk/latest/RequesterUI/OverviewofMturk.html>

contains the query S is formalized as the *query-driven maximum quasi-clique* (QMQ) search, which aims to find the largest λ -quasi-clique containing S . The QMQ search problem is a new graph mining problem that is not studied before, and this problem is proved to be NP-Hard and inapproximable. To solve this problem, we propose the notion of *core tree* to organize dense subgraphs recursively, which reduces the search space and effectively helps find the solution within a few tree traversals. To optimize a currently available solution to a better solution, we introduce three maximization operations: *Add*, *Remove* and *Swap*. We propose two iterative maximization algorithms, DIM and SUM, to approach QMQ by deterministic and stochastic means respectively. With extensive experiments on real datasets, we demonstrate that our algorithms significantly outperform the state of the art algorithms in running time and/or the quality.

We make the following contributions:

- We define the problem of query-driven maximum quasi-clique search, a novel cohesive subgraph query not studied before, to solve the cohesion-persistent story search problem.
- We propose core tree as a recursive representation of a graph, which helps quickly find a tentative solution to the QMQ search problem within a few tree traversals by greatly reducing the solution search space.
- We introduce *Add*, *Remove* and *Swap* to search for new solutions and efficiently optimize a tentative solution to a better neighboring solution. Building on this, we propose deterministic and stochastic iterative maximization algorithms for QMQ search – DIM and SUM.
- We perform an extensive experimental study on three real datasets, which demonstrates that our algorithms significantly outperform several baselines in running time and/or the quality.

We will discuss the details of the QMQ search in Chapter 4.

1.3.2 Context-Aware Story-Telling

Mining transient stories and their relatedness implicit in social streams is a challenging task, since these streams are noisy and surge quickly. To address the challenges discussed in Section 1.2.2, we propose **CAST** [43], which is a Context-Aware Story-Teller specifically designed for streaming social content. **CAST** takes a noisy social stream as the input, and outputs a “story vein”, which is a human digestible and evolving summarization graph by linking highly related stories together. More precisely, we model the social stream as a post network, and define stories by a new cohesive subgraph

type called (k, d) -Core in the post network, in which every node should have at least k neighbors and two end nodes of every edge should have at least d common neighbors. (k, d) -Core is more compact than k -Core, and thus a better definition for stories than k -Core. Unlike quasi-cliques whose decision problem is NP-Hard, (k, d) -Core can be detected in polynomial time. We propose deterministic and randomized context search to support the iceberg query, which builds the relatedness between stories as social streams flow. We call the relatedness graph between stories as a story vein. We perform detailed experimental study on real Twitter streams and the results demonstrate the creativity and value of our approach.

The main contributions of **CAST** are summarized below:

- We define a new cohesive subgraph called (k, d) -Core to represent transient stories and propose two efficient algorithms, **Zigzag** and **Node-First**, to identify maximal (k, d) -Cores from the post network;
- Given a story, we propose deterministic and randomized context search to support the iceberg query for highly related stories, which builds the story vein on the fly;
- Our experimental study on real Twitter streams shows that story vein can be digested and effectively help build an expressive context-aware story-teller on streaming social content.

We will discuss the details of **CAST** in Chapter 5.

1.3.3 Incremental Event Evolution Tracking

In this thesis, an event is a post cluster which may contain many related stories. Since an event usually has a relatively long (e.g., weeks or even months) life cycle, the tracking of event evolution patterns in social streams can greatly help understand and summarize the event over the time. To implement the event evolution tracking, we propose Incremental Cluster Evolution Tracking (ICET, [44]), which focuses on tracking the evolution patterns of clusters in highly dynamic networks. There are several previous works on data stream clustering using a node-by-node approach for maintaining clusters. However, handling of bulk updates, i.e., a subgraph at a time, is critical for achieving acceptable performance over very large highly dynamic networks. We propose a subgraph-by-subgraph incremental tracking framework for cluster evolution in this thesis. To effectively illustrate the techniques in our framework, we consider the event evolution tracking task in social streams as an application, where a social stream and an event are modeled as a dynamic post network and a dynamic cluster respectively.

By monitoring through a fading time window, we introduce a skeletal graph to summarize the information in the dynamic network, and formalize cluster evolution patterns using a group of primitive evolution operations and their algebra. Two incremental computation algorithms are developed to maintain clusters and track evolution patterns as time rolls on and the network evolves. Our detailed experimental evaluation on large Twitter datasets demonstrates that our framework can effectively track the complete set of cluster evolution patterns from highly dynamic networks on the fly.

In summary, the problem we study is captured by the following questions: how to incrementally and efficiently track the evolution behaviors of clusters in large-scale networks, which are noisy and highly dynamic? Our main contributions are the following:

- We propose an incremental computation framework for cluster evolution on highly dynamic networks;
- We filter out noise by introducing a *skeletal graph*, based on which we define a group of primitive evolution operations for nodes and clusters, and introduce their algebra for incremental tracking;
- We leverage the incremental computation by proposing two algorithms based on bulk updating: ICM for the incremental cluster maintenance and eTrack for the cluster evolution tracking;
- Our application on event evolution tracking in large Twitter streams demonstrates that our framework can effectively track all kinds of cluster evolution patterns from highly dynamic networks in real time.

The details of ICET will be discussed in Chapter 6.

1.3.4 Targeted Crowdsourcing

The approaches proposed in this dissertation are based on two hypotheses: users prefer correlated posts to individual posts in social stream modeling, and structural approach is better than frequency/LDA-based approaches in event and story modeling. We use crowdsourcing-based user studies to verify these two hypotheses. To make sure the crowdsourcing-based user studies have a high quality, we use multiple techniques to control the quality of workers, as explained below.

- In the beginning, we set the Minimum Time Constraint and Approval Rate Constraint to filter out the MTurk workers who provide answers within a very short time and have a very low historical approval rate. Most of bots and spammers will be removed after this step.
- For the remaining workers, we perform the qualification test, which is

a series of questions with known answers. These qualification questions will be treated as the golden standard and worker’s qualification will be measured in terms of the ratio of questions answered correctly. If the ratio is higher than a predefined threshold, this worker will be treated as a qualified worker.

- All qualified workers will submit their work on the real crowdsourcing tasks. Since there is no known answer for the crowdsourcing tasks, the quality of workers will be measured by cross-comparison with peers in an iterative way, which is captured by Expectation-Maximization with Qualification (EMQ). EMQ is capable of measuring user’s quality in crowdsourcing and punishing Smart Spammers from among all qualified workers, by assigning low quality scores to them. By understanding these probabilities as users’ quality scores, EMQ achieves a better performance than other competing approaches.

The details of targeted crowdsourcing will be explained in Chapter 7.

1.4 Thesis Outline

The rest of this dissertation is structured as follows. Chapter 2 explores the related work. In Chapter 3, we introduce the preprocessing techniques on social streams and the modeling of a social stream as a post network. Chapter 4 studies the cohesion-persistent story search problem in social streams. Chapter 5 discusses the context-aware story-telling for streaming social content. In Chapter 6, we propose the incremental clustering evolution tracking, which is an incremental computation framework for event evolution tracking on social streams. Chapter 7 performs targeted crowdsourcing for ground-truth finding and hypothesis verification, with an extensive discussion on user quality control. In Chapter 8, we summarize this dissertation and list possible directions for future research in social stream mining.

Chapter 2

Related Work

Related work on mining unstructured social streams can be classified into three categories: (1) Graph mining, including graph clustering, community detection, dense subgraph mining, etc.; (2) Social media analytics, especially, event detection in Twitter streams; (3) Topic detection and tracking, which is traditionally studied on news articles and recently applied to social media data. In this chapter, we introduce the related work in each category.

2.1 Graph Mining

Graphs are seemingly ubiquitous to model the relationship between objects in many applications. When modeling an unstructured social stream as an evolving network of posts, graph mining becomes a powerful way to detect and track meaningful patterns in social streams. Related work of this dissertation in graph mining falls into one of the following topics: (1) Graph clustering; (2) Community detection and search; (3) Dense subgraph mining and (4) Subgraph relatedness computation. They are discussed below.

2.1.1 Graph Clustering

DBSCAN [30] is a density-based clustering method, which groups together points that are closely packed together. Compared with partitioning-based approaches (e.g., K-Means [30]) and hierarchical approaches (e.g., BIRCH [30]), density-based clustering (e.g., DBSCAN [30]) is effective in finding arbitrarily-shaped clusters, and is robust to noise. Application of these clustering approaches to a network that is changing with the time is a very challenging problem. CluStream [3] is a framework that divides the clustering process into an online component which periodically generates detailed summary statistics for nodes and an offline component which uses only the summary statistics for clustering. However, CluStream is based on K-Means only. DenStream [17] presents a new approach for discovering clusters in an evolving data stream by extending DBSCAN. DStream [18] uses an online component which maps each input data record into a grid and an offline

component which generates grid clusters based on the density. Another related work is by Kim et al. [38], which first clusters individual snapshots into quasi-cliques and then maps them over time by looking at the density of bipartite graphs between quasi-cliques in adjacent snapshots. Although [38] can handle birth/growth/decay/death of clusters, it is not incremental and the split and merge patterns are not supported. In contrast, our event tracking approach on dynamic post networks is incremental and is able to track composite behaviors like merging and splitting.

2.1.2 Community Detection and Search

Communities in graphs can be defined from global or local perspectives [25]. A community in global sense is a dense subgraph with very few ties to the outside of this subgraph, which is measured by “modularity”, a function which evaluates the goodness of graph partitioning. Louvain method [14], based on modularity optimization, is the state-of-the-art community detection approach which outperforms others. However, Louvain method is not robust to combat noises, such as the meaningless posts like “Good night :)” in social streams. Local definitions of communities focus on the subgraph under study, but neglecting the rest of the graph. Clique is a very strict definition of community, where a member is a friend of every other member. However, finding cliques in a graph is an NP-complete problem. It is possible to relax the notion of clique, by defining a community as clique-like structures, as we will discuss further in related work of dense subgraph mining.

On the application level, our query-driven quasi-clique search shares some similar intuitions with the community search ([19, 71]), which is finding the communities containing the querying set of people. However, since the definitions of communities in [71] and [19] are very different from a quasi-clique, the comparison between them and our query-driven quasi-clique search is not applicable.

2.1.3 Dense Subgraph Mining

Typical dense subgraphs studied in the literature include densest subgraph [73], clique, k -Plex, quasi-clique and k -Core [47]. Densest subgraph is a subgraph that maximizes the average degree. The densest subgraph can be found in polynomial time by solving a parametric maximum-flow problem [73], while finding the densest subgraph with a fixed size is known to be NP-Hard [7]. The maximum clique problem is a well-known NP-Hard problem. In real applications, the clique definition is too strict making it less

likely for large cliques to exist in practical graphs. E.g., there may be a large subgraph where most but not all node pairs are adjacent. This has motivated relaxations to cliques, some popular examples of which include k -Plex, quasi-clique, and k -Core. Since the degree constraints of k -Plex and quasi-clique are correlated with the size $|V_i|$, the NP-Hardness of the maximum clique problem carries over to k -Plexes and quasi-cliques [8]. In contrast, the complexity of the k -Core generation is in polynomial time [47]. However, if the size of a k -Core is distinctly larger than k , the average edge degree of this k -Core may be very low, and in this case, k -Core is not compact enough to describe a story. In this dissertation, we define a new dense subgraph called (k, d) -Core to overcome these challenges.

Maximal/Maximum Quasi-clique Detection. By definition, a maximal quasi-clique is a quasi-clique that cannot be a subgraph of any other quasi-cliques in the given graph. The maximum quasi-clique is the quasi-clique that has the largest number of nodes, out of all quasi-cliques in the given graph. The size of a maximal quasi-clique may be much smaller than the size of the maximum quasi-clique.

A negative breakthrough result by Arora et al. [6] together with results of Feige et al. [23], and more recently Hastad [31], shows that no polynomial time algorithm can approximate the maximum clique problem within a factor of $n^{1-\epsilon}$ ($\epsilon > 0$), unless $P = NP$. Thus, it is very unlikely that general heuristic algorithms can provide results with guaranteed optimality to the maximum clique problem. Related prior work on the maximal/maximum quasi-clique detection is typically based on local search ([1, 15, 33, 50, 61, 62]). Especially, Abello et al. [1] developed efficient semi-external memory algorithms for GRASP [64] to extract maximal quasi-cliques. Brunato et al. [15] extended two existing stochastic local search algorithms used for the classical maximum clique problem to the maximal quasi-clique problem. Pattillo et al. [62] established several fundamental properties of the maximum quasi-clique problem, but their quasi-clique is defined using edge density: $\mathcal{D}(G_i) = \frac{2|E_i|}{|V_i|(|V_i|-1)} \geq \lambda$. Based on depth-first search, Liu et al. [50] proposed an efficient algorithm called Quick to find maximal quasi-cliques using several pruning techniques, and Uno et al. [74] proposed a reverse search method to enumerate all quasi-cliques. However, none of them studied the *query-driven quasi-clique* problem as defined in Chapter 4. To the best of our knowledge, this thesis is the first study on the query-driven quasi-clique search problem.

Notice that there is another definition for the quasi-clique based on edge density: $\mathcal{D}(G_i) = \frac{2|E_i|}{|V_i|(|V_i|-1)} \geq \lambda$, studied in [1, 61, 74]. We do not adopt

this definition because it has the potential to introduce undesired low degree nodes into quasi-cliques: a quasi-clique has edge density $\mathcal{D}(G_i) \geq \lambda$ cannot prevent the occurrence of a node in G_i which has very low degree. In contrast, we use the definition that every node in a quasi-clique should have degree higher than $\lambda \cdot (|V_i| - 1)$. It is easy to show that a λ -quasi-clique under our node degree definition is at least a λ -quasi-clique under the edge density definition, but the converse is invalid. Thus, our definition is stronger than the edge density-based definition. Besides, Tsourakakis et al. [73] proposed a new dense subgraph called optimal quasi-clique, and defined constrained optimal quasi-clique problem to find the optimal quasi-clique containing a given node. However, their optimal quasi-cliques (Problem 2 in [73]) is defined as the subgraph $G_i(V_i, E_i)$ that maximizes $|E_i| - \frac{\lambda|V_i|(|V_i|-1)}{2}$, which is a fundamentally different problem from popular definitions of quasi-cliques based on edge density or node degree.

2.1.4 Subgraph Relatedness Computation

In this thesis, we model a story as a dense subgraph of posts, and the relatedness between stories can be measured by the relatedness between subgraphs. The relatedness between nodes in a graph is a well-studied problem, with popular algorithms such as HITS, Katz and Personalized PageRank [11]. For the relatedness between dense subgraphs, traditional measures like Jaccard coefficient, Cosine similarity and Pearson’s Correlation [53] are not effective if these dense subgraphs have no overlap on node sets. Recently, a propagation and aggregation process was used to simulate the information flow between nodes, which was studied in the context of top- k structural similarity search in [45] and authority ranking in [49].

2.2 Social Media Analytics

In related work, there are two main research directions on social media analysis. The first direction is the frequency-based approaches, which treat each social post as a statistical unit and use histogram-based analysis to mine the patterns in social media, e.g., bursty events. The second direction is the entity based approach, which extracts entities from each social posts and builds a network of entities from social media for event identification. They are introduced below separately.

2.2.1 Frequency Based Peak Detection

Most previous works detect events by discovering topic bursts from a document stream. Their major techniques either detect the frequency peaks of event-indicating phrases over time in a histogram, or monitor the formation of a cluster from a structure perspective. A feature-pivot clustering is proposed by Fung et al. [26] to detect bursty events from text streams. Sarma et al. [67] design efficient algorithms to discover events from a large graph of dynamic relationships. Weng et al. [77] build signals for individual words and apply wavelet analysis on the frequency of words to detect events from Twitter. A framework for tracking short, distinctive phrases (called “memes”) that travel relatively intact through on-line text was developed by Leskovec et al. [48]. Twitinfo [55] represents an event it discovers from Twitter by a timeline of related tweets. Marcus et al. [54] presents TweeQL, a streaming SQL-like interface to the Twitter API, making common tweet processing tasks simpler. Sakaki et al. [66] investigated the real-time interaction of events such as earthquakes in Twitter and proposed an algorithm to monitor tweets and to detect a target event based on classifiers.

2.2.2 Entity Network Based Approaches

Recently, Agarwal et al. [2] discover events that are unraveling in microblog streams, by modeling events as correlated keyword graphs. Angel et al. [5] study the maintenance of dense subgraphs with size smaller than a threshold (e.g., 5) under streaming edge weight updates. Both [2] and [5] model the social stream as an evolving entity graph, but suffer from certain drawbacks: (1) many posts are ignored in the entity recognition phase and post attributes like time; (2) author of the post cannot be integrated; (3) they cannot handle subgraph-by-subgraph bulk updates, which are key to efficiency. In contrast, these drawbacks are addressed in the post network defined in this dissertation.

2.3 Topic Detection and Tracking

Topic detection and tracking is an extensively studied field [51], with the most common approaches based on Latent Dirichlet Allocation (LDA) [13]. Techniques for topic detection and tracking cannot be applied to story relatedness tracking, because they are usually formulated as a classification problem [4], with an assumption that topics are *predefined* before tracking, which is unrealistic for social streams. Recent works (e.g., [32]) suffer from

this problem. Besides, the lack of training data set for story relatedness tracking on noisy social streams renders the existing works [59, 68] on Story Link Detection (SLD) inapplicable, because SLD is trained on well-written news articles. Jin et al. [37] present Topic Initiator Detection (TID) to automatically find which web document initiated the topic on the Web. In text streams, Hierarchical Dirichlet Processes (HDP, [27]) is proposed to track and connect topics incrementally. Since HDP is computed based on the document-word matrix, it is difficult to integrate HDP-based approaches with other signals, e.g., time stamps, GPS, authors, etc.

There is less work on evolution tracking. A framework for tracking short, distinctive phrases (called “memes”) that travel relatively intact through on-line text was developed in [48]. The evolution of communities in dynamic social networks is tracked in [79]. However, these existing works cannot track composite evolution patterns of communities, e.g., merging or splitting of communities. Moreover, all existing works have to re-compute communities from each network snapshot, which is time-consuming and results in lots of redundant computation. Unlike them, we focus on the incremental tracking of cluster evolution patterns in highly dynamic networks, where we maintain each cluster by gradually adding or removing nodes from it.

Chapter 3

Modeling Unstructured Social Streams

This chapter focuses on the modeling of social streams, which is the first step in social stream mining. We introduce the social stream preprocessing techniques in Section 3.2. The construction of the post network from a social stream is discussed in Section 3.3. We define stories and events, from both the social stream perspective and the post network perspective, in Section 3.4. We provide a comparison between the structural modeling used in this dissertation and other modeling methods in Section 3.6.

3.1 Motivation

Social websites like Twitter have a great impact on many people’s digital lives. As the social content streams fast, it may easily lead to “information anxiety”, which is the gap between the information we receive and the information we are able to digest [40]. The current generation of information-seeking on social media works just like the traditional search on web pages, in which users input several keywords and the output will be a long list of tweets or posts with keywords contained, ranked by time freshness. For instance, Twitter Search³ returns a huge list of posts to a given keyword query, with little aggregation or semantics, and leaves it to the users to sift through the large collection of results to figure out the very small portion of useful information. Since a post like a tweet only contains a small piece of information, users are required to manually aggregate and digest search results, which is time-consuming and painful. The noisy and redundant nature of social streams degrades user’s experience further.

On the other hand, since a post like tweet only conveys a very small piece of information, it would be ideal if we can group the posts talking about the same information together. In this dissertation, we define “story” and “event” as two kinds of post structures that organize the posts telling

³<https://twitter.com/search-home>

similar information together. We are aiming to build the next generation social stream mining technologies, which provide users an organized and summarized view of what’s happening in the social world. Instead of showing users a long list of posts, our new social stream mining technologies try to present users with well-organized stories and events.

3.2 Social Stream Preprocessing

In a social media like Twitter, new posts emerge quickly in every second and old posts will be naturally outdated as the time rolls. Posts in social streams such as tweets are usually written in an informal way. To design a processing strategy that can quickly and robustly extract the meaningful information of a post, we focus on the entity words contained in a post, since entities depict the topic. For example, given a tweet “iPad 3 battery pointing to thinner, lighter tablet?”, the entities are “iPad”, “battery” and “tablet”. However, traditional Named Entity Recognition tools [21] only support a narrow range of entities like Locations, Persons and Organizations. [5] reported that only about 5% of tweets has more than one named entity. NLP parser based approaches [39] are not appropriate due to the informal writing style of posts and the need for high processing speed. To broaden the applicability, we treat each noun in the post text as a candidate entity. Technically, we obtain nouns from a post text using a Part-Of-Speech Tagger⁴, and if a noun is plural (POS tag “NNS” or “NNPS”), we obtain its singular form. In practice, we find this preprocessing technique to be robust and efficient. In the Twitter dataset we used in experiments (see Section 6.7), each tweet contains 4.9 entities on average. We describe a post p as a triple (L, τ, u) , where p^L is the list of entities, p^τ is the time stamp, and p^u is the author. We formally define a post and a post stream as follows.

Definition 1 (Post). *A post p is a triple (L, τ, u) , where L is the list of entities in the post, τ is the time stamp when this post is generated, and u is the user who created it.*

We let p^L denote L in the post p for simplicity, and analogously for p^τ and p^u . We use $|p^L|$ to denote the number of entities in p .

Definition 2 (Social Stream) *A social stream is a first-in-first-out queue of posts ordered by time of arrival, in which each post p is represented as a triple (L, τ, u) as defined in Definition 1.*

⁴POS Tagger, <http://nlp.stanford.edu/software/tagger.shtml>

3.3 Post Network Construction

Our modeling method for social streams in this dissertation is based on constructing a network of posts and maintaining the network over a moving time window, as posts stream in and fade out. This network is used for subsequent analysis. Essentially, our modeling method is based on the hypothesis that the correlation between posts should be considered, and grouped posts provide a better user experience than individual posts. In this section, we describe the construction of post network based on post correlations.

3.3.1 Post Similarity Computation

Our initial data analysis shows that post topics do not have a strong correlation with authors in Twitter. In other words, a typical Twitter user may create posts with different topics at different time moments. Fortunately, we found that post topics are highly correlated with entities and time moments, i.e., posts talking about the same topic typically have similar entities and very close time stamps.

Traditional similarity measures such as TF-IDF based cosine similarity, Jaccard Coefficient and Pearson Correlation [53] only consider the post content. However, clearly time stamps should play an important role in determining post similarity, since posts created closer together in time are more likely to discuss the same event. We introduce the notion of fading similarity to capture both content similarity and time proximity. For example, with Jaccard coefficient as the underlying content similarity measure, the *fading similarity* is defined as

$$S_F(p_i, p_j) = \frac{|p_i^L \cap p_j^L|}{|p_i^L \cup p_j^L| \cdot e^{|p_i^T - p_j^T|}} \quad (3.1)$$

We use an exponential function to incorporate the decaying effect of time lapse between the posts. The unit of time difference is Δt , typically in hours. It is trivial to see that $0 \leq S_F(p_i, p_j) \leq 1$ and that $S_F(p_i, p_j)$ is symmetric.

Post similarity measures the similarity between two posts p_1 and p_2 by a score between 0 and 1, which is assessed by considering both the post content and time. That is to say, if two posts share many common entities and their posting time is very close, they will be similar.

3.3.2 Post Network

To find the correlation between posts, we build a post network $G(V, E)$ based on the following rule: if the fading similarity between two posts (p_i, p_j) is

3.3. Post Network Construction

higher than a given threshold λ , we create an edge $e(p_i, p_j)$ between them and set the edge similarity $s(p_i, p_j) = S_F(p_i, p_j)$. Obviously, a lower λ retains more semantic similarities but results in much higher computation cost, and we set $\lambda = 0.3$ empirically on Twitter streams to gain a balance between edge sparsity and information richness. Consider a time window of observation and consider the post network at the beginning. While we move forward in time and new posts appear and old posts fade out, $G(V, E)$ is dynamically updated at each moment, with new nodes/edges added and old nodes/edges removed. On the scale of Twitter streams with millions of tweets per hour, $G(V, E)$ is truly a large and fast dynamic network. The formal definition for the post network is given below.

Definition 3 (Post Network) *Given two posts p_i, p_j in a social stream Q and a threshold ϵ ($0 < \epsilon < 1$), there is an edge between p_i and p_j if the post similarity $s(p_i, p_j) \geq \epsilon$. The post network corresponding to Q is denoted as $G(V, E)$, where each node $p \in V$ is a post in Q , and each edge $(p_i, p_j) \in E$ is constructed if the similarity $s(p_i, p_j) \geq \epsilon$.*

Intuitively, an edge in the post network connects two posts if they are similar enough. The post network can be viewed as a structural representation of the original unstructured social stream, by organizing meaningful information from noisy buzzes. Specifically, posts with very few edges can be essentially treated as noise and ignored.

3.3.3 Linkage Search

Removing a node and associated edges from $G(V, E)$ is an easy operation. In contrast, when a new post p_i appear, it is impractical to compare p_i with each node p_j in V_t to verify the satisfaction of $S_F(p_i, p_j) > \lambda$, since the node size $|V_t|$ can easily go up to millions. To solve this problem, first we construct a post-entity bipartite graph, and then perform a two-step random walk process to get the hitting counts. The main idea of linkage search is to let a random surfer start from post node p_i and walk to any entity node in p_i^L on the first step, and continue to walk back to posts except p_i on the second step. All the posts visited on the second step form the candidates of p_i 's neighbors. Supposing the average number of entities in each post is d_1 and the average number of posts mentioning each entity is d_2 , then linkage search can find the neighbor set of a given post in time $O(d_1 d_2)$. In our Twitter dataset, d_1 and d_2 are usually below 10, which supports the construction of a post network on the fly.

3.4 Stories and Events

Story and event are two concepts we use in this dissertation to aggregate and organize posts with similar meaningful information together. In related work, there are many different definitions for story and event, and some studies even use these two terms interchangeably, without proper distinguishing. In this section, we distinguish and define stories and events dually on two levels: social streams level and post network level, as explained below.

3.4.1 Stories and Events in Social Streams

In social streams, both a story and an event are a set of posts talking about very similar information. Their main difference is on the granularity of information they carry: a story is assumed to only talk about a single thing, while an event is assumed to talk about many things with high relatedness. Thus, we can say that an event contains many highly related stories. For example, we consider “Ukrainian crisis” is an event happening from November 2013 to May 2014, which contains lots of stories like “the annexation of Crimea by the Russian Federation” and “War in Donbass” in March 2014. In the following, we give the definitions of stories and events in social streams.

Definition 4 (Story in Social Stream) *A story is a set of posts that talk about a single topic in a short time span.*

Definition 5 (Event in Social Stream) *An event is a set of posts that talk about a series of highly related topics.*

Usually, the information carried by a story can be described by a few sentences or a small set of keywords. Since posts in social streams are usually very short, we consider a post describes at most one story in most cases. However, a story can be described by many posts, e.g., two posts “Crimea was annexed by the Russian Federation” and “Crimea voted on 6 March to formally accede as part of the Russian Federation” are talking about the same story. An event is a post cluster which contains many highly related stories, where the relatedness between stories is measured on both the content similarity and the time closeness. Typically, an event has a relatively long time span and follows clear evolution patterns, e.g., emerging, growing and decaying. A typical event contains lots of details and cannot be described by a short post. For example, “Ukrainian crisis” is an event, which consists of a series of related stories, e.g., “Crimean Crisis”, “War in Donbass”, “Ukraine President Elections”, etc.

Definition 4 and 5 are given on the conceptual level. There are several open questions for these two definitions, some of which are listed below:

- Given a set of posts, how to determine whether these posts talking about a story, an event, or none of them?
- How to quantify the relatedness of two stories?
- How to measure the evolution patterns of an event?

Since social streams are unstructured and difficult to analyze, the answers of these questions depend on the modeling of social streams. In previous sections, we discussed the modeling of a social stream as a post network. In the following, we explain stories and events with reference to a post network.

3.4.2 Stories and Events in Post Network

In Section 3.3, we discussed the construction of a post network from a social stream: a post in social stream is modeled as a node and two posts with high similarity are connected by an edge. As a result, a story or an event in social streams corresponds to a subgraph in the post network. This subgraph corresponding to a story or an event should be connected, since we assume posts in a story or an event carry very similar information.

Cohesion. Let's start from the definition of a story in the post network. Given a connected subgraph $G_i(V_i, E_i)$ of the post network, how can we determine whether this subgraph corresponds to a story or not? The answer is to make use of a measure called "cohesion", as defined below.

Definition 6 (Cohesion) *Given a connected subgraph $G_i(V_i, E_i)$, its cohesion $\mathcal{C}(G_i)$ is defined as the ratio between the minimum degree and the maximum possible degree in G_i . Supposing $\deg(v, G_i)$ is the degree of node v in G_i , we have*

$$\mathcal{C}(G_i) = \frac{\min_{v \in V_i} \deg(v, G_i)}{|V_i| - 1} \quad (3.2)$$

Why the cohesion is so important for defining a story? The reason is similar posts are connected together in the post network. If two posts are similar with each other in both the text content and the creation time, it is likely that these two posts are telling the same story, and they will be connected by an edge in the post network. A story in a social stream is a subgraph in the post network. If nodes in a subgraph are highly connected with each other, it is very likely that the posts corresponding to this subgraph are telling the same story. To guarantee that posts in the same story are

similar to each other, we consider coreness, edge density and cohesion of this post subgraph. Among these three, *cohesion* defined in Eq. (3.2) is most effective in ensuring that every post in the story is similar to the majority of posts in the same story. Compared with coreness $\mathcal{K}(G_i)$ in k -Core where $\mathcal{K}(G_i) \geq k$ and edge density $\mathcal{D}(G_i) = \frac{2|E_i|}{|V_i|(|V_i|-1)}$, *cohesion* can make sure a node connects to the majority of other nodes, regardless of the story size, while $\mathcal{K}(G_i)$ and $\mathcal{D}(G_i)$ cannot guarantee that.

Cohesion-Persistent Story. The first instantiation is defining a story as a connected subgraph $G_i(V_i, E_i)$ with enough high cohesion. Clearly, a story is a special kind of dense subgraphs. In related work, there are many definitions for dense subgraphs, e.g., quasi-clique, k -Plex, k -Core [47] and k -Truss [34]. Supposing $deg(v, G_i)$ is the degree of node v in G_i , these dense subgraphs are defined as:

- Quasi-clique: $|deg(v, G_i)| \geq \lambda(|V_i|-1)$ for any node $v \in V_i$ and $\lambda \in (0, 1]$;
- k -Plex: $|deg(v, G_i)| \geq |V_i| - k$ for every node $v \in V_i$;
- k -Core: $|deg(v, G_i)| \geq k$ for every node $v \in V_i$;
- k -Truss: for every edge $(v_i, v_j) \in E_i$, $|N(v_i) \cap N(v_j)| \geq k$.

Out of them, only for λ -quasi-cliques, the cohesion is always at least λ as the subgraph size changes. The intuition of a cohesion-persistent story is best captured by a λ -quasi-clique, as the degree of nodes grows with the node size in a quasi-clique. Given several posts already read by the user as the input, the finding of the cohesion-persistent story containing these posts is called the cohesion-persistent story search problem, as discussed in Chapter 4. Formally, we introduce the *cohesion-persistent story*, as defined below.

Definition 7 (Cohesion-Persistent Story) *A cohesion-persistent story $G_i(V_i, E_i)$ in a post network $G(V, E)$ is defined as a λ -quasi-clique, where $0 < \lambda \leq 1$ and for any node $v \in V_i$, we have $|deg(v, G_i)| \geq \lambda(|V_i| - 1)$.*

(k, d) -Core Story. Definition 7 provides a theoretically ideal way to define a story. However, notice that finding the maximum clique or quasi-cliques from a graph is an NP-Hard problem [8, 15]. Even worse, [31] proved that there are no polynomial algorithms that provide any reasonable approximation to the maximum clique problem. Since a clique is a special case of quasi-clique or k -Plex, these hardness results carry over to maximum quasi-clique and maximum k -Plex problems. There are a lot of heuristic algorithms that provide no theoretical guarantee on the quality [1, 8, 15]. Since most of them are based on local search [15], they do not scale to large networks, because local search involves the optimization of solutions by iteratively moving to a better neighbor solution in an exponential search space.

Although quasi-cliques are theoretically ideal to instantiate a story, the intractable performance of quasi-clique computation brings difficulty to apply this definition on the real-world large post network. In Chapter 4, we will discuss the cohesion-persistent story search problem, which aims to approach the maximum quasi-clique containing given nodes using heuristic rules. However, due to the performance, this approach cannot solve the problem of finding all stories in the post network. These challenges motivate us to seek an alternative instantiation of a story, which is cohesive enough and efficient to compute on real-world post networks.

The good news is that k -Cores can be exactly found in polynomial time. By adjusting k , we can generate k -Cores with any desired cohesion score $k/(|V_S| - 1)$. For example, increasing k will improve the cohesion because the minimal edge degree is increased and $|V_S|$ is decreased in the same time. However, k -Cores are not always capable of capturing our intuition on stories: although each post has at least k neighbors, the fact that two posts are connected by a single capillary may be not a strong enough evidence to prove that they tell the same story. Sometimes, posts only share some common words but discuss different stories, e.g., “Google acquires Motorola Mobility” and “Bell Mobility acquires Virgin Mobile”. To address this challenge, we make a key observation that *the existence of more common neighbors between two connected posts suggests a stronger commonality in story-telling*. Supposing p_i and p_j are connected by an edge, $N(p_i)$ and $N(p_j)$ are neighbor sets of p_i and p_j respectively, if there exists post $p_l \in N(p_i) \cap N(p_j)$, then we call p_l a witness for the post similarity $s(p_i, p_j)$. We capture this intuition in the following definition, where we formalize a story as a (k, d) -Core.

Definition 8 ((k, d)-Core Story) A (k, d) -Core story in the given post network $G(V, E)$ is defined by a maximal (k, d) -Core $G_i(V_i, E_i)$, where k, d are numbers with $k > d > 0$ and

- $G_i(V_i, E_i)$ is a connected subgraph;
- For every post $p \in V_i$, $|N(p)| \geq k$;
- For every edge $(p_i, p_j) \in E_i$, $|N(p_i) \cap N(p_j)| \geq d$.

Density-Based Event. An event tries to organize a set of posts talking about many related things together. Especially, an event may include multiple related stories. In the post network, an event is also modeled as a connected subgraph. Unfortunately, the various cohesive subgraph options we discussed in story modeling are not suitable for defining an event, since these cohesive subgraphs have a clear center and are designed to talk about

mainly a single thing. An ideal definition of events should allow multiple centers, and these centers should be highly related.

