# Finding dense subgraphs and events in social media

Oana Balalau

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0020

EDITE - ED 130

# Doctorat ParisTech

# T H È S E

pour obtenir le grade de docteur délivré par

## TÉLÉCOM ParisTech

### Spécialité « Informatique »

*présentée et soutenue publiquement par*

Oana-Denisa Bălălău

le 17 mai 2017

# Recherche de sous-graphes denses et d'événements dans les médias sociaux

Directeur de thèse: **Mauro Sozio**

**Jury**

| | |
|---|---|
| **M. Carlos Castillo**, Directeur de recherche, Eurecat | Examinateur |
| **Mme. Chloé Clavel**, Maître de Conférences, Télécom ParisTech | Examinatrice |
| **M. Jean-Loup Guillaume**, Professeur, Université de La Rochelle | Rapporteur |
| **M. Michel Habib**, Professeur, IRIF | Rapporteur |
| **Mme. Clémence Magnien**, Directrice de recherche CNRS, LIP6 | Examinatrice |
| **M. Mauro Sozio**, Maître de Conférences, Télécom ParisTech | Directeur de thèse |
| **M. Emmanuel Viennet**, Professeur, Université Paris 13 | Examinateur |

T H È S E

**TÉLÉCOM ParisTech**
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

# Abstract

Event detection in social media is the task of finding mentions of real-world events in collections of posts. Social media contains information shared by hundreds of millions of users across the world and is one of the richest dataset created by human activity. The motivation behind our work is two-folded: first, finding events that are not covered by mainstream media and second, studying the interest that users show for certain types of events.

In order to solve our problem, we start from a graph based characterization of the data in which nodes represent words and edges count word co-occurrences. Graphs represent a powerful abstraction also for other types of data, spanning from road networks, social networks, brain networks and much more. Density is a very good measure of importance and cohesiveness in graphs and dense subgraphs have been found to represent different communities in social networks, or to indicate the presence of real-world events in a graph-of-words. Taking into account the special properties of real-word networks, we can develop algorithms that efficiently solve hard problems.

The contributions of this thesis are: devising efficient algorithms for computing different types of dense subgraphs in real-world graphs, presenting a novel dense subgraph definition and providing an efficient graph-based algorithm for event detection.

# Acknowledgements

First and foremost, I would like to express my sincere gratitude towards my adviser Mauro Sozio. He has helped me understand the challenges of research but also enjoy the rewards and, in the most difficult moments, he has been a source of encouragement, optimism, and energy. Every discussion and meeting we had felt as a small step in the right direction.

I would like to sincerely thank my committee members Carlos Castillo, Chloé Clavel, Jean-Loup Guillaume, Michel Habib, Clémence Magnien and Emmanuel Viennet for their valuable feedback on my work.

I want to thank Maximilien Danisch with whom I had the pleasure to collaborate, as his motivation and enthusiasm are contagious and inspiring. I thank Manthos Letsios for the privilege of advising him during his master internship. I especially thank my high school computer science teacher, Maria Tătulea, for the patience and dedication with which she guided us through the years and, most importantly, for helping me find my path in life.

My Ph.D. memories are inseparable from the wonderful moments I spent at Télécom ParisTech. I suppose that when you are happy to go to work, it means you are going to the right place. For this I want to thank my past and present colleagues: Roxana Gabriela Horincar, Fabian M. Suchanek, Mikaël Monet, Imen Ben Dhia, Katerina Tzompanaki, Thomas Rebele, Antoine Amarilli, Ziad Ismail, Danai Symeonidou, Mouhamadou Lamine Ba, Raman Samusevich, Arnaud Guerquin, Ioana Ileana, Atef Shaar, Pierre-Alexandre Murena, Jean-Louis Dessalles, Pierre Senellart, Antoine Saillenfest, Marie Al-Ghossein, Miyoung Han, Mostafa Haghir Chehreghani, Jacob Montiel Lopez, Quentin Lobbé, Jonathan Lajus, Modou Gueye, Sébastien Montenez, Albert Bifet, and Talel Abdessalem. I thank my office colleague, Jean-Benoît Griesner, for the fun moments and for our conversations. I especially thank Luis Galárraga for becoming my friend, for his kindness and for the joy he brings with him.

My years in Paris would not have been the same without the great friends I had: Sarthak Ghosh, Alejandro Macías, Zenaida Iordan, Jean Pichon, Anca Nițulescu and Marco Calemme. Even from far, my friends from Romania have been part of my life and I am grateful to them for that. I especially thank Mihaela Drăgoi for her support and encouragements and for sharing with me the depth and sensibility with which she sees the world.

Finally, I thank my family, Andreea, Elena, and Ion, for their unconditional love and support. They have taught me what it means to be a good person, and they have shared their love for knowledge and their strength in face of the difficulties of life. They have been the roots that keep me safe while helping me grow tall.

# Contents

# Chapter 1

# Introduction

A graph provides a powerful abstraction for representing real-world data, while it is beautiful in its apparent simplicity. Since Leonard Euler first formally studied the problem of the existence of a path that visits each edge of a graph exactly once, applications of graph theory have provided notable results in many areas of research. In recent years, graphs have become an important tool in data mining, highlighting the potential of graph mining, the field that studies finding interesting patterns in graphs. Graphs can model a variety of different types of data, spanning from road networks, social networks, brain networks and much more. For example, graphs have been used to study voting theory [69], and to study the small-world phenomenon in a principled way [50] as well as by an extensive experimental evaluation on the Facebook social network [11]. Besides the applications in social networks, graphs can help in understanding and modeling information networks, such as the hyperlink structure of the internet [12] or the connections between the terms used in a collection of tweets [8].

One of the essential tasks in graph mining is finding dense subgraphs, as density can be used to measure the importance and the cohesiveness of a subgraph. Dense subgraphs have other significant properties besides density, such as a small diameter. In communication graphs of wireless networks and peer-to-peer networks, the diameter of the graph is important in the design of routing algorithms [41]. Communities are by definition dense subgraphs, and promising results in [23] show a clear separation in communities of liberal and conservative blogs in a blog citation network, where nodes represented blogs and edges citations between two blogs. In the Web graph, dense subgraphs might be an indication of link spam, websites that through high interlinking attempt to increase their pagerank in order to modify the results of a query in a search engine [42]. In [47], frequent dense subgraphs are used to predict the functions of unknown genes in biological networks. Dense subgraphs in a graph-of-words representation of a stream of tweets might indicate the presence of real-world events, as shown in [8]. Because of the diversity and relevance of applications, it is important to develop algorithms that are well-suited for real-world networks. In this dissertation, we will develop new algorithms for dense subgraphs and we will focus on a particular application, event detection in social media.

Since the beginning of its mainstream adoption in the mid-1990s, the Internet has continuously challenged and shaped the way we perceive and interact with the

world. Nowadays, the news is often read not from their traditional support, the paper, but online, from news sites or even from social media. One of the most well-known newspapers, The New York Times, has been online since 1996. Social media differ from traditional media through an important characteristic: every user can contribute to a story with its personal experience and point of view. This richness of content can help in many situations, from crisis [21] in which the authorities can have access to much more information about affected areas, to cases of conflict of interests which disrupt the normal flow of information, and to coverage of smaller events which otherwise might be overlooked in traditional media. As a natural consequence, people will often turn to social media.

In September 2013 protests started in several cities in Romania, and at that time the protests were some of the largest in the country since the fall of the communists in 1989. The protests gave voice to independent journalists and NGOs that were against a bill that would speed up the process of constructing the largest open-pit mine in Europe in the northern part of Romania. The mine would have used 12,000 metric tons of cyanide annually, and although security measures would have been put in place they could offer little reassuring as Romania had already suffered the consequences of a cyanide spill in 2000, which was called the worst environmental disaster after Chernobyl [1]. The protests were largely not covered by the press in the first week and speculations of corruption were fuelled by a large amount of money that the mining company spent on publicity, which was estimated by Forbes[2] to be around 5.5 million Euro. Rumors of corruption surrounded also the initial acquisition of the territories and later efforts for starting the construction of the mine [3]. As a result of the protests, the Senate rejected the bill on Roșia Montană in November 2013. The large social movement was possible due to mobilization on social media and because of that, it was called the Romanian Autumn [4].

The Yemen civil war has started in 2015 and it continues to these days. According to a report by the United Nations, *"Millions of people in Yemen need assistance to ensure their basic survival. An estimated 14 million are food insecure (including 7 million severely food insecure);14.4 million lack access to safe drinking water or sanitation; 14.7 million lack adequate healthcare; and 3.3 million are acutely malnourished."* The war continues largely because of a power struggle between the two neighborhood countries, Iran and Saudi Arabia, where Saudi Arabia is backed by the UK and the US. Despite the huge humanitarian catastrophe, there have been little coverage from media and actions from political figures according to the United Nations Refugee Agency [5]. In the course of our research, we have discovered activists that use Twitter in order to bring awareness about the situation in Yemen.

The protests in Romania and the coverage of the conflict in Yemen highlight the

[1] http://news.bbc.co.uk/2/hi/europe/642880.stm

[2] http://www.forbes.ro/cati-bani-a-cheltuit-rosia-montana-gold-corporation\-in-publicitatea-din-presa-scrisa_0_8686-10173

[3] http://www.huffingtonpost.co.uk/stephen-mcgrath/rosia-montana-and-dirty-p_b_4123235.html

[4] http://ireport.cnn.com/docs/DOC-1051083

[5] http://www.acnur.org/t3/fileadmin/Documentos/Publicaciones/2016/10449.pdf?view=1

potential of social media. However, because social media were designed as a platform for users to share personal stories, not necessary events of large or small scale, much of the available data is interesting only to a small group of users, and in most of the cases only to the followers of a person. Besides personal stories, social media contain spam, publicity, and false news[6]. Even when users report events, because of the brevity of posts, sentences could contain misspelled words, abbreviations and uncommon syntactical structure, increasing the difficulty of tasks such as entity recognition and part-of-speech recognition. Another important issue is the large content available on social media, shared by 1.8 billion users on Facebook and 317 million users on Twitter[7]. In order to better understand the information shared by the users, Osborne et al. [74] compared Twitter, Facebook and Google Plus to see which site covered more news and where the news was first reported and found that for news coverage all three performed equally well, but Twitter was the first to report breaking news. For this reason, we will focus on event detection in a stream of tweets. Before dealing with the event detection task we will solve general graph mining problem that will help in understanding the structure of graphs-of-words, that is graphs that are constructing from tweets. In the following section, we detail our contributions.

## 1.1 Research Topics and Contributions

In this dissertation we focus on providing new definitions and insight on general graph problems, designing efficient algorithms that can deal with real-world networks and solving concrete data mining problems. Our research interest follows two main directions:

- improving state-of-the-art algorithms for graph mining by taking into account properties of real-world networks;
- event detection by means of a graph mining approach.

Graph mining is concerned with extracting patterns from graphs that satisfy certain properties. We study the problem of finding dense subgraphs while exploring different definitions of density.

Event detection is the data mining task concerned with finding abnormalities in a stream of data and in our case a stream of tweets. We tackle the problem of uniquely identifying an event and presenting a short but informative description of the event in the form of a subgraph.

In the following, we provide an overview of the contributions of this thesis.

*Finding $k$ dense subgraphs with maximum total sum of densities.*

Finding the densest subgraph has application in many areas, such as spam detection and community detection. Most often we are interested in finding more than

[6] https://www.theguardian.com/technology/2016/dec/15/facebook-flag-fake-news-fact-check

[7] http://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/

one community, which might correspond to finding several dense subgraphs. We define the problem of finding $k$ dense subgraphs of limited overlap and maximum total density and we present several heuristics for solving efficiently this NP-hard problem. One interesting direction is to search for minimal densest subgraphs, that is densest subgraphs which don't contain a proper subgraph of equal density. In order to compute such subgraph, we propose an efficient solution based on the linear programming formulation for computing densest subgraphs [22]. In addition, we improve the computation of the densest subgraph by pruning the search space and we design heuristics that efficiently enforce the maximum overlap between subgraphs. We have conducted an extensive evaluation to prove the quality of the solutions of our algorithms. Our heuristics find dense subgraphs on carefully chosen subgraphs of the original input graph, however, for future work, we will consider holistic approaches, such as graph decompositions. We show in a case study promising results for solving the problem of $k$ dense subgraphs of limited using the density-friendly decomposition [93].

Part of this work is a collaboration with Yahoo! and Hong Kong University, and is presented in the article: Oana Denisa Balalau, Francesco Bonchi, T-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. Finding Subgraphs with Maximum Total Density and Limited Overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, 2015.

*Heaviest-k subgraph: can we find exact solutions in real-world graphs?*

Densest subgraphs might provide unsatisfactory results when applied to certain data mining tasks, such as event detection. There are two main problems that appear: first, the output subgraph is usually too large for a user to understand its meaning, and second, the average degree measure tends to group together subgraphs corresponding to different events. In order to alleviate these problems, we study a variant of the densest subgraph problem in a weighted graph, the heaviest-$k$ subgraph. Although a $NP$-hard problem, exact solutions for small $k$ values can be computed efficiently on large real-world graphs. We build upon a branch-and-bound strategy and we use the intuition that heavy subgraphs should contain heavy edges. In order to speed up our algorithm, we further observe how to efficiently compute upper bounds and to avoid duplicate solutions. This algorithm can be easily modified in order to find other types of subgraphs, such as the heaviest clique of size $k$, as we will show in this dissertation.

This work is presented in the article: Matthaios Letsios, Oana Denisa Balalau, Maximilien Danisch, Emmanuel Orsini, and Mauro Sozio. Finding heaviest k-subgraphs and events in social media. In *The Sixth IEEE ICDM Workshop on Data Mining in Networks (DaMNet 2016), Barcelona Spain, 12*, 2016.