In this dissertation, we consider a graph cluster (or called partition) is the best way to define an event. The intuition is that, since post network is constructed by pairwise post similarity, a graph cluster with high internal connectivity and low external connectivity indicates that posts inside the cluster talk about very related things. Various clustering approaches can be applied on a post network to extract clusters, and among them, we choose density-based clustering [30] as the best modeling for events. In density-based clustering (e.g., DBSCAN [20]), the threshold $MinPts$ is used as the minimum number of nodes in an ε -neighborhood, required to form a cluster. We adapt this and use a weight threshold δ as the minimum total weight of neighboring nodes, required to form a cluster. The reason we choose density-based approaches is that, compared with partitioning-based approaches (e.g., K-Means [30]) and hierarchical approaches (e.g., BIRCH [30]), density-based methods such as DBSCAN define clusters as areas of higher density than the remainder of the data set, which is effective in finding arbitrarily-shaped clusters and is robust to noise. Moreover, density-based approaches are easy to adapt to support single-pass clustering. In the post network, we consider ε to be a *similarity threshold* to decide the connectivity, which can be used to define the weight of a post in density-based clustering. According to density-based clustering, nodes in the post network are distinguished into three types: core posts, border posts and noisy posts, by a threshold δ applied on the post weight. Formally, we define an event on post network below.

Definition 9 (Density-Based Event) *An event in the post network $G(V, E)$ is a cluster C obtained by density-based clustering using density parameters ε and δ , where ε is a similarity threshold to remove the edge if its similarity is lower than ε , and δ is the minimum weight of a core node.*

The details of density-based clustering for event identification can be found in Section 6.4.1.

3.5 Context and Evolution

As discussed in Chapter 1, detecting stories with high cohesion, tracking story context and event evolution patterns are the most important problems we focus on in this dissertation. The notion of cohesion has been defined in Definition 6. In this section, we introduce *context* for stories and *evolution* for events respectively.

Story Context. Transient stories that we identified from the post network may be highly related, e.g., two stories “the launch of Blackberry 10” and “BlackBerry Super Bowl ad” are highly related. We try to exploit and integrate signals from different perspectives (e.g., content or time) to compute the relatedness between stories. Here, we introduce different types of relatedness dimensions, which capture the story relatedness from three perspectives, and quantify the relatedness by a value in $[0, 1]$.

- **Content Similarity.** By viewing a story as a document and a post entity as a term, existing document similarity measures can be used to assess the story relatedness. However, TF-IDF based Cosine similarity fail to be effective, since TF vectors of stories tend to be very sparse.
- **Temporal Proximity.** Stories that happen closer in time are more likely to correlate together.
- **Edge Connectivity.** Edges associated with posts of two stories can be used to determine the story relatedness. This approach calculates the strength of edge connectivity between posts of two stories. These edges serve as the bridge between two stories, and the connectivity strength can be measured by various ways, e.g., the Jaccard Coefficient.

Story context search aims at finding the neighboring stories of a given story in the post network efficiently. In database research literature, we use iceberg query to describe a kind of queries which aims to find results with scores above a given threshold. We introduce story context search, which is a kind of iceberg query on stories, as stated below.

Definition 10 (Story Context Search) *Given a set of stories \mathbb{S} , a threshold γ ($0 < \gamma < 1$) and a story S , the story context search for S is to find the subset of stories $\mathbb{S}' \subseteq \mathbb{S}$, where for each $S' \in \mathbb{S}'$, the relatedness $Cor(S, S') \geq \gamma$.*

The computation of $Cor(S, S')$ will be discussed in Chapter 5.

Event Evolution. Event evolution happens when the time window moves on social streams. We use E to denote an event and e is a snapshot of E at a specific moment. For simplicity, if we talk about event e , it actually means event E at moment t . Let S_t denote the set of events at moment t . We analyze the evolutionary process of events at each moment and abstract them into four primitive patterns and two composite patterns. The four primitive patterns are *emerge*, *disappear*, *grow* and *decay*. The two composite patterns are *merge* and *split*, which can be decomposed into a series of *emerge* and *disappear* patterns. From moment t to $t + 1$, they are defined below.

3.6. Comparing with Other Modeling Methods

Cases	Add	Remove
If p is a noise post	-	-
If p is a border post to the neighboring event e	<i>grow</i>	<i>decay</i>
If p is a core post without neighboring event	<i>emerge</i>	<i>disappear</i>
If p is a core post to exactly one neighboring event e	<i>grow</i>	<i>decay</i>
If p is a core post to multiple neighboring events $\{e_1, e_2, \dots, e_n\}$	<i>merge</i>	<i>split</i>

Table 3.1: Event evolution patterns

- **emerge**: add event e to the event set S_t ;
- **disappear**: remove event e from the event set S_t ;
- **grow**: increase the size of e by adding new posts;
- **decay**: decrease the size of e by removing old posts;
- **merge**: remove a list of events $\{e_1, e_2, \dots, e_n\}$ from S_t and add a new event e , where $e = e_1 + e_2 + \dots + e_n$;
- **split**: remove an old event e from S_t and add a list of events $\{e_1, e_2, \dots, e_n\}$, where $e = e_1 + e_2 + \dots + e_n$.

Compared with primitive patterns, *merge* and *split* are not very common in event evolution. To a specific event, *emerge/disappear* or *merge/split* can only happen once, but *grow/decay* may happen at each moment.

In the following, we explain how to track event evolution incrementally as the post network gets updated. Conceptually, we call a post a core post if this post is similar to lots of other posts. Suppose a post p is added into the post network. If p is a noise post, we simply ignore p . If p is a border post to the neighboring event e , *grow* e . If p is a core post without neighboring event, a new event *emerges*. If p is a core post that is a neighbor of exactly one event e , *grow* e . If p is a core post that is a neighbor of multiple events $\{e_1, e_2, \dots, e_n\}$, *merge* them into a new event. The analysis of event evolution patterns for removing a post p from post network is very similar. We show evolution patterns of various cases in Table 3.1.

3.6 Comparing with Other Modeling Methods

In this thesis, we view each post as a node, and the relationship between posts as an edge. By this way, a post stream can be transformed into a

network evolving with the time, which is called a post network. Events and stories can be viewed as substructures in this post network. However, in related work, there are other alternative perspectives for social media search, in which content and frequency based approaches are major players [13, 48]. In this section, we briefly compare them, and show the advantages of the structure based approach. Notice that the experimental study of the structural perspective versus content and frequency perspectives will be extensively discussed in Chapter 5 and 6.

- **Structure vs. LDA Approaches.** In content perspective, topic detection and tracking is an extensively studied field [51], with the most common approaches based on Latent Dirichlet Allocation (LDA) [13]. Techniques on topic tracking are usually formulated as a classification problem [4], with an assumption that topics are *predefined* before tracking, which is unrealistic for social streams. Recent works [27, 32] fall prey to this problem. The lack of training data set for story relatedness tracking on noisy social streams renders the existing works [59, 68] on Story Link Detection (SLD) inapplicable, because SLD is trained on well-written news articles. In text streams, Hierarchical Dirichlet Processes (HDP, [27]) was proposed to track and connect topics incrementally. However, since HDP is computed based on the document-word matrix, it is difficult to integrate HDP-based approaches with other signals, e.g., time stamps, GPS signals, authors, etc.
- **Structure vs. Frequency.** In related work, frequency based approaches are commonly used in story and event detection. Weng et al. [77] build signals for individual words by applying wavelet analysis on the frequency based signals of words to detect events from Twitter. A framework for tracking short, distinctive phrases (called “memes”) that travel relatively intact through on-line text was developed in [48]. Twit-info [55] represents an event it discovers from Twitter by a timeline of related tweets. [66] investigated the real-time interaction of events such as earthquakes in Twitter and proposed an algorithm to monitor tweets and to detect a target event based on classifiers. To summarize, the common technique behind the above is detecting popular items based on the ranking of frequency, and these items are typically terms, hash-tags or short phrases. Compared with frequency based approaches, our structural approach represents a story as a cohesive subgraph in post network, which has a compact internal structure to describe the story, and contains rich information. For example, frequency based approaches cannot track the merge or split of two events, while structural approach

is capable to track, by capturing the merge or split of two events as the merge or split of the underlying subgraphs in post network.

We also perform a detailed user study to verify the hypothesis that structural method is better than content and frequency based methods using crowdsourcing in Chapter 7.

3.7 Discussion and Conclusion

Our modeling method for social streams in this dissertation is based on constructing a network of posts and maintaining the network over a moving time window. To track stories and events from social streams, our modeling method contains following steps:

- Post information extraction, in which we extract post text content and time stamps from social streams;
- Entity extraction, where entities in post content are extracted using NLP tools;
- Post similarity computation based on entities and time stamps;
- Graph based algorithms to mine social patterns like stories and events.

The limitations of this modeling work flow is that we ignore some information on each step. For example, in post information extraction, we ignore the author of a post, and in entity extraction, we use entities to represent post content. However, based on data analysis, we ensure the information we ignored is less meaningful than the information we kept. For example, we ignore authors because data analysis shows post topics of an author on Twitter are roughly random. We use entities to represent post content, rather than keywords or hashtags, because we found that keywords will generate post networks with too many edges (i.e., too high edge density) and hashtags will generate post network which is too sparse.

One drawback of our post similarity computation is that we ignore the word ambiguity problem. For example, “apple” may mean the Apple company or a kind of fruit; “Microsoft” and “MSFT” may mean the same thing. Distinguishing word ambiguity in informally written tweets is a very hard problem in the NLP community and we consider the deep analysis on this problem beyond the research scope of this dissertation.

There are still some space for the improvement. First, currently the entities returned by Stanford NLP tools are mostly nouns, which indicates better entity recognition approaches may be proposed to generate entities with higher quality. E.g., emotions and sentiments can be extracted as at-

3.7. Discussion and Conclusion

tributes of entities and provide more input to the similarity function. Second, deep analysis of sentence structure of tweets may result in a better post content similarity computation method. Third, since the current post similarity computation only combines content similarity and time proximity, better similarity functions may be proposed to incorporate more meaningful information into consideration. Fourth, it is possible to combine structural approaches with frequency or LDA-based approaches, which may result in a better hybrid approach than pure structural approaches.

Chapter 4

Cohesion-Persistent Story Search

A cohesion-persistent subgraph is a subgraph with its cohesion higher than a given threshold λ ($0 < \lambda \leq 1$). Recall that cohesion is defined as the ratio between the minimum degree and the maximum possible degree in a given subgraph (Definition 6). By this definition, a cohesion-persistent subgraph with threshold λ is a λ -quasi-clique. The quasi-clique is an appealing way to model a story in the post network, due to the benefit that the cohesion of a λ -quasi-clique is always higher than λ ($0 < \lambda \leq 1$), regardless of the size of the story. We call the story defined by a quasi-clique as a cohesion-persistent story. In this chapter, we are interested in the search problem of cohesion-persistent stories, where given a query set of posts, we try to find the maximum quasi-clique that contains this query set of posts. In applications, the query can be posts liked or found interesting/useful by the user, and the answer will be the most popular story related to the querying posts.

4.1 Introduction

Just like any real-world networks, it is common for the post network to have a skewed degree distribution, with the result that they feature “dense subgraphs”. Since we model a social stream as a post network, finding dense subgraphs of the post network corresponds to the story detection problem on social streams. While several alternative definitions have been proposed for dense subgraphs (e.g., see [25, 47]), *quasi-cliques* constitute an appealing way to model a story, since its cohesion is guaranteed to above a given threshold. Specifically, a λ -quasi-clique is defined as a connected subgraph in which the ratio between the degree of each node and the highest possible degree is at least λ , where $\lambda \in (0, 1]$. There has been some prior work on finding quasi-cliques from graphs [15, 62, 63]. However, none of them can handle the quasi-clique *search* problem, which is not studied before, and has many applications in the real-world, especially, the *cohesion-persistent story search*

on social streams. In particular, we focus on a new graph mining problem in this chapter, namely *query-driven maximum quasi-clique search*. Given a graph $G(V, E)$, a set of query nodes $S \subseteq V$, and a parameter $\lambda \in (0, 1]$, we are interested in the problem of finding the largest λ -quasi-clique containing the node set S .

The conceptual definition of a story is a set of posts telling the same thing. On the post network, this requires that every node has a sufficiently high connection strength with other nodes in the same subgraph. To help determine a post set tells the same thing or not, we use the notion “cohesion” to describe the ratio between the minimum degree and the highest possible degree in a connected subgraph. Clearly, the ideal definition of a story requires the cohesion to be persistent, or robust, regardless of the size of the subgraph. There are many definitions for dense subgraphs, e.g., densest subgraph [73], k -Plex, quasi-clique, k -Core [47] and k -Truss [34]. Of these, only for λ -quasi-cliques, the cohesion is always at least λ as the subgraph size changes. The finding of the largest cohesion-persistent subgraph containing a given node set can be modeled as the Query-driven Maximum Quasi-clique (QMQ) search problem. Since the maximum clique problem is NP-Hard [61] and no polynomial time algorithm can approximate it within a factor of $n^{1-\epsilon}$ ($\epsilon > 0$) [31], it is not surprising that finding the maximum quasi-clique is also NP-Hard and not approximable in polynomial time. As for heuristic approaches, existing studies on quasi-clique maximization (without any query) are mainly based on local search [15, 33, 62], in which a solution moves to the best neighboring solution iteratively, updated node-by-node. However, *none of the existing approaches are designed for quasi-clique search and can efficiently handle the QMQ problem*. A detailed comparison with these and other related works appears in Section 2.1.3.

Challenges. To the best of our knowledge, this chapter is the first study on the quasi-clique search problem. Solving this difficult problem raises the following challenges:

- Given a set of query nodes S , how to find a λ -quasi-clique containing S as efficiently as possible?
- How can we design an effective and uniform objective function for solutions that guides the maximization of λ -quasi-clique containing S ?
- Given a solution, how to devise efficient iterative maximization techniques to search for a better solution?
- As the iterative maximization is a local search process, how can we prevent the algorithm being trapped into a local maximum?

Main Idea. In this chapter, we propose an efficient framework for the QMQ search problem, which has two components: an offline component that builds a tree representation \mathbb{T} of the given graph G , and an online component which responds to the maximum quasi-clique search for a given set of query nodes S . In the offline component, we propose *core tree*, a tactfully designed hierarchical structure to help find a *pre-solution* to the QMQ search problem in a few tree traversal operations, where the pre-solution is a maximal k -Core for some k , which contains the query nodes S and whose cohesion is very close to λ . The idea is that a λ -quasi-clique can be obtained from the pre-solution very quickly. The notion of k -Core, as defined in [47], is a connected subgraph in which every node has degree at least k . In our proposed core tree, G is the root and each tree node is a maximal k -Core, for some k ; if a tree node \mathbb{T}_i is a subgraph of another tree node \mathbb{T}_j , we say \mathbb{T}_i is a child of \mathbb{T}_j . The core tree can be built recursively and efficiently, and the pre-solution can be quickly retrieved from the core tree.

In the online component, we introduce three maximization operations: *Add*, *Remove* and *Swap*. By scheduling these operations using different strategies, we propose two iterative maximization algorithms for the QMQ search, called Deterministic Iterative Maximization (DIM) and Stochastically Updated Maximization (SUM). Of these, DIM is a deterministic approach that, in each iteration, greedily moves from the current solution to the best available neighboring solution. SUM, on the other hand, is a stochastic approach, which moves from the current solution to one of its good neighboring solutions with a probability proportional to the “marginal gain” associated with such a move. Intuitively, SUM has the potential to break the limitation of getting trapped in a local maximum and thus find better results than DIM, at the expense of potentially longer iteration steps than DIM. Both DIM and SUM are efficient, specifically DIM takes $O(Tn)$ time while SUM takes $O(Tn^2)$ time for each simulation, where T is the total number of iterations, and n is the average solution size on each iteration.

Contributions. We make the following contributions in this chapter:

- We define the problem of query-driven maximum quasi-clique search, a novel cohesive subgraph query with many real-world applications (Section 4.2).
- We propose core tree as a recursive representation of a graph, which helps quickly find a pre-solution to the QMQ search problem within a few tree traversals by reducing the solution search space (Section 4.3).
- We introduce *Add*, *Remove* and *Swap* to search for new solutions and

4.2. Problem Overview

$G(V, E)$	the given network
S	the query node set
$G_i(V_i, E_i)$	a connected subgraph of G
$Q_S^\lambda(V_Q, E_Q)$	a query-driven quasi-clique
$\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q)$	the query-driven maximum quasi-clique
\mathbb{T}, \mathbb{T}_i	core tree, tree node
$\text{deg}(v, G_i)$	Degree of node v in subgraph G_i
$\mathcal{F}(G_i)$	objective function for subgraph G_i

Table 4.1: Notation.

efficiently optimize a pre-solution to a λ -quasi-clique as well as the current solution to a better neighboring solution. Building on this, we propose deterministic and stochastic iterative maximization algorithms for QMQ search – DIM and SUM (Section 4.4).

- We perform an extensive experimental study on three real datasets, which demonstrates that our algorithms significantly outperform several baselines in running time and/or the quality. (Section 4.5).

Section 4.6 concludes this chapter. Major notations used are listed in Table 6.1.

4.2 Problem Overview

Problem. Given a set of query nodes S , we define a query-driven quasi-clique w.r.t. S as a quasi-clique that contains all nodes in S . The formal definition follows, where for a subgraph $G_i \subseteq G$ and a node v in G_i , we use $\text{deg}(v, G_i)$ to denote the degree of v in G_i .

Definition 11 (Query-Driven Quasi-Clique). *Given an undirected graph $G(V, E)$, a set of query nodes $S \subseteq V$, and a parameter $0 < \lambda \leq 1$, a query-driven quasi-clique w.r.t. S and λ is a subgraph $Q_S^\lambda(V_Q, E_Q)$ of G , such that:*

- Q_S^λ is connected and $S \subseteq V_Q \subseteq V$;
- For every node $v \in V_Q$, $\text{deg}(v, Q_S^\lambda) \geq \lambda \cdot (|V_Q| - 1)$.

For simplicity, we use Q_S^λ and Q interchangeably. The problem of query-driven maximum quasi-clique search is to find the largest λ -quasi-clique containing the query nodes S , as formalized below.

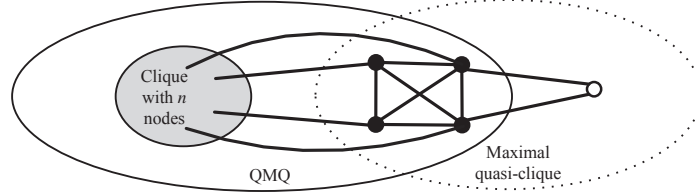


Figure 4.1: The query S is marked by solid nodes, with $\lambda = 0.5$. A maximal quasi-clique and the query-driven maximum quasi-clique (QMQ) are circled by a dotted ellipse and a solid ellipse, respectively.

Problem 1 Given an undirected graph $G(V, E)$, a set of query nodes $S \subseteq V$ and a parameter $0 < \lambda \leq 1$, find the subgraph $\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q)$ in $G(V, E)$, such that

$$\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q) = \arg \max_Q \{|V_Q| \mid Q = Q_S^\lambda(V_Q, E_Q)\} \quad (4.1)$$

We denote by $\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q)$ the query-driven maximum quasi-clique (QMQ) for the query S and parameter λ . Notice that it is possible that there is no qualified λ -quasi-clique containing all nodes in S , and in this case we return NULL.

Hardness. It is well-known that the problem of finding the maximum clique is NP-Hard [61]. Even worse, a negative breakthrough result by Arora et al. [6] together with results of Feige et al. [23], and more recently Hastad [31], imply that there is unlikely to be any polynomial time approximation algorithm for this problem within a factor of $n^{1-\epsilon}$ ($\epsilon > 0$). According to Abello et al. [1], general heuristics which provide answers with guaranteed approximation to the maximum clique are unlikely to exist. Since a clique is a special case of a quasi-clique, these hardness results carry over to the maximum quasi-clique problem [62]. In particular, the query-driven maximum quasi-clique (QMQ) search is also NP-Hard and inapproximable, since when $S = \emptyset$, QMQ search reduces to the traditional maximum quasi-clique problem.

Maximum vs. Maximal Quasi-Cliques. The QMQ search problem should not be confused with the extensively studied *maximal* quasi-clique problem. A maximal λ -quasi-clique cannot be the subset of any other λ -quasi-clique. However, its size may be arbitrarily smaller than that of the maximum quasi-clique. Fig. 4.1 illustrates these ideas with an example of a maximal λ -quasi-clique and the QMQ, both containing the query set S .

Solution Overview. The hardness of the QMQ search problem indicates

that polynomial-time algorithms which provide answers with guaranteed optimality to QMQ unlikely exist. The hardness and inapproximability results for QMQ naturally motivate the quest for good heuristics to solve the problem. In related work, most algorithms for finding maximum as well as maximal quasi-cliques are based on local search [15, 33, 62]. However, traditional local search does not scale to large networks, because the search space of neighboring solutions in the iterative optimization increases exponentially with the network size. Thus, using traditional approaches, it is difficult to quickly find an initial solution corresponding to a quasi-clique containing S .

To address these challenges, in this chapter, we propose an efficient framework for the QMQ search problem on large-scale networks. The architecture of this framework is illustrated in Fig. 4.2. There are two major components: an offline component which recursively identifies and organizes dense subgraphs in the given network, and an online component which responds to the QMQ search for query nodes S . The offline component, discussed in Section 4.3, transforms the network G into a dense subgraph (DSG) tree to support efficient retrieval of query-driven quasi-cliques. The online component quickly locates the tree node with the closest cohesion to λ (called *pre-solution*) on the DSG tree for query nodes S and starts the iterative maximization process to approach the QMQ, as discussed in Section 4.4. In case no qualified solution is found, the algorithm will return NULL.

To simplify the presentation, we assume throughout the chapter that $S \neq \emptyset$, i.e., the query nodes are non-empty. This assumption does not result in any loss of generality. For the special case $S = \emptyset$, we can augment the graph $G = (V, E)$ to a new graph G' by adding a new node v into G with edges connecting v to every node in G . Let $S' = \{v\}$. Then Q_S^λ is a λ -quasi-clique in G iff $Q_{S'}^\lambda$ is a λ -quasi clique in G' , where $Q_{S'}^\lambda = Q_S^\lambda + \{v\}$.

4.3 Pre-solution on Core Tree

In this section, we propose *core tree*, a convenient representation of a graph where each tree node is a certain dense subgraph. As we will see in Section 4.3.2, core tree facilitates the quick look-up of pre-solution, which is a connected subgraph containing query set S and has cohesion very close to λ . Meanwhile, we are able to confine the size of the QMQ to a small range between the size of a tree node and its parent, which accelerates the QMQ search process.

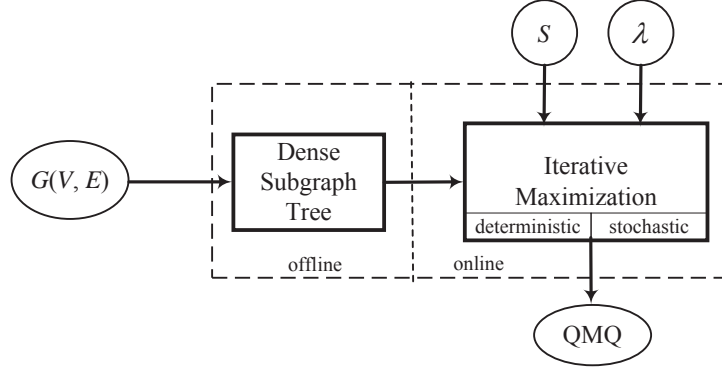


Figure 4.2: Architecture of the efficient query-driven maximum quasi-clique search by DSG tree.

4.3.1 Core Tree: Definition and Properties

In the following, we calibrate a dense subgraph using three measures and define the core tree for a given graph.

Measures. As remarked earlier, degree distributions of real-world networks tend to be skewed, making it common for such networks to have embedded dense subgraphs. Several notions of dense subgraphs have been proposed in the literature, including densest subgraphs, k -cores, k -plexes, and quasi-cliques [47, 73]. In principle, we could have defined a “dense subgraph (DSG) tree” representation of a graph using any of the above notions of dense subgraphs. We provide the rationale for using k -Cores for defining our DSG tree later in this section. To quantify the properties of a dense subgraph, we make use of the CCD measure, namely *Coreness*, *Cohesion* and *Density*, as defined below.

Definition 12 (CCD Measure). Given a connected subgraph $G_i(V_i, E_i)$, the CCD measure of G_i is defined as

- *Coreness*: $\mathcal{K}(G_i) = \min_{v \in V_i} \deg(v, G_i)$;
- *Cohesion*: $\mathcal{C}(G_i) = \frac{\mathcal{K}(G_i)}{|V_i| - 1}$;
- *Density*: $\mathcal{D}(G_i) = \frac{|E_i|}{\binom{|V_i|}{2}} = \frac{2|E_i|}{|V_i|(|V_i| - 1)}$.

Cohesion is the ratio between the minimum degree and the maximum possible degree. Clearly, a λ -quasi-clique is a connected subgraph with cohesion at least λ . Density, also called local clustering coefficient [76], can be

4.3. Pre-solution on Core Tree

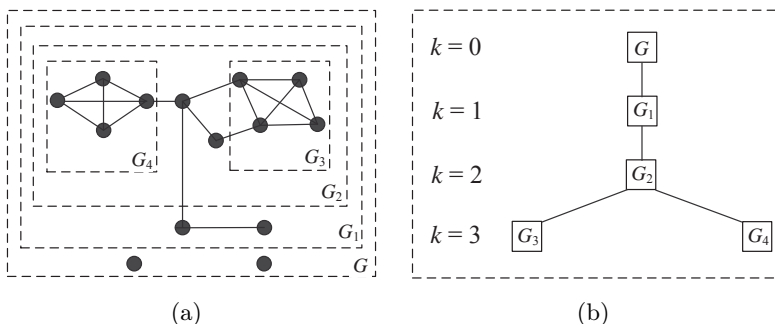


Figure 4.3: A graph G with k -Cores identified recursively in (a) and its corresponding core tree in (b).

viewed as the ratio between the average degree and the maximum possible degree. Both cohesion and density fall in the range $[0, 1]$.

Definition. Conceptually, a core tree is a representation of the given graph as a recursively inclusive tree structure. Each tree node in a core tree is a maximal k -Core.

Definition 13 (Maximal k -Core). A maximal k -Core is a connected subgraph $G_i(V_i, E_i)$ such that:

- $\mathcal{K}(G_i) \geq k$;
- G_i is not a subgraph of any other k -Core.

Definition 14 (Core Tree). Given an undirected graph $G(V, E)$, the core tree of G is a tree \mathbb{T} whose nodes correspond to maximal k -Cores of G , for some k , such that:

- *Root:* The root of \mathbb{T} corresponds to G and has depth 0;
- *Parent-Child:* Whenever \mathbb{T}_i is a tree node of \mathbb{T} at depth k corresponding to a maximal k -Core G_i , and G_j is a maximal $(k + 1)$ -Core which is a subgraph of G_i , then \mathbb{T}_i has a child \mathbb{T}_j corresponding to G_j ; \mathbb{T}_j has depth $k + 1$.

We show an example in Fig. 4.3 to illustrate the core tree. When $k = 1$, isolated nodes are removed. As k increases, the size of k -Core decreases along the path from the root, making the cohesion monotone increasing.

Note that G is trivially a maximal 0-core. It follows that every node of \mathbb{T} at depth k is a maximal k -core. It is possible that multiple tree nodes of \mathbb{T} may correspond to the same subgraph of G . E.g., suppose G has no

isolated nodes. Then G is a maximal 0-core as well as a maximal 1-core. The apparent redundancy in the core tree can be eliminated by a concise storage, as introduced below.

Concise Storage. The height of a core tree \mathbb{T} is the largest k for which G has a k -core (called degeneracy in [47]). This is usually a small number (e.g., below 100) for real world large graphs [28]. Conceptually, since a tree node is a subgraph of its parent, a subgraph may appear repeatedly in multiple levels and bring redundancy. However, there is no need to materialize any subgraphs. Instead, we can represent a tree node \mathbb{T}_i corresponding to $G_i(V_i, E_i)$ by the format:

$$\mathbb{T}_i = (k, V_i, \text{parent}, \text{children}, \text{leftover})$$

where k is the depth of \mathbb{T}_i , *parent* and *children* respectively denote the ID and set of IDs of the parent and children of the tree node \mathbb{T}_i , V_i is the set of nodes in the subgraph of G that \mathbb{T}_i corresponds to, and *leftover* is the set of nodes that are in V_i but not in *any* child.

We can always induce G_i from G by V_i , and for simplicity, we use \mathbb{T}_i and G_i interchangeably. In addition, as an inverted index, for every node v in G , we also remember the ID of the tree node \mathbb{T}_i with highest k that contains v .

Properties. Core trees enjoy many desirable properties by virtue of effectively organizing dense subgraphs at different granularity levels. These properties facilitate efficient search for QMQ. We state and establish these properties below.

Proposition 1 *If \mathbb{T}_i and \mathbb{T}_j are siblings on the core tree \mathbb{T} , then $V_i \cap V_j = \emptyset$.*

Proof: Since \mathbb{T}_i and \mathbb{T}_j are siblings, their corresponding subgraphs G_i and G_j are both maximal k -Cores. Suppose $V_i \cap V_j \neq \emptyset$ and $G_k(V_k, E_k)$ is the union of G_i and G_j , i.e., $V_k = V_i \cup V_j$, $E_k = E_i \cup E_j$. Then G_k satisfies the definition of a k -Core, which contradicts the maximality of G_i and G_j , so $V_i \cap V_j = \emptyset$. \square

Proposition 2 *Cohesion of a tree node is non-decreasing along any root-to-leaf path on the core tree.*

Proof: Based on Def. 14, $\mathcal{K}(G_i)$ is non-decreasing and $|V_i|$ is non-increasing, so cohesion is non-decreasing. \square

Proposition 3 *Given any two nodes \mathbb{T}_i and \mathbb{T}_j on the core tree, if $V_i \cap V_j \neq \emptyset$ and $|V_i| \leq |V_j|$, then $V_i \subseteq V_j$.*

Proof: Since $V_i \cap V_j \neq \emptyset$, \mathbb{T}_i and \mathbb{T}_j are on the same path from the root to a leaf. Since $|V_i| \leq |V_j|$, G_i should be a subgraph of G_j , so $V_i \subseteq V_j$. \square

Prop. 1 indicates the *disjointness* property, which is useful to help prune the search space: once we find the query set S is contained in a tree node \mathbb{T}_i , we can reduce the search space to the branch of \mathbb{T}_i and safely ignore other tree nodes in different root-to-leaf paths. Prop. 2 shows the *monotone increasing* property of cohesion on the path from the root to a leaf in the core tree. Prop. 3 presents the *inclusion* property: if two tree nodes share common graph nodes, they must be on the same root-to-leaf path. Both Prop. 2 and 3 are essential for quasi-clique maximization.

Why Core Trees? Popular dense subgraph definitions include densest subgraph, k -Plex, clique, quasi-clique, k -Core and k -Truss ([34, 47, 73]). In the following, we briefly discuss each of them, and argue why k -Core is the best fit for instantiating a dense subgraph (DSG) tree. First, the monotone property of *Cohesion* shown in Prop. 2 is crucial for the fast search of the QMQ. Densest subgraph is a subgraph that maximizes the average degree, and cohesion of densest subgraphs is not necessarily monotone as the average degree increases. k -Plex is defined as a subgraph $G_i = (V_i, E_i)$, with $\mathcal{K}(G_i) \geq |V_i| - k$, and $\mathcal{C}(G_i) = \frac{|V_i| - k}{|V_i| - 1} = 1 - \frac{k-1}{|V_i| - 1}$. A natural way to define a k -Plex tree is to start with G as the root with $k = |V|$, the number of nodes in G . When we decrease k , the resulting $(k - 1)$ -Plexes are guaranteed to be subgraphs of the subgraph corresponding to the given k -Plex. However, in general, depending on the size of the $(k - 1)$ -Plexes, the cohesion may be more or less than that of the parent. Specifically, consider a k -Plex H with n nodes and a $(k - 1)$ -Plex subgraph H' with n' nodes. We can show that $\mathcal{C}(H') \geq \mathcal{C}(H)$ iff $n' \leq 1 + \frac{k-2}{k-1}(n - 1)$. In general, there is no guarantee on the value of n' and so cohesion is not monotone on a root to leaf path of a DSG tree defined using k -Plexes. Clique and quasi-clique are special cases of query-driven quasi-cliques with $S = \emptyset$, so we cannot assume they are already available. Moreover, all the aforementioned dense subgraph problems are NP-Hard, and are hard to approximate. In contrast, maximal k -Cores can be exactly found in polynomial time [57], so the construction cost of core tree is very cheap. k -Truss [34] is a special kind of $(k - 1)$ -Core, which has the added constraint that every edge should be contained in at least $(k - 2)$ triangles. The time complexity of k -Truss detection is much higher than the k -Core detection – $O(|V| \cdot |E|)$ vs. $O(|V| + |E|)$ – and it has no distinct advantages over k -Cores for the QMQ search. We thus conclude that k -Cores are the ideal choice for instantiating a DSG tree.

Algorithm. Given a graph $G(V, E)$, the procedure for generating a core tree from G is shown in Algorithm 1. Specifically, given a k -Core G_i , we generate all $(k + 1)$ -Cores by recursively removing nodes with degree less

Algorithm 1: Core Tree Generation

Input: $G(V, E)$
Output: Core tree \mathbb{T}

- 1 set G as the root of \mathbb{T} ;
- 2 $Set = \{(G, 0)\}$;
- 3 **while** $Set \neq \emptyset$ **do**
- 4 get (G_i, k) from Set and remove from Set ;
- 5 recursively remove nodes from G_i if degree less than $k + 1$;
- 6 **for** each connected component G_j in the remaining graph of G_i **do**
- 7 set G_j as a child of G_i in \mathbb{T} ;
- 8 add $(G_j, k + 1)$ into Set ;

than $k + 1$, and these $(k + 1)$ -Cores will be added as the children of G_i in core tree \mathbb{T} . Since the highest k is the degeneracy of G , the time complexity of core tree generation is $O(|E| + |V|)$, similarly to the graph degeneracy computation [56].

4.3.2 Search for Pre-Solution

Henceforth, by a *solution*, we mean a λ -quasi-clique containing the query nodes S . By a *pre-solution*, we mean a maximal k -core containing S whose cohesion is closest to that of a λ -quasi-clique. It may fall shy of the required cohesion to be a solution, hence the term pre-solution. A graph G may or may not have a λ -quasi-clique containing S . If it does, it can be obtained efficiently from the pre-solution. Below, we give the formal definition of a pre-solution.

Definition 15 (Pre-Solution). *Given a core tree \mathbb{T} , λ and a query set S , the pre-solution for the QMQ \overline{Q}_S^λ is a tree node \mathbb{T}_i with a set of graph nodes V_i , such that*

$$PreSolution(\lambda, S) = \arg \min\{|\mathcal{C}(\mathbb{T}_i) - \lambda| \mid S \subseteq V_i\} \quad (4.2)$$

Core trees offer two benefits for accelerating QMQ search: (1) given a query node set S and parameter λ , the pre-solution for the QMQ search can be generated efficiently by traversing the core tree; (2) given the pre-solution or the current solution for the QMQ, we can find a better neighboring solution efficiently, by reducing the search space, exploiting the core tree. We use the term graph node and tree node to distinguish between nodes of G and nodes of core tree \mathbb{T} . Recall that each tree node corresponds to a subgraph of G which is a maximal k -core, for some k .