*Finding cliques in real-world graphs: k-clique listing and k-clique decomposition*

Cliques are the quintessential of dense subgraphs. The problem of $k$-clique listing is a hard problem, but efficient algorithms that take into account the structure of the underlying network can deal with large graphs. This standard problem in graph mining has been overlooked in recent research works in the favor of listing triangles

or finding the maximal clique. We propose an algorithm based on the work of [25], for which we show better guarantees and a straightforward parallelization method. We extend our contribution by devising an algorithm for computing the $k$-clique decomposition and an approximation of the $k$-clique densest subgraph. We show the performance of our algorithms in comparison with the state-of-the-art through an extensive experimental evaluation.

This research work is a collaboration with Maximilien Danisch and Mauro Sozio.

*Event detection in social media*

Event detection is a topic that has received a lot of attention in the field of data mining. There are several benefits of finding events in social media: we can detect certain events much earlier than the mainstream media covers them, we can find events that have received little media attention, and we have access to an important source of information when dealing with difficult situations that affect many people. We started our work by intensely crawling the social platform Twitter, which allows access to 1% of the tweets through a dedicated API. We have collected several datasets, each corresponding to a set of keywords or a geographic location. From this data we can create graphs of word co-occurrence, in which every tweet corresponds to a clique, building on the intuition that an event will contain a set of important keywords that will induce a dense subgraph. We experiment with different types of dense subgraphs and we propose our own definition of density which we find better suited for our task. We devise an event detection algorithm that is tailored to explore the key features that describe an event: the location and time-frame. In the end, we present a 14 months study that confirms the efficiency of our approach, and finally, we highlight interesting results concerning campaigns intended to bring awareness about the civil war in Yemen.

This research work is a collaboration with Mauro Sozio.

## 1.2 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2 we present basic notions that will be used throughout the rest of the thesis, and the previous work on the topics we will tackle. In Chapter 3 we study the problem of finding several overlapping subgraphs. In Chapter 4 we develop an exact algorithm for solving the heaviest-k-subgraph problem and in Chapter 5 we present algorithms for another type of dense subgraph, the clique. Our contributions to the task of event detection in social media are detailed in Chapter 6. We conclude with Chapter 7, where we offer final remarks and future research directions.

# Chapter 2

# Preliminaries

In this chapter, we first introduce notions from graph theory, linear programming, probability theory and event detection in Section 2.1. Then, we present the previous work in Section 2.2. We will address the state-of-the-art in finding dense subgraphs, finding multiple dense subgraphs, the heaviest $k$-subgraph, listing $k$-cliques and finding events in social media.

## 2.1 Definitions and Background

### 2.1.1 Graphs

Historically, one of the first graph problems that was rigorously studied is the Seven Bridges of Königsberg, which Leonhard Euler solved in 1736. The solution was kept and we can see in Figure 2.1 Euler's drawing of the seven bridges. We shortly introduce notions of graph theory that will appear in the course of the thesis.

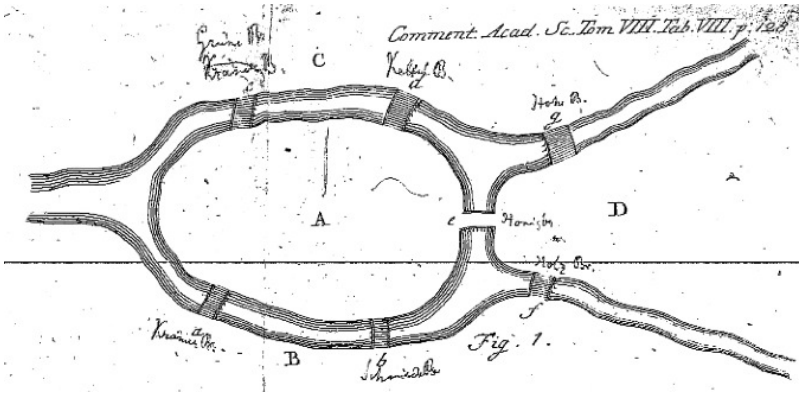


Figure 2.1: Euler drawing in "Solutio problematis ad geometriam situs pertinentis"

**Definition 2.1.1** (Graph)**.** A *graph* is a mathematical structure composed of nodes and edges, where nodes represent objects and edges the relation between objects. Formally, a graph $G = (V, E)$ is defined to be a pair of sets where $V$ is the set of nodes (called also vertices) and $E$ the set of edges, where an edge is a pair consisting of two nodes. When edges are nonempty sets of nodes of unrestrictive cardinality, the pair of sets $(V, E)$ is a *hypergraph*.

**Definition 2.1.2** (Directed/Undirected Graph)**.** In an *undirected graph* the edges are unordered pairs, while in a *directed graph* edges are ordered pairs.

**Definition 2.1.3** (Weighted/Unweighted Graph)**.** A *weighted graph* has a weight function associated to the set of edges, $w : E -> \mathbb{R}^+$, while an *unweighted graph* is a graph where the edges have unit weight.

**Definition 2.1.4** (Induced Subgraph)**.** Given a graph $G = (V, E)$ and $H = (V_H, E_H)$, where $V_H \subseteq V$ and $E_H \subseteq E$ and every endpoint of an edge from $E_H$ is included in $V_H$, we call $H$ a *subgraph* of $G$. An *induced subgraph* is a subgraph $H$ such that it contains all the edges in $E$ whose both endpoints are in $V_H$.

**Definition 2.1.5** (Average Degree Density)**.** Given a graph $H = (V_H, E_H)$, the average degree density is defined as half the average degree:

$$\rho(H) = \frac{|E_H|}{|V_H|}$$

**Definition 2.1.6** (Densest Subgraph)**.** Given a graph $G = (V, E)$ and an induced subgraph $H = (V_H, E_H)$, the subgraph $H$ is a *densest subgraph* if for any induced subgraph $S$ of $G$, $\rho(H) \geq \rho(S)$.

There are several variants of this problem, perhaps the most studied are the following ones: finding a maximum density subgraph of exactly $k$ nodes, finding a subgraph at most $k$ nodes or finding a subgraph of least $k$ nodes.

**Definition 2.1.7** (Edge Density)**.** Another definition of density used in literature is the *edge density*, which for a given graph $H = (V_H, E_H)$ equals:

$$\rho_e(H) = \frac{2|E_H|}{|V_H|(|V_H| - 1)}$$

**Definition 2.1.8** (Clique)**.** A graph is a *clique* if any two nodes are connected by an edge, that is if the graph has the edge density equal to 1. A $k-clique$ is a clique with $k$ nodes.

Two natural problems are finding a clique of maximum size (*maximum clique* problem) or enumerating/counting all the cliques of a certain size (k-clique enumeration/counting).

**Definition 2.1.9** (Quasi-Clique)**.** A *quasi-clique* is a graph for which, given a rational number $\gamma$, $0 \leq \gamma \leq 1$, its edge density $\rho_e$ is larger or equal to $\gamma$.

It is often the case that finding cliques might be too restrictive, so we prefer to search for quasi-cliques. The value for $\gamma$ is data-dependent so trying several values is advisable.

**Definition 2.1.10** (Maximum Weighted Clique)**.** In weighted graphs, the *maximum weighted clique* is a clique that maximizes the sum of edge weights.

**Definition 2.1.11** ($k$-core)**.** The *k-core* of a graph $G = (V, E)$ is a maximal induced subgraph of G, $H = (V_H, E_H)$ s.t. $\forall v \in V_H$, $d_H(v) \geq k$, where $d_H(v)$ is the degree of node $v$ in $H$.

**Definition 2.1.12** (Core Number of a Vertex)**.** The *core number* of a vertex $v$ is the highest value of $k$ such that $v$ is contained in a $k$-core.

**Definition 2.1.13** (Core Decomposition)**.** For all $k$, the set of all $k$-cores of the graph $G$ is the *core decomposition* of $G$.

**Definition 2.1.14** (Degeneracy)**.** The degeneracy of a graph $G$ is the highest value of $k$ such that $G$ has a $k$-core.

**Definition 2.1.15** (Tree/Forest)**.** A tree is an acyclic connected graph and a forest is a disjoint set of trees.

**Definition 2.1.16** (Arboricity)**.** The arboricity of a graph is the minimum number of edge-disjoint forests in which the graph can be decomposed.

Both the degeneracy and the arboricity are measures used to quantify the sparseness of a graph. For a graph $G$ with degeneracy $d$ and arboricity $a$, the following inequality holds:

$$a \leq d \leq 2a - 1.$$

### 2.1.2 Linear programming

Many graph problems can be expressed as optimization problems: finding the densest subgraph, finding the shortest path, finding the maximum flow. When both the constraints and the objective function are linear functions, we can express the problem as *a linear programming problem.*

**Definition 2.1.17** (Linear Programming Problem)**.** A linear programming problem can be defined as the problem of maximizing or minimizing a linear objective function by assigning values to a set of variables such that we satisfy a set of linear equations or inequalities.

Linear programming ($LP$) can be solved using polynomial time algorithms [44], while if we restrict the variables to take integer values, then the problem becomes integer linear programming ($ILP$), an NP-hard problem.

The canonical representation of a linear program is:

$$\begin{aligned} \max \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{PrimalLP}$$

where the known coefficients $c$ and $b$ are vectors and $A$ is a matrix, while the vector $x$ has to be determined.

**Definition 2.1.18** (Primal/Dual)**.** Every linear maximization problem can be expressed as a minimization problem and reciprocally every minimization problem can be expressed as a maximization problem. The initial problem is called the primal and the second the dual.

The dual of the previous LP is:

$$\begin{aligned} \min \quad & \mathbf{y}^{\mathrm{T}}\mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^{\mathrm{T}}A \leq \mathbf{c}^{\mathrm{T}} \\ & \mathbf{y} \geq \mathbf{0} \end{aligned} \tag{DualLP}$$

where $y$ represents the vector of variables and $c$, $b$, and $A$ are the coefficients previously introduced.

**Theorem 2.1.1.** The dual of a dual linear program is the initial primal linear program.

**Theorem 2.1.2** (Weak Duality Theorem)**.** If $\mathbf{x}$ is a feasible solution for the PrimalLP and $\mathbf{y}$ is a feasible solution for the DualLP, then:

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} \geq \mathbf{y}^{\mathrm{T}}\mathbf{b}.$$

**Theorem 2.1.3** (Strong Duality Theorem)**.** The *strong duality theorem* states that if the primal or the dual problem is feasible and the solution bounded, then this is true for both problems and the optimal values are the same:

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{y}^{\mathrm{T}}\mathbf{b}.$$

Linear programming has been a fundamental tool in the analysis and design of approximation algorithm. A linear programming formulation can help develop approximation algorithms by means of two main techniques: linear programming relaxation or the primal-dual schema.

**Definition 2.1.19** (Approximation Algorithm)**.** An *$\alpha$-approximation algorithm* is a polynomial algorithm for an optimization problem which always returns a solution of value within a factor of $\alpha$ of the optimal solution. More precisely, for a maximization problem with optimum solution $OPT$, the value returned by the $\alpha$-approximation algorithm $APPX$, where $\alpha < 1$, will always be $\alpha OPT \leq APPX \leq OPT$. For a minimization problem, $\alpha > 1$, the following inequality holds for an $\alpha$-approximation algorithm, $OPT \leq APPX \leq \alpha OPT$.

Hard problems like set cover have a natural representation using integer linear programming. The integer linear programs can be transformed in linear programs through a technique called *relaxation*, by allowing the variables to take real values. The LP-relation provides fractional solutions which are converted to integral solution through a technique called *LP-rounding*. The approximation guarantee will be then computed by comparing the cost of the fractional solution and the integral solutions. The first step in finding the approximation guarantee is computing the *integrality gap*, that is the maximum ratio between the solution of the integer linear program and its relation. For a minimization problem, let $M_{LP}$ be the solution of the LP and $M_{ILP}$ the solution of the ILP, the integrality gap is $IG = \frac{M_{ILP}}{M_{LP}}$, while for a maximization problem the ratio is reversed. We note that the integrality gap is always at least 1, because in the minimization problem the fractional solution will always be smaller or equal, while in a maximization problem the fractional solution

will be greater or equal, due to the relaxation. The second step is designing a rounding scheme which will transform the fractional solution into an integer solution with a cost of at most $rM_{LP}$, where $r$ is the rounding ratio. Because of the integrality gap, every rounded solution will have a cost of at least $IG * M_{LP}$, so the rounding ratio is necessarily greater or equal to the integrality gap. In conclusion, the rounding ratio will give the approximation guarantee of the algorithm.

Another method of designing approximation algorithms is using the primal-dual schema, which can give much faster algorithms as it exploits the special structure of individual problems. The first step is to compute the LP-relation of the problem and the dual of this relaxed problem. Then, a solution is constructed iteratively starting from an initial feasible solution $\mathbf{y}$ of the dual problem and then obtaining the corresponding integral solution of the initial LP. We note that any feasible solution of the dual provides a lower bound on the optimum solution of the primal.

### 2.1.3 Probability Theory

**Definition 2.1.20** (Random Variable)**.** A *random variable* $X$ is a function which assigns values to the output of a random experiment. Random variables can be discrete or continuous.

**Definition 2.1.21** (Expectation)**.** Suppose $X$ is a discrete random variable that can take value $x_1$ with probability $p_1$, value $x_2$ with probability $p_2$, and so on, up to $x_k$ with probability $p_k$. The *expectation* of $X$ is:

$$E[X] = x_1p_1 + x_2p_2 + \cdots + x_kp_k.$$

**Definition 2.1.22** (Standard Deviation)**.** The *standard deviation* is a measure of quantifying the dispersion of the data, taking small values if the data is close to the expected value and high values when the data is spread over a wide range of points. The formula of the standard deviation:

$$\sigma(X) = \sqrt{E[(X - E[X])^2]} = \sqrt{E[X^2] - (E[X])^2}$$

The expectation and the standard deviation are often denoted as $\mu(X)$ or $E[X]$ and $\sigma(X)$, respectively.

**Definition 2.1.23** (Markov's Inequality)**.** If $X$ is a non-negative random variable and $t \in R^+$, we can bound the probability of $X$ being greater or equal than $t$:

$$\Pr[X \geq t] \leq \frac{E[X]}{t}$$

**Definition 2.1.24** (Chebyshev's Inequality)**.** The intuition behind the inequality is if the deviation from the mean is small, it is unlikely that we will have a large deviation from the mean. Let $X$ be a random variable and $k \in R^+$, then:

$$\Pr[|X - E[X]| \geq k\sigma(X)^2] \leq \frac{1}{k^2}$$

**Definition 2.1.25** (Chernoff's Bounds)**.** When considering the case of a random variable that is the sum of a number of independent random variables, we can specify tighter bounds, improving on Markov's inequality and Chebyshev's inequality. Let $X = \sum_{i=1}^{n}(X_i)$, with $X_i$ independent and $P[X_i = 1] = p_i$ and $P[X_i = 0] = 1 - p_i$. Let $\mu = E[X] = \sum_{i=1}^{n}(p_i)$, $0 < \delta \leq 1$, we get the bound on the lower tail:

$$P[X < (1-\delta)\mu] < \epsilon^{-\mu\delta/2}.$$

For the upper tail and $\delta > 0$:

$$P[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}.$$

**Definition 2.1.26** (Binomial Distribution)**.** Given a number of $n$ experiments and a probability $p$, the binomial distribution $\mathcal{B}(n,p)$ of parameters $n$ and $p$ is the discrete probability distribution of the number of successes in $n$ independent experiments, with the probability of success equal to $p$.

Let $X$ be the random variable that follows the binomial distribution $\mathcal{B}(n,p)$. The probability of getting $k$, $0 \leq k \leq n$ successes in $n$ trials is:

$$\mathcal{B}(k;n,p) = \Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

The expected value of $X$ is $\mu(X) = np$ and the standard deviation is $\sigma(X) = \sqrt{np(1-p)}$.

**Definition 2.1.27** (Normal Distribution)**.** The normal distribution is a continuous probability distribution that can be used to approximate the binomial distribution for $n$ large enough.

The probability density of the normal distribution $\mathcal{N}(\mu,\sigma^2)$ is:

$$f(x \mid \mu,\sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $\mu$ is mean or expectation of the distribution and $\sigma$ is its standard deviation.

### 2.1.4 Event detection

**Definition 2.1.28** (Event)**.** A happening of importance that takes place at a specific time and place, along with all necessary preconditions and future consequences.

Part of the definition was given in [3] ("a specific thing that happens at a specific time and place along with all necessary preconditions and unavoidable consequences.") and is perhaps one of the most complete definition existing. However, it is not straightforward to identify the necessary preconditions and future consequences, and also to evaluate the importance of an event.

**Definition 2.1.29** (Activity)**.** An activity is a set of actions that have a common goal.

**Definition 2.1.30** (Topic)**.** A topic is an event or an activity and all the related events and activities.

The definitions of an activity and a topic were formalized during the Topic Detection and Tracking program [1].

**Definition 2.1.31** (Trend)**.** A trend is a topic that receives more attention over time [52].

**Definition 2.1.32** (Twitter storm)**.** A Twitter storm or a tweet storm is a trend that has a sudden increase in mentions and usually a much shorter time span. Twitter storms are organized by a person or a group of people in order to bring awareness on a controversial topic or on breaking news.

We note that although the definitions offer clarifications, ambiguities are unavoidable as these notions don't have a precise mathematical interpretation which can be rigorously tested.

[1] `http://www.itl.nist.gov/iad/mig/tests/tdt/1999/doc/tdt98_overview/tsld003.htm`

## 2.2 Related Work

Our work has been initially inspired by the research of [8], in which the authors propose a graph mining approach for real-time story identification. The stream of news is represented as a graph, where nodes are entities and edges qualify the strength of relations between entities. Throughout this thesis, we will be concerned with general graph mining problems but also with their applications to real data mining tasks and, in our particular case, event detection.

### 2.2.1 Graph Mining

Graph mining is an important tool in the general framework of data mining. When data can be modeled in terms of entities and connections between entities, graphs are the quintessential representation. In the literature, graphs have been used for representing social networks, road networks, collaboration networks, product similarity networks and also for modeling the content of posts of social networks users.

Having a mathematical representation of data provides the freedom for defining and computing mathematical properties which would be otherwise hard to quantify. Examples include computing all the shortest paths, the diameter of the graph, connected components and dense subgraphs. In this thesis, we focus on finding dense subgraphs due to the relevance for the task of finding events in social media.

#### 2.2.1.1 Dense subgraphs

The densest subgraph problem has applications in event detection but also in other areas, for example in community detection (Fortunato [38] elaborated an extended survey on methods of community discovery), finding patterns in gene annotation graphs (the work of Saha et al. [86]) and link spam detection (Gibson et al. [42] discovered that dense subgraphs in web graphs often are link spam, i.e. websites that try to influence search engine rankings). Finding the densest subgraph is a well-studied problem in literature and it can be solved in polynomial time despite the fact there is an exponential number of subgraphs to consider. One of the earliest proposals of a polynomial algorithm is by [82], which gives a solution requiring in the worst case $n$ max-flow computations on a network with $n + 2$ nodes. Goldberg [43] gives a faster algorithm that needs only $O(log(n))$ max-flow computations. Goldberg constructs a network in which a subset of the edges receives a weight that depends on an input parameter, $\alpha$. This parameter will correspond in the end to the density of the densest subgraph, and in order to find the value, binary search is performed. In [22], the authors prove that an optimum solution for the densest can be computed using linear programming techniques. Further, approximation algorithms for both undirected and directed graphs are presented. The intuition of the approximation algorithm and of its guarantee comes from the dual of the initial LP for the densest, as in the classical primal-dual schema. The algorithm for the undirected case is very simple: at each iteration, we remove the node of minimum degree and we compare the density of the remaining graph with a global maximum. At the end of the algorithm, we output the subgraph of maximum density found. We

note that the approximation algorithm for the undirected graph has many interesting properties, such as providing the optimum solution for the core decomposition. Bahman et al. [12] provide approximation algorithms that can be implemented both in a data streaming model or in a distributed model. In this work [12], the authors provide a $1/(2+2e)$ approximation algorithm for computing the densest subgraph in both directed and undirected graphs, that makes $O(log_{1+e}(n))$ passes over the input. In addition, an $1/(3+3e)$ approximation algorithm for the $DalkS$ problem is presented. The approximation algorithm for the undirected graph is very similar to the algorithm in [22], however, at each step, we remove not the node of minimum degree but all the nodes that have the degree smaller than $2+2\epsilon$ times the density of the current graph. The algorithm has a straightforward implementation in the MapReduce framework. For a dynamic setting where edges are added and removed, Epasto et al. [34] present an approach that provides a $1/(2+2e)$ approximation of the current densest subgraph and requires $O(poly\ log(n+r))$ amortized cost per update with high probability, where $n$ is the maximum number of nodes in the graph and $r$ the total number of update operations. The algorithm works by computing nested sets of nodes that have certain properties, such as representing subgraphs of increasing density, and then it maintains those properties as edges are added and removed.

#### 2.2.1.2 Heaviest $k$-subgraph

When enforcing a limit on the size of the subgraph, finding the densest subgraph of exactly k vertices ($DkS$), the problem becomes NP-hard (reduction from the Clique problem). Variants of the problem, finding a densest subgraph with at most k vertices($DamkS$) or a densest subgraph with at least k vertices ($DalkS$) are also NP-hard [49]. For ($DalkS$), [7], [49] give 1/3-approximation algorithms, which is based on the approximation algorithm for the densest presented in [22]. In a weighted graph, the $DkS$ problem has been referred as the problem of finding the heaviest k-subgraph $HkS$. Both $DkS$ and $HkS$ are NP-hard and no polynomial-time algorithms with a fixed performance guarantee are known. In [37] the authors give a polynomial algorithm that computes a solution within a factor of $n^{\alpha}$, $\alpha < \frac{1}{3}$, from the optimum solution for $DkS$, with the addition of a factor of $O(\log n)$ for $HkS$. The algorithm calls three different procedures and outputs the densest from the three solutions. The first procedure is a simple baseline, where $k$ vertices are returned from a set of $k/2$ edges that are selected arbitrarily. The second procedure is a greedy algorithm where the nodes are first sorted non-increasing by degree, and the first $k/2$ nodes are retained and then the solution is completed with $k/2$ that have the highest number of neighbors in the first set retained. The third procedure entails computing walks of length two for every pair of nodes in the graph, and then carefully constructing a solution containing the $k/2$ nodes with the largest number of paths of length two and the $k/2$ best-connected nodes to the previous nodes. For $HkS$, Asahiro et al. [9] describe a greedy algorithm that has an approximation ratio of $O(\frac{k}{n})$. As in the case of the $Dalks$ problem, this algorithm is also inspired by the approximation algorithm for the densest presented in [22], however, there are no fixed parameter guarantees. The state-of-the algorithm for $DkS$ is due to Bhaskara

et al. [17] and gives a $O(n^{\frac{1}{4}+\epsilon})$ approximation guarantee for any $\epsilon > 0$. In [17], the authors count selected subgraphs of constant size in G and use these counts to find the vertices of the dense subgraph.

#### 2.2.1.3 Finding multiple densest subgraphs

Unlike its single-subgraph counterpart, the problem of finding a set of $k$ dense subgraphs has received considerably less attention. In [97], the authors maintain the top $k$ edge-disjoint subgraphs in a dynamic stream of graphs, taking into account departures and arrivals of graphs. The authors first efficiently compute the top $k$ in a single graph in the following manner: the densest subgraph is computed and then all its edges are removed from the graph, and then the procedure restarts on the remaining graph. The algorithm stops when it has found the top $k$ densest subgraphs, and a pruning technique based on the guarantee of the approximation algorithm for the densest subgraph is used to speed up the computation. For maintaining the top $k$ subgraphs in the moving window of graphs, the authors use the same pruning technique as in the single graph computation. In [96], the authors find top $k$ dense subgraphs that are vertex-disjoint. In this approach, instead of iterating until no other dense subgraph of a density larger than the current $k$ solutions can be found, the authors iterate until they have found $k$ subgraphs. Recent work by [40] defines the overlap between subgraphs using a distance function and then penalizes the overlap as part of the objective function. Their approach is based on the max-sum diversification problem, where the goal is to maximize the sum of a submodular function and a distance function. Angel *et al.* [8] focus on maintaining the set of all overlapping subgraphs exceeding a density threshold under streaming edge weight updates. The main contribution consists in the design of an efficient data structure for storing subgraphs that have a large overlap while allowing fast access to the information related to the subgraphs and fast insertion of new dense subgraphs. In [70], the authors proposed an efficient heuristic for maintaining the top-$k$ node disjoint dense subgraphs in a dynamic graph. The main intuition of the algorithm is that high degree nodes contribute to dense subgraphs and the nodes in a dense subgraph are strongly connected to each other.

Apart from that, existing research has considered tangentially related problems, such as finding nested subgraphs containing a set of query nodes and exhibiting non-increasing densities [92]. Another property that the authors enforce is the uniformity of communities, that is the weights of edges in a community should be in a small range of values. Recent work by the same authors [93] has defined the density friendly-decomposition, in which the nested subgraphs have also non-increasing densities, but the first subgraph in the decomposition is a densest subgraph. They design an exact algorithm based on the max-flow algorithm for the densest subgraph and an approximation algorithm based on the approximation algorithm for densest. In [30], the authors discover overlapping dense components containing a query node [30]. More precisely, the authors find the quasi-clique components containing a query node, where a quasi-clique component is higher level subgraph, in which a node is a quasi-clique in the original graph and there is an edge between two nodes if they share at least a given number of nodes. In [42], an algorithm is

designed for finding dense bipartite subgraphs in massive graphs. The results of the algorithm when applied to the web graph retrieve many link spams, that is websites that through massive interlinking try to improve their pagerank.

Chen and Saad [23] propose a matrix-blocking model to identify dense subgraphs that best cover the input graph. In this model, not every node has to belong to a dense subgraph and the number of dense subgraphs is not given in input. The blocking algorithm works by grouping together similar columns and rows using a cosine similarity function and obtaining, in the end, dense regions corresponding to dense subgraphs. The original blocking algorithm requires a threshold for which the value of the cosine similarity function is considered good, in [23] the authors remove this threshold, replacing it however with a threshold for estimating the quality of a dense subgraph.

#### 2.2.1.4 Listing triangles

Triangle listing and counting (i.e. 3-clique listing) have been studied intensively in recent years, with the algorithms proposed in [25, 55, 63] being perhaps the most efficient ones in practice, when the input graph fits into main memory. In particular, the compact-forward algorithm developed in [55] has the running time $O(m \cdot \sqrt{m})$ and a space complexity of $O(m)$, while in the case when the degree distribution of the input graph follows a power law distribution with exponent $\alpha$ the running time is in $O(m \cdot n^{\frac{1}{\alpha}})$. The compact-forward algorithm is based on two main ideas: sorting the nodes in non-increasing order of degree and maintaining for each node a list of its neighbors that appear later in the order. The algorithm uses the order for iterating over the nodes but also for avoiding outputting the same subgraph twice, as it considers a neighbor of $v$ only if it appears later in the order. The lists are used to compute the intersection of neighborhoods, taking into account that each triangle will be visited once. The algorithm presented in [25] has running time $O(m \cdot a)$, where $a$ is the arboricity of the graph and $m$ is the number of edges. In this algorithm, the nodes are again sorted in non-increasing order of degree but the order is required only for the theoretical guarantees. An additional data structure is used to compute the intersection of neighborhood and after a node is processed, it is removed from the graph in order to avoid duplicates.