Main Idea. Given a graph G , λ , and S , let \mathbb{T} be the corresponding core tree. Here are the major steps in the search for a pre-solution, which we denote by \mathbb{T}_Q :

1. For each graph node $s \in S$, find the tree node \mathbb{T}_s that contains s and has the largest depth in \mathbb{T} . Let \mathbb{T}_S denote the lowest common ancestor in \mathbb{T} , of the set of the tree nodes $\{\mathbb{T}_s | s \in S\}$;
2. If \mathbb{T}_S is not a λ -quasi-clique, the pre-solution is simply \mathbb{T}_S , i.e., $\mathbb{T}_Q = \mathbb{T}_S$.
3. If \mathbb{T}_S is a λ -quasi-clique, we climb up the core tree \mathbb{T} from \mathbb{T}_S until we find an ancestor \mathbb{T}_l which is a λ -quasi-clique but its parent \mathbb{T}_u is not a λ -quasi-clique.
4. If $|\mathcal{C}(\mathbb{T}_l) - \lambda| \leq |\mathcal{C}(\mathbb{T}_u) - \lambda|$, set the pre-solution $\mathbb{T}_Q = \mathbb{T}_l$; otherwise, set $\mathbb{T}_Q = \mathbb{T}_u$.

Notice that \mathbb{T}_Q is sometimes a λ -quasi-clique (i.e., a solution) and sometimes not (a non-solution). Notice also that \mathbb{T}_s is not necessarily a leaf node. For example, in Fig. 4.3, supposing s is a node in G_1 but not in G_2 , \mathbb{T}_s is G_1 . From the leaf to the root, let $\{\mathbb{T}_s | s \in S\}$ be the set of those tree nodes having the first appearance of query nodes. These are the maximal k -cores with the highest k containing the query nodes. Since the root of core tree \mathbb{T} is the whole graph G , given query S , the lowest common ancestor \mathbb{T}_S for tree node set $\{\mathbb{T}_s | s \in S\}$ always exists. If \mathbb{T}_S is not a λ -quasi-clique, it is still possible to find a subgraph of \mathbb{T}_S which is a query-driven λ -quasi-clique for S . In this case, we let $\mathbb{T}_Q = \mathbb{T}_u = \mathbb{T}_S$ and jump to the optimization phase to be discussed in Section 4.4. Otherwise, \mathbb{T}_S is a λ -quasi-clique, and we climb up the core tree from \mathbb{T}_S and find an ancestor \mathbb{T}_l , which is a λ -quasi-clique but whose parent \mathbb{T}_u is not. According to Prop. 2, such a tree node pair \mathbb{T}_l and \mathbb{T}_u always exists for given \mathbb{T}_S and λ . In the extreme case that $\mathbb{T}_l = \text{Root}$ and $\mathbb{T}_u = \text{NULL}$, we return G as the final solution directly. Otherwise, the pre-solution \mathbb{T}_Q is chosen from \mathbb{T}_l and \mathbb{T}_u , depending on which tree node has a closer cohesion with λ . Pre-solution \mathbb{T}_Q is the start of the iterative optimization discussed in Section 4.4.

Lower/Upper Bounds of QMQ. By leveraging core tree, we can confine the size of the QMQ \overline{Q}_S^λ to a small range. Prop. 4 reveals the relationship between $\mathbb{T}_l, \mathbb{T}_u$ and \overline{Q}_S^λ .

4.3. Pre-solution on Core Tree

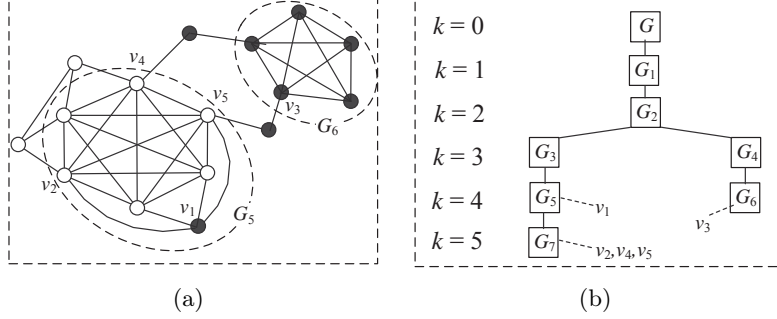


Figure 4.4: (a) A small graph shown and (b) its corresponding core tree, showing deepest tree nodes containing graph nodes v_1, \dots, v_5 ; G_5 and G_6 in (b) are annotated by dotted circles in (a).

Proposition 4 *Given tree nodes \mathbb{T}_l and \mathbb{T}_u on core tree corresponding to subgraphs G_l and G_u , both containing the query S , if $\mathcal{C}(G_l) \geq \lambda \geq \mathcal{C}(G_u)$, then the QMQ $\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q)$ for S and λ satisfies:*

$$|V_l| \leq |\overline{V}_Q| \leq |V_u| \quad (4.3)$$

Proof: Since G_l contains S and $\mathcal{C}(G_l) \geq \lambda$, G_l is a query-driven λ -quasi-clique for S ; so $\overline{Q}_S^\lambda(\overline{V}_Q, \overline{E}_Q)$, being a maximal λ -quasi-clique for given S and λ , satisfies $|V_l| \leq |\overline{V}_Q|$. Notice that $\mathcal{C}(\overline{Q}_S^\lambda) \geq \mathcal{C}(G_u)$, so supposing $|\overline{V}_Q| > |V_u|$, we get that the minimum degree $\mathcal{K}(\overline{Q}_S^\lambda) > \mathcal{K}(G_u)$. This implies \overline{Q}_S^λ is a $\mathcal{K}(\overline{Q}_S^\lambda)$ -Core, and given that $\overline{V}_Q \cap V_u \supseteq S \neq \emptyset$, \overline{Q}_S^λ should be a subgraph of G_u . This contradicts the assumption that $|\overline{V}_Q| > |V_u|$, so we have $|\overline{V}_Q| \leq |G_u|$. \square

Prop. 4 indicates that the size of the QMQ is bounded by the size of G_l and G_u . Although in general, the QMQ \overline{Q}_S^λ shares many common graph nodes with G_l and G_u , \overline{Q}_S^λ is not necessarily a supergraph of G_l or a subgraph of G_u . E.g., in Fig. 4.4(a), for $S = \{v_2, v_4, v_5\}$ and $\lambda = 0.4$, the 4-Core G_5 containing S has cohesion $2/3$, the QMQ (denoted by hollow nodes) has cohesion $3/7$, and the former is not a subgraph of the latter. However, as a maximal k -Core containing S with a very close cohesion to λ , the pre-solution picked from G_l and G_u serves as an ideal candidate for the iterative maximization towards \overline{Q}_S^λ .

Example. The search for pre-solution is illustrated in Fig. 4.4. Suppose $\lambda = 0.2$ and $S = \{v_1, v_2\}$. Then G_5 is the lowest common ancestor \mathbb{T}_S .

Then we climb up the core tree to find $G_l = G_3$ and $G_u = G_2$ that satisfy $\mathcal{C}(G_l) \geq \lambda \geq \mathcal{C}(G_u)$. According to Prop. 4, the QMQ will be a subgraph of G whose size is “sandwiched” between those of G_3 and G_2 . The pre-solution \mathbb{T}_Q is set to G_2 or G_3 , depending on whichever has a cohesion closer to λ . As another example, suppose $\lambda = 0.5$ and $S = \{v_1, v_3\}$. We then get $\mathbb{T}_S = G_2$, which is not a λ -quasi-clique. In this case, we set the pre-solution $\mathbb{T}_Q = G_2$ and the iterative maximization process will check the existence of the QMQ inside G_2 .

4.4 Query-Driven Quasi-Clique Maximization

In this section, we discuss our iterative maximization techniques for searching for a QMQ based on a given pre-solution.

4.4.1 Objective and Operations

Objective. Let $G_i(V_i, E_i)$ be any connected subgraph of G , with $S \subseteq V_i$. G_i may or may not be a λ -quasi-clique. For convenience, if G_i is a λ -quasi-clique, we call G_i a *solution*; otherwise, we call G_i a *non-solution*. Recall that a pre-solution may be a solution or a non-solution. Our objective, given such a graph G_i , is to find a solution and maximize its node size. In order to facilitate our search, we propose a new objective function below. Recall, $\mathcal{D}(G_i)$ denotes the density of G_i (see Def. 12).

$$\mathcal{F}(G_i) = \begin{cases} |V_i| + \mathcal{D}(G_i) & \text{if } G_i \text{ is a } \lambda\text{-quasi-clique} \\ \mathcal{C}(G_i) & \text{otherwise} \end{cases} \quad (4.4)$$

$\mathcal{F}(G_i)$ is designed tactfully and has the following benefits:

- When G_i is a λ -quasi-clique, since $0 \leq \mathcal{D}(G_i) \leq 1$, the term $|V_i|$ dominates the objective function $\mathcal{F}(G_i)$ and $\mathcal{F}(G_i) > 1$. Thus, maximizing $\mathcal{F}(G_i)$ prefers a solution with higher size. On the other hand, the density part $\mathcal{D}(G_i)$ encourages preferring the subgraph with a higher density, among those with the same cardinality. Clearly, a subgraph with a higher density has a higher potential to attract more nodes. In this case, $\mathcal{F}(G_i)$ is optimized to make the size of G_i as large as possible.
- When G_i is not a λ -quasi-clique, the cohesion $\mathcal{C}(G_i)$ stimulates the optimization process by rewarding subgraphs with a higher cohesion. In this case, $\mathcal{F}(G_i) < \lambda \leq 1$ and $\mathcal{F}(G_i)$ is designed to transform G_i from a non-solution to a solution as soon as possible.

Operation	Explanation
<i>Add:</i> $G'_i = G_i + \{v\}$	add a new node v into a λ -quasi-clique G_i such that $\mathcal{F}(G'_i) > \mathcal{F}(G_i)$ Goal: increase the cardinality $ V_i $
<i>Remove:</i> $G'_i = G_i - \{v\}$	remove an existing node v ($v \notin S$) from G_i such that $\mathcal{F}(G'_i) > \mathcal{F}(G_i)$ Goal: increase the cohesion $\mathcal{C}(G_i)$
<i>Swap:</i> $G'_i = G_i + \{v_1\} - \{v_2\}$	add a new node v_1 into a λ -quasi-clique G_i and remove another node v_2 ($v_2 \notin S$) such that $\mathcal{F}(G'_i) > \mathcal{F}(G_i)$ Goal: increase the density $\mathcal{D}(G_i)$

Table 4.2: Three Maximization Operations.

- As another benefit, the maximization of $\mathcal{F}(G_i)$ naturally prevents the cycling of quasi-cliques, i.e., a previous quasi-clique appearing again after a series of add and remove operations (defined in the next paragraph). To appreciate this point, in related work, specifically, Reactive Local Search (RLS, [9]) relies on an extra parameter T to prevent cycling: every time a node is added or removed from their current solution, it cannot be considered for removal or addition for the next T iterations, where T needs to be tuned for different graphs, a tedious process.

Operations. The QMQ search can be viewed as a process of maximizing the objective function $\mathcal{F}(G_i)$ where G_i is initialized to \mathbb{T}_Q , where the pre-solution \mathbb{T}_Q may be a λ -quasi-clique or not. We introduce three operations: *Add*, *Remove* and *Swap* defined in Table 4.2 and explained below.

- *Add:* $G'_i = G_i + \{v\}$. *Add* operation is applied when both G_i and G'_i are λ -quasi-cliques.
- *Remove:* $G'_i = G_i - \{v\}$. If G_i is not a λ -quasi-clique, *Remove* operation is applied to improve the cohesion. To make $\mathcal{F}(G'_i) > \mathcal{F}(G_i)$, we need to ensure $\mathcal{K}(G'_i) \geq \mathcal{K}(G_i)$, i.e., we cannot remove a node v that decreases the coreness.
- *Swap:* $G'_i = G_i + \{v_1\} - \{v_2\}$. *Swap* is applied when adding a node v_1 makes G_i lose the λ -quasi-clique property, but removing another node v_2 at the same time restores the property. Instead of viewing *Swap* as a combination of *Add* and *Remove*, we view swap as an atomic operation since that allows us to move from one λ -quasi-clique to another. The motivation for swapping nodes is that the node swapped in may increase the density. Note that edges associated with a node are added or removed automatically with the node.

If the pre-solution $\mathbb{T}_Q = G_l$, which is a λ -quasi-clique, to ensure $\mathcal{F}(G'_i) > \mathcal{F}(G_i)$, *Add* and *Swap* will be used in the iterative maximization. If the pre-solution $\mathbb{T}_Q = G_u$, since $\mathcal{C}(G_u) < \lambda$, on each iteration before $\mathcal{C}(G_i) \geq \lambda$ is satisfied, G_i should be a subgraph of G_u with coreness $\mathcal{K}(G_i) \geq \mathcal{K}(G_u)$. Prop. 5 explains this case and shows that *Add* cannot help improve the $\mathcal{F}(G_i)$ score, and similarly for *Swap*. Thus, when the optimization process starts from G_u , we can only use *Remove* to improve the $\mathcal{F}(G_i)$ score until G_i becomes a λ -quasi-clique.

Proposition 5 *Suppose the iterative optimization starts from $\mathbb{T}_Q = G_u$ and the current state G_i is a non-solution, obtained by zero or more Remove operations to G_u . Then adding a node v to G_i cannot make $\mathcal{F}(G_i + \{v\}) > \mathcal{F}(G_i)$.*

Proof: Recall that a Remove operation is applied to a current state G_i only when the resulting \mathcal{F} score does not decrease. We prove the result by induction on the number of Remove operations applied to $\mathbb{T}_Q = G_u$.

Base Case: Zero Remove operations are applied to G_u . Then since $G_i = G_u$ is a maximal k -Core, adding a node into G_u will decrease the coreness and make $\mathcal{F}(G_i + \{v\}) < \mathcal{F}(G_i)$.

Induction: Suppose G_i is a non-solution obtained from G_u after m Remove operations. Notice that if any of the predecessors of G_i obtained during the Remove sequence is G_l , we could not have applied Remove to it while keeping the \mathcal{F} score non-decreasing. From this, it follows that G_i cannot be a subgraph of G_l . By the rule for applying the Remove operation, $\mathcal{K}(G_i) \geq \mathcal{K}(G_u)$. Let $G'_i = G_i + \{v\}$. By Eq. (4.4), since G_i is not a λ -quasi-clique, we have $\mathcal{F}(G_i) = \mathcal{C}(G_i) = \frac{\mathcal{K}(G_i)}{|V_i|-1}$. To make $\mathcal{F}(G'_i) = \frac{\mathcal{K}(G'_i)}{|V_i|} > \frac{\mathcal{K}(G_i)}{|V_i|-1}$, we have to make $\mathcal{K}(G'_i) > \mathcal{K}(G_i)$. Since $\mathcal{K}(G_l) = \mathcal{K}(G_u) + 1$, we have $\mathcal{K}(G'_i) \geq \mathcal{K}(G_l)$. Notice that G'_i and G_l share the node set S , we conclude by Proposition 1 that G'_i should be a subgraph of G_l . This conflicts with our inference that G_i is not a subgraph of G_l , so $\mathcal{F}(G_i + \{v\}) > \mathcal{F}(G_i)$ does not hold for any v . \square

4.4.2 Efficient Solution Search and Rank

Given a connected subgraph G_i which may be a solution or a non-solution containing S , in this section, we discuss efficient techniques for identifying neighboring subgraphs of G_i with better \mathcal{F} score, using three search operations. According to Eq. (4.4), if G_i is not a λ -quasi-clique, improving its \mathcal{F} score amounts to improving its cohesion. If it is, then improving its \mathcal{F} score amounts to increasing the size, and, if the size cannot be increased, improving the density.

We let $E(V_1, V_2)$ denote the set of edges connecting the node sets V_1 and V_2 . Below, we discuss the solution search using *Add*, *Swap* and *Remove* respectively.

Search by Add. *Add* is designed to increase the existing quasi-clique size. The solution set for *Add* is

$$Add(G_i) = \{v | \mathcal{F}(G'_i) > \mathcal{F}(G_i) > 1, G'_i = G_i + \{v\}\} \quad (4.5)$$

That is, $Add(G_i)$ is the set of single nodes that could be added to G_i to improve its \mathcal{F} score. According to Eq. (4.4) and Def. 12, since $|V'_i| = |V_i| + 1$ is fixed, The only thing that discriminates between different candidate nodes $v \in Add(G_i)$ in terms of \mathcal{F} score is $|E'_i|$. Thus, we can start from G_i and rank neighboring nodes $v \notin V_i$ by the edge size $|E(G_i, \{v\})|$, provided G'_i is still a λ -quasi-clique. The node v with the highest $|E(G_i, \{v\})|$ and satisfying $\mathcal{C}(G_i + \{v\}) \geq \lambda$ produces the best solution G'_i with the highest \mathcal{F} score. Notice that we do not need to check every neighboring node of G_i to find out the best solution: since G_i is the subgraph of some tree node \mathbb{T}_I with $\mathcal{K}(\mathbb{T}_I) \leq \mathcal{K}(G_i)$, if there is a graph node v in \mathbb{T}_I but not in G_i with $|E(G_i, \{v\})| \geq \mathcal{K}(\mathbb{T}_I)$, then for any graph node v' not in \mathbb{T}_I , we have $|E(G_i, \{v'\})| < |E(G_i, \{v\})|$. Otherwise, if $|E(G_i, \{v'\})| \geq \mathcal{K}(\mathbb{T}_I)$, then $|E(\mathbb{T}_I, \{v'\})| \geq |E(G_i, \{v'\})| \geq \mathcal{K}(\mathbb{T}_I)$, making $\mathbb{T}_I + \{v'\}$ a $\mathcal{K}(\mathbb{T}_I)$ -core, violating its maximality. For example, in Fig. 4.4(b), supposing G_i is the subgraph of the tree node G_3 , if there is a graph node v in G_3 but not in G_i having $|E(G_i, \{v\})| \geq 3$, then we cannot find a node v' outside G_3 but having $|E(G_i, \{v'\})| \geq |E(G_i, \{v\})|$. Thus, starting from the deepest tree node that contains G_i , we can climb up the core tree level by level to check whether in that tree node \mathbb{T}_I we can find a graph node v not in G_i but with $|E(G_i, \{v\})| \geq \mathcal{K}(\mathbb{T}_I)$. Once found, we stop the solution search and the current best solution is guaranteed to be the best solution out of all solutions in $G(V, E)$. This approach can be extended to find exact top n solutions (shown in Alg. 3). With guaranteed quality, since we do not need to check every neighbor of G_i , the search space of *Add* is distinctly smaller than that of existing approaches [1, 15].

Search by Swap. Let $G'_i = G_i + \{v_1\} - \{v_2\}$ and $G''_i = G_i + \{v_1\}$. *Swap* is designed to improve the density when G''_i is not a λ -quasi-clique, but both G_i and G'_i are. The solution set of *Swap* is

$$\begin{aligned} Swap(G_i) = \{(v_1, v_2) | \mathcal{F}(G'_i) > \mathcal{F}(G_i) > 1 \geq \lambda > \mathcal{F}(G''_i), \\ G'_i = G_i + \{v_1\} - \{v_2\}, \\ G''_i = G_i + \{v_1\}\} \end{aligned} \quad (4.6)$$

Swap indicates $\deg(v_2, G_i) \geq \lambda(|V_i| - 1)$, $\deg(v_2, G_i'') < \lambda(|V_i|)$ and $\deg(v_1, G_i') > \deg(v_2, G_i)$. In practice, if $Add(G_i) = \emptyset$ and there is a node v_2 ($v_2 \notin S$) in G_i satisfying $\deg(v_2, G_i'') < \lambda(|V_i|)$ and $\deg(v_1, G_i') > \deg(v_2, G_i)$, the swap operation may happen. We rank the node pair (v_1, v_2) by $\deg(v_1, G_i') - \deg(v_2, G_i)$ for *Swap* solutions.

Search by Remove. *Remove* is designed to improve the cohesion of a non-solution G_i . We define the neighboring (solution or non-solution) set of *Remove* as

$$\begin{aligned} Remove(G_i) = \{v | \mathcal{F}(G_i') > \mathcal{F}(G_i), \mathcal{F}(G_i) < \lambda \leq 1, \\ G_i' = G_i - \{v\}\} \end{aligned} \quad (4.7)$$

Clearly, v is a node in G_i . We can rank candidate v in G_i by the coreness change $\Delta\mathcal{K} = \mathcal{K}(G_i') - \mathcal{K}(G_i)$ resulting from removing v from G_i . If $\Delta\mathcal{K} < 0$, then $v \notin Remove(G_i)$, since the \mathcal{F} score decreases. Thus, we cannot simply remove the node with the minimum degree inside G_i : suppose v_1 and v_2 both have the minimum degree in G_i and they are connected; then removing of v_1 will make the coreness of the remaining graph lower than before, which results in a lower \mathcal{F} score. If multiple candidate nodes have the same coreness change $\Delta\mathcal{K}$, we rank them further by their degree in G_i . Removing the node with the highest $\Delta\mathcal{K}$ and lowest degree in G_i maximizes the gain on \mathcal{F} and density, while a higher density has a higher potential to increase the cohesion.

4.4.3 Iterative Maximization Algorithms

Since the maximum clique problem is NP-Hard and is inapproximable within a factor of $n^{1-\epsilon}$ ([6, 23, 31]), the design of heuristic algorithms for QMQ search with a guaranteed quality is unrealistic. In this section, we propose two iterative maximization algorithms, DIM and SUM, to approach the QMQ. DIM (Deterministic Iterative Maximization) is a deterministic approach which greedily moves to the best neighboring solution or non-solution on each iteration. In contrast, SUM (Stochastically Updated Maximization) is a stochastic approach which moves to one of several good neighboring solutions or non-solutions with a probability, which has the potential to find better results than DIM. Both DIM and SUM are based on local search on core tree. In practice, we find that both of them find solutions with high quality, if a QMQ containing the query nodes exists.

If the current state G_i is a non-solution, according to Prop. 5, both DIM and SUM will try to remove nodes from G_i and make it become a solution (i.e., a λ -quasi-clique containing S). Once G_i is a solution, DIM

and SUM will try to apply *Add* or *Swap* to grow it. In this section, since $\mathcal{F}(G_i)$ provides a uniform way to evaluate G_i , no matter whether G_i is a solution or non-solution, for ease of presentation, we use the term “answer” as a generic term that may refer to a solution or a non-solution.

Local Search. Readers are referred to [33, 52] for complete details of local search. Here, we briefly revisit major steps:

1. Given the current answer, find feasible candidate answers in the neighborhood by one of the operations;
2. Evaluate candidates using the objective function \mathcal{F} ;
3. If the marginal gain is positive, move to the best candidate, and then jump to step 1. Otherwise, return the current solution or NULL if the current answer is a non-solution.

Deterministic Iterative Maximization. DIM follows the greedy strategy by always moving to the best available neighboring answer. Major steps of the DIM approach are shown in Alg. 2. The pre-solution \mathbb{T}_Q is set as the starting point of the maximization. If G_i is a non-solution ($\mathcal{C}(G_i) < \lambda$), we repeat *Remove* operations until G_i becomes a λ -quasi-clique or there is no qualified λ -quasi-clique containing S in G (line 4-8). If G_i is already a solution, we repeat *Add* and *Swap* operations iteratively, until $\mathcal{F}(G_i)$ cannot be improved any more (line 9-23). In particular, we start from *Add* operations by exploring neighbors on core tree level by level, and if qualified nodes can be found for *Add*, we pick the best node to add (line 10-18); otherwise, we try to *Swap* nodes (line 20-23). We repeat *Add* and *Swap* until the \mathcal{F} score cannot be improved. An example for the scheduling of operations is shown in Fig. 4.5(a), in which *Remove* operations are performed first if \mathbb{T}_Q is not a λ -quasi-clique, followed by *Add* or *Swap* operations. Supposing the total number of iterations of DIM is T and on each iteration, on an average, n answers are explored, time complexity of DIM is $O(Tn)$.

DIM always picks the best available neighboring answer on each iteration. While this greedy heuristic is effective and is followed by lots of local search methods, it may converge to a local maximum.

In the following, we propose SUM (Stochastically Updated Maximization), which is a stochastic approach, moving with a probability to one of the good neighboring answers. SUM has the potential to break the local maximum limitation and find better results than DIM.

Stochastically Updated Maximization. Given the current answer G_i , we define the marginal gain $\Delta\mathcal{F} = \mathcal{F}(G'_i) - \mathcal{F}(G_i)$ for a neighboring answer

Algorithm 2: DIM

Input: Pre-solution \mathbb{T}_Q, S, λ
Output: the QMQ \overline{Q}_S^λ

```

1 if  $\mathbb{T}_Q$  is the root node  $G$  and  $\mathcal{C}(\mathbb{T}_Q) \geq \lambda$  then return  $G$  ;
2  $G_i = \mathbb{T}_Q$ ;
3 while  $G_i$  is not a  $\lambda$ -quasi-clique do
4   Rank node  $v$  in  $G_i$  by  $\Delta = \mathcal{K}(G_i - \{v\}) + \mathcal{D}(G_i - \{v\}) - \mathcal{K}(G_i) - \mathcal{D}(G_i)$ ;
5   Remove the node not in  $S$  with the highest positive  $\Delta$ ;
6   if  $G_i$  is not connected or not changed then return NULL ;
7 while true do
8    $AddSet = \emptyset$  and  $\mathbb{T}_I =$  the deepest tree node with  $G_i$ ;
9   while  $\max |E(G_i, v)| < \mathcal{K}(\mathbb{T}_I)$  for  $v$  in  $\mathbb{T}_I$  but not in  $G_i$  and  $AddSet$  do
10    Add  $v$  into  $AddSet$  if  $\mathcal{C}(G_i + \{v\}) \geq \lambda$ ;
11    if  $\mathbb{T}_I$  is not the root then
12       $\mathbb{T}_I =$  the parent of  $\mathbb{T}_I$ ;
13    else Break WHILE loop;
14  if  $AddSet \neq \emptyset$  then
15    Add  $v = \arg \max \{|E(G_i, v)| \mid v \in AddSet\}$  into  $G_i$ ;
16  else
17     $v_1 = \arg \max \{|E(G_i, v)| \mid v \in V - V_i\}$ ;
18     $G_i'' = G_i + \{v_1\}, G_i' = G_i + \{v_1\} - \{v_2\}$ ;
19    if it exists  $v_2 \in V_i - S$  such that  $\deg(v_2, G_i'') < \lambda(|V_i|)$  and
20       $\deg(v_1, G_i') > \deg(v_2, G_i)$  then  $G_i = G_i'$  ;
    else Break WHILE loop and return  $G_i$ ;

```

G_i' . As discussed in Section 4.4.2, for each operation, neighboring answers are ranked by $\Delta\mathcal{F}$ in descending order, and we use the function $\mathbb{F}(x)$ to denote the marginal gain of a solution ranked at the x -th place, e.g., $\mathbb{F}(1)$ denotes the maximum marginal gain.

In related work, GRASP [1] is a well-known randomized local search method based on the threshold $\delta = \min \mathbb{F}(x) + c(\max \mathbb{F}(x) - \min \mathbb{F}(x))$ where $c \in [0, 1]$. The new answer is picked randomly from all answers with $\mathbb{F}(x) \geq \delta$. If $c = 1$, GRASP always moves to the best answer like DIM. If $c = 0$, GRASP performs a random selection from all neighboring answers. However, GRASP has two drawbacks: (1) the parameter c is difficult to tune; (2) GRASP totally ignores the actual distribution of marginal gains as x varies. In our proposed Stochastically Updated Maximization (SUM) algorithm, we overcome these limitations by picking the new answer based on the ‘‘inflection point’’ $(x^*, \mathbb{F}(x^*))$ on $\mathbb{F}(x)$. The inflection point is the point when the $\mathbb{F}(x)$

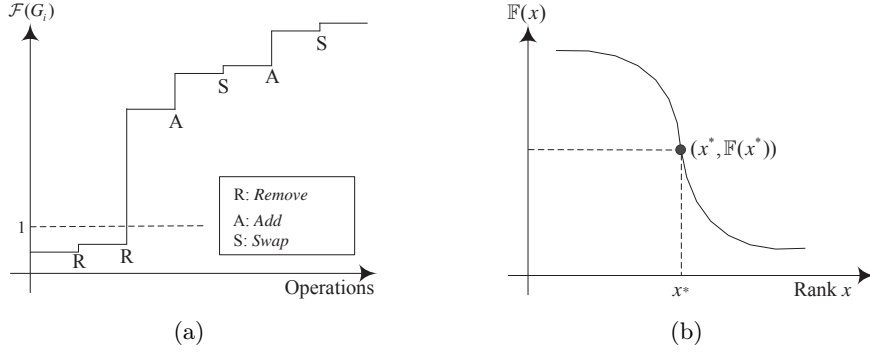


Figure 4.5: (a) An example for the QMQ maximization. (b) Illustrating the inflection node on $\mathbb{F}(x)$, where solutions with rank $x \leq x^*$ are preferred.

change is maximized. That is,

$$x^* = \arg \max_x (\mathbb{F}(x) - \mathbb{F}(x+1)) \quad (4.8)$$

The inflection point splits $\mathbb{F}(x)$ into a high marginal gain part and a low marginal gain part. Fig. 4.5(b) shows an illustration of the inflection point on $\mathbb{F}(x)$. SUM draws an answer from all high marginal gain answers with rank $x < x^*$, using the probability:

$$\mathbb{P}(x) = \frac{\mathbb{F}(x)}{\sum_{\mathbb{F}(x) \geq \mathbb{F}(x^*)} \mathbb{F}(x)} \quad (4.9)$$

$\mathbb{P}(x)$ can be understood as the ratio between answer x 's marginal gain and the sum of all marginal gains higher than $\mathbb{F}(x^*)$. It is possible that an answer is not the best but is picked as the new answer, though its probability of being picked is lower than that of the best answer. On the other hand, since non-greedy answers can be selected with a non-zero probability, SUM has the potential to avoid being trapped in a local maximum. Clearly, the output of DIM is just one possible output of SUM, and SUM may produce much better results than DIM.

Major steps of SUM are shown in Alg. 3. As can be seen, SUM and DIM have a similar scheduling strategy for maximization operations. However, SUM probabilistically picks the new answer from a set of good answers. The preparation for the solution set *SolGain* can be found in lines 5-8 for *Remove*, lines 14-20 for *Add* and lines 24-28 for *Swap*. Supposing the total number of iterations of SUM is T and on each iteration the marginal gain is evaluated on n answers, the time complexity of each simulation of SUM is $O(Tn^2)$.

4.5 Experimental Study

All experiments are conducted on a computer with Intel 2.90 GHz CPU and 8 GB RAM. All algorithms are implemented in Java. We conduct experiments on three real data sets, as explained below.

LiveJournal Social Network. LiveJournal is an online blogging website with a friendship network. We model each user as a node, and friendship ties as edges in LiveJournal social network. Available at SNAP⁵, this dataset has 3,997,962 nodes and 34,681,189 edges in total. A social circle in LiveJournal social network is best modeled by a quasi-clique, since every user is required to declare friendship with the majority of members in a social circle so defined. The query-driven maximum quasi-clique corresponds to the largest social circle containing the query users, which supports applications like social community search.

DBLP Co-Authorship Network. To simulate social collaborations, we use the DBLP⁶ Co-Authorship Network. Each author is modeled as a node, and if two authors have collaborated on at least one paper, there will be a link between them. In total, DBLP co-authorship network has 1,206,881 nodes and 8,894,745 edges. A quasi-clique in the co-authorship network is a group of authors closely working with each other. Given a set of query researchers, the query-driven maximum quasi-clique search seeks to find the largest collaboration network between them.

Youtube Sharing Network. This dataset, available at SNAP⁷, is based on Youtube’s social network, where two people are connected if they have similar interests in videos and become friends. There are a total of 1,134,890 nodes and 2,987,624 edges. Given a set of users, the query-driven maximum quasi-clique is the largest video sharing group containing these query users.

4.5.1 Core Tree Construction

Core tree construction is an offline step. The running time for core tree construction on three datasets is shown in Table 4.3, where we also show the highest k (i.e., degeneracy [47]) for each dataset. Considering that LiveJournal and DBLP datasets have 4M and 1.2M nodes respectively, DBLP dataset takes more time than LiveJournal dataset, which indicates that apart from the graph size, graph structure is a significant factor influencing the perfor-

⁵<http://snap.stanford.edu/data/com-LiveJournal.html>

⁶<http://dblp.uni-trier.de/xml/>

⁷<http://snap.stanford.edu/data/com-Youtube.html>

4.5. Experimental Study

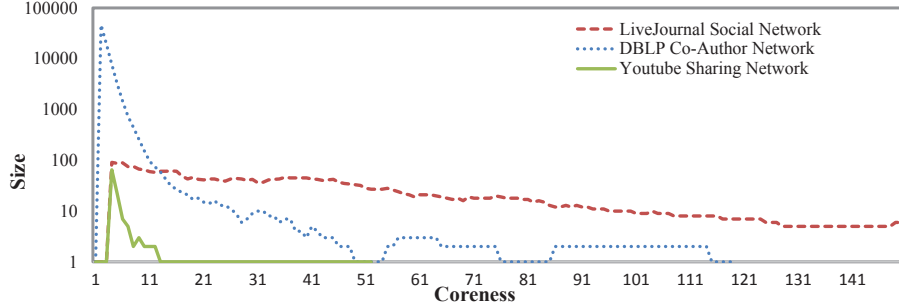


Figure 4.6: The number of maximal cores first rapidly increases, and then slowly decreases with the coreness k on three data sets.

mance of the core tree construction. Specifically, in Fig. 4.6, we show the number of maximal k -Cores for each k , for the three data sets. As we can see, from the root to leaves, the number of maximal cores first rapidly increases, and then slowly decreases with the coreness k for all datasets. Especially, there is a huge number of 2-Cores and 3-Cores in DBLP co-authorship network, because there are many graduate students only publishing papers with their supervisors or a few collaborators. This is a dominant factor in the core tree construction time, as this means the core tree for DBLP has many more nodes than for the other datasets.

4.5.2 Performance Evaluation

Baselines. To the best of our knowledge, there is no previous study on query-driven maximum quasi-clique search. Therefore, we adapt previous approaches for maximum quasi-clique detection (without query) as baselines. In our evaluation, we designed two categories of baselines based on related work: *operation baselines* and *optimization baselines*. For operation baselines, we use Add-MC and Remove-MC, which are discussed in [15] and are extended from traditional maximum clique problem. These baselines are compared with *Add*, *Remove* and *Swap* discussed in Section 4.4.2.

- **Add-MC:** Call a node u in a λ -quasi-clique G_i a *critical* node if $\lambda(|V_i| - 1) \leq \text{deg}(u, G_i) < \lambda|V_i|$. Then we add a node v with the maximum $\text{deg}(v, G_i + \{v\})$ to the quasi-clique G_i such that: (i) $\text{deg}(v, G_i + \{v\}) \geq \lambda|V_i|$ and (ii) v is adjacent to every critical node of G_i .
- **Remove-MC:** Remove a node v from G_i whose removal results in the largest set of Add-MC nodes.

For optimization baselines, we use RLS (Reactive Local Search) [15],

4.5. Experimental Study

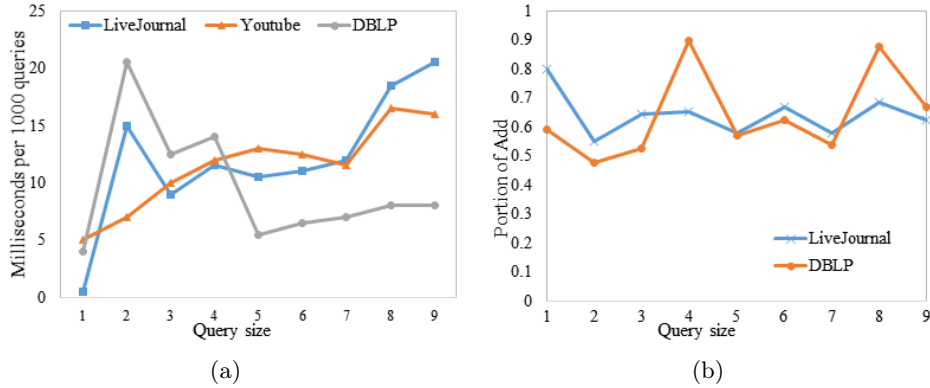


Figure 4.7: (a) Running time of 1000 pre-solution queries as the increasing of query set size. (b) Portion of pre-solution chosen from G_l , out of all pre-solutions, for different query set sizes. $\lambda = 0.9$.

GRASP [1], and DFS-Tree [50, 74], which are compared with DIM and SUM.

- **RLS**: At each step, choose the best Add-MC node. If no nodes can be added, choose the best Remove-MC node. Every time a node is added or removed, it cannot be removed or added again for the next move.
- **GRASP**: A randomized local search method based on thresholds. We set $c = 0.5$ and adapt GRASP on core tree by picking a new solution randomly from all solutions with objective scores higher than the average.
- **DFS-Tree**: Build depth-first search trees for quasi-cliques by setting each query node as the root, and return the largest tree node in these trees that contains all query nodes, adapted from [50, 74].

Pre-Solution on Core Tree. Finding a pre-solution is one of the key steps of our method. In this section, we evaluate the efficiency of our algorithm for finding a pre-solution. We generate queries with q nodes by randomly choosing q nodes from a subgraph within radius 2, and this subgraph is itself randomly chosen from the original graph. The core tree built in the offline step provides an extremely efficient way to find the pre-solution for a given query. In Fig. 4.7(a), we show the running time of 1000 pre-solution queries for different query sizes. With the help of core tree, the pre-solution search for λ and S reduces to a few tree traversal operations and is extremely efficient: *1000 pre-solution searches can be done in a few milliseconds, on data sets with millions of nodes.* Except for two special cases – the whole graph is a λ -quasi-clique or there is no tree node that is a λ -quasi-clique containing

S – a pre-solution will be chosen from either G_l or G_u , depending on the proximity of their cohesion to λ . As shown in Fig. 4.7(b), the majority of pre-solutions are chosen from G_l , for which *Add* and/or *Swap* are applicable.

Maximization. The efficiency of an iterative maximization process is determined by two key aspects: the cost of a single iteration and the convergence rate. We evaluate the performance of iterative maximization from these two aspects respectively.

Fig. 4.8(a) shows the running time for searching neighboring solutions using *Add* and Add-MC with different sizes (i.e., number of nodes) of current solution G_i on LiveJournal. Due to given graph structure, different G_i may have very different numbers of neighbors, resulting in a skewed distribution of running time. On average, *Add* operation is 2.6 times faster than Add-MC: each *Add* takes 24 ms, while each Add-MC takes 62 ms. This can be explained by the observation that an exploration using *Add* has a remarkably smaller search space than Add-MC, since *Add* searches solutions by exploring neighboring nodes level by level and stops immediately if top n solutions are already found on core tree, with n set to a small number, e.g., 10. But Add-MC needs to check every neighboring node to find top solutions. Notice that for SUM, *Add* finds top- n solutions whereas for DIM, only the top 1 solution is needed. Fig. 4.8(b) compares the ratio of the search space for *Add* to that for Add-MC. As query size increases, the ratio between the search spaces of *Add* and Add-MC decreases, indicating that *Add* only needs to search a fraction of the search space of Add-MC, as the query size grows. The added value of *Add* over Add-MC in the quality of results will be discussed in detail below (see Table. 4.4).

Next, given a pre-solution as the starting point of optimization, we measure the number of iterations taken by each optimization method before convergence. For every query size, we repeat the experiment 100 times and show in Fig. 4.8(c) the average number of iterations for each of the four maximization methods: the numbers show a common decreasing trend as the query size increases. This makes sense because larger query size is more likely to result in a larger solution, which sets a higher bar to add new nodes. Fig. 4.8(c) also shows that SUM has the highest number of iterations, but we will see later that SUM also succeeds in finding larger quasi-cliques more often than all other methods.

Total Running Time. Unlike RLS and GRASP, DFS-Tree baseline is not an iterative optimization method. Instead, “DFS-Tree” represents a category of methods [50, 74] that find maximal quasi-cliques by exploring the space of quasi-cliques in a depth-first search tree, in which each tree node corresponds

to a quasi-clique. Since the DFS-Tree method is not originally designed for the QMQ search problem, to use it as a baseline, we adapt it by applying pruning techniques and returning the largest node of the DFS tree (corresponding to a quasi-clique) containing all query nodes. In Fig. 4.8(d), we show the average running time of these methods on LiveJournal, as the query size $|S|$ increases. RLS, GRASP, DIM, and SUM have comparable running times, with RLS being slower than the other three. This is explained by the fact that RLS uses Add-MC and Remove-MC in place of the faster *Add* and *Remove*. DFS-Tree quickly explodes as the query size increases. In fact, its running time trend is opposite to that of DIM and SUM. This is because the DFS-Tree method needs to build a DFS tree for every query node in S and the cost of searching the largest tree node containing all query nodes increases with $|S|$, making DFS-Tree based methods consume more memory and perform worse than our core tree based methods, DIM and SUM. The running time of GRASP is comparable to SUM.

Quality Comparison. As discussed in Section 4.2, there is unlikely to be any heuristic algorithm with guaranteed approximation for the optimal QMQ. To measure the quality of proposed approaches empirically, we designed the following experiment: given the same pre-solution, we run RLS, GRASP, DFS-Tree, DIM and SUM independently to get the QMQ w.r.t. S and λ . We rank these QMQs by size, and the method producing the largest QMQ will earn a hit score of one point. By randomly selecting queries with size 3, we repeat the experiment 100 times, and list the final score for each method in Table 4.4. Notice that multiple methods may generate the same final QMQ and may earn a hit score in a given round, so the sum of scores is higher than 100. The score of GRASP is the lowest, because it uniformly selects a neighboring solution from a set of above-average solutions, and in many cases, GRASP ends up with not-the-best final solutions. Theoretically, DFS-Tree should yield a high score overall, but since we have to prune the search path to control its memory consumption [50, 74], it generates an overall score only slightly higher than GRASP. Both RLS and DIM greedily move to a new solution with the highest marginal gain, but since DIM uses *Swap* to improve the density while RLS does not provide *Swap*, DIM achieves a higher quality score than RLS. Finally, SUM achieves the highest overall score, because not only is SUM capable of using *Swap* to improve the density, it is able to strike a balance between greed and opportunity: while it is more probable to move to a new solution with a higher marginal gain, it retains the potential to jump out of the local maximum and thus achieve a better final solution, thanks to the non-zero probability of choosing

a neighboring solution that is not (locally) the best.

4.6 Discussion and Conclusion

In this chapter, we discussed the cohesion-persistent story search problem. Given a query consisting of a set of nodes S in a graph $G(V, E)$, and parameter $\lambda \in (0, 1]$, we focus on finding the largest λ -quasi-clique containing S , which has lots of applications in real world networks. This is NP-hard and is hard to approximate, calling for clever heuristic solutions. To quickly find a dense subgraph containing S with cohesion close to λ , as the pre-solution of the final solution, we propose the notion of core tree by recursively organizing maximal cores of G . We make use of three optimization operations: *Add*, *Remove* and *Swap*. Then, we propose two iterative maximization algorithms, DIM and SUM, to approach the query-driven maximum quasi-clique in terms of deterministic and stochastic means respectively. With extensive experiments on three real datasets, we demonstrate that our algorithms significantly outperform several natural baselines based on the state of art, in running time and/or the quality of the solution found. This work raises a number of open questions. It's interesting to ask if we can apply the operations in bulk mode, at the level of subgraphs, instead of the node-by-node mode that we have taken. It is also important to investigate whether our techniques can be extended and generalized to search other kinds of dense subgraphs such as k -plexes or optimal quasi-cliques [73].

Algorithm 3: SUM

Input: Pre-solution $\mathbb{T}_Q, S, \lambda, n$
Output: the QMQ \overline{Q}_S^λ

```

1 if  $\mathbb{T}_Q$  is the root node  $G$  and  $\mathcal{C}(\mathbb{T}_Q) \geq \lambda$  then return  $G$  ;
2  $G_i = \mathbb{T}_Q$ ;
3 while  $G_i$  is not a  $\lambda$ -quasi-clique do
4   Set  $SolGain = \emptyset$ ;
5   for each node  $v$  in  $G_i$  but not in  $S$  do
6      $Solution = G_i - \{v\}$ ,  $Gain = \mathcal{F}(solution) - \mathcal{F}(G_i)$ ;
7     Add  $(Solution, Gain)$  into  $SolGain$ ;
8   if  $SolGain \neq \emptyset$  then
9      $G_i =$  Stochastically pick an answer from  $SolGain$  with gain higher
10    than  $\mathbb{F}(x^*)$ ;
11  else Return NULL ;
12 while true do
13    $AddSet = \emptyset$  and  $\mathbb{T}_I =$  the deepest tree node with  $G_i$ ;
14   while top  $n$ -th  $|E(G_i, v)| \leq \mathcal{K}(\mathbb{T}_I)$  for  $v$  in  $\mathbb{T}_I$  but not in  $G_i$  and
15    $AddSet$  do
16     Add  $v$  into  $AddSet$  if  $\mathcal{C}(G_i + \{v\}) \geq \lambda$ ;
17     if  $\mathbb{T}_I$  is not the root then
18        $\mathbb{T}_I =$  the parent of  $\mathbb{T}_I$ ;
19     else Break WHILE loop;
20   if  $AddSet \neq \emptyset$  then
21     Stochastically pick  $v$  from nodes in  $AddSet$  with gain higher than
22      $\mathbb{F}(x^*)$  and let  $G_i := G_i + \{v\}$ ;
23   else
24     Set  $SolGain = \emptyset$ ;
25     for each node  $v_1 \in V - V_i$  with  $E(G_i, v_1) > deg(v_2, G_i) + 1$  do
26        $G''_i = G_i + \{v_1\}$ ,  $G'_i = G_i + \{v_1\} - \{v_2\}$ ;
27       if it exists  $v_2 \in V_i - S$  such that  $deg(v_2, G''_i) < \lambda(|V_i|)$  and
28        $deg(v_1, G'_i) > deg(v_2, G_i)$  then
29         Add  $(G'_i, \mathcal{F}(G'_i) - \mathcal{F}(G_i))$  into  $SolGain$ ;
30     if  $SolGain \neq \emptyset$  then
31        $G_i =$  Stochastically pick an answer from  $SolGain$  with gain
32       higher than  $\mathbb{F}(x^*)$ ;
33     else Break WHILE loop and return  $G_i$  ;

```

4.6. Discussion and Conclusion

Data Sets	LiveJournal	DBLP	Youtube
Time (seconds)	346	486	29
Highest Coreness	361	119	52

Table 4.3: Running time for core tree construction.

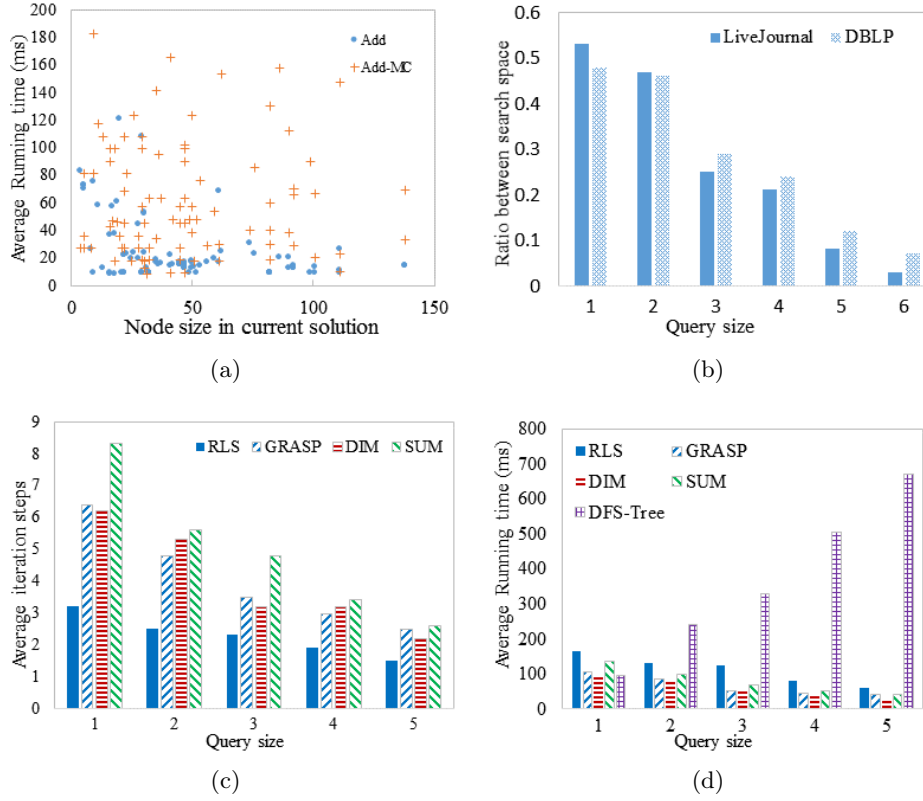


Figure 4.8: (a) Running time of *Add* and *Add-MC* for different solution sizes, with query node size $|S| = 3$; (b) ratio between the search spaces of *Add* and *Add-MC*, given current solution, as query size increases; (c) average #iterations for various methods on LiveJournal data set; (d) average running time of different methods on LiveJournal, as query size increases; $\lambda = 0.9$ by default.

Method	SUM	DIM	DFS-Tree	GRASP	RLS
Score	77	41	11	9	17

Table 4.4: Aggregated quality scores for different methods from 100 tests.

Chapter 5

Context-Aware Story-Telling

Stories that we identified from the post network may be highly related, e.g., two stories “the launch of Blackberry 10” and “BlackBerry Super Bowl ad” are highly related. The detection of story relatedness, or called story context search, is a fundamental task of social stream mining. Story context search aims at finding the neighboring stories of a given story in the post network efficiently. In this chapter, We try to exploit and integrate signals from different perspectives (e.g., content or time) to compute the relatedness between stories. After that, effective and efficient story context search algorithms are proposed to build a network of stories in social streams.

5.1 Introduction

There are many previous studies [48, 55, 66, 77] focusing on detecting new emerging stories from social streams. They serve the need for answering “*what’s happening now?*”. However, in reality, stories usually do not happen in isolation, and existing studies fail to track the relatedness between them. For example, “Crimea votes to join Russia” (on May 6, 2014) and “President Yanukovich signs compromise” (on February 21, 2014) are two separate stories, but they are actually highly related under the same event “Ukraine Crisis”. In this chapter, our goal is to design a *context-aware story-teller* for streaming social content, which not merely detects trending stories in a given time window of observation, but also builds the “context” of each story by measuring its relatedness with other stories on the fly. As a result, our story-teller has advantages on responding advanced user queries like “*tell me related stories*”, which is crucial to help digest social streams.

Building a context-aware story-teller over streaming social content is a highly challenging problem, as explained below:

- The effective organization of social content. It is well known that posts such as tweets are usually short, written informally with lots of grammatical errors, and even worse, a correctly written post may have no significance and be just noise.

- Identification of transient stories from time window. Story detection should be robust to noise and efficient to support the single-pass tracking essential in the streaming environment.
- Story context search on the fly. Story relatedness computation should be interpretable and efficient to support online queries.

To the best of our knowledge, no training data set for the context-aware story-telling on social streams is publicly available, which renders the existing works [59, 68] on Story Link Detection (SLD) not applicable, because SLD is trained on well-written news articles. Furthermore, we cannot apply topic tracking techniques [27, 32] to story context search, because topic tracking is usually formulated as a classification problem [4], with an assumption that topics are predefined before tracking, which is unrealistic for story context search on social streams.

To address above challenges, we propose **CAST** in this chapter, which is a Context-Aware Story-Teller specifically designed for social streams. **CAST** takes a noisy social stream as the input, and outputs a “story vein”, which is a human digestible and evolving summarization graph by linking highly related stories. The major workflow of **CAST** is illustrated in Figure 5.1. First of all, we treat each post as a node and add an edge between two posts if they are similar enough. For example, “Australian authorities update search for MH370” and “new MH370 search area is closer to Australia” are two similar posts and we add an edge between them. By this way, a post network will be constructed for social posts in the same observing time window. Second, we propose the new notion of (k, d) -Core to define a transient story, which is a cohesive subgraph in post network. In a (k, d) -Core, every node has at least k neighbors and two end nodes of every edge have at least d common neighbors. We propose two algorithms, **Zigzag** and **NodeFirst**, for the efficient discovery of maximal (k, d) -Cores from post network. After that, we define the *iceberg query* as finding highly related stories for a given story. Two approaches are proposed, deterministic context search (DCS) and randomized context search (RCS), to implement the iceberg query with high efficiency. The story vein is constructed based on the iceberg query and serves as the backend of context-aware story-teller on social streams, which discovers new stories and recommends related stories to users at each moment. Typical users of **CAST** are daily social stream consumers, who receive overwhelming (noisy) buzzes and wish to digest them by an intuitive and summarized representation in real time.

The main contributions of this chapter are summarized below:

- We define a new cohesive subgraph called (k, d) -Core to represent sto-

5.2. Story Vein

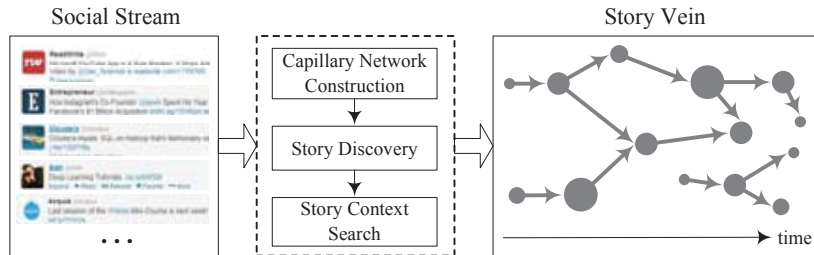


Figure 5.1: Illustrating the workflow of **StoryVein**, which has three major steps: (1) post network construction, (2) transient story discovery and (3) story context search.

Q	a social stream
$s(p_i, p_j)$	the similarity between posts p_i and p_j
$G(V, E)$	post network
$\mathbb{G}(V, E)$	story vein
$S(V_S, E_S)$	a story S
$Cor(S, S')$	the relatedness between story S and S'

Table 5.1: Major Notations.

ries and propose two efficient algorithms, **Zigzag** and **NodeFirst**, to identify maximal (k, d) -Cores from the post network (Section 5.3);

- We propose deterministic and randomized context search to support the iceberg query for highly related stories, which builds the story vein on the fly (Section 5.4);
- Our experimental study on real Twitter streams shows that **StoryVein** can digest and effectively build an expressive context-aware story-teller on streaming social content (Section 5.5).

We conclude this chapter in Section 5.6. Major notations are shown in Table 5.1.

5.2 Story Vein

In this chapter, we propose **CAST**, an effective context-aware story-teller for social streams. As social posts flow in, **CAST** is able to discover new stories and track the “vein” between stories, in which each story is a group of highly similar posts telling the same thing in the social stream, and each vein is a relatedness link between two stories. We define a story vein as follows,

where we use $Cor(S_i, S_j)$ to denote the relatedness between stories S_i and S_j .

Definition 16 (Story Vein) *Given a post network $G(V, E)$ and a threshold γ ($0 < \gamma < 1$), the output of **CAST** can be represented by a directed graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$, where each node $S \in \mathbb{V}$ is a story, and each edge $(S_i, S_j) \in \mathbb{E}$ means the story relatedness $Cor(S_i, S_j) \geq \gamma$ and S_i happens earlier than S_j .*

The motivation behind **CAST** is, real world stories are not isolated but commonly correlated with each other. Intuitively, the context of S in $\mathbb{G}(\mathbb{V}, \mathbb{E})$ is the neighboring stories of S . In particular, we use *upper* and *lower context* to represent the stories connected by incoming and outgoing edges of S , respectively. The target of **CAST** is to discover new stories and build the contexts of these stories. As the social stream Q gets updated over time, the story vein $\mathbb{G}(\mathbb{V}, \mathbb{E})$ will be also updated in real time. The dynamic nature of context-aware story-telling raises challenges both in the discovery of new stories and in the tracking of story context. In the following, we will address these challenges by discussing transient story discovery in Section 5.3 and story context tracking in Section 5.4.

5.3 Transient Story Discovery

In this section, we describe the motivation and algorithms for identifying transient stories as a new kind of cohesive subgraph, (k, d) -Core, from the post network.

5.3.1 Defining a Story

Edge weights in a post network have very natural semantics: the higher the post similarity, the more likely two posts are to talk about the same story. It is well-known that a cohesive subgraph is a sub-structure with high edge density and very low internal commute distance [75]. Suppose $S(V_S, E_S)$ is a cohesive subgraph in $G(V, E)$. Since nodes in $S(V_S, E_S)$ have a high density to connect with each other, it is very likely that all nodes in $S(V_S, E_S)$ share the same content and tell the same story. Based on this observation, we model a story in social streams as a cohesive subgraph in post network.

There are many alternative ways to define a cohesive subgraph. *Clique* may be the best candidate in spirit, because any two nodes have a sufficiently high similarity and thus share the same content. However, in real world data sets, cliques are too restrictive for story definition and this calls for some

relaxations. There are several choices for clique relaxations, e.g., quasi-clique, k -Core, k -Plex, etc [47]. Given a connected subgraph $S(V_S, E_S)$, supposing $N(p)$ is the neighbor set of node $p \in V_S$ in $S(V_S, E_S)$, these clique relaxations are defined as:

- Quasi-clique: $|N(p)| \geq \lambda(|V_S| - 1)$ for every post p with $0 < \lambda \leq 1$;
- k -Plex: $|N(p)| \geq |V_S| - k$ for every post $p \in V_S$;
- k -Core: $|N(p)| \geq k$ for every post $p \in V_S$.

Notice that finding the maximum clique, quasi-clique or k -Plex from a graph is an NP-Hard problem [8, 15]. Even worse, [31] proved that there are no polynomial algorithms that provide any reasonable approximation to the maximum clique problem. Since a clique is a special case of quasi-clique or k -Plex, these hardness results carry over to maximum quasi-clique and maximum k -Plex problems. There are a lot of heuristic algorithms that provide no theoretical guarantee on the quality [1, 8, 15]. Since most of them are based on local search [15], they do not scale to large networks, because local search involves the optimization of solutions by iteratively moving to a better neighbor solution in an exponential search space.

The good news is that k -Cores can be exactly found in polynomial time. By adjusting k , we can generate k -Cores with desired edge density $2|E_S|/|V_S|$. For example, increasing k will improve edge density because the minimal edge degree is increased and $|V_S|$ is decreased in the same time. However, k -Cores are not always capable of capturing our intuition on stories: although each post has at least k neighbors, the fact that two posts are connected by a single edge may be not a strong enough evidence to prove that they tell the same story. Sometimes, posts only share some common words but discuss different stories, e.g., “Google acquires Motorola Mobility” and “Bell Mobility acquires Virgin Mobile”. We show an example of 3-Core in Figure 5.2(a), where p_1 and p_2 are connected but they belong to two separate cliques. To address this challenge, we make a key observation that *the existence of more common neighbors between two edge-connected posts suggests a stronger commonality in story-telling*. Supposing p_i and p_j are connected by an edge and there exists post $p_l \in N(p_i) \cap N(p_j)$, then we call p_l a witness for the post similarity $s(p_i, p_j)$. We capture this intuition in the following definition, where we formalize a story as a maximal (k, d) -Core.

Definition 17 (Story in CAST) *A story in social stream Q is defined by a maximal (k, d) -Core $S(V_S, E_S)$ in post network $G(V, E)$ associated with Q , where k, d are numbers with $k > d > 0$ and*

- $S(V_S, E_S)$ is a connected subgraph;
- For every post $p \in V_S$, $|N(p)| \geq k$;

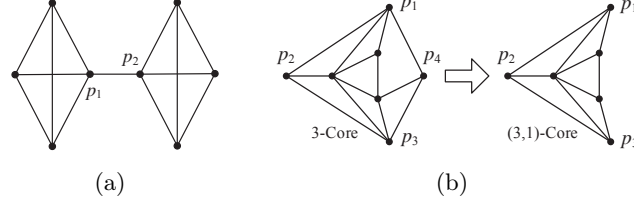


Figure 5.2: (a) A 3-Core without similarity witness between p_1 and p_2 . (b) The illustration of generating a (3,1)-Core from a 3-Core.

- For every edge $(p_i, p_j) \in E_S$, $|N(p_i) \cap N(p_j)| \geq d$.

Why (k, d) -Core? In (k, d) -Cores, we use k to adjust the edge density, and use d to control the strength of similarity witness. It is easy to see that a maximal (k, d) -Core is a subgraph of a maximal k -Core. However, compared with k -Cores, (k, d) -Cores have a more cohesive internal structure enhanced by at least d common neighbors as witnesses of commonality between two nodes connected by an edge. This enhancement makes posts in (k, d) -Core more likely to tell the same story. We show an example of 3-Core and (3,1)-Core in Figure 5.2(b). As we can see, p_4 is in 3-Core but not in (3,1)-Core, because the similarity between p_1 and p_4 is not witnessed by other posts. Besides, [65] defines a kind of cohesive subgraph called k -Dense, in which every edge has at least k witnesses. It is easy to infer that a k -Dense is a $(k+1, k)$ -Core. Thus, k -Dense is a special case of (k, d) -Core, but provides no flexibility to adjust k and d independently. Therefore, our proposed (k, d) -Core is better than k -Core and k -Dense in capturing a story.

Non-overlap Property. Similar to maximal k -Cores, a maximal (k, d) -Core cannot be a subgraph of any other (k, d) -Cores. Maximal (k, d) -Cores have the following important property.

Proposition 6 *The maximal (k, d) -cores of a graph are pairwise disjoint.*

Proof: Suppose $S_i(V_i, E_i)$ and $S_j(V_j, E_j)$ are two different maximal (k, d) -Cores and $V_i \cap V_j \neq \emptyset$. Now we can construct a connected subgraph $S(V_i \cup V_j, E_i \cup E_j)$. Since each node in S has at least degree k and each edge in S has at least d similarity witnesses, S exactly satisfies the definition of a (k, d) -Core. This conclusion contradicts the assumption that S_i and S_j are maximal. \square

5.3.2 Story Formation

We now discuss the computation of (k, d) -Cores. We first review the k -Core generation process [47]: given a network $G(V, E)$, iteratively remove the nodes with degree less than k from $G(V, E)$, until all the remaining nodes have a degree at least k . The result will be a set of maximal k -Cores, which are obtained in polynomial time. The k -Core generation process forms the basis of our (k, d) -Core generation algorithms. We propose the first solution for (k, d) -Core generation in Algorithm 4. We call it a **Zigzag** algorithm, because the basic idea is to repeatedly change the property of the current network between two states: the first state is k -Core set G_1 obtained by removing nodes recursively, and the second state is $(d + 1, d)$ -Core set G_2 obtained by removing edges recursively. This Zigzag process will terminate if each connected component in the result set is a (k, d) -Core, or the result set is empty. It is easy to see Algorithm 4 takes polynomial time and the result is exact.

We notice that the transition costs between two states are not symmetric: the computational costs of G_1 and G_2 are very different. If we can reduce the frequency of the transition with a higher overhead, the overall performance of **Zigzag** can be optimized. The following lemma formalizes the property.

Proposition 7 *Given a network $G(V, E)$ with $|V| < |E|$ and integers k, d ($k > d$), for each iteration in **Zigzag**, the computation of k -Core set G_1 is more efficient than the computation of $(d + 1, d)$ -Core set G_2 .*

Proof: To compute G_1 , we need to check the degree of every node. To compute G_2 , we need to check the common neighbor size of the two end nodes for every edge. Given that $|V| < |E|$ in most networks and the checking of node degree is more efficient than the checking of common neighbors, we complete the proof. \square

Proposition 7 suggests that if we reduce the computation frequency of the $(d + 1, d)$ -Core set G_2 , the performance will be improved. Following this, we propose Algorithm 5 to improve the performance of Algorithm 4, by applying node deletions as much as possible. The heuristic is, whenever an edge is deleted, we check whether this deletion makes the degree of its end nodes smaller than k , and if it does, a recursive node deletion process starting from end nodes will be performed (Line 7). We call Algorithm 5 **NodeFirst** because it greedily invokes the node deletion process when possible. Since **NodeFirst** avoids to perform a complete edge deletion process as **Zigzag** did, the network converges very fast to the set of maximal (k, d) -Cores. The following propositions indicate that **NodeFirst** produces exactly the same results as **Zigzag**, but its performance is better than **Zigzag**.

Algorithm 4: (k, d) -Core Generation: **Zigzag**

Input: $G(V, E), k, d$
Output: All maximal (k, d) -Cores

- 1 Generate a set of k -Cores G_1 by removing nodes with degree less than k recursively from $G(V, E)$;
- 2 **while** G_1 is not empty **do**
- 3 Generate a set of $(d + 1, d)$ -Cores G_2 by removing edges with witnesses less than d recursively from G_1 ;
- 4 **if** G_1 equals G_2 **then**
- 5 | break while loop;
- 6 Generate a set of k -Cores G_1 by removing nodes with degree less than k recursively from G_2 ;
- 7 **if** G_1 equals G_2 **then**
- 8 | break while loop;
- 9 return G_1 ;

Proposition 8 Given a network $G(V, E)$ and numbers k, d ($k > d$), both **Zigzag** and **NodeFirst** generate the same result, which is the set of all maximal (k, d) -Cores in $G(V, E)$.

Proof: Suppose $S_{(k,d)}$ is the set of all maximal (k, d) -Cores in $G(V, E)$. Both **Zigzag** and **NodeFirst** try to remove nodes or edges in each iteration, so the sizes of nodes and edges follow a monotone decreasing trend. With this message, we show the proof by following algorithm logics: **Zigzag** converges to $S_{(k,d)}$ because $S_{(k,d)}$ is the first time when a k -Core set and a $(d + 1, d)$ -Core set are equal (Line 7 in Algorithm 4); **NodeFirst** converges to $S_{(k,d)}$ because it is the first time when no edges with witness less than d can be found in a k -Core set (Line 8 in Algorithm 5). \square

Proposition 9 On each iteration, **NodeFirst** is more efficient than **Zigzag**.

Proof: We measure the efficiency by the number of node degree check or edge similarity witness check operations. **Zigzag** takes $|V| + |E|$ operations, while **NodeFirst** takes $|V| + 1$ operations. Thus, **NodeFirst** is more efficient than **Zigzag**. \square

5.4 Story Context Tracking

Transient stories we identified from the post network may be highly correlated, e.g., “the launch of Blackberry 10” and “BlackBerry Super Bowl

Algorithm 5: (k, d) -Core Generation: **NodeFirst**

Input: $G(V, E), k, d$
Output: All maximal (k, d) -Cores

- 1 Generate a set of k -Cores G' by removing nodes with degree less than k recursively from $G(V, E)$;
- 2 **while** G' is not empty **do**
- 3 Find an edge $e(p_i, p_j)$ with less than d witnesses;
- 4 **if** $e(p_i, p_j)$ exists **then**
- 5 delete (p_i, p_j) from G' ;
- 6 **if** $Deg(p_i) < k$ or $Deg(p_j) < k$ **then**
- 7 Remove nodes with degree less than k recursively from G' ;
- 8 **else**
- 9 break while loop;
- 10 return G' ;

ad". In this section, we exploit and integrate signals from different measures in story relatedness computation, and propose deterministic context search (DCS) to construct the veins for a story S on the fly. The performance of DCS is proportional to the size of S 's neighboring posts. In the case when the neighboring post size is huge, we propose randomized context search (RCS) to boost the performance.

We remark that DCS and RCS proposed in this section are two general computation frameworks, which can be applied to different definitions of stories. As long as a story is a connected subgraph in a post network, DCS and RCS will apply to find the context between stories. That is to say, stories defined as λ -quasi-cliques and (k, d) -Cores can work seamlessly as two instantiations of story concept in the same context tracking framework. For simplicity, we take (k, d) -Core as the structure to instantiate story and explain its context search in this section.

5.4.1 Story Relatedness Dimensions

Recall that stories correspond to (k, d) -cores in the post network and let $S_i(V_i, E_i)$ and $S_j(V_j, E_j)$ denote two stories. Here we introduce different types of relatedness dimensions, which capture the story relatedness from different perspectives, and quantify the relatedness by a value in $[0, 1]$. Notice that node overlap is a very common evidence to assess the relatedness between two subgraphs [69]. However, since Proposition 6 shows that (k, d) -Cores generated by **Zigzag** or **NodeFirst** are pairwise disjoint in nodes,

node overlap is not useful in story relatedness computation.

Dimension 1: Content Similarity. By viewing a story as a document and a post entity as a term, existing document similarity measures can be used to assess the story relatedness. However, TF-IDF based Cosine similarity fail to be effective, since TF vectors of stories tend to be very sparse. We exploit another popular measure, LDA-based symmetric KL-divergence [59]. Supposing d is the document representation of story S and $\theta(d)$ is the topic distribution of d produced by LDA, we have

$$C_1(S_i, S_j) = \frac{KL(\theta(d_i) \parallel \theta(d_j)) + KL(\theta(d_j) \parallel \theta(d_i))}{2} \quad (5.1)$$

where $KL(\theta(d_i) \parallel \theta(d_j)) = \sum_x \theta_x(d_i) \log \frac{\theta_x(d_i)}{\theta_x(d_j)}$.