From a theoretical point of view, the state-of-the-art algorithms for triangle counting and listing are [6] and [18], respectively. The running time of [6] is $O(m^{\frac{2\omega}{\omega+1}})$, where $\omega$ denotes the fast matrix multiplication exponent, however, the algorithm requires the graph to be represented as an adjacency matrix, so the space complexity is $O(n^2)$. The algorithm in [6] computes for a given adjacency matrix $M$ representing a graph $G$, the matrix $M^3$, as for each node $v$, the value $M^3_{vv}$ represents twice the number of triangles containing nodes $v$. In the recent work of [18], the algorithms are output sensitive, i.e., they are fast if the number of triangles in the input graph is small. The main contribution of the authors is in designing a randomized algorithm that lists with high probability all the triangles in a graph by using two alternating procedures: eliminating edges that belong to few triangles by listing those triangles and eliminating nodes of a low degree by enumerating the paths of length two that contain the nodes.

Several algorithms have been proposed in the case when the graph does not fit into main memory, such as [51] in the MapReduce architecture. The authors sample paths of length two, which they refer to as *wedges*, and use the wedges to estimate the clustering coefficient of the graph and the total number of triangles. In [14], the authors study the problem of computing the number of triangles every node belongs to, referred as the local triangle problem. The algorithms work in the semi-streaming model, requiring $O(\log(|V|))$ sequential scans over the edges of the graph. The main contribution of the paper is to efficiently compute the intersection of sets, as this can be used to estimate the number of triangles in which a node is included, by summing over the intersections between the set of his neighbors and the sets of neighbors of his neighbors. In [27], the authors present an algorithm that can deal with large graphs by reading from the hard disk partitions of the graph and counting the triangles in those partitions. The algorithm is I/O-efficient, as all the necessary information is loaded into the main memory. The authors prove the correctness of their approach and provide two partitioning schemes, sequential partitioning, and dominating-set-based graph partitioning. The first schema reads sequentially the graph from the disk and performs well if the graph presents high locality, that is nodes are given consecutive IDs based on their proximity. The second schema is based on the minimum dominating set of a graph, that is the minimum set of nodes such that every node $v$ in the graph is either in the set of it is connected to a node in the set. Partitions of the graph are created starting from subsets of the dominating set.

#### 2.2.1.5 $k$-cliques

Perhaps surprisingly, the $k$-clique listing and counting problems have received not much attention in recent years. This might be due to the computational challenges of the problem, with visualizing or even storing all $k$-cliques being not feasible if the input graph is both large and dense. Most recent studies focus on counting or listing triangles and maximal cliques.

A sequential algorithm for counting and listing $k$-cliques is presented in [25]. Its running time is $O(m \cdot a^{k-2})$ where $a$ is the arboricity of the graph. The algorithm computes the cliques recursively, using a procedure that has as parameters an integer $k$, a set $C$ and a graph $G$, where initially $k$ is the size of the clique, $G$ the input graph and $C$ the empty set. The procedure is as follows: for a given $k > 2$, first the nodes in the graph are sorted in a non-increasing degree order and then we iterate over the nodes; for each node $v$ a subgraph induced by its neighborhood is computed, and the recursive procedure is called for the computed subgraph, $k - 1$ and the set $C \cup \{v\}$; if $k = 2$ the recursion is over and we output for each edge from the subgraph the reunion between its nodes and the nodes from $C$. For efficiency issues, the procedure doesn't pass a new subgraph, but reuses the initial graph $G$, with additional labels.

#### 2.2.1.6 Maximal cliques

Most maximal cliques listing algorithms are based on the seminal algorithm developed by Bron and Kerbosch [20, 24, 94]. The Bron and Kerbosch algorithm [20] is a branch and bound approach, where larger cliques are generated first and cliques that share many nodes are generated sequentially. The algorithm is a recursion in which the solution $S$ is extended. Besides the set $S$, a set of nodes $C$ and $N$ are used to store the candidate nodes that can be added to the set $S$, and the nodes that have already served as an extension of the set $C$ at previous steps, respectively. In order to have a maximal clique, both set $C$ and $N$ have to be empty. If set $C$ is not empty we can extend the current clique with nodes from $C$, while is $N$ is not empty, the current clique has been contained in another clique. Note that each time we add a node $v$ in the solution $S$, we update the sets $C$ and $N$ such that we remove nodes that are not neighbors of $v$. Intuitively, the algorithms find all maximal cliques containing the first node $v_1$, then all the maximal cliques containing the second node $v_2$ but not also the first $v_1$ and so on.

In [94], the similar pruning techniques as in [20] are used, but the maximal cliques are listed in an encoded format in the output. The algorithm is not output-sensitive so the running time can be bounded theoretically, having a worst case running time of $O(3^{n/3})$, where $n$ is the number of nodes in the graph and $3^{n/3}$ is the maximum number of maximal cliques in a graph with $n$ nodes. A further improvement of the algorithm is to select from the set $C$ a node $v$ which has the maximum overlap between its neighborhood and the set $C$, allowing to construct first the larger maximal cliques and reducing the number of recursive calls. The state of the art algorithm for this problem has running time $O(c \cdot n \cdot 3^{\frac{c}{3}})$ [35], where $c$ is the core number of the input graph. Such running time is almost tight as the largest number of maximal cliques in a graph with core number $c$ is $(n - c) \cdot 3^{\frac{c}{3}}$. In order to obtain this running time, the algorithm iterates over the nodes in non-decreasing core order, and then for each node the algorithm presented in [94] is called on the subgraph induced by its neighborhood.

Recent works have focused on distributed algorithms that can deal with large real-world graphs via a distributed computation on smaller blocks of data [24, 28]. In [28], the nodes are partitioned into two sets, using as criteria if the number of neighbors can fit in a block of data. The nodes in the first set are feasible nodes and the nodes in the second set are hubs. If we consider all the maximal cliques containing at least one feasible node and all the maximal cliques containing only hubs, we obtain all the maximal cliques in the graph. Therefore we can compute the maximal cliques containing nodes in the first set, and then compute the maximal cliques in the subgraph induced by the hubs. We note that we can again use the same strategy on the subgraph induced by the hubs, as the hubs degrees will be greatly reduced after removing the feasible nodes.

In Table 2.1, we summarize some of the techniques we have presented.

| Method | Component | Ref. | Technique |
|---|---|---|---|
| Exact polynomial alg. | Densest subgraph | [43] | Solving $\log_n$ maximum flow computation |
| | | [22] | Solving a linear program |
| | Triangle counting | [25] | Degree ordering and neighbourhood intersection |
| | | [18] | Fast matrix multiplication and filtering of nodes and edges |
| Exact suprapoly. alg. | Densest-$k$ subgraph | [59] | Branch-and-bound |
| | Maximal clique | [20] | Recursive backtraking |
| | $k$-cliques counting | [32] | Ordering and neighbourhood intersection |
| Approximation alg. | Densest subgraph | [22] | Iteratively removing min degree node |
| | Densest $k$-subgraph | [17] | Counting selected subgraphs of constant size |
| | Dynamic densest subgraph | [34] | Maintaining subsets of nodes that satisfy certain properties |
| | Top-$k$ dense subgraphs | [40] | Adapted max-sum diversification |
| Efficient heuristics | Top-$k$ dense subgraphs | [13] | Iterative computation of dense subgraphs |
| | Dynamic top-$k$ dense subgraphs | [70] | Tracking of high degree nodes and data structures that satisfy certain properties |

Table 2.1: Short summary of existing techniques

### 2.2.2 Event Detection

*An event is a happening of importance that takes place at a specific time and place, along with all necessary preconditions and future consequences.* There has been a lot of interest in discovering events in an automatic fashion by studying mainstream media or social media. Event detection in traditional media has been the focus of the Topic Detection and Tracking program since 1997, program which was sponsored by the U.S. Defence Advanced Research Projects Agency [2]. The purpose of the program was to find and track news in streams of articles [4]. Event detection in social media faces additional challenges, due to several factors: tasks like entity recognition or part-of-speech recognition became difficult because of abbreviations and spelling mistakes, text can be very short giving little information on the context (only 140 characters are allowed on Twitter), and the data is very large and noisy.

**Event detection in mainstream media.** The most used technique for event detection in mainstream media is document clustering [5]. In this approach, new articles arrive and have to be assigned to existing clusters or to a new cluster, where a cluster corresponds to a topic or event. An article is represented as a vector which has as dimensions terms and for each element of the vector we compute the TF-IDF score of the associated term. Every vector is compared with the centroids and if the smallest similarity is below a threshold, a new cluster is initialized. Different similarity functions are proposed in [5]: cosine, weighted sum, language models, and Kullbach-Leiblar divergence. In [75], the authors present a faster algorithm that deals with the most time-consuming aspect of the previous algorithm: finding the closest cluster to a point. As the search can be linear, in [75] the authors propose a locality sensitive hashing algorithm that works in a bounded amount of time.

**Social media vs mainstream media.** Interesting research has been conducted in order to discover similarities and differences in event coverage in mainstream media vs social media. In [102], the authors compare the content available on Twitter with the news in New York Times using variants of the Latent Dirichlet Allocation algorithm. Some of the findings are that traditional media and Twitter cover similar topic categories, but with different distributions of interest and Twitter users are less likely to produce opinionated tweets about global topics, however, they will actively retweet such topics helping the spread of global news. In [72] the authors look at the case of climate change and how awareness campaigns, natural disasters, governmental meetings and others are covered in the media. Mainstream media seems to cover more disasters and government meetings caused by climate change while in Twitter the actions of individuals appear to be the most prominent.

Several approaches have been proposed for event detection in social media, which can be classified according to whether they focus on a particular class of events (such as earthquakes, computer security breaks etc.) or whether they focus on

[2] `http://www.itl.nist.gov/iad/mig/publications/proceedings/darpa98/html/tdt10/tdt10.htm`

general events. Another criterion that can be used to classify the related work is the technique, and the most successful techniques are clustering, graph mining approaches, support vector machine, and gradient boosted decision tree. Events are uniquely identified by different sets of elements: a set of documents, a set of entities and a time frame, a set of keywords and a timeframe or a keyword and a timeframe. There are several extensive surveys in [10, 48, 62, 71].

**Detecting unspecified events in social media.** One of the best-known works in event detection in social media is [101], where the authors propose an event detection algorithm that clusters the wavelet-based signal of terms. More precisely, the frequency of terms in posts over time is a time series, which can be processed using wavelet analysis. The authors filter trivial terms using auto-correlation of signals and cluster the remaining terms employing as a similarity metric the cross-correlation of signals. Clusters are finally ranked according to a score which takes into account the sum of cross-correlation of terms and the total number of terms associated with the cluster. In [15] the authors cluster tweets and then label the resulting clusters. The clusters don't necessarily correspond to events, so a second step of classification distinguishes between events and nonevents. The authors propose several features as input for the classification algorithm and also make the difference between real-world events and Twitter-specific events, such as online contests. In [29], the authors propose an algorithm based on the continuous wavelet transformation and Latent Dirichlet Allocation (LDA) topic inference. In the first step, the frequency of hashtags over time is analyzed using continuous wavelet transformation and hashtags that present a peak in their pattern of occurrence are selected. For each selected hashtag, LDA is run on the tweets containing it, in order to infer the topics associated with that tag. In [8], an algorithm is developed for maintaining overlapping dense subgraph with size limit in a dynamic graph. Each dense subgraph represents an event, where the nodes are entities describing the event and the importance of an event is given by the weight of the subgraph. The main contribution consists in designing an efficient data structure for storing subgraphs that have a large overlap while allowing fast access to the information related to the subgraphs and fast insertion of new dense subgraphs. In [45], the authors leverage the intuition that during an event users will mention more often other users in order to engage in conversation or they will retweet posts of other users. Terms are selected from tweets containing mentions, and for each is computed the deviation from the expected frequency for given time period. An event is uniquely identified by a term and a time window and to provide a better description of events, additional terms sharing similar temporal patterns with the initial term are added.

The previously approaches use textual and quantitative features to detect and place an event in a temporal context. However, most often an important feature is overlooked: an event has to have a *location*. The location is very important because there can be events of the same type, i.e. that can be described by the same keywords or posts, that occur at the same time but in different places. In the following, we mention a few works that take into account the location in their algorithms. The first approach, proposed in [56], uses geotagged tweets in order to infer normal

crowd behavior and an event is detected each time the expected frequency of tweets in a region increases. A well-known topic inference algorithm, Latent Dirichlet Allocation, is modified in [78] in order to take into account temporal and geographic features. In [1], terms occurring in documents in a time frame are filtered according to two criteria: if a term doesn't deviate from the expected number of occurrences or if the geolocated tweets containing the term are uniformly distributed across space, then the term is removed. The remaining terms are clustered based on the similarity of their geographic distribution.

**Detecting a class of events in social media.** Supervised or semi-supervised techniques can be used when the type of event that we want to discover is known. In [87], a support vector machine algorithm classifies tweets in earthquake related or not. In order to account for possible noise and misclassification, the authors model the number of earthquake-related tweets as a Poisson process and infer the moment when an earthquake occurs using the exponential distribution. In [48], the authors present a platform for automatic classification of messages during a crisis event. More precisely, first tweets are crawled using certain criteria like location or keywords and then human annotators label tweets using different categories, such as "needs". An automatic classifier is trained based on the labels and retrained as new labels arrive. The authors mention that automatic classification using a pre-existing dataset is not a good solution as crisis events can have very specific aspects that differentiate them. In [83], the authors find controversial events, that is events in which users express opposing opinions, surprise or disbelief. They investigate which Twitter posts related to a list of celebrities are controversial. The authors estimate the controversy of an event using a regression model, where some of the features proposed are positive and negative polarity words, bad words, mentions of the celebrities in news articles in the same time period.