Dimension 2: Temporal Proximity. Stories that happen closer in time are more likely to correlate together. Given a story $S_i(V_i, E_i)$, we can get the histogram of post volume along time dimension on each bin with size equal to the time window sliding step Δt . Supposing $Dist$ is the normalized distribution (with the sum equal to 1) of story S 's histogram, we take the complement of L_1 Norm as an example to measure the temporal proximity between S_i and S_j :

$$C_2(S_i, S_j) = 1 - \sum_{t=1}^{len} |Dist_i(t) - Dist_j(t)| \quad (5.2)$$

where len is the length of the observation time window.

Dimension 3: Edge Connectivity. Capillaries associated with posts of two stories can be used to determine the story relatedness. This approach calculates the strength of edge connectivity between posts of S_i and S_j in post network. These edges serve as the bridge between two stories, and the connectivity strength can be measured by various ways, e.g., the Jaccard Coefficient based on the portion of bridge edges:

$$C_3(S_i, S_j) = \frac{|E(V_i, V_j)|}{|E(V_i, V) \cup E(V_j, V)|} \quad (5.3)$$

where $E(A, B)$ is the edge set between node set A and B .

Baseline: Hybrid Relatedness Model. As a baseline approach, story relatedness can be computed by a hybrid model on all relatedness dimensions, with the form:

$$Cor_H(S_i, S_j) = \prod_{k=1}^3 C_k(S_i, S_j) \quad (5.4)$$

It is easy to see that $0 \leq Cor_H(S_i, S_j) \leq 1$. One drawback of the hybrid relatedness model is the performance. First, LDA computation for large corpus is expensive [70]. Second, to construct the context of story S in the story vein, the hybrid relatedness model needs to compute $Cor_H(S, S')$ between S and every other story S' in $\mathbb{G}(\mathbb{V}, \mathbb{E})$, which is redundant and time-consuming.

5.4.2 Story Context Search

Story context search aims at finding the neighbors of a story in the story vein $\mathbb{G}(\mathbb{V}, \mathbb{E})$ efficiently. However, simply applying the hybrid relatedness model shown in Eq.(5.4) results in an inefficient pairwise computation. To overcome the challenge, we introduce iceberg query on story vein, as stated below.

Definition 18 (Iceberg Query) *Given a story vein $\mathbb{G}(\mathbb{V}, \mathbb{E})$, a threshold γ ($0 < \gamma < 1$) and a story S ($S \notin \mathbb{V}$), the iceberg query for S is to find the subset of all stories $\mathbb{V}_S \subseteq \mathbb{V}$, where for each $S' \in \mathbb{V}_S$, $Cor(S, S') \geq \gamma$.*

Iceberg query is the key technique to support **CAST**. Iceberg query grows the story vein on the fly: for each new story S , it can quickly build possible relatedness links between S and the remaining stories. Iceberg query does not need to explore all stories in \mathbb{V} to generate the exact \mathbb{V}_S , which improves the performance.

In this section, we propose two approaches to implement iceberg query on story vein. The first approach is deterministic, which integrates content similarity, temporal proximity and edge connectivity together by aggregating all capillaries between two stories. The second approach is randomized, which improves the performance of the deterministic approach by filtering the flow from S to S' . They are discussed separately below.

Deterministic Context Search. The basic idea of deterministic context search (DCS) follows a propagation and aggregation process. To initialize, we treat story $S(V_S, E_S)$ as *source* and every other story $S'(V_{S'}, E_{S'})$ as *target*. In the propagation phase, we start from each post in *source* and broadcast the post similarity to neighboring posts along capillaries. In the aggregation phase, we aggregate the post similarity received by posts in each *target*, and denote it by $PA_D(S, S')$. In detail, we have

$$PA_D(S, S') = \sum_{p \in V_S, p' \in V_{S'}, (p, p') \in E} s(p, p') \quad (5.5)$$

Since $s(p, p')$ is computed by combining content similarity and temporal proximity (see Eq.(3.1)), $PA_D(S, S')$ naturally integrates the above discussed three story relatedness dimensions. The relatedness between S and S' is assessed by the ratio between the post similarity aggregated by S' and propagated by S :

$$Cor_D(S, S') = \frac{PA_D(S, S')}{PA_D(S, G)} \quad (5.6)$$

where $PA_D(S, G)$ is the post similarity sum of all capillaries associated with posts in S . When $Cor_D(S, S') \geq \gamma$, there will be a story link between S and S' in story vein $\mathbb{G}(\mathbb{V}, \mathbb{E})$. Let S^τ denote the average time stamp of S , i.e., $S^\tau = \frac{1}{|V_S|} \sum_{p \in V_S} p^\tau$. If $S^\tau < S'^\tau$, the edge direction is from S to S' . Otherwise, the edge direction is from S' to S .

The sketch of DCS is shown in Algorithm 6. Notice that post similarity is not aggregated on every story. Some stories may receive zero or little post similarity, and can be omitted. On average, a source story visits at most $\min\{|V|, \frac{|S_V| \cdot |E|}{|V|}\}$ neighboring posts in DCS. Compared with the hybrid relatedness model, which requires to access the whole post network $G(V, E)$ in pairwise computation, DCS improves the performance significantly.

The propagation and aggregation process is also extensively discussed in the structural similarity search problem [45]. In [45], the similarity between two nodes are modeled as the authority score of corresponding node pair, and the propagation and aggregation process on node-pairs is used to compute the similarity score accurately.

Randomized Context Search. To improve the performance of the deterministic context search further, we propose *randomized context search* (RCS) based on random walk. The motivation behind RCS is, in the propagation phase, it is preferable to only visit the neighboring posts with a high post similarity. Given a post p in story S , suppose $\max(p)$ and $\min(p)$ are the maximum and minimum post similarity associated with p , respectively. Technically, we use α ($0 \leq \alpha \leq 1$) as a throttle to propagate high post similarities: only if $s(p, p') \geq \min(p) + \alpha(\max(p) - \min(p))$, RCS propagates $s(p, p')$ to p' by a uniformly random probability.

There are two special cases in RCS:

- $\alpha = 0$: Post p propagates to every neighboring post.
- $\alpha = 1$: Post p only propagates to the neighboring post with the highest post similarity.

We empirically choose $\alpha = 0.5$. Suppose N is the total number of simulations we run for the source story S . On each simulation, a ran-

5.4. Story Context Tracking

Algorithm 6: Deterministic Context Search (DCS)

Input: $G(V, E)$, $\mathbb{G}(\mathbb{V}, \mathbb{E})$, new story $S(V_S, E_S)$, γ
Output: In-neighbor set $\mathbb{N}_I(S)$, out-neighbor set $\mathbb{N}_O(S)$

- 1 $PA_D(S, G) = 0$, $N(S) = \emptyset$, $\mathbb{N}_I(S) = \emptyset$, $\mathbb{N}_O(S) = \emptyset$;
- 2 $S^\tau = \frac{1}{|V_S|} \sum_{p \in V_S} p^\tau$;
- 3 **for** each $p \in V_S$ **do**
- 4 **for** each neighbor p' of p in $G(V, E)$ **do**
- 5 **if** p' is in a story S' **then**
- 6 **if** $S' \notin N(S)$ **then**
- 7 add S' into $N(S)$;
- 8 add $s(p, p')$ into $PA_D(S, G)$ and $PA_D(S, S')$;
- 9 **for** each $S' \in N(S)$ **do**
- 10 $Cor_D(S, S') = \frac{PA_D(S, S')}{PA_D(S, G)}$;
- 11 **if** $Cor_D(S, S') \geq \gamma$ **then**
- 12 $S'^\tau = \frac{1}{|V_{S'}|} \sum_{p' \in V_{S'}} p'^\tau$;
- 13 **if** $S^\tau < S'^\tau$ **then** add S' into $\mathbb{N}_O(S)$;
- 14 **else** add S' into $\mathbb{N}_I(S)$;
- 15 **return** $\mathbb{N}_I(S)$ and $\mathbb{N}_O(S)$;

dom surfer starts from a post p in S and randomly visits a neighbor p' if $s(p, p') \geq \min(p) + \alpha(\max(p) - \min(p))$. The aggregated post similarity from source S to target S' on simulation i can be described as

$$R_i(S, S') = \begin{cases} s(p, p') & \text{if } p \in V_S, p' \in V_{S'} \\ 0 & \text{otherwise} \end{cases} \quad (5.7)$$

The aggregated post similarity by S' on N simulations is denoted by $PA_R(S, S')$:

$$PA_R(S, S') = \sum_{i=1}^N R_i(S, S') \quad (5.8)$$

We show the sketch of RCS in Algorithm 7. Similar to DCS, the relatedness between S and S' is computed by $Cor_R(S, S') = \frac{PA_R(S, S')}{PA_R(S, G)}$. The vein threshold and direction is determined by the same way of DCS. Clearly, in RCS, a source story visits at most N neighboring posts and usually $N \ll |V|$. The value of N is decided by the time budget of story context search in real **CAST** system. Compared with DCS, RCS achieves better performance by accessing smaller number of neighboring posts connected by capillaries with higher post similarity.

Algorithm 7: Randomized Context Search (RCS)

Input: $G(V, E)$, $\mathbb{G}(\mathbb{V}, \mathbb{E})$, new story $S(V_S, E_S)$, γ , α , n
Output: In-neighbor set $\mathbb{N}_I(S)$, out-neighbor set $\mathbb{N}_O(S)$

- 1 $PA_R(S, G) = 0$, $N(S) = \emptyset$, $\mathbb{N}_I(S) = \emptyset$, $\mathbb{N}_O(S) = \emptyset$, $count = 0$;
- 2 $S^\tau = \frac{1}{|V_S|} \sum_{p \in V_S} p^\tau$;
- 3 **while** $count < n$ **do**
- 4 randomly select a node $p \in V_S$;
- 5 randomly select a neighbor p' of p in $G(V, E)$;
- 6 **while** $s(p, p') < \min(p) + \alpha(\max(p) - \min(p))$ **do**
- 7 randomly select a neighbor p' of p in $G(V, E)$;
- 8 **if** p' is in a story S' **then**
- 9 **if** $S' \notin N(S)$ **then**
- 10 add S' into $N(S)$;
- 11 add $s(p, p')$ into $PA_R(S, G)$ and $PA_R(S, S')$;
- 12 **for** each $S' \in N(S)$ **do**
- 13 $Cor_R(S, S') = \frac{PA_R(S, S')}{PA_R(S, G)}$;
- 14 **if** $Cor_R(S, S') \geq \gamma$ **then**
- 15 $S'^\tau = \frac{1}{|V_{S'}|} \sum_{p' \in V_{S'}} p'^\tau$;
- 16 **if** $S^\tau < S'^\tau$ **then** add S' into $\mathbb{N}_O(S)$;
- 17 **else** add S' into $\mathbb{N}_I(S)$;
- 18 **return** $\mathbb{N}_I(S)$ and $\mathbb{N}_O(S)$;

5.4.3 Interpretation of Story Vein

Since story vein is described in graph notation, it becomes very important to explain the story vein to end users who may be non-technical. There are various options to present a (k, d) -Core in human consumable form. The first option is annotating a (k, d) -Core by the most representative posts inside, e.g., posts with the highest edge degree. The second option is rendering a (k, d) -Core into a word cloud. Since we extracted entities from each post, the frequency of an entity in a (k, d) -Core can be used to indicate the font size of that entity in the word cloud. In this paper, we provide both options to aid human perception. We will show some interpretation examples in experimental study.

5.5 Experimental Study

All experiments are conducted on a computer with Intel 2.66 GHz CPU, 8 GB RAM, running 64-bit Windows 7. All algorithms are implemented in Java. We empirically set the threshold $\epsilon = \gamma = 0.2$ for the construction of post network and story vein. We set $\alpha = 0.5$ to filter out capillaries with low post similarity in RCS.

All data sets are crawled from Twitter.com via Twitter4J⁸ API, with a time span from Jan. 1 to Feb. 1, 2012. Although our story teller **CAST** works regardless of the domain, we make the data set domain-specific in order to facilitate evaluation of the generated results.

CNN-News. By simulating a social news stream, we collect tweets created in the first half year of 2014 from CNN channels, which include @cnn, @cnbrk, @cnnmoney, @cnlive and @cnni. This data set has 10872 tweets and serves for the quality analysis.

Tech-Lite. We built a technology domain dataset called Tech-Lite by aggregating timelines of users listed in Technology category of “Who to follow”⁹ and their retweeted users. Tech-Lite has 352,328 tweets, 1402 users and the streaming rate is 11700 tweets/day.

Tech-Full. Based on the intuition that users followed by users in Technology category are most likely also in the technology domain, we obtained a larger technology social stream called Tech-Full by collecting all the timelines of users that are followed by users in Technology category. Tech-Full has 5,196,086 tweets, created by 224,242 users, whose streaming rate is about 7216 tweets/hour.

5.5.1 Tuning Post Network

Post similarity computation shown in Eq. (3.1) influences the structure of post network directly. We can tune the content similarity function $s_T(p_i^T, p_j^T)$ and temporal proximity function $s_\tau(|p_i^\tau - p_j^\tau|)$ to make the post network concise and expressive. Many set-based similarity measures such as Jaccard coefficient [53] can be used to compute the similarity $s_T(p_i^T, p_j^T)$ between posts. Since entity usually appears once in one tweet, similarity measures such as Cosine similarity and Pearson correlation [53] will degenerate to a form very similar to Jaccard, so we use Jaccard as our similarity function and omit further discussion of alternatives.

⁸Twitter4J. <http://twitter4j.org/>

⁹http://twitter.com/who_to_follow/interests

5.5. Experimental Study

Methods	Capillaries	Stories	Vein Links
No-Fading	1159364	373	495
Exp-Fading	327390	87	275
Reci-Fading	357132	123	384

Table 5.2: The number of edges in the post network, and the number of stories and relatedness links in the story vein as the changing of temporal proximity functions. We define a story as a (5, 3)-Core.

Temporal proximity function $s_\tau(|p_i^\tau - p_j^\tau|)$ determines how similarity to older posts is penalized, compared to recent posts. We compared three different functions: (1) reciprocal fading (“Reci-Fading”) with $D(|p_i^\tau - p_j^\tau|) = \frac{1}{|p_i^\tau - p_j^\tau| + 1}$, (2) exponential fading (“Exp-Fading”) with $D(|p_i^\tau - p_j^\tau|) = e^{-|p_i^\tau - p_j^\tau|}$ and (3) no fading (“No-Fading”) with $D(|p_i^\tau - p_j^\tau|) = 1$. For any posts p_i, p_j , clearly $e^{|p_i^\tau - p_j^\tau|} \geq |p_i^\tau - p_j^\tau| + 1 \geq 1$. Exp-Fading penalizes the posts in the old part of time window severely (see Table 5.2): the number of capillaries and stories generated by Exp-Fading is lower than by other approaches. Since No-Fading does not penalize old posts in the time window, too many capillaries and stories will be generated without considering recency. Reci-Fading is in between, which is a more gradual penalty function than Exp-Fading. In the rest of experiments, we use Exp-Fading, since it generates the most concise post network with an emphasis on recent posts.

5.5.2 Quality Evaluation

In this section, we evaluate the quality of **CAST** on two tasks: story discovery and context search, by comparing with baselines.

Ground Truth. We build the ground truth of tech stories in Jan 2012 by browsing news articles on main stream technology websites like CNET.com, TechCrunch.com, and ReadWrite.com, etc. We pick headlines with high occurrence and build a ground truth of top 10 stories. They include CES related stories, SOPA & PIPA related stories, Facebook IPO, Yahoo co-founder Jerry Yang resignation, RIM CEO changes, etc.

Baseline 1: Peak Detection. Some recent work on event detection [48, 55, 67] can be used for story discovery, and they share the same spirit that aggregates the frequency of topic-indicating phrases at each moment to build a histogram and generates stories by detecting volume peaks in the histogram. We design three baselines to capture the major techniques used

5.5. Experimental Study

Rank	HashtagPeaks	MentionPeaks	EntityPeaks
1	#CES	@CNETNews	google
2	#SOPA	@thatdrew	ces
3	#EngadgetCES	@m4tt	apple
4	#opengov	@TNWapple	video
5	#gov20	@TNWinsider	sopa
6	#CES2012	@TNWapps	twitter
7	#PIPA	@TGW	year
8	#opendata	@jonrussell	facebook
9	#StartupAmerica	@CNET	app
10	#win7tech	@harrisonweber	iphone
Precision	0.5	0.2	0.7
Recall	0.4	0.2	0.5

Figure 5.3: Top 10 results of HashtagPeaks, MentionPeaks and EntityPeaks on Tech-Lite dataset. The ground truth for precision and recall is top 10 major stories selected from main stream technology news websites.

by these approaches.

- *HashtagPeaks*: aggregate frequent hashtags;
- *MentionPeaks*: aggregate frequent mentions.
- *EntityPeaks*: aggregate frequent entities.

We show top 10 results of HashtagPeaks, MentionPeaks and EntityPeaks in Tech-Lite dataset with time stamps between Jan 1 and Feb 1, 2012 in Figure 5.3. Their precision and recall are also listed by comparing with the ground truth. Notice that multiple peaks may correspond to the same story in ground truth, where the precision and recall differ. As we can see, MentionPeaks is the worst because most of these mentions are not topic-indicating phrases. Hashtags are the “twitter” way to indicate an event, but it requires manual assignation by human. EntityPeaks is the best out of three baselines, since entities are extracted from the social stream preprocessing stage to annotate stories. Although these baselines are able to indicate some words about a story, they’re not qualified for a successful story-teller because the internal and external structure of the story is missing. In particular, they cannot show how users interact with each other, how posts are clustered, and how the story is distributed along time dimension.

Baseline 2: Topic Modeling. Topic modeling is a typical way to detect topics from text document corpus. As we mentioned earlier, existing work on topic detection and tracking falls into a classification problem, and prevalent topic modeling models like Latent Dirichlet Allocation (LDA) [13] are mainly trained on formal-writing news articles. We design a baseline using LDA, and treat top 100 posts of each topic as a story. Post time stamps are totally

5.5. Experimental Study

Rank	LDA-Detected Story Description
1	blog post, great buy in january, start sign
2	report amazon kindle fire
3	check real youtube interview video series
4	win chance south beach trip
5	small business company story, local community
6	make great start, things learn
7	bad thing in life, feel good
8	send email hey, glad to hear, follow tweet
9	jan travel, days ago, back home tonight
10	president obama, iran war killed
11	daily top stories, tech news, health care
12	apple tv iphone ipad, ces event, app store
13	watch night show, tonight fun, miss baby
14	twitter mobile app, android apps, tweet free
15	Iowa caucus results, Santorum and Romney
16	play half game, nyc new york city, winter fans
17	super bowl, man football party
18	happy new year, hope wonderful year, friends family
19	love word, perfect friend, people
20	google internet search ad, https link

Table 5.3: Top 20 stories detected by LDA from Tech-Full and described by high frequency words. We treat top 100 posts of each topic as a story.

ignored in LDA. We set the topic number as 50, and after rendering topics to stories, we show the top 20 stories in Table 5.3. As we can see, the topic cohesion of LDA-detected stories is not very high: posts sharing some common words are very easily classified into the same topic, even though they are not exactly talking about the same story. Besides, LDA model cannot deal with noisy posts, so the quality is compromised in social streams.

CAST on Tech-Lite. Recall that our proposed story-teller **CAST** uses a (k, d) -Core to represent a story, which is a cohesive subgraph in the post network. Figure 5.4 shows top 10 transient stories generated by **CAST**. Each story is represented as a $(5,3)$ -Core. It is worth noting that there are various ways to present stories on the user interface. In this experiment, we render a story into an entity cloud. Each cloud is annotated by a short description beside. Interestingly, we observe the curve of tweet frequency goes down on every weekend, which reflects the living habit in reality. Compared with the ground truth, our story-teller achieves a precision of 0.9 and a recall of 0.8.

5.5. Experimental Study

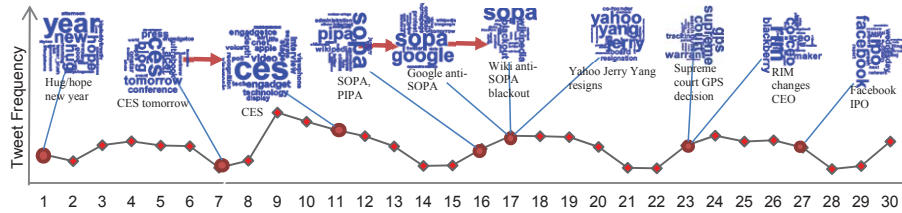


Figure 5.4: Top 10 transient stories generated by our proposed story-teller on Tech-Lite. Each story is represented as a (5,3)-Core, and we render a story into an entity cloud for human reading. Some transient stories may be related, as linked by lines. The curve on the bottom is the breakdown of tweet frequency on each day in Jan 2012.

For precision, the only case we fail is that “hug/hope new year” is not a story in ground truth. The reason may be lots of people tweeting about “happy new year” but this is not formally reported in technology news websites, since it is a well-known fact. For recall, the ground truth story not appearing in our top 10 is “Apple iBooks 2 Release” on Jan 19. This story is ranked on the 13th position in our story-teller.

We can see that stories may be related with each other. In Figure 5.4, there are multiple stories about CES (Consumer Electronic Show) 2012, held from Jan 8 to Jan 13. Meanwhile, stories related to SOPA & PIPA are highly related, even though each of them tells a relatively different story.

When inspecting the top 50 transient stories returned by **CAST**, we observe two major advantages compared with news reports. First, our story-teller can identify stories that are hardly noticeable in news websites. For example, “Rupert Murdoch joins Twitter” is a story widely spread in social streams, but not commonly reported by news articles. Second, the formation of a transient story in social streams generally occurs earlier than the first publishing of the corresponding news articles. [48] also observed this phenomenon, and the typical time lag is around 2.5 hours.

CAST on Tech-Full. Figure 6.1 shows an example to illustrate the story vein on Tech-Full. To help understand, we group stories densely connected by vein links in rectangles, where each rectangle can be viewed as a story series. As we can see, “CES” and “SOPA & PIPA” are two very different story series, but they can be connected by indirect links via posts such as “CES 2012: Microsoft Keynote With Steve Ballmer” and “Microsoft opposes SOPA”. **CAST** will track the relatedness between stories and build a highly informative story vein for story-teller.

5.5. Experimental Study

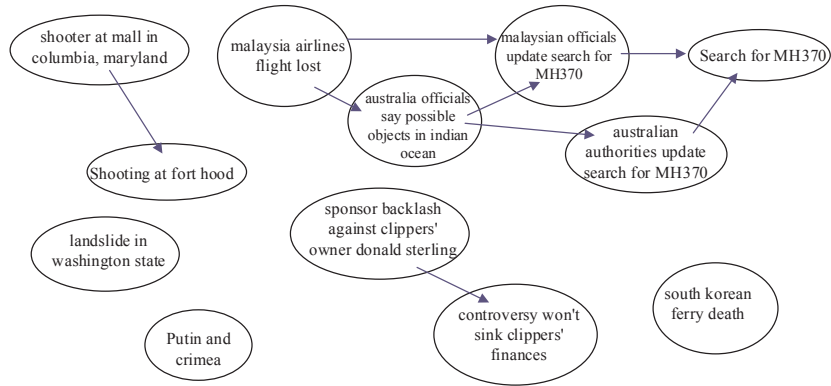


Figure 5.5: A fragment of story vein tracked from CNN-News, which has a time span from January 1 to June 1, 2014.

CAST on CNN-News. CNN-News simulates the daily social stream received by a Twitter user and has a time span of six months. We show a fragment of the story vein tracked from CNN-News in Figure 5.5. In this example, “MH370 lost and search” is a story series with an internal structure, by organizing individual stories together. Users can click on each story to see the details, or traverse along story veins to read highly related stories happening earlier or later. Compared with Twitter Trends Search¹⁰ which shows trending topics, **CAST** has distinct advantages on providing both the internal structure and the external context of a specific story.

5.5.3 Performance Testing

In this section, we test the performance of proposed algorithms. All tests are performed on Tech-Full dataset with the time window set to one week. On average, the post network has a node size 710,000 and edge size 4,020,000.

Story Identification. Recall that both k -Core and (k, d) -Core generations have polynomial time complexity. We test the running time of generating all maximal k -Cores and (k, d) -Cores from the post network and show the result in Figure 5.7(a). Let $k = d + 1$. As d increases, the running time of k -Core generation drops. The reason is, although a bigger k means more nodes need to be removed at each iteration, the total number of iterations drops even more quickly. The running time for (k, d) -Cores is nearly stable, but much higher than k -Core generation. This observation implies that

¹⁰<https://twitter.com/search-home>

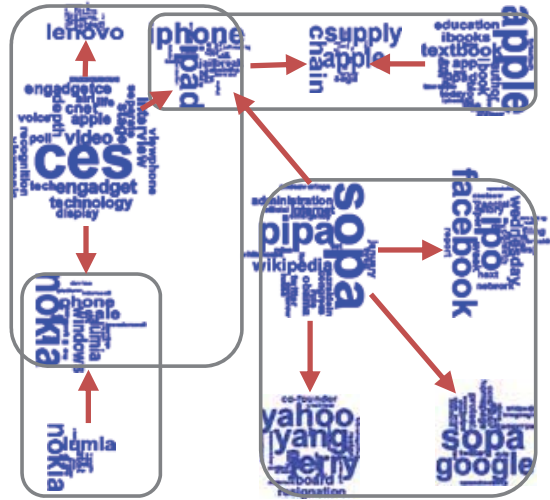


Figure 5.6: An example to illustrate our context-aware story-teller. Each tag cloud here is a single story identified from the post network. Sets of stories with higher relatedness are grouped together in rectangles to aid readability.

NodeFirst should be much faster than **Zigzag**, since **NodeFirst** performs k -Core generation whenever possible. This conclusion is supported by the experimental results in Figure 5.7(a). The relationship between d and the number of connected components we can find from the post network is discovered in Figure 5.7(b). We see that with increasing d , both the numbers of $(d+1)$ -Cores and $(d+1, d)$ -Cores drop. Since a $(d+1, d)$ -Core is at least a $(d+1)$ -Core but more cohesive, we may find multiple $(d+1, d)$ -Cores from a $(d+1)$ -Core by removing edges that do not satisfy the (k, d) -Core definition. Thus, given the same post network, the number of $(d+1, d)$ -Cores is usually higher than the number of $(d+1)$ -Cores.

Iceberg Queries. In **CAST**, we use iceberg query to construct possible vein links between a given story and all other stories in the post network. As discussed in Section 5.4, we treat the given story as source and all other stories as targets, and the story relatedness threshold γ will govern the construction of story vein links.

In Figure 5.7(c), we treat each story in the post network as the query and show the total running time of iceberg queries for all stories. As d increases, we observe that the number of $(d+1, d)$ -Cores decrease gradually from Figure 5.7(b). This will make the total size of transient stories smaller and naturally the total running time of iceberg queries decrease. In experiments, we can

5.6. Discussion and Conclusion

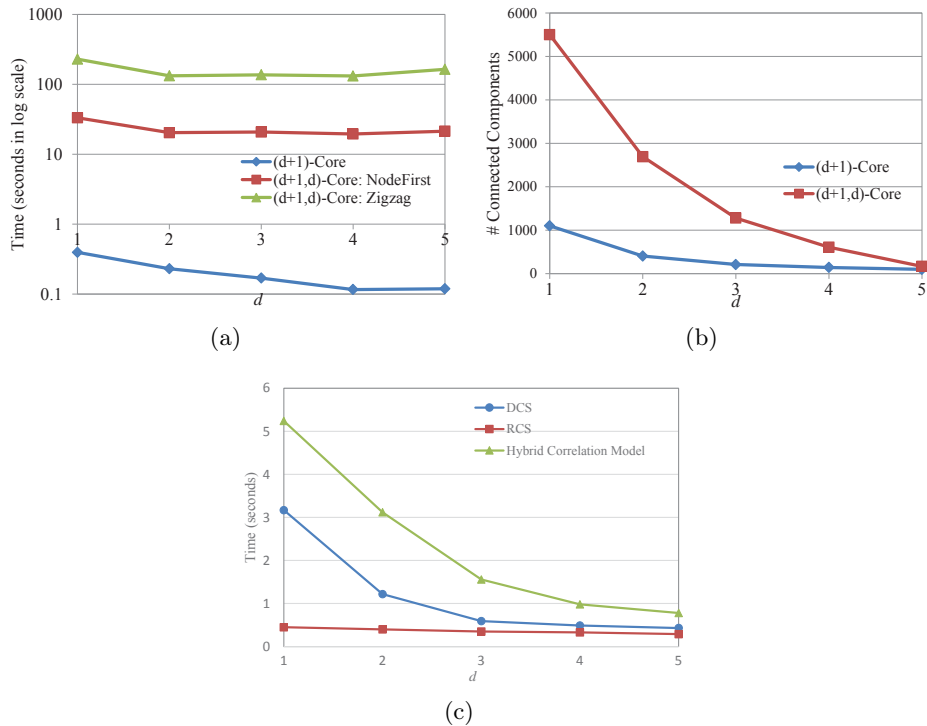


Figure 5.7: (a) Running time of $(d+1)$ -Core generation and $(d+1, d)$ -Core generation (**Zigzag**, **NodeFirst**). (b) The number of connected components generated by $(d+1)$ -Cores and $(d+1, d)$ -Cores. (c) Running time of different context search approaches. All experiments are running on Tech-Full dataset with the time window set to one week.

see that the performance of RCS is remarkably better than the performance of DCS. For the quality, we show the accuracy of RCS in Table 5.4, as the number of simulations n grow. We define n as a number proportional to the neighboring post size of story S and as we can see, 20% simulations already produce an accuracy higher than 95%.

5.6 Discussion and Conclusion

In this chapter, we focus on two problems: (1) Efficiently identify transient stories from fast streaming social content; (2) Perform Iceberg query to build the structural context between stories. To solve the first problem, we transform social stream in a time window to a post network, and model

5.6. Discussion and Conclusion

$\frac{n}{ \text{neighboring posts of } S }$	10%	20%	30%	40%
$Avg(1 - \frac{ Cor_D(S,S') - Cor_R(S,S') }{Cor_D(S,S')})$	0.91	0.95	0.98	0.99

Table 5.4: The accuracy of RCS, as the number of simulations n grow. We define n as a number proportional to the neighboring post size of story S and measure the accuracy based on DCS.

transient stories as (k, d) -Cores in the post network. Two polynomial time algorithms are proposed to extract maximal (k, d) -Cores. For the second problem, we propose deterministic context search and randomized context search to support the iceberg query, which allows to perform context search without pairwise comparison. We perform detailed experimental study on real Twitter streams and the results demonstrate the effectiveness and value of our proposed context-aware story-teller **CAST**. In future work, we are interested in mining opinions from transient stories, e.g., the sentiment on a political event. Besides, we are interested in various visualization techniques to present transient stories to end users in a friendly way.

Chapter 6

Incremental Event Evolution Tracking

In Chapter 3, we discussed the modeling of an event as a density cluster. Density-based clustering is superior to other clustering methods such as K-Means or hierarchical clustering for event identification, since it is robust to noisy posts and can quickly find core posts in social streams. As the post network changes over time, the maintenance of density clusters without computation from scratch is very important and this is a highly challenging task. In this chapter, we focus on the tracking of event evolution patterns, which corresponds to the incremental density-based cluster maintenance on the post network level.

6.1 Introduction

People easily feel overwhelmed by the information deluge coming from social websites. There is an urgent need to provide users with tools which can automatically extract and summarize significant information from highly dynamic social streams, e.g., report emerging bursty events, or track the evolution of one or more specific events in a given time span. There are many previous studies [10, 26, 48, 55, 66, 67] on detecting new emerging events from text streams; they serve the need for answering the query “*what’s trending now?*” over social streams. However, in many scenarios, users may want to know more details about an event and may like to issue advanced queries like “*how’re things going?*”. For example, for the event “SOPA (Stop Online Piracy Act) protest” happening in January 2012, existing event detection approaches can discover bursty activities at each moment, but cannot answer queries like “how SOPA protest has evolved in the past few days?”. An ideal output to such an evolution query would be a “panoramic view” of the event history, which improves user experience. In graph perspective, we can model social streams as dynamically evolving post networks and model events as clusters over these networks, obtained by means of a clustering approach that

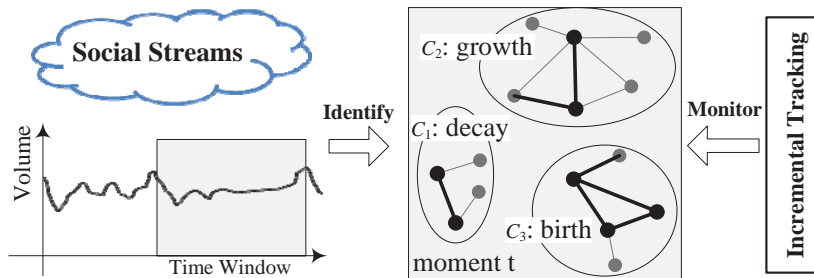


Figure 6.1: Post network captures the correlation between posts in the time window at each moment, and evolves as time rolls on. The skeletal graph is shown in bold. From moment t to $t + 1$, the incremental tracking framework will maintain clusters and monitor the evolution patterns on the fly.

is robust to the large amount of noise present in social streams. Accordingly, we consider the above kind of queries as an instance of the **cluster evolution tracking** problem, which aims to track the cluster evolution patterns at each moment from such dynamic networks. Typical cluster evolution patterns include birth, death, growth, decay, merge and split. Event *detection* can be viewed as a subproblem of cluster *evolution tracking* in social streams.

In this chapter, we propose an incremental tracking framework for cluster evolution over highly dynamic networks. To illustrate the techniques in this framework, we consider the *event evolution tracking* task in social streams as an application, where a social stream and an event are modeled as a dynamic post network and a post cluster respectively. The reasons we deploy our framework on this application are: social streams usually surge very quickly, making it ideal for the performance evaluation, and events are human-readable, making it convenient to assess the quality. In detail, since a significant portion of social posts like tweets are just noise, we first define a *Skeletal Graph* as a compact summary of the original post network, from which post clusters can be generated. Then, as we will discuss later, we monitor the network updates with a fading time window, and capture the evolution patterns of networks and clusters by a group of primitive evolution operations and their algebra. Moreover, we extend node-by-node evolution to subgraph-by-subgraph evolution to boost the performance of evolution tracking of clusters. Figure 6.1 shows an overview of major modules we use for cluster evolution tracking in social streams.