**Sub-event detection in social media.** In [26] the authors use only non-textual features in order to discover sub-events related to a given an event. They present a model which is based on the intuition that users are more likely to post tweets during an event and after the event has finished to retweet or communicate with other users. The classifier will use only tweets and retweets related to a given event, and more precisely it will classify a window $w_i$ as containing a sub-event if the counts of tweets and retweets in that window and in the windows $w_{i-2}, w_{i-2}, w_{i+1}, w_{i+2}$ satisfy a linear inequality. The inequality is obtained through logistic regression on an annotated dataset. In [68], the authors provide a technique for sub-event detection and summarization. Tweets are represented as a weighted graph-of-words and sub-events are identified using the notion of core decomposition. More precisely, when a sub-event occurs, users will use terms related to the sub-event, and consequently, these terms will have high core numbers in the graph representation. The authors recompute the graph at the end of 60 seconds time windows and compare the sum of the core numbers of the $t$ terms belonging to the highest cores with the average sum over the last time windows, and if the sum exceeds the threshold the corresponding terms are outputted to summarize the sub-event.

**Trend detection in social media.** A trend differs from an event as the trend can be unrelated to real events. The Twitter platform offers the possibility to users to see trending terms and hashtags, but without providing a summary of the corresponding trends. In 2015, the data science team at Twitter released a whitepaper [3] describing challenges and different trend detection approaches. The algorithms study the time series of terms occurrences over time and identify abnormal patterns. The first algorithm models the number of occurrences of terms in tweets as a Poisson distribution and finds trending terms when the probability of having a certain number of occurrences is very low according to the probabilistic model. A second approach is a data-driven approach that doesn't make any supposition on the underlying distribution of the time series, using only tagged time series corresponding to events and non-events and comparing them with the new untagged time series.

**Evaluating the performance of algorithms.** Testing the performance of event detection algorithms is a very difficult task, due to many factors such as the fact an event cannot be defined in a mathematical way which leads to a subjective evaluation, human annotators might differentiate between sub-events of the same event, and authors have tested their algorithm's performance on many datasets obtained using different criteria. As a possible solution, in [67], the authors propose an annotated corpus of 120 million tweets containing more than 500 events, obtained by evaluating the results of several event detection algorithms using crowdsourcing. Potential issues in using the corpus are that the annotated set of events might still be incomplete and that the dataset is not available but has to be queried but has to be queried from Twitter, leading to a smaller sample than the original due to tweet and account deletion. In [100], a comparison based on the qualitative assessment of the results and the run-time performance was performed on several event detection algorithms. The algorithm presented in [101] performed best when compared with the topic modelling algorithm LDA, and algorithms presented in [99], [98] and [29].

In the Table2.2 we summarize algorithms that have been used for event detection. We specify the technique, the features of importance of each algorithm, and the type of the event identified.

[3]https://blog.twitter.com/2015/trend-detection-social-data

| Ref. | Technique | Features selected | Event type |
|---|---|---|---|
| [88] | Online clustering using TF-IDF, initial filtering of tweets | Terms | Unspecified |
| [65] | Finding 'bursty terms', grouping of terms via co-occurrence | Terms | Unspecified |
| [80] | Online clustering via locality sensitive hashing | Terms | Unspecified |
| [81] | Online clustering via TF-IDF with boosting of proper nouns | Terms | Breaking news |
| [87] | Support vector machine + Kalman filters for location | context terms ( terms appearing before and after a query term) | Earthquakes |
| [83] | Gradient Boosted Decision Trees | Sentiments, bad words, entities | Controversial events |
| [56] | Modelling of crowd behaviour via geotagged tweets, identifying regions of interest and normal crowd behaviour | Geolocation | Unspecified |
| [15] | Online clustering via TF-IDF and classification of clusters | Terms, mentions, replies, retwees, topical coherence | Unspecified |
| [64] | Generative language modelling. | Tweet credibility: emoticons, post length, shouting, capitalization,etc. | Specified |
| [101] | Wavelet-based signals of terms, modularity partitioning | Signal of terms based on TF-IDF | Unspecified |
| [29] | Wavelet-based signals of terms and LDA topic inference for event summarization | Hashtags | Unspecified |
| [76] | Improving technique from [75] with Wikipedia information | Edits, page views and new page creation from Wikipedia | Unspecified |
| [60] | Binomial distribution of bigrams. | Bigrams | Unspecified |
| [16] | TD-IDF, normalized TD-IDF, entropy | Unigrams, bigrams | Unspecified |
| [8] | Dynamic overlapping subgraphs in a graph of entities | Entities | Unspecified |
| [26] | Linear classifier based on the intuition that the number of tweets increases after an event and the number of retweets decreases | Frequency of tweets, retweets | Specified, subevent |
| [45] | Computing intervals on which the sum of the difference between the frequency and the expected frequency of a term is maximized. | Mentions and terms | Unspecified |
| [68] | Top 10 terms with highest core values in a core decomposition of a weighted graph. | Terms | Specified, subevents |

Table 2.2: Event detection algorithms

# Chapter 3

# Dense Subgraphs with Limited Overlap

## 3.1 Introduction

Finding dense subgraphs in large graphs has emerged as a key primitive in a variety of real-world application domains [57], ranging from biology [39, 54] to finance [33]. In the Web domain, Gibson *et al.* [42] have observed that dense subgraphs might correspond to a thematic group of pages or spam link farms. In the context of social networks, finding dense subgraphs has been employed for organizing social events and community detection [90], as well as for expert team formation [19, 96]. Angel *et al.* [8] have shown how finding dense subgraphs in the entity co-occurrence graph constructed from micro-blogging streams can be used to automatically detect important events.

Many of the aforementioned applications ask for finding several (possibly overlapping) dense subgraphs, which might correspond to communities in social networks or important events. Perhaps surprisingly, such a problem has not been studied in a principled way to the best of our knowledge. In this chapter, we aim at filling this gap. In a first attempt to give a formal definition of the problem of finding overlapping dense subgraphs, one could aim at finding at most $k$ subgraphs with maximum aggregate total density. However, it turns out that such a formulation might lead to finding several subgraphs being very similar to each other and in particular sharing a large fraction of nodes of a relatively dense subgraph. Such a solution is not really interesting as the dense subgraphs to be found should ideally exhibit some appreciable degree of diversity among each other. Therefore, we enforce an upper bound on the pairwise Jaccard coefficient between the sets of nodes of the subgraphs.

A natural heuristic for our main problem is the following one: Greedily find one densest subgraph in the current graph, remove all its vertices and edges, and iterate until $k$ subgraphs are found or the current graph is empty (see e.g., [96]). This heuristic, although reasonable, might potentially deliver arbitrarily bad solutions in terms of our objective function, as we show in the remainder. Another observation is that such an approach would produce subgraphs which are pairwise disjoint. By allowing some limited amount of overlap, one could find more interesting (i.e., denser) solutions, while maintaining enough diversity among the subgraphs

extracted. Another drawback of the previous heuristic is that algorithms for finding densest subgraphs (based on linear programming and maximum flow) cannot cope with large graphs containing millions of edges.

In this chapter, we present an efficient algorithm for our problem which comes with provable guarantees in some cases of interest. We introduce the concept of minimality of a densest subgraph, (roughly speaking a densest subgraph is minimal if it does not contain any other densest subgraph) and develop efficient algorithms for finding minimal densest subgraphs, which will be pivotal in solving our main problem. Our algorithm for finding minimal densest subgraphs turns out to be the fastest known algorithm for the exact computation of a densest subgraph as it can handle large graphs containing up to 10 million edges, as shown by our experimental evaluation. We finally devise an efficient heuristic so to find subgraphs with limited overlap on even larger input graphs.

More in detail, the contributions of this chapter are summarized as follows:

- We define the $(k, \alpha)$-Dense Subgraph with Limited Overlap problem ($(k, \alpha)$-DSLO): given an integer $k > 0$ as well as a real number $\alpha \in [0, 1]$, find at most $k$ subgraphs that maximize the total aggregate density, i.e., the sum of the average degree of each subgraph, under the constraint that the maximum pairwise Jaccard coefficient between the set of nodes in the subgraphs be at most $\alpha$. We prove that $(k, \alpha)$–DSLO is NP-hard even when $\alpha = 0$ (disjoint subgraphs).

- We improve the LP-based approach by Charikar [22], thus achieving the fastest known exact algorithm for the densest subgraph problem. Our algorithm has the desirable property of producing *minimal* densest subgraphs and uses our fast LP solver as a subroutine. In particular, we prove that the number of calls to an LP-solver is logarithmic in the number of nodes with high probability. This allows us to deal with large real-world graphs.

- We devise an algorithm (MinAndRemove) for the $(k, \alpha)$-DSLO problem. We prove that in the case the input graph contains $k$ disjoint densest subgraphs, our algorithm is guaranteed to find an optimum solution for $(k, \alpha)$-DSLO. In the general case, we show empirically that our algorithm can find solutions that are very close to an optimum solution of our problem.

- We present a fast heuristic (FastDSLO) for $(k, \alpha)$-DSLO which, albeit less accurate than MinAndRemove, is able to find dense subgraphs with limited overlap on even larger graphs (containing up to 100 million edges).

- We present a case study in which we compare how well a certain type of graph decomposition, the density friendly decomposition [93], performs for the $(k, \alpha)$-DSLO problem.

In Section 3.2 we define our main problem formally and prove its NP-hardness. All our algorithms are presented in Section 3.3, while Section 3.4 contains an experimental evaluation on large real-world graphs. Finally, in Section 3.5 we draw our conclusions and discuss interesting directions for future work.

## 3.2 Definition and Complexity

In this section, we define our problem formally and we study its computational complexity.

Given an undirected graph $G = (V, E)$, we define its density $\rho(G)$ to be $\frac{|E|}{|V|}$, which corresponds to half the average degree of the nodes in $G$. For a set of vertices $S \subseteq V$, we denote the subgraph of $G$ induced by $S$ as $G(S) = (S, E(S))$, where $E(S) = \{\{u, v\} \in E | u, v \in S\}$).

In a first attempt to give a formal definition of our problem, one could aim at finding at most $k$ subgraphs with maximum aggregate total density. However, it turns out that such a formulation might lead to finding several subgraphs being very similar to each other and in particular sharing a large fraction of nodes of a relatively dense subgraph. Such a solution is not really interesting as the dense subgraphs to be found should ideally exhibit some appreciable degree of diversity among each other. Therefore, we enforce an upper bound $\alpha \in [0, 1]$ on the pairwise Jaccard coefficient between the sets of nodes of the subgraphs, with one indicating that the two subgraphs contain exactly the same nodes and zero indicating that they are disjoint. Our problem can then be formalized as follows.

**Definition 3.2.1** ($(k, \alpha)$-DSLO)**.** Given an undirected graph $G = (V, E)$, an integer $k > 0$, as well as a rational number $\alpha \in [0, 1]$, find a set of sets of vertices $\mathcal{S} = \{S_1, \ldots, S_{\bar{k}}\}$ with $\bar{k} \leq k$, and $S_i \subseteq V, \forall S_i \in \mathcal{S}$, such that

$$\sum_{i=1}^{\bar{k}} \rho(G(S_i))) \quad \text{is maximum, and}$$

$$\frac{|S_i \cap S_j|}{|S_i \cup S_j|} \leq \alpha \quad \forall S_i, S_j \in \mathcal{S}. \tag{3.1}$$

**Theorem 3.2.1.** $(k, \alpha)$-DSLO is NP-hard.

We prove NP-hardness by reducing the well-known maximum independent set problem to a special case of $(k, \alpha)$-DSLO i.e., to the case where $\alpha = 0$. Particularly, we show that given a graph $G$ and a positive integer $k$, it is NP-hard to find $k$ disjoint subsets $S_1, S_2, \ldots, S_k$ of nodes such that the sum of densities $\sum_{i=1}^{k} \rho(S_i)$ is maximized. In particular, we consider a decisional version of the problem, in which we are given a target $\tau$ and the problem is to decide if there are $k$ disjoint subsets whose sum of densities is at least $\tau$.

Our hardness proof reduces from the NP-hardness of the maximum independent set on degree bounded graphs [79]. In particular, for any fixed $\Delta \geq 3$, given a graph $G$ with maximum degree at most $\Delta$ and an integer $k$, it is NP-hard to decide if $G$ contains an independent set of size $k$.

**Reduction Construction.** Given an instance $G = (V, E)$ of the maximum independent set problem with maximum degree at most $\Delta$, we construct a graph $\widehat{G}$ and select a threshold $\tau$ such that $G$ has an independent set of size $k$ *iff* $\widehat{G}$ contains $k$ disjoint subsets whose sum of densities is at least $\tau$.

*Node Gadget.* We choose some $N = n^4$, where $n = |V|$. Given a node $u$, we create a gadget graph $G_u$ as follows. Let $C_u$ be a set of $N$ independent nodes, which forms

the *core* of $G_u$. Let $A_u$ be a set $\Delta$ independent nodes, which are the *arms* of $G_u$. The graph $G_u$ is a complete bipartite graph between $C_u$ and $A_u$, and so contains $N\Delta$ edges.

*Interaction between Node Gadgets.* For distinct nodes $u$ and $v$ in $V$, the cores $C_u$ and $C_v$ are disjoint. If $\{u, v\} \notin E$, then $A_u$ and $A_v$ are disjoint; but if $\{u, v\} \in E$, then $|A_u \cap A_v| = 1$, i.e., the two gadgets $G_u$ and $G_v$ share exactly one arm. Moreover, each arm node can be shared by at most 2 gadget graphs. Since $G$ has degree at most $\Delta$, each arm of a gadget graph can be potentially used to connect with another gadget according to $G$. This completes the description of graph $\widehat{G}$.

Before we state our threshold $\tau$, we consider the densities of subgraphs in $\widehat{G}$.

**Lemma 3.2.1.** The density of any subset of nodes in $\widehat{G}$ is at most $\Delta$. Moreover, if the density of a subset $S$ is at least $\Delta - \frac{1}{2}$, then $S$ must contain all the arms $A_u$ of some $u$.