We notice that at a high level, our method resembles previous work on density-based clustering over streaming data, e.g., DenStream [17], DStream

6.1. Introduction

$S_F(p_1, p_2)$	the fading similarity between posts p_1 and p_2
(ε, δ)	similarity threshold, priority threshold
$w^t(p)$	the priority of post p at moment t
$G_t(V_t, E_t)$	the post network at moment t
G_{old}	the old subgraph that lapses at moment $t + 1$
G_{new}	the new subgraph that appears at moment $t + 1$
$\overline{G}_t(\overline{V}_t, \overline{E}_t)$	the skeletal graph at moment t
$\overline{C}, \overline{S}_t$	a component, a component set in \overline{G}_t
C, S_t	a cluster, a cluster set in G_t
$\mathcal{N}(p)$	post p 's neighbor set with similarity larger than ε
$N_c(p)$	the cluster set of post p 's neighboring core posts

Table 6.1: Notation.

[18] and cluster maintenance in [2] and [5]. However, there are several major differences with this body of work. First, our approach works on highly dynamic networks and provides users the flexibility in choosing the scope for tracking by means of a fading time window. Second, the existing work can only process the adding of nodes/edges one by one, while our approach can handle adding, deleting and fading of nodes, in bulk mode, i.e., *subgraph by subgraph*. This is an important requirement for dealing with the high throughput rate of dynamic networks. Third, the focus of our approach is tracking and analyzing the cluster evolution dynamics in the whole life cycle. By contrast, the previous works focus on clustering streaming data, which is a sub-task in our problem.

On the application side, comparing with topic tracking approaches, we note that they are usually formulated as a classification problem [4]: when a new story arrives, compare it with topic features in the training set by decision trees or k -NN [78], and if it matches sufficiently, declare it to be on a topic. Since these approaches assume that topics are *predefined before tracking*, we cannot simply apply them to event evolution tracking in social streams. Comparing with existing event detection and tracking approaches [48, 55, 66, 67], our framework has advantages in tracking the whole life cycle and capturing composite evolution behaviors such as merging and splitting.

The problem is formalized in Sec. 6.2. For convenience, we summarize the major notations used in this chapter in Table 6.1.

6.2 Problem Formalization

We formally define dynamic network and dynamic clusters here, and then introduce the problem this paper seeks to solve.

Dynamic Network. A dynamic network is a network with node and edge updates over time. We define a snapshot of an dynamic network at moment t as a weighted graph $G_t(V_t, E_t)$, where an edge $e(u, v) \in E_t$ connects nodes u, v in V_t and $s(u, v)$ is the similarity between them. For the problem studied in this paper, we assume a dynamic network is the input and $s(u, v) \in (0, 1]$ is a value usually set by a specific similarity function. From time t to $t + 1$, we use ΔG_{t+1} to describe the updating subgraph applied to G_t , i.e., $G_t + \Delta G_{t+1} = G_{t+1}$. In real applications, the size of ΔG_{t+1} is typically much smaller than the size of G_t or G_{t+1} , and we make this as an assumption throughout this Chapter. Naturally, a dynamic network $G_{1:i}$ from moment 1 to i can be described as a continuous updating subgraph sequence applied at each moment. The formal definition is given below.

Definition 19 A dynamic network $G_{i:j}$ from moment i to j is denoted as $(G_i; \Delta G_{i+1}, \dots, \Delta G_j)$, where $G_i(V_i, E_i)$ is a weighted graph with node set V_i and edge set E_i , and ΔG_{t+1} ($i \leq t < j$) is an updating subgraph at moment $t + 1$ such that $G_t + \Delta G_{t+1} = G_{t+1}$.

When the network evolves from G_t to G_{t+1} , we reasonably assume that at moment $t + 1$, only a small portion of G_t is incrementally updated, i.e., $|V_{t+1} - V_t| + |V_t - V_{t+1}| \ll |V_t|$ and $|E_{t+1} - E_t| + |E_t - E_{t+1}| \ll |E_t|$. This assumption generally holds in practice, and when it doesn't, we can shorten moment interval sufficiently to make the assumption hold. For simplicity, we express ΔG_{t+1} as a sequence of node additions and deletions, e.g., $\Delta G_{t+1} := +v_1 - v_2$ means adding node v_1 and all the edges incident with v_1 in a single operation, and analogously, deleting node v_2 and its incident edges in a subsequent operation.

Dynamic Clusters. Let's suppose C_t is a subgraph in G_t , and $isCluster(C_t)$ is a boolean function to validate whether C_t is a cluster or not, with the exact definition given in Sec. 6.4.2. In the following, we define a dynamic density cluster.

Definition 20 A dynamic cluster $C_{i:j}$ from moment i to j is denoted as $(C_i; \Delta C_{i+1}, \dots, \Delta C_j)$ where $isCluster(C_i) = True$, and ΔC_{t+1} ($i \leq t < j$)

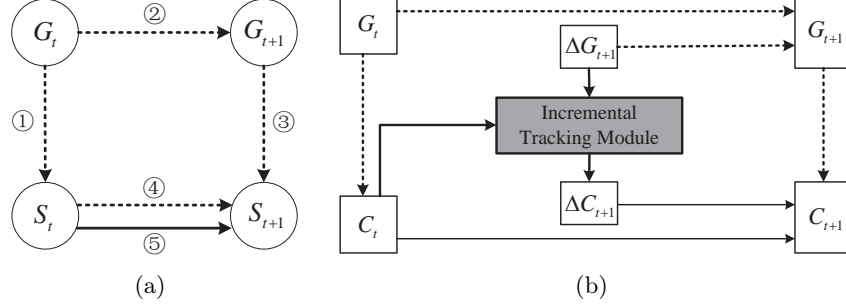


Figure 6.2: (a) The commutative diagram between dynamic networks G_t, G_{t+1} and cluster sets S_t, S_{t+1} . The “divide-and-conquer” baseline and our *Incremental Tracking* are annotated by dotted and solid lines respectively. (b) The workflow of incremental tracking module, which shows our framework tracks cluster evolution dynamics by only consuming the updating subgraph ΔG_{t+1} .

is an updating subgraph at moment $t + 1$ that makes $C_t + \Delta C_{t+1} = C_{t+1}$ and $isCluster(C_{t+1}) = True$.

The Problem. We focus on addressing the following problem:

Problem 2 *Supposing $G_{i:j} = (G_i; \Delta G_{i+1}, \dots, \Delta G_j)$ is a large dynamic network and $isCluster(C_t)$ is a binary validation function for cluster candidate C_t , the problem of incremental cluster evolution is to generate an updating subgraph sequence $(\Delta C_{i+1}, \dots, \Delta C_j)$ with $C_t + \Delta C_{t+1} = C_{t+1}$ and $isCluster(C_{t+1}) = True$, where $i \leq t < j$.*

The cluster evolution patterns can be observed from the updating sequence. For example, if $C_t \neq \emptyset$ but $C_{t+1} = \emptyset$, it means C_t dies at moment $t + 1$. $C_t = \emptyset$ but $C_{t+1} \neq \emptyset$, a new cluster C_{t+1} is born at moment $t + 1$. Typical cluster evolution patterns include birth, death, growth, decay, merge and split. In this paper, we aim to track the complete set of cluster evolution patterns in real time.

6.3 Incremental Tracking Framework

We illustrate the relationship between dynamic networks G_t, G_{t+1} and cluster sets S_t, S_{t+1} at consecutive moments as a commutative diagram in Fig-

ure 6.2(a). The traditional approaches for tracking dynamic network related problems usually follow a “divide-and-conquer” spirit [38], which consists of three components: (1) decompose a dynamic network into a series of snapshots for each moment, (2) apply graph mining algorithms on each snapshot to find useful patterns, (3) match patterns between different moments to generate a dynamic pattern sequence. Applied to our problem, to track cluster evolution patterns, these steps are:

Step ①: At moment t , identify the cluster set S_t from G_t ;

Step ②: At moment $t + 1$, as the network evolves, generate G_{t+1} from G_t using ΔG_{t+1} ;

Step ③: Again, identify the cluster set S_{t+1} from G_{t+1} ;

Step ④: Generate cluster evolution patterns from time t to $t + 1$ by tracing the correspondence between S_t and S_{t+1} .

However, this approach suffers from both performance and quality. Firstly, repeated extraction of clusters from large networks from scratch is a very expensive operation (steps ① and ③), and tracing the correspondence between cluster sets at successive moments is also expensive (step ④). Secondly, the step of tracing correspondence, since it is done after two cluster sets are generated, may lead to loss of accuracy. In contrast, the method we propose is incremental tracking of cluster evolution, which corresponds to step ⑤ in Figure 6.2(a). The workflow of this incremental tracking from time t to $t + 1$ is illustrated in Figure 6.2(b). More precisely, for the very first snapshot of the dynamic network, say G_0 , our approach will generate the corresponding event set S_0 from scratch. After this, we update the existing cluster set by the changed parts and move to the next moment recursively, by only applying step ⑤, i.e., we incrementally derive S_{t+1} from S_t and ΔG_{t+1} . The experiments on real data set show that our incremental tracking approach outperforms the traditional baselines in both performance and quality.

6.4 Skeletal Graph Clustering

The functional relationships between different types of objects defined in this paper are illustrated in Figure 6.3. As an example, the arrow from G_t to \overline{G}_t with label Ske means \overline{G}_t is derived from G_t by function Ske , i.e., $\overline{G}_t = Ske(G_t)$. See Table 6.1 for notations used. The various objects and their relationships will be explained in the rest of the paper.

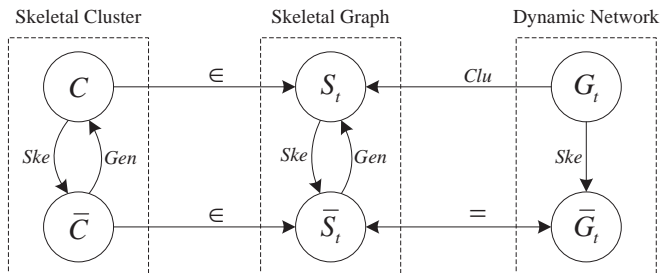


Figure 6.3: The functional relationships between different types of objects defined in this paper, e.g., the arrow from G_t to \bar{G}_t with label *Ske* means $\bar{G}_t = Ske(G_t)$. Refer to Table 6.1 for notations.

6.4.1 Node Prioritization

In reality, many posts tend to be just noise, so it is essential to identify those nodes that play a central role in clusters. On web link graph analysis, there is a lot of research on node authority ranking, e.g., HITS and PageRank [53]. However, most of these methods are iterative and not applicable to the *single-pass* computation on streaming data. Node prioritization is a technique to quickly differentiate and rank the processing order of nodes by their roles in a single pass. It is extremely useful in big graph mining, where there are too many nodes to be processed and many of them are of little significance. However, to the best of our knowledge, there is insufficient study on single-pass node prioritization in a streaming environment.

In this paper, we perform node prioritization based on density parameters (ε, δ) , where $0 < \varepsilon < 1$, and $\varepsilon \leq \delta$. In density-based clustering (e.g., DBSCAN [20]), the threshold *MinPts* is used as the minimum number of nodes in an ε -neighborhood, required to form a cluster. We adapt this and use a weight threshold δ as the minimum total weight of neighboring nodes, required to form a cluster. The reason we choose density-based approaches is that, compared with partitioning-based approaches (e.g., K-Means [30]) and hierarchical approaches (e.g., BIRCH [30]), density-based methods such as DBSCAN define clusters as areas of higher density than the remainder of the data set, which is effective in finding arbitrarily-shaped clusters and is robust to noise. Moreover, density-based approaches are easy to adapt to support single-pass clustering. In the post network, we consider ε to be a *similarity threshold* to decide connectivity, and can be used to define the post priority.

6.4. Skeletal Graph Clustering

Definition 21 Given a post $p = (L, \tau, a)$ in post network $G_t(V_t, E_t)$ and similarity threshold ε , the priority of p at time t ($t \geq p^\tau$), is defined as

$$w^t(p) = \frac{1}{e^{|t-p^\tau|}} \sum_{q \in \mathcal{N}(p)} S_F(p, q) \quad (6.1)$$

where $\mathcal{N}(p)$ is the subset of p 's neighbors with $S_F(p, q) > \varepsilon$.

Notice that post priority decays as time moves forward. Thus, post priority needs to be continuously updated. In practice, we only store the sum $\sum_{q \in \mathcal{N}(p)} S_F(p, q)$ with p to avoid frequent updates and compute $w^t(p)$ on demand.

Skeletal Graph. With post priority computed, we use δ as a *priority threshold* to differentiate nodes in $G_t(V_t, E_t)$:

- A post p is a *core post* if $w^t(p) \geq \delta$;
- It is a *border post* if $w^t(p) < \delta$ but there exists at least one core post $q \in \mathcal{N}(p)$;
- It is a *noise post* if it is neither core nor border, i.e., $w^t(p) < \delta$ and there is no core post in $\mathcal{N}(p)$.

Intuitively, a post is a core post if it shares enough common entities with many other posts. Neighbors of a core post are at least border posts, if not core posts themselves. Core posts play a central role: if a core post p is found to be a part of a cluster C , its neighboring (border or core) posts will also be a part of C . This property can be used in the single-pass clustering: if an incoming post p is “reachable” from an existing core post q , post p will be assigned to the cluster with q . Core posts connected by edges with similarity higher than ε will form a summary of $G_t(V_t, E_t)$, that we call the skeletal graph.

Definition 22 Given post network $G_t(V_t, E_t)$ and density parameters (ε, δ) , we define the skeletal graph as the subgraph of $G_t(V_t, E_t)$ induced by posts with $w^t(p) \geq \delta$ and edges with similarity higher than ε . We write $\overline{G}_t = \text{Ske}(G_t)$.

Ideally, $\overline{G}_t(\overline{V}_t, \overline{E}_t)$ will retain important information in G_t . Empirically, we found that adjusting the granularity of (ε, δ) to make the size $|\overline{V}_t|$ roughly equal to 20% of $|V_t|$ leads to a good balance between the quality of the skeletal graph in terms of the information retained and its space complexity. More tuning details can be found in Section 6.7.1.

6.4.2 Skeletal Cluster Identification

One of the key ideas in our incremental cluster evolution tracking approach is to use the updating subgraph ΔG_{t+1} between successive moments to maintain the skeletal clusters. Post clusters are constructed from these skeletal clusters. Maintaining skeletal clusters can be done efficiently since the skeletal graph is much smaller in size than the post graph it's obtained from. Besides efficiency, skeletal cluster has the advantage of giving the correspondence between successive post clusters in a very small cost.

Definition 23 Given $G_t(V_t, E_t)$ and the corresponding skeletal graph $\overline{G}_t(\overline{V}_t, \overline{E}_t)$, a skeletal cluster \overline{C} is a connected component of \overline{G}_t . A post cluster is a set of core posts and border posts generated from a skeletal cluster \overline{C} , written as $C = \text{Gen}(\overline{C})$, using the following expansion rules:

- All posts in \overline{C} form the core posts of C .
- For every core post in C , all its neighboring border posts in G_t form the border posts in C .

In what follows, by cluster, we mean a post cluster, distinguished from the explicit term skeletal cluster. By definition, a core post only appears in one (post) cluster. If a border post is associated with multiple core posts in different clusters, this border post will appear in multiple (post) clusters.

6.5 Incremental Cluster Evolution

In this section, we discuss the incremental evolution of skeletal graph and post clusters under the fading time window.

6.5.1 Fading Time Window

Fading (or decay) function and sliding time window are two common aggregation schemes used in time-evolving graphs (e.g., see [17]). Fading scheme puts a higher emphasis on newer posts, as captured by fading similarity in Eq. (3.1). Sliding time window scheme (posts are first-in, first-out) is essential because it provides a scope within which a user can monitor and track the evolution.

Since clusters evolve quickly from moment to moment, even within a given time window, it is important to highlight new posts and degrade old posts using the fading scheme. Thus, we combine these two schemes and introduce a *fading time window*, as illustrated in Figure 6.4. In practice, users

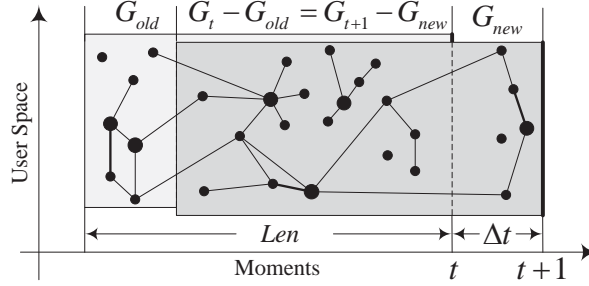


Figure 6.4: An illustration of the fading time window from time t to $t + 1$, where post priority may fade w.r.t. the end of time window. G_t will be updated by deleting subgraph G_{old} and adding subgraph G_{new} .

can specify the length of the time window to adjust the scope of monitoring. Users can also choose different fading functions to penalize old posts and highlight new posts in different ways. Let Δt denote the interval between moments. For simplicity, we abbreviate the moment $(t + i \cdot \Delta t)$ as $(t + i)$. When the time window slides from moment t to $t + 1$, the post network $G_t(V_t, E_t)$ will be updated to be $G_{t+1}(V_{t+1}, E_{t+1})$. Suppose $G_{old}(V_{old}, E_{old})$ is the old subgraph (of G_t) that lapses at moment $t + 1$ and $G_{new}(V_{new}, E_{new})$ is the new subgraph (of G_{t+1}) that appears (see Figure 6.4). Clearly,

$$G_{t+1} = G_t - G_{old} + G_{new} \quad (6.2)$$

Let Len be the time window length. We assume $Len > 2\Delta t$, which makes $V_{old} \cap V_{new} = \emptyset$. This assumption is reasonable in applications, e.g., we set Len to 1 week and Δt to 1 day.

6.5.2 Network Evolution Operations

We analyze the evolution process of networks and clusters at each moment and abstract them into five primitive operators: $+$, $-$, \odot , \uparrow , \downarrow . We classify the operators based on the objects they manipulate: nodes or clusters, and define them below.

Definition 24 *Primitive node operations:*

- $G_t + p$: add a new post p into $G_t(V_t, E_t)$ where $p \notin V_t$. All the new edges associated with p will be constructed automatically by linkage search (explained in Sec. 3.3);
- $G_t - p$: delete a post p from $G_t(V_t, E_t)$ where $p \in V_t$. All the existing edges associated with p will be automatically removed from E_t .

- $\odot G_t$: update the post priority scores in G_t .
- Composite node operations:
 - $G_t \oplus p = \odot(G_t + p)$: add a post p into $G_t(V_t, E_t)$ where $p \notin V_t$ and update the priority of related posts;
 - $G_t \ominus p = \odot(G_t - p)$: delete a post p from $G_t(V_t, E_t)$ where $p \in V_t$ and update the priority of related posts.

Definition 25 *Primitive cluster evolution operations:*

- $+C$: generate a new cluster C ;
 - $-C$: remove an old cluster C ;
 - $\uparrow(C, p)$: increase the size of C by adding post p ;
 - $\downarrow(C, p)$: decrease the size of C by removing post p .
- Composite cluster evolution operations:
- $Merge(S) = +C - S$: merge a set of clusters S into a new single cluster C and remove S ;
 - $Split(C) = -C + S$: split a single cluster C into a set of new clusters S and remove C .

In particular, composite node operations are designed to conveniently describe the adding/deleting of posts with priority scores updated in the same time, and composite cluster operations are designed to capture the advanced evolution patterns of clusters. Each operator defined above on a single object can be extended to a set of objects, i.e., for a node set $\mathcal{X} = \{p_1, p_2, \dots, p\}$, $G_t + \mathcal{X} = G_t + p_1 + p_2 + \dots + p$. This is well defined since $+$ is associative and commutative. We use the left-associative convention for ‘ $-$ ’: that is, we write $A - B - C$ to mean $(A - B) - C$. These operators will be used later in the formal description of the evolution procedures. Figure 6.5(a) depicts the role played by the primitive operators in the tracking of cluster evolutions from dynamic networks.

6.5.3 Skeletal Graph Evolution Algebra

The updating of skeletal graphs from \bar{G}_t to \bar{G}_{t+1} is the core task in cluster evolution tracking. If we ignore the node priorities for a moment, the following formula shows different ways to compute the overlapping part in G_{t+1} and G_t , as illustrated in Figure 6.4:

$$G_{t+1} - G_{new} = G_t - G_{old} = G_{t+1} \ominus G_{new} = G_t \ominus G_{old} \quad (6.3)$$

However, at the skeletal graph level, $Ske(G_{t+1} - G_{new}) \neq Ske(G_t - G_{old})$: some core posts in $G_t - G_{old}$ may no longer be core posts due to the removal of edges incident with nodes in G_{old} or simply due to the passing of time; some non-core posts may become core posts because of the adding of edges with nodes in G_{new} . To measure the changes in the overlapping part, we define the following three components.

Definition 26 *Updated components in overlap:*

- $\bar{S}_+ = Ske(G_{t+1} - G_{new}) - Ske(G_{t+1} \ominus G_{new})$: components of non-core posts in $G_t - G_{old}$ that become core posts in $G_{t+1} - G_{new}$ due to the adding of G_{new} ;
- $\bar{S}_- = Ske(G_t - G_{old}) - Ske(G_t \ominus G_{old})$: components of core posts in $G_t - G_{old}$ that become non-core posts in $G_{t+1} - G_{new}$ due to the removing of G_{old} ;
- $\bar{S}_\odot = Ske(G_t \ominus G_{old}) - Ske(G_{t+1} \ominus G_{new})$: components of core posts in $G_t - G_{old}$ that become non-core posts in $G_{t+1} - G_{new}$ due to the passing of time.

Based on Definition 26, from moment t to $t + 1$, the changes of core posts in the overlapping part, i.e., $G_{t+1} - G_{new}$ (equivalently, $G_t - G_{old}$ - see Figure 6.4), can be updated using the components \bar{S}_+ , \bar{S}_- and \bar{S}_\odot . That is,

$$\begin{aligned}
 & Ske(G_{t+1} - G_{new}) - Ske(G_t - G_{old}) \\
 = & (Ske(G_{t+1} - G_{new}) - Ske(G_{t+1} \ominus G_{new})) \\
 & - (Ske(G_t - G_{old}) - Ske(G_t \ominus G_{old})) \\
 & - (Ske(G_t \ominus G_{old}) - Ske(G_{t+1} \ominus G_{new})) \\
 = & \bar{S}_+ - \bar{S}_- - \bar{S}_\odot
 \end{aligned} \tag{6.4}$$

Let \bar{S}_{old} and \bar{S}_{new} denote the sets of skeletal clusters in G_{old} and G_{new} respectively. The following theorem characterizes the iterative and incremental updating of skeletal graphs from moment t to $t + 1$, and it plays a central role in the cluster evolution.

Theorem 1 *From moment t to $t + 1$, the skeletal graph evolves by removing core posts in G_{old} , adding core posts in G_{new} and updating core posts in the overlapping part. That is*

$$\bar{S}_{t+1} = \bar{S}_t - \bar{S}_{old} - \bar{S}_- - \bar{S}_\odot + \bar{S}_{new} + \bar{S}_+ \tag{6.5}$$

Proof: Since operator ‘-’ does not update post priority, we have $Ske(G_{t+1} - G_{new}) = Ske(G_{t+1}) - Ske(G_{new}) = \bar{S}_{t+1} - \bar{S}_n$, $Ske(G_t - G_{old}) = Ske(G_t) -$

6.5. Incremental Cluster Evolution

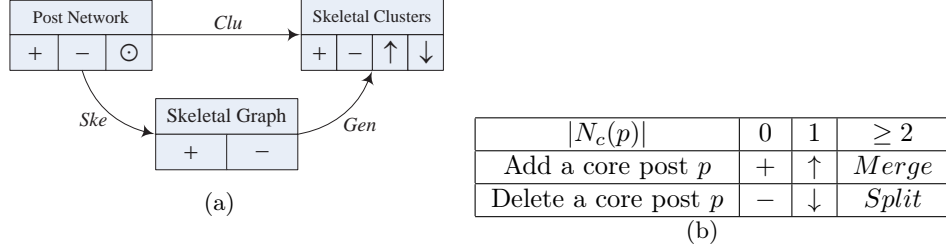


Figure 6.5: (a) The relationships between primitives and evolutions. Each box represents an evolution object and the arrows between them describe inputs/outputs. (b) The evolutionary behavior table for clusters when adding or deleting a core post p .

$Ske(G_{old}) = \bar{S}_t - \bar{S}_{old}$. Then, $\bar{S}_{t+1} - \bar{S}_{new} - \bar{S}_t + \bar{S}_{old} = \bar{S}_+ - \bar{S}_- - \bar{S}_\ominus$ and we get the conclusion. \square

Theorem 1 indicates that we can incrementally maintain skeletal clusters \bar{S}_{t+1} from \bar{S}_t . Since we define (post) clusters based on skeletal clusters, this incremental updating of skeletal clusters benefits incremental updating of cluster evolution essentially.

6.5.4 Incremental Cluster Evolution

Let $S_t = Clu(G_t)$ denote the set of clusters obtained from the post network G_t . Notice that noise posts in G_t do not appear in any clusters, so the number of posts in S_t is typically smaller than $|V_t|$. Next, we explore the incremental cluster evolution problem from two levels: the node-by-node updating level and subgraph-by-subgraph updating level.

Node-by-Node Evolution. The basic operations underlying cluster evolution are the cases when S_t is modified by the addition or deletion of a cluster that includes only one post. In the following, we analyze and show the evolution of clusters by adding or deleting a post p . When adding p , we let $N_c(p)$ denote the set of clusters that p 's neighboring core posts belong to *before* p is added. When deleting p , let $N_c(p)$ denote the set of clusters that p 's neighboring core posts belong to *after* p is removed. $|N_c(p)| = 0$ means p has no neighboring core posts. Notice that *Merge* and *Split* are composite operations and can be decomposed into a series of cluster primitive operations. We show the evolution behaviors of clusters in Figure 6.5(b) and explain the detail below.

(a) **Addition:** $S_t + \{p\}$

If p is a noise post after being added into G_t , ignore p . If p is a border post, add p to each cluster in $N_c(p)$. Else, p is a core post and we do the following:

- If $|N_c(p)| = 0$: apply $+C$, where $C = \{p\} \cup \mathcal{N}(p)$;
- If $|N_c(p)| = 1$: apply $\uparrow(C, \{p\} \cup \mathcal{N}(p))$, where C is the lone cluster in $N_c(p)$;
- If $|N_c(p)| \geq 2$: apply $Merge = +C - \sum_{C' \in N_c(p)} C'$ and $C = N_c(p) \cup \{p\} \cup \mathcal{N}(p)$.

(b) Deletion: $S_t - \{p\}$

If p is a noise post before being deleted from G_t , ignore p . If p is a border post, delete p from each cluster in $N_c(p)$. Else, p is a core post and we do the following:

- If $|N_c(p)| = 0$: apply $-C$ where $p \in C$;
- If $|N_c(p)| = 1$: apply $\downarrow(C, \{p\} \cup \mathcal{N}(p))$;
- If $|N_c(p)| \geq 2$: apply $Split = -C + \sum_{C' \in N_c(p)} C'$, where $p \in C$ before the deletion.

Subgraph-by-Subgraph Evolution. When dynamic networks such as post networks in social streams surge quickly, the node-by-node processing for cluster evolution will lead to a poor performance. To accelerate the performance, we consider the subgraph-by-subgraph updating approach. Let $Clu(G_{new}) = S_{new}$ and $Clu(G_{old}) = S_{old}$ be the cluster sets of the graphs G_{new} and G_{old} , and S_t be the set of all clusters at moment t . As the time window moves forward to moment $t + 1$, if we add G_{new} to the network G_t , clusters will evolve as follows:

$$\begin{aligned}
 Clu(G_t + G_{new}) &= Gen(Ske(G_t + G_{new})) \\
 &= Gen(Ske(G_t) + Ske(G_{new}) + \bar{S}_+ - \bar{S}_\odot) \quad (\text{Definition 26}) \\
 &= S_t + S_{new} + S_+ - S_\odot
 \end{aligned} \tag{6.6}$$

where $S_+ = Gen(\bar{S}_+)$ and $S_\odot = Gen(\bar{S}_\odot)$. Similarly, if we remove G_{old} from the network G_t , clusters evolve as follows:

$$\begin{aligned}
 Clu(G_t - G_{old}) &= Gen(Ske(G_t - G_{old})) \\
 &= Gen(Ske(G_t) - Ske(G_{old}) - \bar{S}_-) \quad (\text{Definition 26}) \\
 &= S_t - S_{old} - S_-
 \end{aligned} \tag{6.7}$$

where $S_- = Gen(\bar{S}_-)$. Based on Equation (6.6) and (6.7), from moment t to $t + 1$, the set of clusters can be incrementally updated by the iterative computation

$$\begin{aligned}
S_{t+1} &= Clu(G_{t+1}) = Clu(G_t - G_{old} + G_{new}) \\
&= S_t - S_{old} - S_- + S_{new} + S_+ - S_{\odot}
\end{aligned} \tag{6.8}$$

Equation (6.8) can be also verified by applying *Gen* function on both sides of Equation (6.5). Naturally, Equation (6.8) provides a theoretical basis for the incremental computation of cluster evolution: as the post network evolves from G_t to G_{t+1} , we do not compute S_{t+1} from G_{t+1} . Instead, we incrementally update S_t by means of the five cluster sets appearing in Equation (6.8), using simple set operations. Since the sizes of G_{old} and G_{new} are usually very small compared with G_t , these five cluster sets are also of small size and so we can generate S_{t+1} quickly from them. The details of incremental computation are discussed in Section 6.6.

6.6 Incremental Algorithms

The traditional approach of decomposing an evolving graph into a series of snapshots suffers from both quality and performance, since clusters are generated from scratch and matched heuristically at each moment. To overcome this limitation, we propose an incremental tracking framework, as introduced in Section 6.5 and illustrated in Figure 6.2(b). In this section, we leverage our incremental computation by proposing Algorithm 8 for the incremental cluster maintenance (ICM) and Algorithm 9 for the cluster evolution tracking (eTrack) respectively. Since at each moment $|V_{old}| + |V_{new}| \ll |V_t|$, our algorithms can save a lot computation by adjusting clusters incrementally, rather than generating them from scratch.

Bulk Updating. Traditional incremental computation on dynamic graphs usually treats the addition/deletion of nodes or edges one by one [18, 22]. However, in a real scenario, since social posts arrive at a high speed, the post-by-post incremental updating will lead to very poor performance. In this paper, we speed up the incremental computation of S_t by *bulk updating*. Clearly, updating S_t with a single node $\{p\}$ is a special case of bulk updating. Here, a bulk corresponds to a cluster of posts and we “lift” the post-by-post updating of S_t to the bulk updating level. Recall that $N_c(p)$ is the neighboring cluster set of p where p is a core post. To understand the bulk updating in Algorithm 8, for a cluster C , we define $N_c(\overline{C})$ as the neighboring cluster set of posts in \overline{C} , i.e., $N_c(\overline{C}) = \cup_{p \in \overline{C}} N_c(p)$ where $\overline{C} = Ske(C)$. When C is added into or deleted from S_t as a bulk, the size of $N_c(\overline{C})$ will decide the evolution patterns of clusters from moment t to $t + 1$ after C is added or

deleted, as shown in Figure 6.5(b). Since C is usually a small subgraph, we consider a bulk operation can be done in constant time. Since internal edges of a subgraph \overline{C} can be ignored when determining $N_c(\overline{C})$, bulk updating is more efficient than node-by-node updating.

Incremental Cluster Maintenance (ICM). The steps for incremental cluster maintenance (ICM) from any moment t to $t + 1$ are summarized in Algorithm 8. The ICM algorithm follows the iterative computation shown in Equation (6.8), that is $S_{t+1} = S_t - S_{old} - S_- - S_{\odot} + S_{new} + S_+$. As analyzed in Section 6.5.4, each bulk addition and bulk deletion has three possible evolution behaviors, decided by the size of $N_c(\overline{C})$. Lines 3-13 deal with deleting a bulk C , where three patterns $\{-, \downarrow, Split\}$ are handled. Lines 15-26 deal with adding a bulk C and handle another three patterns $\{+, \uparrow, Merge\}$. Supposing there are n bulk updates in ICM, the time complexity of ICM is $O(n)$. Since a bulk operation is generally completed in constant time, ICM is an efficient single-pass incremental computation algorithm.

Cluster Evolution Tracking (eTrack). Given a dynamic network $G_{i:j}$ and the set of clusters S_i at the start moment i , the eTrack algorithm will track the primitive cluster evolution operations at each moment, working on top of the ICM algorithm (Line 3). We summarize the steps of eTrack in Alg. 9. Basically, eTrack monitors the changes of clusters effected by ICM at each moment. If the cluster is not changed, eTrack will take no action; otherwise, eTrack will determine the corresponding cases and output the cluster evolution patterns (Lines 4-12). Notice that in Lines 5-8, if a cluster C in S_t has ClusterId id , we use the convention that $S_t(id) = C$ to access C by id , and $S_t(id) = \emptyset$ means there is no cluster in S_t with ClusterId id . Especially, lines 7-8 mean a cluster in S_t evolves into a cluster in S_{t+1} by deleting the posts in $S_t(id) - S_{t+1}(id)$ first and adding the posts in $S_{t+1}(id) - S_t(id)$ later. As an efficient monitoring algorithm, once we get S_{t+1} incrementally by ICM, the time complexity of eTrack is linear in the number of clusters in S_t and S_{t+1} at each moment.

6.7 Experiments

In this section, we first discuss how to tune the construction of post network and skeletal graph to find the best selection of entity extraction and density parameters. Then, we test the quality and performance of cluster evolution tracking algorithms on two social streams: Tech-Lite and Tech-Full that we crawled from Twitter. Our event detection baseline covers the major techniques reported in [48, 55, 66, 67]. Our evolution tracking baseline captures

6.7. Experiments

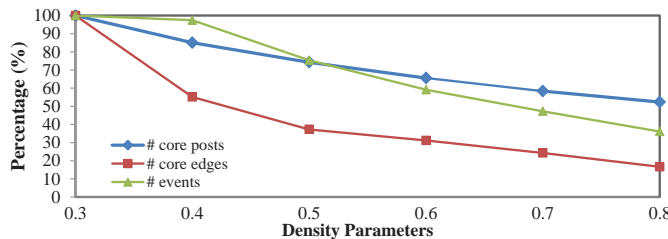


Figure 6.6: The trends of the number of core posts, core edges and events when increasing δ from 0.3 to 0.8. We set $\delta = \varepsilon = 0.3$ as the 100% basis.

the essence of the state of the art algorithms reported in [38]. All algorithms are implemented in Java. We use the graph database Neo4J¹¹ to store and manipulate the post network.

Datasets. All datasets are crawled from Twitter via Twitter API. Although our cluster evolution tracking algorithm works regardless of the domain, in order to facilitate evaluation, we make the dataset domain specific. The crawling of datasets is performed as follows. We built a technology domain dataset called Tech-Lite by aggregating all the timelines of users listed in the Technology category of “Who to follow”¹² and their retweeted users. Tech-Lite has 352,328 tweets, 1402 users and the streaming rate is about 11700 tweets/day. Based on the intuition that the followers of users in Technology category are most likely to be in the same domain, we obtained a larger technology social stream called Tech-Full by collecting all the timelines followed by users in the Technology category. Tech-Full has 5,196,086 tweets, created by 224,242 users, whose streaming rate is about 7216 tweets/hour. Both Tech-Lite and Tech-Full include retweets and have a time span from Jan. 1 to Feb. 1, 2012. Since each tweet corresponds to a node in the post network, both Tech-Lite and Tech-Full produce highly dynamic networks. Notice that the performance of our single-pass incremental approach is mainly affected by the streaming rate, rather than the dataset size.