*Proof.* The result follows from the following statement. Suppose $S$ is a subset of nodes in $\widehat{G}$ that intersects the cores of some $r$ gadgets. Suppose for some $D > 0$, for each $i \in [r]$, $S$ contains $d_i$ arm nodes of gadget graph $G_i$, where $d_i \leq D$. (Observe that each arm node can belong to more than one gadget graph $G_i$.) Then, we show that the density of $S$ is at most $D$.

For $i \in [r]$, suppose $S$ contains $n_i$ nodes from the core $C_i$ of gadget graph $G_i$. The total number of edges induced by $S$ is $\sum_{i\in[r]} n_i d_i$. If none of the $G_i$'s share an arm, the number of nodes in $S$ is $\sum_{i\in[r]}(n_i + d_i)$. Observe that for each pair of $G_i$'s that share an arm, the number of nodes decreases by 1. Since each $G_i$ can share arms with at most $d_i$ other gadgets, the maximum number of pairs that can share an arm is $\frac{\sum_{i\in[r]} d_i}{2}$.

Hence, $|S| \geq \sum_{i\in[r]}(n_i + d_i) - \frac{\sum_{i\in[r]} d_i}{2} = \sum_{i\in[r]}(n_i + \frac{d_i}{2})$.

Therefore, the density of $S$ is at most
$\frac{\sum_{i\in[r]} n_i d_i}{\sum_{i\in[r]}(n_i + \frac{d_i}{2})} \leq \max_{i\in[r]} \frac{n_i d_i}{n_i + \frac{d_i}{2}} \leq \frac{ND}{N + \frac{D}{2}}$, where the last inequality follows because the expression $\frac{n_i d_i}{n_i + \frac{d_i}{2}}$ is increasing in both $n_i$ and $d_i$.

Finally, observing that $\frac{ND}{N + \frac{D}{2}} \leq D$, we finish the proof of the statement. □

The following lemma completes the reduction proof.

**Lemma 3.2.2.** The graph $G$ (with maximum degree at most $\Delta$) contains an independent set of size $k$ *iff* the graph $\widehat{G}$ contains $k$ disjoint subsets whose sum of densities is at least $\tau = k\Delta - \frac{1}{n}$.

*Proof.* The forward direction is easy because each point in the independent set corresponds to a disjoint gadget graph, which has density $\frac{\Delta N}{N+\Delta} \geq \Delta - \frac{1}{n^2}$, because $\Delta < n$ and $N = n^4$. Hence, the sum of the densities of the $k$ disjoint gadget graphs is at least $k\Delta - \frac{1}{n}$.

For the backward direction, observe that each of the $k$ disjoint subsets $S_i$'s in $\widehat{G}$ must have density at least $\Delta - \frac{1}{2}$. Otherwise, there exists $k - 1$ disjoint subsets whose sum of densities is at least $(k-1)\Delta + \frac{1}{2} - \frac{1}{n} > (k-1)\Delta$. This implies that there exists a subset in $\widehat{G}$ with density strictly larger than $\Delta$, which is impossible by Lemma 3.2.1.

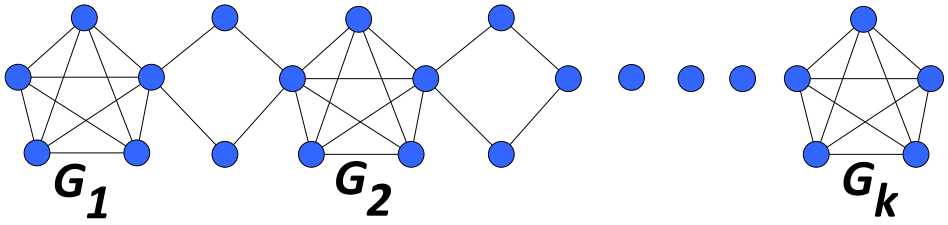


Figure 3.1: A graph $G$ where NAIVE gives poor results. Notice that $G$ is a non-minimal densest subgraph.

Hence, we conclude that each of $k$ disjoint subsets $S_i$'s must have density at least $\Delta - \frac{1}{2}$, which implies by Lemma 3.2.1 that each $S_i$ must contain all the arms of some core $A_u$ for some $u \in V$. This means the $k$ subsets $S_i$'s correspond to $k$ independent nodes in $G$. □

## 3.3 Algorithms

As $(k, \alpha)$-DSLO is NP-hard, we devise heuristics that work well in practice while also exhibiting provable guarantees in some cases of interest. We start by considering one natural heuristic for the disjoint case ($\alpha = 0$): at each step, we compute the densest subgraph in the current graph (using, for example, the approach in [22]), we remove all its nodes and edges from the current graph, iterating until we find exactly $k$ subgraphs or until the current graph contains no edges. We hereinafter refer to this heuristic as NAIVE.

This simple heuristic gives unfortunately very poor results in the worst case, as illustrated in Figure A.1.

In that example the density $\rho(G)$ of the graph is 2 (there are $10k + 4(k-1)$ edges and $5k + 2(k-1)$ nodes), like every subgraph $G_i$ of $G$. NAIVE would find the whole graph $G$ and then would stop giving a solution with a total density equal to 2, while an optimum solution to our problem is composed of the disjoint subgraphs $G_1, \ldots, G_k$ with a total density equal to $2k$. This highlights one of the limitations of NAIVE and paves the way to the definition of minimal dense graphs.

**Definition 3.3.1.** (Minimal dense graphs) An undirected graph $G$ with density $\rho(G)$ is a *minimal dense graph* if for any proper subgraph $H$ of $G$, $\rho(H) < \rho(G)$. Moreover, we say that $G$ is a minimal densest subgraph if it is minimal and has maximum density.

Notice that the graph $G$ in Figure A.1 is a non-minimal densest subgraph.

Finding minimal dense graphs plays an important role in solving our problem, as illustrated by the following variant of NAIVE. At each step we compute a *minimal* densest subgraph, we remove its nodes and edges from the current graph, and we iterate until $k$ subgraphs are found or there are no edges left in the current graph. Including at each step a minimal dense subgraph $H$, rather than a supergraph $G$ of $H$, would not decrease the total density of our solution (as $H$ is as dense as $G$) and might actually increase it, as fewer edges are removed from the current graph

after including $H$ in the solution. It turns out that this simple variant of NAIVE finds $k$ disjoint densest subgraphs if they exist. This is proved in Section 3.3.2. In Section 3.3.1, we show how to efficiently compute minimal densest subgraphs.

Drawing inspiration from the techniques developed for computing minimal densest subgraphs, we then address the $(k, \alpha)$-DSLO problem. The main challenge here is to take full advantage of the overlap between subgraphs so to maximize our objective function. Computing minimal subgraphs is desirable also in this case, in order to minimize the number of edges that are removed at each step. This is discussed in Section 3.3.2. In Section 3.4, we perform an extensive evaluation of our algorithms showing the effectiveness of our heuristics for $(k, \alpha)$-DSLO on large real-world graphs and that minimal densest subgraphs can be computed efficiently on large graphs containing millions of edges.

### 3.3.1 Finding Minimal Densest Subgraphs

Our algorithm for computing minimal densest subgraphs is inspired by the linear programming (LP)-based algorithm for the densest subgraph developed by Charikar [22]. In order to have some provable guarantees for our problem, we need to dive deeper into the structure of the solutions of the LP. We start by recalling the algorithm in [22].

Given a graph $G = (V, E)$, Charikar [22] proposed the following LP formulation for the densest subgraph problem. For each edge $ij \in E$ we introduce a variable $x_{ij}$ taking values in $[0, 1]$, while for each node $v$ we introduce variable $x_v$ (taking values in $[0, 1]$). We have the following linear program:

$$\max \quad \sum_{ij \in E} x_{ij} \tag{BasicLP}$$

$$\text{s.t.} \quad x_{ij} \leq y_i \qquad \forall ij \in E \tag{3.2}$$

$$x_{ij} \leq y_j \qquad \forall ij \in E \tag{3.3}$$

$$\sum_{i \in V} y_i \leq 1 \tag{3.4}$$

$$x_{ij}, y_i \geq 0 \qquad \forall i, j. \tag{3.5}$$

To gain an intuition about the above LP, consider a densest subgraph $H = (S, E(S))$, $S \subseteq V$. We can then define a feasible solution $z^S = (x^S, y^S)$ for BasicLP as follows:

$$x_{ij}^S = \begin{cases} \frac{1}{|S|} & \text{if both } i, j \in S \\ 0 & \text{otherwise.} \end{cases}$$

$$y_i^S = \begin{cases} \frac{1}{|S|} & \text{if } i \in S \\ 0 & \text{otherwise.} \end{cases}$$

Recall that the density of the subgraph $H = (S, E(S))$ is defined as $\rho(H) = \rho(S) = \frac{|E(S)|}{|S|}$. Observe that $z^S$ is feasible for the basic LP, and has objective value $\rho(S)$. Vice versa, given an optimum solution $z^S = (x^S, y^S)$ for BasicLP, one can construct an optimum solution for the densest subgraph problem using the *rounding* algorithm

described in [22]: We first order the $y_i^S$'s by non-increasing order. Let $y_1, \ldots, y_n$ be the variables so ordered. We then find the prefix $y_1, \ldots, y_k$ in such ordering whose corresponding induced subgraph $H$ achieves maximum density, for any value of $k$ in $[2, n]$. It can be proved that $H$ is a densest subgraph.

One can then solve BasicLP using efficient LP solvers such as Gurobi or CPLEX and then compute the densest subgraph by running the algorithm described above. In the rest of this section, we present a more efficient algorithm for computing the densest subgraph, which allows us to deal with large real-world graphs containing millions of edges. The latter algorithm uses the following fact.

**Lemma 3.3.1.** Each optimal solution of BasicLP is a convex combination of points in $\{z^S : S \subseteq V,\ S \text{ is densest subgraph}\}$.

*Proof.* Suppose $z^* = (x^*, y^*)$ is an optimal solution. Then, since the objective value to be maximized is the sum of all $x_{ij}$'s, it follows that if $z^*$ is optimal, it must be the case that $\sum_{i \in V} y_i^* = 1$, and for all $ij \in E$, $x_{ij}^* = \min\{y_i^*, y_j^*\}$.

We prove the result by induction on the number $k$ of non-zero coordinates of $y^*$. If $E$ contains at least one edge, it follows that $k \geq 2$. For the base case $k = 2$, suppose $\{i, j\}$ is the support of $y^*$. Since $x_{ij}^* = \min\{y_i^*, y_j^*\}$ is maximized when $y_i^* = y_j^* = \frac{1}{2}$, the result follows.

For the inductive step, suppose $y^*$ has $k > 2$ non-zero coordinates corresponding to some subset $S \subseteq V$, where $k = |S|$. If all the non-zero $y_i^*$'s are the same for $i \in S$, then it follows that $z^* = z^S$, and the result follows; otherwise, let $\alpha := \min\{y_i^* : i \in S\}$. Observe that $k\alpha \leq 1$.

Define $\widehat{z} = (\widehat{x}, \widehat{y})$ as follows.

$$\widehat{x}_{ij} = \begin{cases} \frac{x_{ij}^* - \alpha}{1 - k\alpha} & \text{if both } i, j \in S \\ 0 & \text{otherwise.} \end{cases}$$

$$\widehat{y}_i = \begin{cases} \frac{y_i^* - \alpha}{1 - k\alpha} & \text{if } i \in S \\ 0 & \text{otherwise.} \end{cases}$$

Hence, $z^* = k\alpha \cdot z^S + (1 - k\alpha)\widehat{z}$, and the number of non-zero coordinates of $\widehat{y}$ is strictly less than $k$. Hence, to complete the inductive step, it suffices to show that $\widehat{z}$ is an optimal solution to the basic LP.

Observe that since the objective function is linear, $p(z^*) = k\alpha \cdot p(z^S) + (1 - k\alpha)p(\widehat{z})$, it is enough to show that $\widehat{z}$ is feasible, because if $p(\widehat{z}) < p(z^*)$, it must be the case that $p(z^S) > p(z^*)$, which violates the optimality of $z^*$.

To check the feasibility of $\widehat{z}$, we have for $ij \in E(S)$, $\widehat{x}_{ij} = \frac{x_{ij}^* - \alpha}{1 - k\alpha} = \frac{\min\{y_i^*, y_j^*\} - \alpha}{1 - k\alpha} = \min\{\widehat{y}_i, \widehat{y}_j\}$.

Moreover, $\sum_{i \in V} \widehat{y}_i = \sum_{i \in S} \frac{y_i^* - \alpha}{1 - k\alpha} = \frac{\sum_{i \in S} y_i^* - k\alpha}{1 - k\alpha} = 1$, because $\sum_{i \in S} y_i^* = 1$. Therefore, $\widehat{z}$ is feasible and this completes the inductive step. □

The following corollary follows from Lemma 3.3.1.

**Corollary 3.3.1.** Suppose $S_1, S_2$ both induce densest subgraphs in $V$. Then, both $S_1 \cap S_2$ and $S_1 \cup S_2$ induce densest subgraphs in $V$.

Another interesting consequence of Lemma 3.3.1 is that we can find a densest subgraph by first solving BasicLP, and then returning the subgraph consisting of

all nodes whose corresponding variables have values strictly larger than zero in our solution.

We now focus on computing a minimal densest subgraph $H$ of a graph $G$ given in input. We recall that any (proper) subgraph of $H$ must have density strictly smaller than $H$.

As we shall use an LP-based approach to find a densest subgraph, we start by showing how to speed up the LP-based algorithm presented in [22]. As proved in Lemma 3.3.1, the subgraph induced by variables with the maximum value in an optimum solution for BasicLP is a densest subgraph. Starting from a solution for BasicLP we can then derive a densest subgraph in $O(n)$, in contrast with the rounding algorithm described in [22], which requires $\Omega(n \log n + m)$ operations, where $n, m$ are the number of nodes and edges in $G$, respectively. To the best of our knowledge, this fact was not known before. We shall refer to this more efficient algorithm for the densest subgraph as FastLP.