6.7.1 Tuning Skeletal Graph

Post Preprocessing. As described in Section 6.4, we extract entities from posts by POS tagger. One alternative approach to entity extraction is using hashtags. However, only 11% of the tweets in our dataset have hashtags, which results in lots of posts in the dataset having no similarity score

¹¹<http://neo4j.org/>

¹²http://twitter.com/who_to_follow/interests

between them. Another approach is simply tokenizing tweets into unigrams and treating unigrams as entities, and we call it the “Unigrams” approach, as discussed in [55]. Table 2(a) shows the comparison of the three entity extraction approaches in the first time window of the Tech-Full social stream. If we use “Unigrams”, obviously the number of entities is larger than other two approaches, but the number of edges between posts tends to be smaller, because tweets written by different users usually share very few common words even when they talk about the same event. The “Hashtags” approach also produces a smaller number of edges, core posts and events, since it generates a much sparser post network. Overall, the “POS-Tagger” approach can discover more similarity relationships between posts and produce more core posts and events given the same social stream and parameter setting.

Density Parameters. The density parameters (ε, δ) control the construction of the skeletal graph. Clearly, the higher the density parameters, the smaller and sparser the skeletal graph. Figure 6.6 shows the number of core posts, core edges and events as a percentage of the numbers for $\varepsilon = 0.3$, as δ increases from 0.3 to 0.8. Results are obtained from the first time window of the Tech-Full social stream. We can see the rate of decrease of $\#events$ is higher than the rates of $\#core$ posts and $\#core$ edges after $\delta > 0.4$, because events are less likely to form in sparser skeletal graphs. More small events can be detected by lower density parameters, but the computational cost will increase because of larger and denser skeletal graphs. However, for big events, they are not very sensitive to these density parameters. We set $\varepsilon = 0.3$, $\delta = 0.5$ as a trade-off between the size and number of events one hand and processing efficiency on the other.

6.7.2 Cluster Evolution Tracking

Ground truth. To generate the ground truth, we crawl news articles in January 2012 from famous technology websites such as TechCrunch, Wired, CNET, etc, without looking at tweets. Then we treat the titles of news as posts and apply our event tracking algorithm to extract event evolution patterns. Finally, a total of 20 major events with life cycles are identified as ground truth. Typical events include “happy new year”, “CES 2012”, “sopa wikipedia blackout”, etc. To find more small and less noticeable events, we use Google Trends for Search¹³, which shows the traffic trends of keywords that appeared in Google Search along the time dimension. If an event-indicating phrase has a volume peak in Google Trends at a specific time,

¹³<http://www.google.com/trends/>

6.7. Experiments

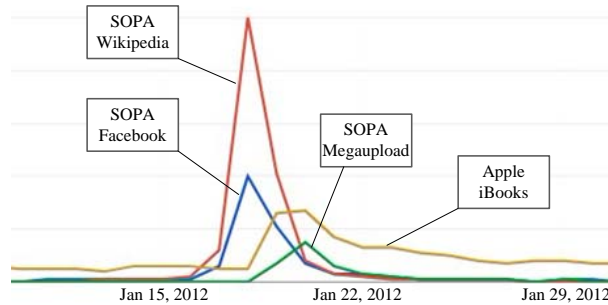


Figure 6.7: Examples of Google Trends peaks in January 2012. We validate the events generated by cTrack by checking the existence of volume peaks at a nearby time moment in Google Trends. Although these peaks can detect bursty events, Google Trends cannot discover the merging/splitting patterns.

HashtagPeaks	UnigramPeaks	Louvain	eTrack
CES	google	Apple iphone ipad	CES conference
SOPA	ces	CES ultrabook tablet	SOPA PIPA
EngadgetCES	apple	Google search privacy	Hug new year
opengov	video	Week's Android games	RIM new CEO
gov20	sopa	Kindle Netflix app	Yahoo jerry yang
CES2012	twitter	Internet people time	Samsung Galaxy Nexus
PIPA	year	Hope weekend	Apple iBooks
opendata	facebook	SOPA Megaupload	Facebook IPO News
StartupAmerica	app	SOPA PIPA Wikipedia	Martin Luther King
win7tech	iphone	Facebook IPO	Tim Cook Apple stock

Figure 6.8: Lists of top 10 events detected from Twitter Technology streams in January 2012 by baseline HashtagPeaks, UnigramPeaks, Louvain and our incremental tracking approach eTrack.

we say this event is sufficiently validated by the real world. We validate the correctness of an event C_i by the following process: we pick the top 3 entities of C_i ranked by frequency and search them in Google Trends, and if the traffic trend of these top entities has a distinct peak at a nearby time to C_i , we consider that C_i corresponds to a real world event widely witnessed by the public. Four examples of Google Trends peaks are shown in Figure 6.7. It is not surprising to find that the birth of events in social streams is usually earlier than its appearance in Google Trends.

Cluster Annotation. Considering the huge volume of posts in a cluster, it is important to summarize and present a post cluster as a conceptual event to aid human perception. In related work, Twitinfo [55] represents an event

it discovers from Twitter by a timeline of tweets, showing the tweet activity by volume over time. However, it is tedious for users to read tweets one-by-one to figure out the event detail. In this paper, we summarize a snapshot of a cluster by a word cloud [29]. The font size of a word in the cloud indicates its popularity. Compared with Twitinfo, word cloud provides a summary of the cluster at a glance and is much easier for human to read.

Comparison with DynDense [5]. DynDense works on entity graphs, with the majority of tweets ignored since they have less than 2 entities. DynDense models events as dense subgraphs with size smaller than N_{max} , usually set as 5. We observed that with less than N_{max} entities, many DynDense results are difficult to interpret as specific real events, e.g., a subgraph with 3 nodes {"HP", "Microsoft", "Google"}. Compared with DynDense, models events as large dense post clusters, which can be validated as real events by checking highly correlated core posts in the cluster.

Baseline 1: Peak-Detection. In recent works [48, 55, 66, 67], events are generally detected as volume peaks of phrases over time in social streams. These approaches share the same spirit that aggregates the frequency of event-indicating phrases at each moment to build a histogram and generates events by detecting volume peaks in the histogram. We design two variants of Peak-Detection to capture the major techniques used by these state-of-the-art approaches.

- Baseline 1a: **HashtagPeaks** which aggregates hashtags;
- Baseline 1b: **UnigramPeaks** which aggregates unigrams.

Notice, both baselines above are for event detection only. Lists of the top 10 events detected by **HashtagPeaks** and **UnigramPeaks** are presented in Figure 6.8. Some highly frequent hashtags like "#opengov" and "#opendata" are not designed for event indication, hurting the precision. **UnigramPeaks** uses the unigrams extracted from the social stream preprocessing stage, which has a better quality than **HashtagPeaks**. However, both of them are limited in their representation of events, because the internal structure of events is missing. Besides, although these peaks can detect bursty words, they cannot discover cluster evolution patterns such as the merging/splitting. For example, in Figure 6.7, there is no way to know "Apple announced iBooks" is a split from the big event "SOPA" earlier, as illustrated in detail in Figure 6.9.

Baseline 2: Community Detection. A community in a large network refers to a subgraph with dense internal connections and sparse connections with other communities. It is possible to define an event as a community of

posts. Louvain method [14], based on modularity optimization, is the state-of-the-art approach community detection method in terms of performance. We design a baseline called “**Louvain**” to detect events defined based on post communities. The top 10 events generated by **Louvain** are shown in Figure 6.8. As we can see, not every result detected by the **Louvain** method is meaningful. For example, “Apple iphone ipad” and “Internet people time” are too vague to correspond to any concrete real events. The reason is, although **Louvain** method can make sure every community has relatively dense internal and sparse external connections, it cannot guarantee that every node in the community is important and has a sufficiently high connectivity with other nodes in the same community. It is highly possible that a low-degree node belongs to a community only because it has zero connectivity with other communities. Furthermore, noise posts are quite prevalent in Twitter and they negatively impact Louvain method.

Baseline 3: Pattern-Matching. We design a baseline to track the evolution patterns of clusters between snapshots. In graph mining, the “divide-and-conquer” approach of decomposing the evolving graph into a series of snapshot graphs at each moment is a traditional way to tackle evolving graph related problems (e.g., [38]). As an example, Kim et al. [38] first cluster individual snapshots into quasi-cliques and then map them in adjacent snapshots over time. Inspired by this approach, we design a baseline for cluster evolution tracking, which characterizes the cluster evolution at consecutive moments, by identifying certain heuristic patterns:

- If $\frac{|C_t \cap C_{t+1}|}{|C_t \cup C_{t+1}|} \geq \kappa$ and $|C_t| \leq |C_{t+1}|$, $C_{t+1} = \uparrow C_t$;
- If $\frac{|C_t \cap C_{t+1}|}{|C_t \cup C_{t+1}|} \geq \kappa$ and $|C_t| > |C_{t+1}|$, $C_{t+1} = \downarrow C_t$.

where C_t and C_{t+1} are any two clusters detected at moment t and $t + 1$ respectively, $\kappa\%$ is the minimal commonality to say C_t and C_{t+1} are different snapshots of the same cluster. A higher $\kappa\%$ will result in a higher precision but a lower recall of the evolution tracking. Empirically we set $\kappa\% = 90\%$ to guarantee the quality. It is worth noting that this baseline generates the same clusters as the eTrack algorithm, but with a non-incremental evolution tracking approach.

Precision and Recall. To measure the quality of event detection, we use **HashtagPeaks**, **UnigramPeaks** and **Louvain** as baselines to compare with our algorithm eTrack. It is worth noting that Baseline 3 is designed for the tracking of event evolution patterns between moments, so we omit it here. We compare the precision and recall of top 20 events generated by baselines and eTrack and show the results in Table 2(b). Compared with the ground

truth, **HashtagPeaks** and **UnigramPeaks** have rather low precision and recall scores, because of their poor ability in capturing event bursts. Notice that multiple extracted events may correspond to the same ground truth event. eTrack outperforms the baselines in both precision and recall. Since there are many events discussed in the social media but not very noticeable in news websites, we also validate the precision of the generated events using Google Trends. As we can see, **HashtagPeaks** and **UnigramPeaks** perform poorly under Trends validation, since the words they generate are less informative and not very event-indicating. eTrack gains a precision of 95% in Google Trends, where the only failed result is “Samsung galaxy nexus”, whose volume is steadily high without obvious peaks in Google Trends. The reason may be that the social stream is very dynamic. **Louvain** is worse than eTrack. The results show eTrack is significantly better than the baselines in quality.

Life Cycle of Cluster Evolution. Our approach is capable of tracking the whole life cycle of a cluster, from birth to death. We explain this using the example of “CES 2012”, a major consumer electronics show held in Las Vegas from January 10 to 13. As early as Jan 6, our approach has already detected some discussions about CES and generated an event about CES. On Jan 8, most people talked about “CES prediction”, and on Jan 9, the highlighted topic was “CES tomorrow” and some hearsays about “ultrabook” which would be shown in CES. After the actual event happened on Jan 10, the event grew distinctly bigger, and lots of products, news and messages are spreading over the social network, and this situation continues until Jan 13, which is the last day of CES. Afterwards, the discussions become weaker and continue until Jan 14, when “CES” was not the biggest mention on that day but still existed in some discussions. Compared with our approach, Baselines 1 and 2 can detect the emerging of “CES” with a frequency count at each moment, but no trajectory is generated. Baseline 3 can track a very coarse trajectory of this event, i.e., from Jan 10 to Jan 12. The reason is, if an event changes rapidly and many posts at consecutive moments cannot be associated with each other, Baseline 3 will fail to track the evolution. Since in social streams the posts usually surge quickly, our approach is superior to the baselines. The illustration of “CES” evolution trajectory and extended discussions can be found in [46].

Cluster Merging & Splitting. Figure 6.9 illustrates an example of cluster merging and splitting generated by algorithm eTrack. eTrack detected the event of SOPA (Stop Online Piracy Act) and Wikipedia on Jan 16, because on that day Wikipedia announced the blackout on Wednesday (Jan 18) to

6.7. Experiments

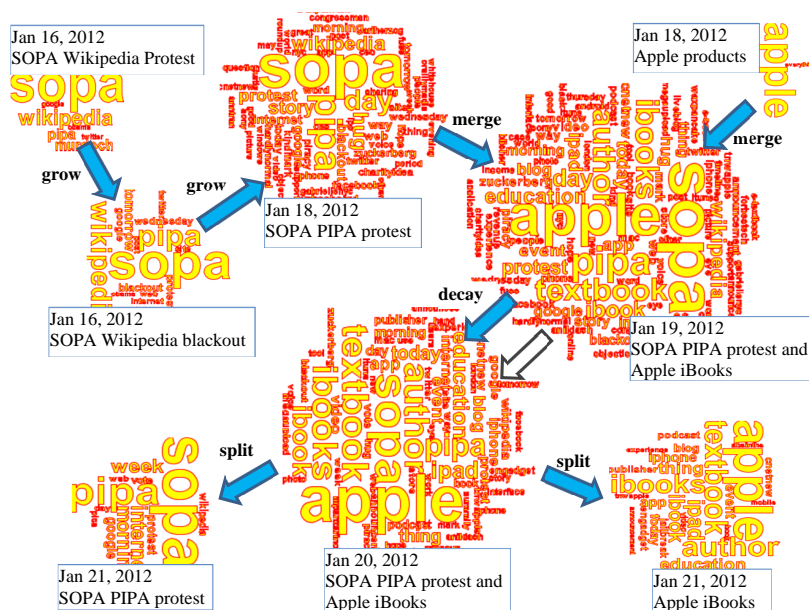


Figure 6.9: The merging and splitting of “SOPA” and “Apple”. At each moment, an event is annotated by a word cloud. Baselines 1 and 2 only works for the detection of new emerging events, and is not applicable for the tracking of merging and splitting dynamics. The evolution trajectories of and Baseline 3 are depicted by solid and hollow arrows respectively.

protest SOPA. This event grew distinctly on Jan 17 and Jan 18, by inducing more people in the social network to discuss about this topic. At the same time, there was another event detected on Jan 18, discussing Apple’s products. On Jan 19, actually the SOPA event and Apple event were merged, because Apple joined the SOPA protest and lots of Apple products such as iBooks in education are directly related to SOPA. This event evolved on Jan 20, by adding more discussions about iBooks 2. Apple iBooks 2 was actually unveiled in Jan 21, while this new product gained lots of attention, people who talked about iBooks did not talk about SOPA anymore. Thus, on Jan 21, the SOPA-Apple event was split into two events, which would evolve independently afterwards. Unfortunately, the above merging and splitting process cannot be tracked by any of the baselines, which output some independent events.

6.7. Experiments

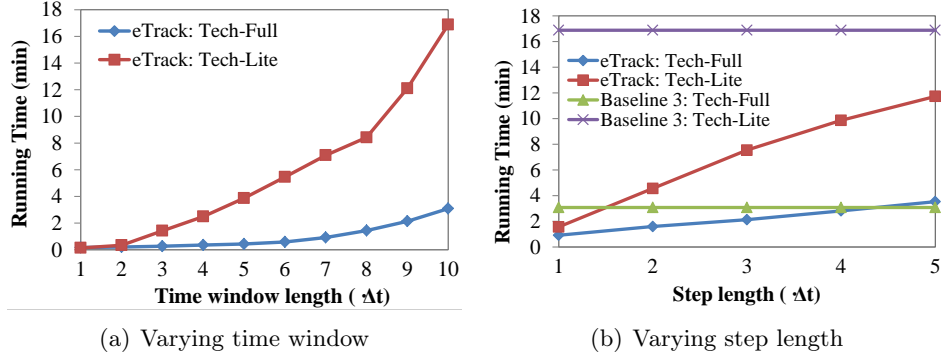


Figure 6.10: The running time on two datasets as the adjusting of the time window length and step length.

6.7.3 Running Time of Evolution Tracking

Remind that Baseline 1 and 2 are for event identification in a fixed time window. For evolution tracking, we measure how the Baseline 3 and eTrack scale w.r.t. both the varying time window width and the step length. We use both Tech-Lite and Tech-Full streams, and set the time step interval $\Delta t = 1$ day for Tech-Lite, $\Delta t = 1$ hour for Tech-Full to track events on different time granularity. The streaming post rates for Tech-Lite and Tech-Full are 11700/day and 7126/hour respectively. Figure 6.10(a) shows the running time of eTrack when we increase the time window length, and we can see for a time window of $10\Delta t$ hours in Tech-Full, our approach can finish the post preprocessing, post network construction and event tracking in just 3 minutes. A key observation is that the running time of eTrack does not depend on the overall size of the dataset. Rather, it depends on the streaming speed of posts in Δt . Thus, Tech-Lite takes more time than Tech-Full since its streaming posts in Δt is higher. Figure 6.10(b) shows if we fix the time window length as $10\Delta t$ and increase the step length of the sliding time window, the running time of eTrack grows nearly linearly. Compared with our incremental computation, Baseline 3 has to process posts in the whole time window from scratch at each moment, so the running time will be steadily high. If the step length is larger than $4\Delta t$ in TechFull, eTrack does not have an advantage in running time compared with Baseline 3, because a large part of post network is updated at each moment. However, this extreme case is rare. Since in a real scenario, the step length is much smaller than the time window length, our approach shows much better efficiency than the baseline approach.

6.8 Discussion and Conclusion

Our main goal is to track the evolution of events over social streams such as Twitter. To that end, we extract meaningful information from noisy post streams and organize it into an evolving network of posts under a sliding time window. We model events as sufficiently large clusters of posts sharing the same topics, and propose a framework to describe event evolution behaviors using a set of primitive operations. Unlike previous approaches, our evolution tracking algorithm performs incremental updates and efficiently tracks event evolution patterns in real time. We experimentally demonstrate the performance and quality of our algorithm over two real data sets crawled from Twitter. As a natural progression, in the future, it would be interesting to investigate the tracking of evolution of social emotions on products, with its obvious application for business intelligence.

Algorithm 8: ICM: Incremental Cluster Maintenance

Input: $S_t, S_{old}, S_{new}, S_-, S_+, S_\odot$
Output: S_{t+1}

```

1  $S_{t+1} = S_t;$ 
  // Delete  $S_{old} \cup S_-$ 
2 for each cluster  $C$  in  $S_{old} \cup S_- \cup S_\odot$  do
3    $\bar{C} = Ske(C);$ 
4    $N_c(\bar{C}) = \cup_{p \in \bar{C}} N_c(p);$ 
5   if  $|N_c(\bar{C})| = 0$  then
6     | remove cluster  $C$  from  $S_{t+1};$ 
7   else if  $|N_c(\bar{C})| = 1$  then
8     | delete  $C$  from cluster  $C'$  where  $C' \in N_c(\bar{C});$ 
9   else
10    | remove the cluster that  $C$  belongs to from  $S_{t+1};$ 
11    | for each cluster  $C' \in N_c(\bar{C})$  do
12      | assign a new cluster id for  $C';$ 
13      | add  $C'$  into  $S_{t+1};$ 
  // Add  $S_{new} \cup S_+$ 
14 for each cluster  $C$  in  $S_{new} \cup S_+$  do
15    $\bar{C} = Ske(C);$ 
16    $N_c(\bar{C}) = \cup_{p \in \bar{C}} N_c(p);$ 
17   if  $|N_c(\bar{C})| = 0$  then
18     | assign a new cluster id for  $C$  and add  $C$  to  $S_{t+1};$ 
19   else if  $|N_c(\bar{C})| = 1$  then
20     | add  $C$  into cluster  $C'$  where  $C' \in N_c(\bar{C});$ 
21   else
22     | assign a new cluster id for  $C;$ 
23     | for each cluster  $C' \in N_c(\bar{C})$  do
24       |  $C = C \cup C';$ 
25       | remove  $C'$  from  $S_{t+1};$ 
26     | add  $C$  into  $S_{t+1};$ 
27 return  $S_{t+1};$ 

```

Algorithm 9: eTrack: Cluster Evolution Tracking

Input: $G = \{G_i, G_{i+1}, \dots, G_j\}$, S_i
Output: Primitive cluster evolution operations

```

1 for  $t$  from  $i$  to  $j$  do
2   obtain  $S_{old}, S_{new}, S_-, S_+$  from  $G_{i+1} - G_i$ ;
3    $S_{t+1} = ICM(S_t, S_{old}, S_{new}, S_-, S_+, S_\odot)$ ;
4   for each cluster  $C \in S_{t+1}$  do
5      $id = ClusterId(C)$ ;
6     if  $S_t(id) \neq \emptyset$  then
7       output  $\downarrow (C, S_t(id) - S_{t+1}(id))$ ;
8       output  $\uparrow (C, S_{t+1}(id) - S_i(id))$ ;
9     else  $+C$ ;
10  for each cluster  $C \in S_i$  do
11     $id = ClusterId(C)$ ;
12    if  $S_{t+1}(id) = \emptyset$  then  $-C$ ;
```

(a) Results of different entity extraction approaches.

Methods	#edges	#coreposts	#coreedges	#events
Hashtags	182905	6232	28964	196
Unigrams	142468	15070	46783	430
POS-Tagger	357132	21509	47808	470

(b) Precision and recall of top 50 events.

Methods	Precision (major events)	Recall (major events)	Precision (G-Trends)
HashtagPeaks	0.40	0.30	0.25
UnigramPeaks	0.45	0.40	0.20
Louvain	0.60	0.55	0.75
eTrack	0.80	0.80	0.95

Table 6.2: Tuning post network.

Chapter 7

Crowdsourcing-Based User Study

User study is an effective way to determine the ground truth, or evaluate the correctness of a hypothesis. Traditionally, user studies are usually conducted in an “offline” mode, e.g., domain experts are employed to mark the truth, or students in a lab are invited to annotate images. With the rising of the Internet, crowdsourcing has recently become a popular mechanism behind user studies. Crowdsourcing is the process of obtaining needed services or content by soliciting contributions from a large group of online users. Compared with the traditional offline evaluation, crowdsourcing has clear advantages on engaging high number of users who are ready to work at flexible time for a fairly low price. However, crowdsourcing may easily fall into the low-quality user problem, since these users are typically not experts of crowdsourcing tasks, and they are motivated by earning money. The quality evaluation of users becomes a crucial problem in crowdsourcing based user study. In this chapter, we will analyze the problem and propose effective techniques like Expectation-Maximization with Qualification (EMQ) to conquer the challenges.

7.1 Introduction

In this thesis, we studied the cohesion, context and evolution problems of stories and events in unstructured social streams. In Chapter 1, we mentioned that the approaches we proposed in this thesis are based on two hypotheses: (1) When modeling social streams, users prefer correlated posts to individual posts; (2) To model stories/events in social streams, structural approach is better than frequency-based approach and LDA-based approach. These two fundamental hypotheses determine that we model social streams as post networks and use graph mining approaches to mine stories and events. In this section, we conduct crowdsourcing-based user study to verify these two important hypotheses in social stream mining. All crowdsourcing tasks are

implemented on Amazon Mechanical Turk (MTurk). Given the fact that a large number of workers on MTurk are bots and spammers, the most critical challenge is user quality control in crowdsourcing. In the related work section, we summarize existing techniques on user quality control, including majority voting, minimum time constraint, qualification test, etc. However, none of them can solve the “Smart Spammer” problem, in which workers pass the qualification test but perform like a spammer to get the reward with minimal work. To deal with the challenge, in this chapter, we propose Expectation-Maximization with Qualification (EMQ), which is capable of measuring user’s quality in crowdsourcing and detecting Smart Spammers among all qualified workers. As an iterative process, EMQ recursively updates the probability of a worker being a valuable worker until convergence, and we call this probability the quality score of a worker. Finally, for a crowdsourcing task composed by a question and several options, the answer of this question is the option obtaining the highest votes, where each vote is weighted by the quality score of a worker.

We organize this chapter as follows. In Section 7.2, we introduce existing techniques for worker quality control in crowdsourcing, and review their pros and cons respectively. In Section 7.3, the two hypotheses that support the modeling of social streams and stories/events in this thesis are introduced. Section 7.4 introduces the quality control workflow used in crowdsourcing tasks of this thesis. Especially, we introduce Expectation-Maximization with Qualification (EMQ), an advanced approach to evaluate the quality of workers. We show experiment results in Section 7.5.

7.2 Related Work

User (or worker) quality control is crucially important in guaranteeing the quality of submitted work in crowdsourcing. As the truth of each crowdsourcing task is either unavailable or very time-consuming to obtain and workers are primarily motivated by the reward, user quality control on crowdsourcing tasks is very challenging. For example, we may consider that a higher reward can lead to a higher quality of answers to this task. However, as pointed by [24], there is no clear correlation between the reward and the final quality. The reason is that increasing the price is believed to attract spammers (i.e., Turkers who cheat, not really performing the job, but using robots or answering randomly). In this section, the state-of-the-art techniques for quality control in crowdsourcing are summarized below.

Majority Voting. The traditional approach to improve the answering qual-

ity of a task is by assigning the same task to a large number of workers, and then doing a majority voting [60]. However, this approach is costly, as each worker needs to be paid.

Minimum Time Constraint. This technique sets a minimum test cost for a task, which forces the workers to read and think about the task for a time span higher than the minimum, e.g., 20 seconds, before making their decision. This action is supported by Amazon MTurk, and has proved to be effective to block bots and spammers [12]: bots will typically submit the work faster than any human, e.g., less than 1 second, and spammers usually do not read the instructions carefully and tend to make decisions faster than normal workers.

Approval Rate Constraint. As mentioned in [16], the Requester can require that all workers meet a particular qualification, such as sufficient accuracy on a small test set or a minimum percentage of previously accepted submissions. In [72], researchers only released HITs (Human Intelligence Tasks) to two groups of turkers: turkers with (1) the Master’s Qualification (a qualification awarded by Amazon) and (2) the default custom qualifications which requires the turkers to have completed at least 1000 HITs with a 95% approval rating. They reported that those turkers with high approval rating can achieve a quality as high as the quality achieved by skilled crowd, which are a group of well-trained graduate students. The disadvantages of this approach is it requires workers having long historical submission records to make the approval rate computed meaningfully.

Check and Reject. This approach does a manual check of the answers provided by the workers and rejects the work if the Requesters feel the submission is of low quality. Also, the Requester has the option of rejecting the work of individual workers, in which case these workers are not paid. However, the manual check of all answers is time-consuming and needs lots of human labor.

Qualification Test. Qualification test is a widely used technique for user quality control in crowdsourcing. For instance, Yashar [58] designed some gold units as a qualification test for the bilingual translation. That is, they provide an English sentence and a Spanish sentence, and then ask turkers yes/no on whether there is a translation between them. As an effective approach, qualification test is widely used in crowdsourcing for the quality control of workers. The downside of this approach is workers need to spend considerable amount of time to complete the qualification test before the actual work.

Expectation-Maximization. Aditya et. al [60] mentioned that the Expectation Maximization algorithm can be used to estimate worker quality. These algorithms collect annotations from humans, and do disagreement-based analysis to deduce the true answers. Ipeirotis et. al. [36] discussed the quality management on Amazon Mechanical Turk. They proposed a solution based on Expectation-Maximization: the algorithm iterates until convergence, following two steps: (1) estimate the correct answer for each task, using labels assigned by multiple workers, accounting for the quality of each worker; and (2) estimate the quality of the workers by comparing their submitted answers to the inferred correct answers. However, the output of EM method changes with the selection of initial seeds, and a bad selection of seeds may produce low-quality results.

Hybrid Approach. By working with Google, Ipeirotis et. al. [35] described Quiz, a gamified crowdsourcing system that simultaneously assesses the knowledge of users and acquires new knowledge from them. Quiz operates by asking users to complete short quizzes on specific topics; as a user answers the quiz questions, Quiz estimates the user’s competence. To acquire new knowledge, Quiz also incorporates questions for which we do not have a known answer; the answers given by competent users provide useful signals for selecting the correct answers for these questions. Their experiments involve over ten thousand users and confirm that Quiz can automatically identify users with the desired expertise and interest in the given topic, with cost below that of hiring workers through paid-crowdsourcing platforms. The downside of Quiz is it may fall into the “smart spammer” problem, where workers perform competent initially but then behave like spammers by giving random answers, because they want to get the reward as quickly as possible.

7.3 Hypotheses in Social Stream Mining

As defined in Chapter 1, in social streams, an event is a set of related stories, and a story is a set of similar posts with high cohesion. The goal of social stream mining is to detect and track stories and events from social streams. To help achieve the goal, we have two fundamental hypotheses for social stream mining in this thesis.

Hypothesis 1 *When modeling social streams, users prefer models in the form of correlated posts to models in the form of individual posts.*

7.3. Hypotheses in Social Stream Mining

1. If you want to know new emerging stories regarding “MH370”, which search result out of these two providing content you consider is easier to understand? Vote for the best option.

Result 1

Result 2

- Option A: Result 1 is much better than Result 2;
- Option D: Result 2 is slightly better than Result 1;
- Option B: Result 1 is slightly better than Result 2;
- Option E: Result 2 is much better than Result 1.
- Option C: They have no difference;

Figure 7.1: Crowdsourcing task for Hypothesis 1.

With Hypothesis 1, we can model a social stream as a post network based on their correlations. Hypothesis 1 is fundamental for the modeling of unstructured social streams in this thesis, since correlated posts provide a better user experience than individual posts. All the data mining techniques proposed in this thesis are essentially based on Hypothesis 1, since these techniques are sophisticated graph mining algorithms which model a collection of posts as a post network.

In Figure 7.1, we show our proposed user study for Hypothesis 1. Supposing the search is for “MH370”, we show result 1 and result 2 to Amazon workers. Result 1 lists tweets containing MH370 one-by-one, ranked by freshness. Result 2 lists grouped tweets, e.g., “MH370 Search Updates”, “MH370 Causes of Disappearance”, etc., with each group telling one event related to MH370. We then provide the following five options:

- Option A: Result 1 is much better than Result 2;
- Option B: Result 1 is slightly better than Result 2;
- Option C: They have no difference;
- Option D: Result 2 is slightly better than Result 1;
- Option E: Result 2 is much better than Result 1.

The crowdsourcing task for testing Hypothesis 1 is asking Amazon workers to answer the question by choosing an option. To be fair, we do not

7.3. Hypotheses in Social Stream Mining

2. Following options try to illustrate the top events detected from tweets in a short time span. Which option you consider that illustrates top events in the best way?

#CES
#SOPA
#EngadgetCES
#opengov
#gov20
#CES2012
#PIPA
#opendata
#StartupAmerica
#win7tech

google
ces
apple
video
sopa
twitter
year
facebook
app
iphone

blog post, great buy in january, start sign
report amazon kindle fire
check real youtube interview video series
win chance south beach trip
small business company story, local community
make great start, things learn
bad thing in life, feel good
send email hey, glad to hear, follow tweet
jan travel, days ago, back home tonight
president obama, iran war killed

Result 1

Result 2

Result 3

Result 4

- Option A: Result 1 is the best;
- Option B: Result 2 is the best;

- Option C: Result 3 is the best;
- Option D: Result 4 is the best.
- Option E: They have no difference.

Figure 7.2: Crowdsourcing task for Hypothesis 2.

inform Amazon workers which kind of algorithms we use to generate individual and correlated results.

Hypothesis 2 *To model stories/events in social streams, structural approach is better than frequency-based approach and LDA-based approach.*

Hypothesis 2 is fundamental to the cohesion, context and evolution problems studied in this thesis. If stories/events are not defined in a structural way, it will be extremely hard to define the notions like story cohesion, story context and event evolution. With Hypothesis 2, we can model a story/event as a dense subgraph inside the post network, and subsequently, the cohesion, context and evolution problems can be defined and studied from the graph mining perspective.

In Figure 7.2, we show our crowdsourcing task for Hypothesis 2. We provide five options, which correspond to results generated by four different story detection approaches:

- Option A: (frequency-based) top hashtags
- Option B: (frequency-based) top entities
- Option C: (LDA-based) LDA approaches

- Option D: (structural) Cluster method
- Option E: They have no difference.

We then ask Amazon workers' preference for the best result, by selecting one of the results. Notice that for the sake of fairness, we do not show the actual approach name in the crowdsourcing task, so the worker who votes for an option has no idea on the algorithmic principle behind this option.

7.4 Quality Control for Crowdsourcing

Distinguishing Workers. Amazon Mechanical Turk (MTurk) is the most popular crowdsourcing Internet marketplace. As reported in January 2011, there are over 500,000 workers from over 190 countries. Besides the normal MTurk workers, it is a well-known fact that a large number of Amazon workers are actually spammers and bots. In this user study, we distinguish the following four different categories of workers:

- **Bots.** A bot is an automatic program that virtually attempts to answer the task. Bots are designed to get the reward, and a bot usually returns the answers within a very short time, typically not realistic for an average human.
- **Spammers.** A spammer is a real worker who provides nonsensical answers. They are real persons and their sole target is to get the reward, without reading the instructions carefully and doing the necessary thinking or work.
- **Unqualified Workers.** An unqualified worker is a real worker who cannot pass the qualification test. In the case of social stream mining, unqualified workers include the workers without comprehension ability: they either cannot understand the instructions very well, or fail to make a rational decision. It is worth noting that an unqualified worker for task A may be a qualified worker for task B.
- **Qualified Workers.** A qualified worker is a real worker who passes the qualification test. Depending on the rule of the qualification test, a qualified worker may fail some qualification questions, given the assumption that the portion of correctly answered questions is larger than a pre-defined threshold, e.g., 0.6. User quality management in crowdsourcing tasks aims to find a sufficiently high number of qualified workers.

Typically, the minimum time constraint can make a distinction between a bot and a real person, because a bot usually returns the answers within a

very short time which is unrealistic for human. However, it is possible that a bot is designed to allow for a minimum elapsed time before it responds to a task. In this case, the historical approval rate constraint can easily filter out bots and spammers, since they usually do random guess and have a very low approval rate. Clearly, a bot and spammer cannot pass the qualification test. A qualification test is designed to distinguish the qualified workers and unqualified workers. Depending on the effectiveness of qualification test, all unqualified workers are rejected and are not allowed to submit the work for the designed crowdsourcing tasks.

Smart Spammers. In this user study, we use the term “Smart Spammers” to refer to a category of qualified workers, who pass the qualification test, but make nonsensical submissions to a part of the subsequent crowdsourcing tasks. Smart Spammers prove their ability to answer the qualifying task, and thereafter the motivation of their behaviors resembles that of a spammer who focuses simply on getting the reward with very little actual work. Notice that a smart spammer may perform like a spammer only in a part of crowdsourcing tasks, but not in all of tasks. Thus, a qualified worker can be either a smart spammer, or a “valuable worker” who really contributes to the crowdsourcing tasks. The detection of the smart spammers is a challenging problem. In this user study, we propose Expectation-Maximization with Qualification (EMQ), an iterative approach to measure worker’s quality by combining workers’ performance in the qualification test and subsequent crowdsourcing tasks. EMQ approach is designed to detect the abuse of the system, by assigning low quality scores to those workers with a high probability of being smart spammers.

Quality Control Workflow. To guarantee maximum effectiveness, we use multiple techniques to control the quality of workers in this user study. The workflow of the quality control is shown in Figure 7.3. In the beginning, we set the Approval Rate Constraint by a sufficiently high threshold to filter out MTurk workers who have a very low historical approval rate. For the remaining workers, we perform the qualification test, which is a series of questions with known answers. These qualification questions will be treated as the golden standard and workers’ qualification will be measured in terms of the ratio of questions answered correctly. If the ratio is higher than a predefined threshold, this worker will be treated as a qualified worker. After this step, nearly all bots, spammers and unqualified workers will be blocked out before the start of real crowdsourcing tasks. All qualified workers will submit their work on the real crowdsourcing tasks. Since there are no predefined answers for crowdsourcing tasks, the quality of workers will be measured

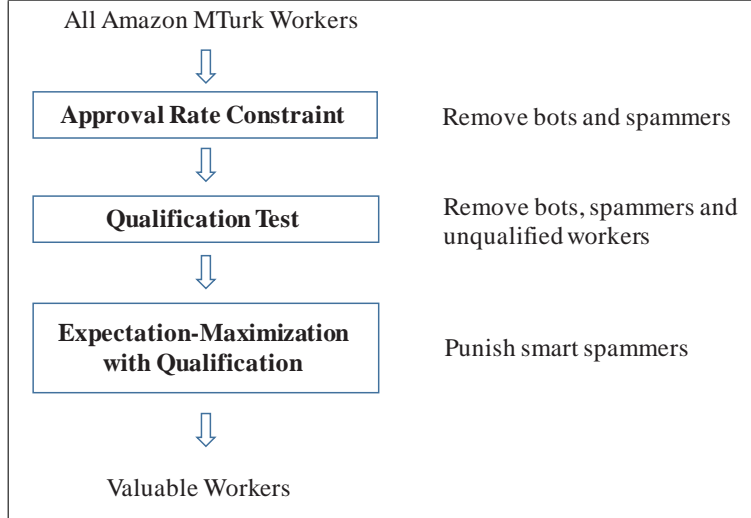


Figure 7.3: The workflow of quality control steps.

by cross-comparison with peers in an iterative way, which is captured by Expectation-Maximization with Qualification (EMQ). In the case of smart spammers who passed the qualification test but submitted random answers to crowdsourcing tasks, their quality scores will be lowered down iteratively due to the fact these random answers deviate from the majority voting distinctly. Thus, EMQ is capable of measuring user’s quality in crowdsourcing and punishing smart spammers from among all qualified workers, by assigning low quality scores to them. We will discuss the EMQ approach in detail in next paragraph.

Expectation-Maximization with Qualification (EMQ). The EMQ approach measures user’s quality in crowdsourcing iteratively. To formalize EMQ, let q denote the quality vector, where q_i denotes the quality score for worker i on each iteration. Assuming there are n workers, we have

$$q = [q_1 \quad q_2 \quad q_3 \quad \cdots \quad q_n] \quad (7.1)$$

Next, we assume V is the voting matrix between workers and questions, where each element V_{ij} is a 0/1 vector describing worker i ’s vote on options of question j . As an example, $V_{12} = [0 \ 0 \ 1 \ 0 \ 0]$ means worker 1 chooses the 3rd option of question 2. Assuming there are m questions, we get

$$V = \begin{bmatrix} V_{11} & V_{12} & \cdots & V_{1m} \\ V_{21} & V_{22} & \cdots & V_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ V_{n1} & V_{n2} & \cdots & V_{nm} \end{bmatrix} \quad (7.2)$$

We also assume D is the distribution of weights on each option for each question, and each question has l options. Thus, D is a $m \times l$ matrix, and for each row D_{j*} , its elements sum up to 1, i.e., $\sum_k D_{jk} = 1$.

During the EMQ process, V will be fixed, q and D will be updated on each iteration. To initialize q , since EMQ uses user's score in qualification test as their initial quality, we set q_i as the percentage of questions answered correctly by worker i in the qualification test, and then normalize q by $q = q/\|q\|_1$. On each iteration, EMQ works as follows:

- Step 1:
 - Computation: $D_{j*} = \sum_{i=1}^n q_i V_{ij}$;
 - Normalization: $D_{j*} = D_{j*}/\|D_{j*}\|_1$;
- Step 2:
 - Computation: $q_i = \sum_{j=1}^m D_{j*} V_{ij}^T$;
 - Normalization: $q = q/\|q\|_1$;

To explain, in Step 1, we treat q_i as the weight of worker i with which to grade each answer provided by that worker in crowdsourcing tasks, and each question will have a distribution of weighted votes on its options, denoted by D_{j*} . In Step 2, we re-calculate the weight of each worker, by cross-comparison between his voting on question j and the current distribution of votes on options of question i . In each iteration, we normalize option weight distribution vectors and quality vector, where $\|v\|_1$ means the normalization by l^1 -norm, i.e., $\|v\|_1 = \sum_i |v_i|$.

The major steps of EMQ are shown in Algorithm 10. Recall that while the correct answer for each qualification question is predefined, we do not know the correct answer for crowdsourcing tasks. Basically, EMQ computes the distribution of answers for each question based on workers' quality vector and voting matrix, and workers' quality vector will be evaluated again based on the distribution matrix between questions and answers. This process will continue, until the quality vector q converges, with l^1 -norm of $(q^o - q)$ less than a small given threshold δ , or the maximum iteration number K is reached. We empirically set $\delta = 0.0001$.

We use the example in Figure 7.4 to illustrate the idea of EMQ algo-

Algorithm 10: Expectation-Maximization with Qualification (EMQ)

Input: Users' answers in qualification test, users' voting matrix V in crowdsourcing tasks, qualification threshold τ , convergence threshold δ , max iteration K

Output: User quality vector q

```

1 for each worker  $i$  do
2   compute  $q_i$ , the percentage of questions answered correctly in the
   qualification test;
3   if  $q_i < \tau$  then
4     mark worker  $i$  as unqualified and remove;
5 set  $q$  as the qualification score vector of qualified workers;
6 set  $k = 0$ ;
7  $q = q / \|q\|_1$ ;
8 while  $k < K$  do
9    $q^o = q$ ;
10   $D_{j*} = \sum_{i=1}^n q_i V_{ij}$ ;
11   $D_{j*} = D_{j*} / \|D_{j*}\|_1$ ;
12   $q_i = \sum_{j=1}^m D_{j*} V_{ij}^T$ ;
13   $q = q / \|q\|_1$ ;
14  if  $\|q^o - q\|_1 < \delta$  then
15    return  $q$ ;
16   $k = k + 1$ ;
17 return  $q$ ;

```

arithm. Suppose that there is a voting example with 3 workers and 3 questions, where each question has two options A and B, as shown in Figure 7.4(a). Figure 7.4(b) assumes that workers' initial quality score vector obtained from the qualification test is $[0.6, 0.8, 1.0]$, which is $[0.2500 \ 0.3333 \ 0.4167]$ after the normalization in initialization. In the first iteration, we compute the distribution of aggregated quality scores between options for each question, as shown in Figure 7.4(c). In detail, for question Q1, we compute $0.2500 + 0.4167 = 0.6667$ for option A and 0.3333 for option B, and similarly for question Q2 and Q3. In turn, to update the quality scores, we compare the weight distribution of options for each question and worker's vote for that question. For example, worker W1 answered A for Q1 and Q2, B for Q3, so W1 will get a total score of $0.6667 + 0.5833 + 0.5833 = 1.8333$. Analogously, workers W2 and W3 get 1.4999 and 1.5001 respectively. After the normalization, we get the quality score vector $[0.3793 \ 0.3103 \ 0.3103]$ in iteration 1, which will be used as the input for iteration 2. This iterative

7.4. Quality Control for Crowdsourcing

Vote	Q1	Q2	Q3
W1	A	A	B
W2	B	A	B
W3	A	B	A

(a)

Iteration 1		
	A	B
Q1	0.6667	0.3333
Q2	0.5833	0.4167
Q3	0.4167	0.5833

(c)

Iteration	W1	W2	W3
0	0.2500	0.3333	0.4167
1	0.3793	0.3103	0.3103
2	0.4082	0.3333	0.2585
3	0.4174	0.3527	0.2299
...
15	0.4270	0.3819	0.1910

(b)

Iteration 15		
	A	B
Q1	0.6181	0.3819
Q2	0.8089	0.1911
Q3	0.1911	0.8089

(d)

Figure 7.4: (a) A voting example with 3 workers and 3 questions, where each question has two options: A and B. (b) The computation of normalized quality vector q on each iteration, where quality scores on iteration 0 are obtained from the qualification test. (c) and (d): The distributions of quality weights among options for each question on iteration 1 and 15, respectively.

process continues and reaches the convergence on the 15th iteration. As we can see in Figure 7.4(b), while W3 has the highest initial quality score, the final quality score of W3 is low, indicating that W3 may be a smart spammer who did well in the qualification test, but provided random answers for crowdsourcing tasks to get the reward. Thus, EMQ is capable to catch smart spammers because their quality scores become very low after many iterations of punishments, even though these smart spammers may get very high initial quality scores in the qualification test.

It is well-known that the Expectation-Maximization (EM) algorithm is a hill-climbing approach, and it can only be guaranteed to reach a local maximum. When there are multiple maximas, whether we will actually reach the global maximum depends on where we start. Clearly, the selection of starting seeds impacts the final user quality upon convergence. Existing user studies [36, 60] based on EM typically set the seed by a uniform distribution, i.e., elements in Q_1 are equal, or by a unreliable random guess. In contrast, we consider that worker's performance in the qualification test serves as a good starting point for EM. EMQ uses the scores obtained from the qualification test as the prior information, which is an ideal seed to measure worker's task-specific quality in crowdsourcing.

7.5 Experiments

In this section, we perform crowdsourcing-based user studies on Amazon MTurk. We employed a total of 945 workers with historical approval rate higher than 85%, out of which 890 workers passed the qualification test. Each qualified worker will work on two crowdsourcing tasks, as shown in Figure 7.1 and 7.2 respectively.

Qualification Test Design. Our tasks expect that the users have some comprehension ability, i.e., given a set of tweets, they can figure out what these tweets are talking about. Thus, a qualification test can be done by providing multiple summarization phrases as options, and then asking turkers to select the best summarization phrase capturing the given tweet. There will be only one correct answer for each qualification question. To avoid the situation that the qualification question becomes naive, we ensure that the summarization phrase itself does not occur in the tweets themselves. For example, given 5 tweets talking about an event “oil price drop” but not necessarily mentioning the phrase “oil price drop”, and then provide users a list of 3 options, e.g., “iphone 6 price”, “oil filter change” and “oil price drop”. If a worker does not select “oil price drop”, she will fail in this question. Very likely, it is a bot or a worker with very low comprehension ability. We assess each worker’s performance in the qualification test by a score, which is computed by the portion of qualification questions answered correct, e.g., if a worker gets correct on four qualification questions out all the five, the qualification score of this worker will be 0.8. In experiments, we set the qualification threshold τ as 0.5 (shown in Algorithm 10), with the intuition that any qualified workers should get correct on at least a half of all qualification questions. Workers with qualification scores lower than 0.5 are considered unqualified and will be rejected to participate in the subsequent crowdsourcing tasks.

We show our qualification test sample used before real crowdsourcing tasks in Figure 7.5. There are a total of five questions and we provide three options for each question. Workers who answered at least three questions correctly will be selected as qualified workers. In our experiment, we have a total of 945 Amazon workers who participated the qualification test, out of which 890 workers passed the test with qualification scores higher than 0.5. Thus, the pass rate is 94.2% and these 890 qualified workers will proceed to work on real crowdsourcing tasks.

EMQ Implementation. All qualified workers are eligible to participate the subsequent crowdsourcing tasks, where the true answer is unknown. We

7.5. Experiments

Qualification Test: select the best summarization phrase capturing the given tweet






No.	Tweets	Summarization Options
1.	 Barack Obama News @ObamaNews · 18h Check out our 2015 year in photos where you'll find powerful moments from the 50th anniversary of the marches from...	A. 2015 moments B. Photoshop C. Powerful women
2.	 Andrew Said @AndrewSaidLA · 3h @bikinginla Yeah asking if a kid run over by a truck was wearing a helmet is like asking if the passengers of #MH370 were wearing seatbelts	A. Childhood life B. MH370 crash C. Truck crash
3.	 CreareMarketing @CreareMarketing · 6h CES 2015: Will new tech deliver the data mother lode? buff.ly/1lmswiA #MarketingTips #Conversions #SmallB...	A. Mother and father B. Database center C. New tech
4.	 TVE7.COM @TVE7COM · 15h Australia : MH370 "hard spots" found in the Indian Ocean Breaking News 5th Sep 2014 - tve7.com/australia-mh37...	A. MH370 new found B. Indian people C. Australia cities
5.	 Mike Chillit @MikeChillit · May 27 #MH370 This didn't come up as a possibility in March 2014. Not sure which engines were on Korean Air.	A. Korean music B. MH370 engine C. New possibility

Figure 7.5: Qualification test sample used before real crowdsourcing tasks.

use EMQ to measure the credibility of each qualified worker's answers to these crowdsourcing tasks. To implement EMQ, we set the initial seed of user quality scores as the qualification scores. By following the steps in Algorithm 10, EMQ iterates and will be able to generate a convergent quality score for each qualified user, which in turn results in a convergent voting score weighted by user quality scores for each crowdsourcing task.

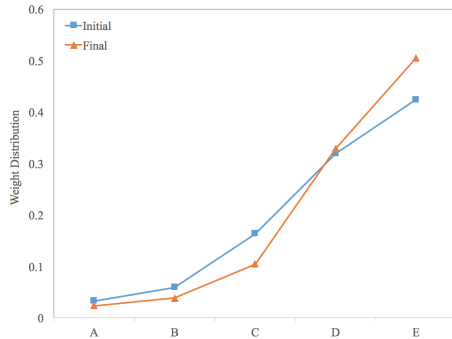
Hypothesis Verification Using EMQ. Each qualified worker will work on two crowdsourcing tasks, as shown in Figure 7.1 and 7.2, which correspond to Hypothesis 1 and 2 (denoted by H1 and H2 for short) respectively. In a neutral manner, to claim Hypothesis 1 and 2 to be true, we require that the majority of qualified workers vote for options D and E in H1 and option D in H2, where each vote is weighted by the quality score. Recall that in Section 7.3, options D and E in H1 indicate users prefer correlated posts than individual posts in social stream search, and option D in H2 means structural approach is the best in social stream modeling. We empirically set 60% as the threshold to claim "majority voting", e.g., if more than 60% of votes choose option D in H2, then we say Hypothesis 2 is true.

Based on the results returned by EMQ, we compute a weighted voting sum (WVS) score for each option in each hypothesis, where WVS of an option is the sum of workers' quality scores who voted for this option. Both Hypothesis 1 and 2 will be validated by these WVS scores. In Hypothesis 1, options D and E correspond to the situation that Result 2 is better than Re-

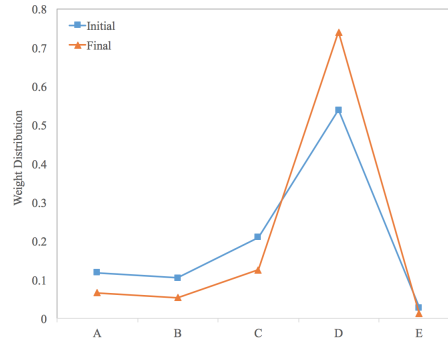
7.5. Experiments

Vote	W1	W2	W3	...	W890
H1 (A/B/C/D/E)	A	A	B	...	D
H2 (A/B/C/D/E)	C	D	E	...	A

(a) Workers' voting table on Hypothesis 1 and 2



(b) EMQ for Hypothesis 1



(c) EMQ for Hypothesis 2

Figure 7.6: Running EMQ for verifying Hypothesis 1 and 2. In (b), the percentage of weighted votes on options D/E increase from 74.4% (before EMQ) to 83.4% (after EMQ). In (c), the percentage of weighted votes on options D increase from 56.7% (before EMQ) to 75.4% (after EMQ).

sult 1. In Hypothesis 2, if result D has higher WVS scores than A, B, C and D, we say structural approaches are better than frequency-based approaches and content approaches, and this hypothesis is true.

We show the results of running EMQ for verifying Hypothesis 1 and 2 in Figure 7.6. Workers' original answers on Hypothesis 1 and 2 are shown in Figure 7.6(a). There are a total of 890 workers who participated in these crowdsourcing tasks, and EMQ takes 9 iterations to converge. H1 has five options: A, B, C, D and E. In Figure 7.6(b), we show the initial distribution of WVS scores before running of EMQ and the final distribution of WVS scores after running EMQ. As we can see, the final distribution is more skewed than the initial distribution, which indicates EMQ iteratively strengthens the WVS scores of options voted by high quality workers, and weakens the WVS scores of options voted by low quality workers. In the end, options D and E collect 32.9% and 50.5% of weighted votes respectively. Recalling that option D means "result 2 is slightly better than result 1" and option E means "result 2 is much better than result 1", we conclude that 83.4% of weighted votes agree that result 2 (in the form of correlated posts) is better than result 1 (in the form of individual posts).

Figure 7.6(c) shows the initial and final distributions of WVS scores for option A, B, C, D and E in Hypothesis 2. As we can see, EMQ procedure lowers down the WVS scores for option A, B and C, while the WVS score for option D is improved. Recall that option A, B are frequency-based approaches, C are LDA-based approaches and D is structural approach. Since option D gets 75.4% of weighted votes finally, we conclude that structural approach is better than other approaches.

In conclusion, Figure 7.6(b) and 7.6(c) show that the majority of high quality workers agree that correlated posts are better than individual posts in social stream modeling, and structural approach is better than other approaches in story/event modeling in social streams.

Detection of Smart Spammers by EMQ. Smart spammers are those workers who passed the qualification test but performed like a spammer in real crowdsourcing tasks. Smart spammers are very difficult to detect by existing techniques. EMQ method is capable to detect smart spammers because their quality scores become very low after many iterations of punishments. It is normal for some workers to have relatively lower final quality scores than their initial quality scores, but if their final quality scores become extremely low, these workers could be smart spammers. We empirically define smart spammers as workers whose initial quality scores are at least five times higher than their final quality scores. Following this definition, we find 44 smart spammers whose convergent EMQ quality scores are five times lower than their initial quality scores. In other words, 4.94% qualified workers are actually smart spammers, which is a portion that cannot be easily ignored in the quality evaluation. In contrast, existing methods cannot detect smart spammers effectively.

7.6 Discussion and Conclusion

The crowdsourcing based hypothesis verification proposed in this chapter has several limitations:

- Crowdsourcing tasks are designed by a few domain experts, which may lose generality and introduce bias that these tasks are not perfectly designed in both content and form to verify the hypothesis. For example, the current crowdsourcing task uses “MH370” event as an example, but maybe this event is not well-known for every worker, or even workers know about MH370 event, their responses are likely to be biased in how this event is presented in crowdsourcing tasks.

- The visualization in crowdsourcing tasks may bring new bias. For example, to verify Hypothesis 2, we use circles with texts and links to visualize structural approaches, but the visualization itself may bring different understanding overhead for workers with different educational levels.
- Due to the diversity of human behaviors in crowdsourcing, the smart spammer detection algorithm by EMQ may be not effective in all situations. For example, some qualified workers may work normally in the first half of tasks, but later they perform like spammers because of some reasons, e.g., being tired or wanting to get the reward quickly.

There is room for the improvement. First, we can employ a larger number of domain experts to design more tasks with different forms and topics, which will reduce the bias introduced by the tasks themselves. Second, better algorithms may be developed to combat smart spammers. Third, we can randomly mix the qualification questions and actual tasks, making it hard for the workers to distinguish them, so that the qualification questions and actual tasks have equal chances to be spammed by workers. This will make it more challenging for smart spammers to pass the qualification test.

Chapter 8

Summary and Future Research

8.1 Summary

In the current social web age, mining unstructured social streams has become a fundamental requirement to satisfy people’s needs in information seeking. Existing user experience on social stream applications like Twitter Search may easily lead to “information anxiety”, in which users input several keywords and the output will be a long list of tweets or posts with keywords contained, ranked by time freshness. Since a post like a tweet or a Facebook update only contains a small piece of information, users are required to digest a long list of search results, which is time-consuming and painful. The noisy and redundant nature of social streams degrades user’s experience further. The social stream mining that we propose aims at providing users with an organized and summarized view of what’s happening in their social world. In this thesis, we examine several important applications in social stream mining, and propose efficient solutions based on graph mining techniques.

Chapter 3 discussed the modeling of unstructured social streams. In detail, we model a social stream as an evolving network of posts. Stories and events are modeled as two special kinds of subgraphs. Specifically, a story is a dense subgraph of posts with high cohesion in each snapshot of a post network, while an event is a cluster of posts across consecutive snapshots as the post network evolves.

In Chapter 4, we instantiate a story as a quasi-clique, and given a query node set S in a graph $G(V, E)$, we try to solve the maximum quasi-clique search problem, which is formalized as finding the largest λ -quasi-clique containing S . To quickly locate the initial solution, we propose k -Core tree by recursively organizing dense subgraphs in G . Three maximization operations are introduced to optimize the solution: *Add*, *Remove* and *Swap*. Then, we propose two iterative maximization algorithms, DIM and SUM, to approach the maximum quasi-clique that contains the given query node set S using deterministic and stochastic approaches respectively.

In Chapter 5, we focus on two problems: (1) efficiently identify transient stories from fast streaming social content; (2) perform iceberg query

to build the structural context between stories. To solve the first problem, we transform social stream in a time window to a capillary network, and model transient stories as (k, d) -Cores in the capillary network. Two efficient algorithms are proposed to extract maximal (k, d) -Cores. For the second problem, we propose deterministic context search and randomized context search to support the iceberg query, which allows to perform context search without pairwise comparison. We perform detailed experimental study on real Twitter streams and the results demonstrate the effectiveness and value of our proposed context-aware story-teller CAST.

Our main goal in Chapter 6 is to track the event evolution patterns from highly dynamic post networks. To that end, we summarize the network by a skeletal graph and monitor the updates to the post network by means of a sliding time window. Then, we design a set of primitive operations and express the cluster evolution patterns using these operations. Unlike previous approaches, our evolution tracking algorithm eTrack performs incremental bulk updates in real time. We deploy our approach on the event evolution tracking task in social streams, and experimentally demonstrate the performance and quality on two real data sets crawled from Twitter.

Finally, Chapter 7 performs crowdsourcing based user studies to validate two important hypotheses. These two hypotheses are fundamental to our mining tasks on social streams. The key problem of the crowdsourcing based user studies is the quality evaluation of workers. We distinguish Amazon MTurk workers into four types: bots, spammers, unqualified workers and qualified workers. While bots and spammers can be detected by traditional techniques like minimum time constraint and approval rate constraint, carefully designed qualification test is required to distinguish unqualified and qualified workers. We discussed detailed qualification test for crowdsourcing tasks in the preceding sections. As a special kind of qualified workers, smart spammers are workers who passed the qualification test but sometimes perform like a spammer in the subsequent crowdsourcing tasks. The detection of smart spammers is a challenging and unsolved problem. In this chapter, we use Expectation-Maximization with Qualification (EMQ), which is capable of measuring user's quality in crowdsourcing and detecting Smart Spammers from among all qualified workers.

8.2 Future Research

This thesis has made substantial progress in the study of cohesion, context and evolution problems in unstructured social stream mining. In future

8.2. Future Research

research, there are still several open opportunities and challenges in social stream mining, as listed below.

- It would be interesting to investigate the incremental evolution of social emotions and sentiments for business intelligence, associated with the evolution of events.
- We look forward to performing advanced analytics on stories and events, such as the spread paths of rumors on social networks, personalized recommendation of new events with GPS signals, etc.
- We are interested in various novel visualization techniques to present transient stories and incremental updating events to end users in a friendly way.

Bibliography

- [1] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.
- [2] Manoj K. Agarwal, Krithi Ramamritham, and Manish Bhide. Real time discovery of dense clusters in highly dynamic graphs: Identifying real world events in highly dynamic environments. *PVLDB*, 5(10):980–991, 2012.
- [3] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.
- [4] James Allan, editor. *Topic detection and tracking: event-based information organization*. Kluwer Academic Publishers, 2002.
- [5] Albert Angel, Nick Koudas, Nikos Sarkas, and Divesh Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *PVLDB*, 5(6):574–585, 2012.
- [6] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, 1998.
- [7] Yuichi Asahiro, Refael Hassin, and Kazuo Iwama. Complexity of finding dense subgraphs. *Discrete Applied Mathematics*, 121(1-3):15–26, 2002.
- [8] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. Clique relaxations in social network analysis: The maximum k -plex problem. *Operations Research*, 59(1):133–142, 2011.
- [9] Roberto Battiti and Marco Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.
- [10] Hila Becker, Mor Naaman, and Luis Gravano. Learning similarity metrics for event identification in social media. In *WSDM*, pages 291–300, 2010.

- [11] Pavel Berkhin. Survey: A survey on pagerank computing. *Internet Mathematics*, 2(1):73–120, 2005.
- [12] Seth Bicknell. How to successfully use amazon’s mechanical turk. Retrieved in January 2015.
- [13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- [14] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech.*, 2008.
- [15] Mauro Brunato, Holger H. Hoos, and Roberto Battiti. On effectively finding maximal quasi-cliques in graphs. In *LION*, pages 41–55, 2007.
- [16] Chris Callison-Burch. Fast, cheap, and creative: Evaluating translation quality using amazon’s mechanical turk. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP 2009, 6-7 August 2009, Singapore, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 286–295, 2009.
- [17] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, 2006.
- [18] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *KDD*, pages 133–142, 2007.
- [19] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. Online search of overlapping communities. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD ’13*, pages 277–288, New York, NY, USA, 2013. ACM.
- [20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.
- [21] David Nadeau et. al. A survey of named entity recognition and classification. *Linguisticae Investigationes*, 30(1):3–26, 2007.
- [22] Li Wan et.al. Density-based clustering of data streams at multiple resolutions. *TKDD*, 3(3), 2009.

- [23] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. In *Foundations of Computer Science*, pages 2–12, 1991.
- [24] Kar en Fort, Gilles Adda, and K. Bretonnel Cohen. Amazon mechanical turk: Gold mine or coal mine? *Computational Linguistics*, 37(2):413–420, 2011.
- [25] Santo Fortunato. Community detection in graphs. *CoRR*, abs/0906.0612, 2009.
- [26] Gabriel Pui Cheong Fung, Jeffrey Xu Yu, Philip S. Yu, and Hongjun Lu. Parameter free bursty events detection in text streams. In *VLDB*, pages 181–192, 2005.
- [27] Zekai Gao, Yangqiu Song, Shixia Liu, Haixun Wang, Hao Wei, Yang Chen, and Weiwei Cui. Tracking and connecting topics via incremental hierarchical dirichlet processes. In *ICDM*, pages 1056–1061, 2011.
- [28] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *ASONAM 2011, Kaohsiung, Taiwan, 25-27 July 2011*, pages 87–93, 2011.
- [29] Martin Halvey and Mark T. Keane. An assessment of tag presentation techniques. In *WWW*, pages 1313–1314, 2007.
- [30] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2011.
- [31] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.
- [32] Yulan He, Chenghua Lin, Wei Gao, and Kam-Fai Wong. Tracking sentiment and topic dynamics from social media. In *ICWSM*, 2012.
- [33] Holger H. Hoos and Thomas Stutzle. *Stochastic local search: Foundations & applications*. Morgan Kaufmann, 2004.
- [34] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1311–1322, 2014.

- [35] Panagiotis G. Ipeirotis and Evgeniy Gabrilovich. Quizz: targeted crowd-sourcing with a billion (potential) users. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 143–154, 2014.
- [36] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP '10*, pages 64–67, New York, NY, USA, 2010. ACM.
- [37] Xin Jin, W. Scott Spangler, Rui Ma, and Jiawei Han. Topic initiator detection on the world wide web. In *WWW*, pages 481–490, 2010.
- [38] Min-Soo Kim and Jiawei Han. A particle-and-density based evolutionary clustering method for dynamic networks. *PVLDB*, 2(1):622–633, 2009.
- [39] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *ACL*, pages 423–430, 2003.
- [40] Bill Kovach and Tom Rosenstiel. *Blur: How to Know What's True in the Age of Information Overload*. Bloomsbury Publishing USA, 2010.
- [41] Pei Lee and Laks V. S. Lakshmanan. Query-driven maximum quasi-clique search. In *SDM*, 2016.
- [42] Pei Lee, Laks V. S. Lakshmanan, and Evangelos E. Milios. Keysee: supporting keyword search on evolving events in social streams. In *KDD*, pages 1478–1481, 2013.
- [43] Pei Lee, Laks V. S. Lakshmanan, and Evangelos E. Milios. Cast: A context-aware story-teller for streaming social content. In *CIKM*, 2014.
- [44] Pei Lee, Laks V. S. Lakshmanan, and Evangelos E. Milios. Incremental cluster evolution tracking from highly dynamic network data. In *ICDE*, pages 3–14, 2014.
- [45] Pei Lee, Laks V. S. Lakshmanan, and Jeffrey Xu Yu. On top-k structural similarity search. In *ICDE*, pages 774–785, 2012.
- [46] Pei Lee, Laks V.S. Lakshmanan, and Evangelos E. Milios. Event evolution tracking from streaming social posts. Technical report, 2013.

- [47] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. 2010.
- [48] Jure Leskovec, Lars Backstrom, and Jon M. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD*, pages 497–506, 2009.
- [49] Pei Li, Jeffrey Xu Yu, Hongyan Liu, Jun He, and Xiaoyong Du. Ranking individuals and groups by influence propagation. In *Advances in Knowledge Discovery and Data Mining - 15th Pacific-Asia Conference, PAKDD 2011, Shenzhen, China, May 24-27, 2011, Proceedings, Part II*, pages 407–419, 2011.
- [50] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *ECML/PKDD (2)*, pages 33–49, 2008.
- [51] Ning Liu. Topic detection and tracking. In *Encyclopedia of Database Systems*, pages 3121–3124. 2009.
- [52] HelenaR. Lourenço, OlivierC. Martin, and Thomas Stützle. Iterated local search. In *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 320–353. Springer US, 2003.
- [53] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schuetze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [54] Adam Marcus, Michael S. Bernstein, Osama Badar, David R. Karger, Samuel Madden, and Robert C. Miller. Processing and visualizing the data in tweets. *SIGMOD Record*, 40(4):21–27, 2011.
- [55] Adam Marcus, Michael S. Bernstein, Osama Badar, David R. Karger, Samuel Madden, and Robert C. Miller. Twitinfo: aggregating and visualizing microblogs for event exploration. In *CHI*, pages 227–236, 2011.
- [56] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering smallest-last ordering and clustering and graph coloring algorithms. *Journal of ACM*, 30(3):417–427, 1983.
- [57] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.

- [58] Matteo Negri and Yashar Mehdad. Creating a bi-lingual entailment corpus through translations with mechanical turk: \$100 for a 10-day rush. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, CSLDAMT '10, pages 212–216, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.
- [59] Tadashi Nomoto. Two-tier similarity model for story link detection. In *CIKM*, pages 789–798, 2010.
- [60] Aditya G. Parameswaran, Stephen Boyd, Hector Garcia-Molina, Ashish Gupta, Neoklis Polyzotis, and Jennifer Widom. Optimal crowd-powered rating and filtering algorithms. *PVLDB*, 7(9):685–696, 2014.
- [61] Panos M. Pardalos and Steffen Rebennack. Computational challenges with cliques, quasi-cliques and clique partitions in graphs. In *SEA*, pages 13–22, 2010.
- [62] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. On the maximum quasi-clique problem. *Discrete Applied Mathematics*, 161(1-2):244–257, 2013.
- [63] Jian Pei, Daxin Jiang, and Aidong Zhang. Mining cross-graph quasi-cliques in gene expression and protein interaction data. In *ICDE*, pages 353–354, 2005.
- [64] Mauricio G. C. Resende. Greedy randomized adaptive search procedures. In *Encyclopedia of Optimization*, pages 1460–1469. 2009.
- [65] Kazumi Saito, Takeshi Yamada, and Kazuhiro Kazama. The k-dense method to extract communities from complex networks. In *Mining Complex Data*, pages 243–257. 2009.
- [66] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *WWW*, pages 851–860, 2010.
- [67] Anish Das Sarma, Alpa Jain, and Cong Yu. Dynamic relationship and event discovery. In *WSDM*, pages 207–216, 2011.
- [68] Chirag Shah, W. Bruce Croft, and David Jensen. Representing documents with named entities for story link detection (sld). In *CIKM*, pages 868–869, 2006.

- [69] Dafna Shahaf, Jaewon Yang, Caroline Suen, Jeff Jacobs, Heidi Wang, and Jure Leskovec. Information cartography: creating zoomable, large-scale maps of information. In *KDD*, pages 1097–1105, 2013.
- [70] David Sontag and Dan Roy. Complexity of inference in latent dirichlet allocation. In *NIPS*, pages 1008–1016, 2011.
- [71] Mauro Sozio and Aristides Gionis. The community search problem and how to plan a successful cocktail party. In *KDD*, pages 939–948, 2010.
- [72] Matthew Staffelbach, Peter Sempelinski, David Hachen, Ahsan Kareem, Tracy Kijewski-Correa, Douglas Thain, Daniel Wei, and Greg Madey. Lessons learned from an experiment in crowdsourcing complex citizen engineering tasks with amazon mechanical turk. *CoRR*, abs/1406.7588, 2014.
- [73] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*, pages 104–112, 2013.
- [74] Takeaki Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, 2010.
- [75] Nan Wang, Srinivasan Parthasarathy, Kian-Lee Tan, and Anthony K. H. Tung. Csv: visualizing and mining cohesive subgraphs. In *SIGMOD Conference*, pages 445–458, 2008.
- [76] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, pages 440–442, 1998.
- [77] Jianshu Weng and Bu-Sung Lee. Event detection in twitter. In *ICWSM*, 2011.
- [78] Yiming Yang, Tom Ault, Thomas Pierce, and Charles W. Lattimer. Improving text categorization methods for event tracking. In *SIGIR*, pages 65–72, 2000.
- [79] Xiaohan Zhao, Alessandra Sala, Christo Wilson, Xiao Wang, Sabrina Gaito, Haitao Zheng, and Ben Y. Zhao. Multi-scale dynamics in a massive online social network. In *IMC*, pages 171–184, 2012.