Our algorithm for finding minimal dense subgraphs employs two subroutines: TryRemove and TryEnhance. The former one takes as input a graph $G$ and a node $u$ and checks whether $u$ can be removed from $G$ without decreasing its density. This is done by computing a densest subgraph in the input graph $(V \setminus \{u\}, E)$ and checking whether the density drops. If this is not the case, $u$ can be removed from $G$. A pseudocode for TryRemove is shown in Algorithm 1.

**Algorithm 1** TryRemove$(u, G)$

1: **Input:** A graph $G = (V, E)$ and a node $u$ to be removed.
2: **Output:** Returns a densest subgraph in $G$ not containing $u$, or *null* if every densest subgraph in $G$ must contain $u$.
3: Solve the BasicLP with input $(V \setminus \{u\}, E)$ and run FastLP to find a densest subgraph $H$ in $(V \setminus \{u\}, E)$.
4: **if** $\rho(H) \geq \rho(G)$ **then**
5: **return** $H$
6: **else**
7: **return** null
8: **end if**

We could then compute $H$ by iterating through all nodes and checking for each such a node whether it can be removed or not. This algorithm would not be efficient, in that, it requires to solve $\Theta(n)$ LPs, one for each node of $G$. Therefore, we devise a much more efficient algorithm which requires solving $O(\log n)$ LPs, with high probability. Such an algorithm employs the subroutine TryEnhance, which receives in input a graph $G$, a node $u$, as well as density $\rho_{\max}$ of a densest subgraph in $G$. It returns a densest subgraph in $G$ which contains $u$ while having the smallest number of nodes (or null in case there is no densest subgraph containing $u$).

This is achieved by solving a carefully defined LP whose objective is to maximize $y_u$ subject to the constraint that the density is equal to $\rho_{\max}$. The main intuition is that for each densest subgraph in $G$ with $S$ nodes there is a solution to the LP where variables have all values $\frac{1}{|S|}$. Therefore, by maximizing $y_u$ we can find a

densest subgraph containing $u$ with the smallest number of nodes. This is proved in Lemma 3.3.2 and follows partially from Lemma 3.3.1. See Algorithm 2 for a pseudocode of TryEnhance.

**Algorithm 2** TryEnhance$(u, G, \rho_{\max})$

1: **Input:** a graph $G = (V, E)$, a node $u \in V$, the density $\rho_{\max}$ of the densest subgraph in $G$.
2: **Output:** Returns a densest subgraph in $G$ containing $u$ with minimum cardinality or *null* if there is no densest subgraph containing $u$.
3: Modify the basic LP by adding the constraint $\sum_{ij \in E} x_{ij} = \rho_{\max}$, maximizing the objective function $y_u$; solve the modified LP by running FastLP.
4: **If** there is no feasible solution to the modified LP **then** return *null*.
5: Run FastLP to find a densest subgraph $H = (\bar{V}, \bar{E})$ starting from the LP solution.
6: **return** $H$

**Lemma 3.3.2.** Given a graph $G = (V, E)$ and a node $u \in V$, the subroutine TryEnhance returns a densest subgraph in $G$ containing $u$ with the smallest number of nodes.

*Proof.* The smallest densest subgraph containing $u$ is unique, because by Corollary 3.3.1, the intersection of all densest subgraphs containing $u$ is also a densest subgraph.

Consider the optimal solution $z = (x, y)$ computed from the modified LP in the subroutine TryEnhance$(u, G, \rho_{\max})$. By Lemma 3.3.1, $z$ is a convex combination of $z^{S_i}$'s, where each $S_i$ induces a densest subgraph in $G$. Since the objective is to maximize $y_u$, and $z^S$ is a feasible solution, it follows that $y_u$ is positive, which means that at least one of the $S_i$'s must contain $u$. Since for each $S_i$ containing $u$, $y_u^{S_i} = \frac{1}{|S_i|}$, it follows that if $z$ is a convex combination of more than one $S_i$'s, the value $y_u$ could be strictly improved by $z^{\overline{S}}$, where $\overline{S}$ is the intersection of all $S_i$'s containing $u$.

Hence, it follows that $z = z^{\overline{S}}$ for some densest subgraph induced by $\overline{S}$, which has to be the smallest densest subset containing $u$. □

The main steps for finding efficiently a minimal densest subgraph are the following ones. At each iteration: 1) we pick one node $u$ from our current graph, uniformly at random; 2) we execute the subroutines TryRemove and TryEnhance with input $u$ and our current graph; 3) our current graph is then set to be the smallest subgraph among the ones returned by the previous subroutines. It turns out that $2\log_{\frac{4}{3}} n$ iterations suffice to find a minimal densest subgraph, with high probability. This is proved in Corollary 3.3.3, while Lemma 3.3.3 proves that our algorithm finds a minimal densest subgraph.

In order to speed up even further our algorithm, we include a preprocessing phase where we remove nodes not belonging to any densest subgraph. This is done as follows. We first run the linear-time greedy algorithm presented by Charikar in [22] so to find a 2-approximation solution to the densest subgraph. Let $\rho_{apx}$ be the density of the graphs so found. As noted in [49], no nodes with degree

smaller than the density of the densest subgraph belongs to any densest subgraph. Therefore, we can safely remove all nodes with degree smaller than $\rho_{apx}$ from the input graph. A pseudocode of our algorithm is shown in Algorithm 3.

**Algorithm 3** FINDMINIMAL($G$)

**Input:** A graph $G = (V, E)$ with $n$ nodes.
**Output:** A minimal densest subgraph in $G$.
Run the greedy algorithm to find a 2-approximation solution for the densest subgraph. Let $\rho_{apx}$ be the density of such a subgraph.
Remove iteratively nodes with degree smaller than $\rho_{apx}$ and let $\bar{G}$ be the graphs so obtained.
Find a densest subgraph $H = (\bar{V}, \bar{E})$ in $\bar{G}$ by running FASTLP. Let $\rho_{\max}$ be its density.
**while** (true) **do**
    **let** $u$ be a node picked uniformly at random from $\bar{V}$
    **let** $H_1 \coloneqq$ TRYREMOVE$(u, H)$
    **let** $H_2 \coloneqq$ TRYENHANCE$(u, H, \rho_{\max})$
    **IF** $H_1$ is *null* **return** $H_2$
    **let** $\widehat{H}$ be the subgraph with minimum number of nodes between $H_1$ and $H_2$, breaking ties arbitrarily
    $H \coloneqq \widehat{H}$
**end while**
**return** $H$

**Lemma 3.3.3.** FINDMINIMAL($G$) returns a minimal densest subgraph of $G$.

*Proof.* Algorithm 3 always terminates. After each iteration of the *while* loop, either TRYREMOVE$(u, H)$ returns *null* and therefore the algorithm terminates or the node $u$ is removed from the current graph. Hence, at each iteration, the number of nodes of the current graph decreases by at least one. Observe that the graph $H$ is always a densest subgraph, throughout the execution of the algorithm. When the algorithm terminates, it must be the case that TRYREMOVE$(u, H)$ returns *null*, for some node $u$ in $H$. This means that every densest subgraph of $H$ must contain $u$. By Lemma 3.3.2, TRYENHANCE$(u, H, \rho_{\max})$ returns the smallest densest subgraph containing $u$, and so it must be minimal. □

We prove that $O(\log n)$ iterations of the *while* loop of FINDMINIMAL($G$) suffice to find a minimal densest subgraph in $G$. To this end, we show that at each iteration the number of nodes in the current subgraph decreases by a constant fraction with constant probability. A standard measure concentration argument concludes the proof. Let $H = (\bar{V}, \bar{E})$ be a densest subgraph of $G$. For a node $u \in \bar{V}$, $\epsilon > 0$, we say that $u$ is $\epsilon$-bad in $H = (\bar{V}, \bar{E})$ if both $H_1 \coloneqq$ TRYREMOVE$(u, H)$ and $H_2 \coloneqq$ TRYENHANCE$(u, H, \rho_{\max})$ contain more than $(1 - \epsilon)|\bar{V}|$ nodes, and neither of them is *null*.

**Lemma 3.3.4.** Given a graph $H = (\bar{V}, \bar{E})$ with maximum density $\rho_{\max}$, the fraction of $\epsilon$-bad nodes in $\bar{V}$ is at most $2\epsilon$, for any $\epsilon > 0$.

**Algorithm 4** FINDALLMINIMAL($V$)

1: **Input:** Graph $G = (V, E)$.
2: **Output:** Returns a list $L$ of all minimal densest subgraphs.
3: $L := \varnothing$
4: **while** (true) **do**
5: $\widehat{H}$ = FINDMINIMAL($G$)
**IF** $\widehat{H}$ is not a densest subgraph **break**;
6: $L := L \cup \{\widehat{H}\}$
7: remove all nodes and edges incident to $\widehat{H}$ from $G$
8: **end while**
9: **return** $L$

*Proof.* For contradiction's sake, suppose the set $B$ of $\epsilon$-bad nodes has size more than $2\epsilon|\bar{V}|$. For each $u \in B$, define $A_u := \bar{V} \setminus \text{TRYREMOVE}(u, H)$. Observe that $u \in A_u$, and by the definition of $\epsilon$-bad, $|A_u| < \epsilon|\bar{V}|$.

Consider an arbitrary order of $B := \{u_1, u_2, \ldots, \}$. Since $B$ contains more than $2\epsilon|\bar{V}|$ points, and each $|A_u| < \epsilon|\bar{v}|$, there exists a smallest $i$ such that $\epsilon|\bar{V}| \leq |\cup_{j=1}^{i} A_{u_j}| < 2\epsilon|\bar{V}|$.

Observe that there exists $\widehat{u} \in B \setminus (\cup_{j=1}^{i} A_{u_j})$. Since for each $j$, TRYREMOVE$(u_j, H)$ is a densest subgraph, then by Lemma 3.3.1, $\cap_{j=1}^{i}$TRYREMOVE$(u_j, H) = \bar{V} \setminus (\cup_{j=1}^{i} A_{u_j})$ induces a densest subgraph that contains $\widehat{u}$ and has size at most $(1-\epsilon)|\bar{V}|$. Therefore, it follows that the algorithm TRYENHANCE$(\widehat{u}, H, \rho\max)$ will return a densest subgraph that has size at most $(1-\epsilon)|\bar{V}|$, thereby contradicting that $\widehat{u}$ is $\epsilon$-bad. □

By taking $\epsilon = \frac{1}{4}$ in Lemma 3.3.4, we have the following corollary.

**Corollary 3.3.2.** At each iteration of FINDMINIMAL($G$), the algorithm either terminates with a minimal densest subgraph, or with probability at least $\frac{1}{2}$, the number of nodes in $\widehat{H}$ is at most $\frac{3}{4} \cdot |\bar{V}|$.

**Corollary 3.3.3.** By standard Chernoff bound, the number of iterations in FINDMINIMAL($G$) is at most a constant times its mean, which is at most $2\log_{\frac{4}{3}} n$, where $n$ is the number of nodes in $G$.

Our algorithm FINDMINIMAL($G$) allows us to compute minimal densest subgraphs in graphs containing millions of edges. In particular, the pruning step and the fact that we can bound the number of iterations by a logarithmic function allows us to save several order of magnitudes in terms of running time.

#### 3.3.1.1 Finding All Minimal Densest Subgraphs

Armed with an efficient algorithm for finding minimal densest subgraphs, we now present an algorithm that computes all such graphs. Our algorithm computes at each step a minimal densest subgraph by running Algorithm 3, it removes all its nodes and edges from the current graph and iterates until no densest subgraph can be found. A pseudocode is shown in Algorithm 4.

Minimal densest subgraphs must be disjoint, for otherwise, their intersection would be a densest subgraph (from Corollary 3.3.1), which would contradict the fact that they are minimal. Lemma 3.3.5 then follows.

**Lemma 3.3.5.** Given an undirected graph $G = (V, E)$, our algorithm FindAllMinimal($V$) finds all minimal densest subgraphs in $G$.

### 3.3.2 Main Algorithms

Drawing inspiration from the theory and the techniques developed in the previous section for finding minimal densest subgraphs, we devise one algorithm for our main problem $(k, \alpha)$-DSLO. In this case, we face the additional challenge of taking full advantage of the overlap between the subgraphs.

Our algorithm is inspired by FindMinimal and proceeds as follows. At each step $i$, we compute a minimal densest subgraph $G_i = (V_i, E_i)$ of our current graph $G = (V, E)$ and we remove $\lceil (1-\alpha)|V_i| \rceil$ nodes (and their edges) from $V$. We remove those nodes that are not well connected with nodes outside $G_i$, as they will contribute less to the total density in the next steps of the algorithm. Formally, for any node $v$ in $V_i$ let $\Delta_G(v)$ be the set of neighbors of $v$ in $G$. We remove the $\lceil (1-\alpha) \rceil |V_i|$ nodes (and their edges) with minimum value $|\Delta_G(v) \setminus V_i|$. We iterate until $k$ subgraphs are found or the current graph becomes empty. Observe that the constraint on the Jaccard coefficient is not violated. A pseudocode of our algorithm is shown in Algorithm 15.

We can prove an interesting property of MinAndRemove. Observe that if there are $k$ disjoint densest subgraphs in $G$ then MinAndRemove would return the same solution of FindAllMinimal. Then, our main theorem follows from Lemma 3.3.5.

**Theorem 3.3.1.** If there are $k$ disjoint densest subgraphs in the input graph $G$, then MinAndRemove computes an optimum solution for $(k, \alpha)$-DSLO, for any $\alpha \geq 0$.

Although, we cannot give any guarantee for the general case of $(k, \alpha)$-DSLO (i.e. for any $\alpha \geq 0$) we show the effectiveness of our main algorithm in our experimental evaluation. In particular, we are able to derive an upper bound on any optimum solution for $(k, \alpha)$-DSLO and show that MinAndRemove is very close to such an upper bound in our experiments (up to a factor of 0.9).

Finally, we present a fast heuristic FastDSLO for finding dense subgraphs with limited overlap. Our heuristic computes at each step $i$ a 2-approximation solution $G_i$ to the densest subgraph problem using the greedy algorithm presented in [22]. Then, similarly to MinAndRemove $\lceil (1-\alpha) \rceil |V_i|$ nodes are removed from the current graph so to satisfy the requirement on the pairwise Jaccard coefficient. A pseudocode is shown in Algorithm 16. Our experimental evaluation shows that although our heuristic is less accurate than MinAndRemove, it is still not too far from an optimum solution while it can handle graphs with more than 100 million edges within a few hours.

**Algorithm 5** MinAndRemove$(G, k, \alpha)$

**Input:** A graph $G = (V, E)$, an integer $k > 0$, $\alpha \in [0, 1]$
**Output:** A list $L$ of at most $k$ subgraphs of $G$, $G_i = (V_i, E_i)$, s.t. the constraint on the pairwise Jaccard coefficient on the $V_i$'s is not violated (Equation (A.1)).
$L := \emptyset$
**while** $< k$ subgraphs are found and $G$ is not empty **do**
    Find a minimal densest subgraph $G_i = (V_i, E_i)$ of $G$ by running Algorithm 3
    $L := L \cup \{G_i\}$
    For each node $v$ in $V_i$, let $\Delta_G(v)$ be the set of neighbors of $v$ in $G$.
    Remove the $\lceil (1-\alpha)|V_i| \rceil$ nodes with minimum value $|\Delta_H(v) \setminus V_i|$ and all their edges from $G$.
**end while**
**return** L

**Algorithm 6** FastDSLO$(G, k, \alpha)$

**Input:** A graph $G = (V, E)$, an integer $k > 0$, $\alpha \in [0, 1]$
**Output:** A list $L$ of at most $k$ subgraphs of $G$, $G_i = (V_i, E_i)$, s.t. the constraint on the pairwise Jaccard coefficient on the $V_i$'s is not violated (Equation (A.1)).
$L := \emptyset$
**while** $< k$ subgraphs are found and $G$ is not empty **do**
    Find a 2-approximation solution $G_i = (V_i, E_i)$ to the densest subgraph problem by running the greedy algorithm in [22].
    $L := L \cup \{G_i\}$
    For each node $v$ in $V_i$, let $\Delta_G(v)$ be the set of neighbors of $v$ in $G$.
    Remove the $\lceil (1-\alpha)|V_i| \rceil$ nodes with minimum value $|\Delta_H(v) \setminus V_i|$ and all their edges from $G$.
**end while**
**return** L

## 3.4 Experiments

We perform our experimental evaluation on a Linux server with Intel Xeon E7-4870 at 2.40GHz, while limiting the total amount of main memory available to 64 GB. We solve linear programs with the Gurobi Optimizer version 5.6.3. All our algorithms are implemented in Java.

We consider 8 datasets in total grouped according to their size[1]. We ignore the direction of the edges in the directed graphs, as it is irrelevant to our purposes. Most of our experiments are conducted on the datasets that contain up to 11 million edges, as illustrated in Table 3.1. We refer to this set of datasets as *large* datasets.

The second group of datasets consists of *very large* datasets containing up to more than 100 million edges where we evaluate our fast heuristic. In Table 3.2 we

[1]Sources: snap.stanford.edu , konect.uni-koblenz.de, law.di.unimi.it/datasets.php

| Name | Nodes | Edges | Description |
|---|---|---|---|
| web-Stanford | 281K | 1.9M | Hyperlink network |
| com-Youtube | 1.1M | 3M | Social network |
| web-Google | 875K | 4.3M | Hyperlink network |
| Youtube-growth | 3.2M | 9.3M | Social network |
| As-Skitter | 1.69M | 11M | Internet topology |

Table 3.1: Large real-world datasets.

specify all the details for this group of datasets. We refer to this set of datasets as *very large* datasets.

| Name | Nodes | Edges | Description |
|---|---|---|---|
| Live Journal | 4.8M | 43M | Social network |
| Hollywood-2009 | 1.1M | 57M | Social network |
| Orkut | 3M | 117M | Social network |

Table 3.2: Very large real-world datasets.

**Densest Subgraph**. We first present in Table 3.3 the running time of several algorithms for computing the densest subgraph or an approximation: the 1/2 densest subgraph approximation algorithm [22] (1/2**DS**), the max-flow based algorithm for the densest subgraph [43] (MF**DS**), the linear programming algorithm for the densest subgraph [22] (LP**DS**), the minimal densest algorithm [13] (Min**DS**). For the experiments we used the code of [32] and [13].

| Name | 1/2**DS** | MF**DS** | LP**DS** | Min**DS** |
|---|---|---|---|---|
| web-Stanford | 6s | 22s | 49s | 260s |
| com-Youtube | 25s | 33s | 293s | 405s |
| web-Google | 27s | 47s | 38s | 43s |
| Youtube-growth | 69s | 80s | 180s | 631s |
| AS-Skitter | 52s | 107s | 87s | 246s |

Table 3.3: Running time for different algorithms for computing densest on large real-world datasets.

| Name | 1/2**DS** | MF**DS** |
|---|---|---|
| Live Journal | 456s | 993s |
| Hollywood-2009 | 398s | 5349s |
| Orkut | 470s | 2868s |

Table 3.4: Running time for different algorithms for computing densest on very large real-world datasets.

We notice that although stated in [13] that the maximum flow based algorithm for the densest subgraph [43] is very slow, we manage to significantly improve the

performance by using the Boost implementation of the Boykov-Kolmogorov max-flow algorithm [32]. The algorithm in [43] obtains better running time on the large datasets as shown in Table 3.3, and as shown in Table 3.4 it can scale to the very large datasets. An interesting future work would be to adapt Golberg's [43] algorithm for finding minimal dense subgraphs.

**Top $k$ dense subgraphs**. We evaluate our algorithm MinAndRemove against two variants of the Naive algorithm. In the first variant, we compute at each step a densest subgraph with the LP-based approach proposed in [22], we remove all its nodes and edges from the graph, and we iterate until $k$ subgraphs have been found or the graph has become empty. We refer to this algorithm as NaiveDensest. In the second variant, we compute at each step the $\frac{1}{2}$-approximation algorithm proposed in [22] instead of a densest subgraph. We refer to this algorithm as NaiveGreedy. The reason to consider also this second variant is that the $\frac{1}{2}$-approximation algorithm is much faster than the LP-based optimal approach.

We tested NaiveDensest on the large datasets and it did not terminate after 16 hours of computation on any dataset. Therefore, in the rest of the experimental evaluation, we focus on evaluating MinAndRemove against NaiveGreedy.

We start by measuring the total density when $k = 10$ while varying $\alpha$ between 0.1 and 0.5. Let $\rho_{\max}$ be the density of a densest subgraph in the input graph. Observe that $k \cdot \rho_{\max}$ gives us an upper bound on the value of any optimum solution for $(k, \alpha)$-DSLO, for any value of $\alpha$. Therefore, we can use such an upper bound to give an idea of how close our results are to an optimum solution. Given that we use only an upper bound, clearly, our results might be even closer to the actual optimum solution.

In Table 3.5, we measure the ratio between the total density computed by our main algorithm and our upper bound, when $k = 10$. We can see that in all cases our algorithm yields a solution that is at least within a factor of 0.44 of the value of an optimum solution, while in many cases it yields an approximation factor of 0.8 (meaning that it reaches the 80% of the upper bound on the optimum objective-function value). We can also see that the quality of the solution increases as a function of $\alpha$ showing that our algorithm takes full advantage of the overlap between the subgraphs. This is not the case for one dataset only ($web-Google$), however, we observe that the results are already very good for this dataset when $\alpha = 0.1$, making it harder to improve upon such a solution. We also recall that our upper bound might be loose and that we might have computed an optimum solution for such a dataset.

Next, we evaluate our algorithm MinAndRemove against NaiveGreedy, when $k = 10$. Table 3.6 shows the ratio between the total density of the subgraphs found by our algorithm and those of NaiveGreedy. We can see that in most cases MinAndRemove yields a solution that is a factor of 1.5 larger than that of MinAndRemove, and always at least 10% denser. We expect that the advantage of MinAndRemove against NaiveGreedy would be even more remarkable for larger values of $k$. Table 3.7 shows the total running time of our algorithm, which is always at most 3.2 hours, while in many cases is as less as 30 mins. We recall that instead the basic LP-based approach by Charikar [22] could not terminate after 16

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| web-Stanford | .717 | .738 | .767 | .790 | .816 |
| com-Youtube | .480 | .521 | .518 | .613 | .623 |
| web-Google | .808 | .808 | .808 | .808 | .808 |
| Youtube-growth | .440 | .467 | .538 | .593 | .579 |
| As-Skitter | .585 | .597 | .599 | .625 | .647 |

Table 3.5: Ratio between the density of MinAndRemove and our upper bound

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| web-Stanford | 1.469 | 1.512 | 1.570 | 1.617 | 1.671 |
| com-Youtube | 1.192 | 1.293 | 1.287 | 1.522 | 1.547 |
| web-Google | 1.595 | 1.595 | 1.595 | 1.595 | 1.595 |
| Youtube-growth | 1.2 | 1.271 | 1.467 | 1.617 | 1.578 |
| As-Skitter | 1.125 | 1.147 | 1.151 | 1.202 | 1.244 |

Table 3.6: Ratio between the density of MinAndRemove and NaiveGreedy

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| web-Stanford | 0.3h | 0.21h | 0.21h | 0.23h | 0.21h |
| com-Youtube | 0.54h | 0.54h | 0.63h | 0.55h | 0.47h |
| web-Google | 2.15h | 2.31h | 2.31h | 2.12h | 2.73h |
| Youtube-growth | 1.5h | 1.74h | 2.19h | 1.8h | 3.13h |
| As-Skitter | 1.29h | 1.47h | 2.85h | 2.53h | 1.78h |

Table 3.7: Running time when varying $\alpha$ for MinAndRemove

| $\alpha = 0.3$ | $k = 2$ | $k = 4$ | $k = 6$ | $k = 8$ | $k = 10$ |
|---|---|---|---|---|---|
| web-Stanford | .991 | .914 | .858 | .808 | .767 |
| com-Youtube | .840 | .744 | .638 | .570 | .518 |
| web-Google | .991 | .915 | .863 | .836 | .808 |
| Youtube-growth | .845 | .738 | .664 | .593 | .538 |
| As-Skitter | .914 | .783 | .693 | .641 | .599 |

Table 3.8: Ratio between the density of MinAndRemove and our upper bound

hours of computation on any of the selected datasets. On the other hand, as expected, NaiveGreedy is faster than our method. However, NaiveGreedy does not ensure optimality in finding the densest subgraph at each step, and this results in consistently less dense solutions.

In Table 3.8, we set $\alpha = 0.3$ and we measure the ratio between the density of MinAndRemove and an upper bound to any optimum solution as a function of $k$. We can see that when $k$ is at most 4 our solution is very close to an optimum solution for $(k, \alpha)$-DSLO (up to a factor of 0.9). For larger values of $k$, our solution is still within a factor of 1/2 of an optimum solution or better. This might depend on the fact that our upper bound becomes loose when $k$ is large.

We then evaluate the impact of computing at each step minimal densest sub-

graphs in MinAndRemove, as opposed to computing densest subgraphs. We perform the following experiment. Starting from the input graph, we remove at each step minimal densest subgraphs until the current graph is left with no edges. We then perform a similar experiment where at each step (possibly non-minimal) densest subgraphs are removed from the current graph. Our experiments show (which are omitted for lack of space) that in the former case we gain a factor of 1% (on average) in the total density, which is significant given that MinAndRemove might deliver near-optimal solutions. We also observe that all our datasets contain at most one densest subgraph. This fact could not be verified prior to our work.

Our experimental evaluation shows that MinAndRemove can handle large graphs containing up to more than 10 million edges within 3 hours or less while delivering near-optimal solutions for our problem. For even larger graphs we resort to our fast heuristic which is evaluated on graphs containing up to more than 100 million edges. Similarly to the previous case, we can derive an upper bound on the optimum solution as a function of the densest subgraph found by our heuristic. Namely, let $\rho_A$ be the density of the densest subgraph found by FastDSLO (which we recall is a 2-approximation for the densest subgraph problem). Then, $2k \cdot \rho_A$ gives an upper bound to any optimum solution for $(k, \alpha)$-DSLO.

In Table A.2, we set $k = 10$ while we measure the ratio between the density of FastDSLO and an upper bound to any optimum solution, as a function of $\alpha$. Although our heuristic turns out to be less accurate than MinAndRemove it delivers solutions with an approximation factor of around 0.2 or more, while it can handle very large graphs with more than 100 million edges within a few hours. We also observe that our upper bound might be even looser in this case, as it is based on an approximation of the density of a densest subgraph. Table 3.10 show the running time of FastDSLO when $k = 10$ and $\alpha$ varies between 0.1 and 0.5. We can observe that the running time is around half an hour for the smaller datasets and does not exceed 2.3 hours on the largest dataset (Orkut).

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| LiveJournal | .244 | .245 | .251 | .285 | .276 |
| Hollywood-2009 | .187 | .190 | .199 | .210 | .231 |
| Orkut | .187 | .200 | .218 | .249 | .271 |

Table 3.9: Ratio between the density of FastDSLO and our upper bound

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| LiveJournal | 0.56h | 0.52h | 0.63h | 0.54h | 0.54h |
| Hollywood-2009 | 0.34h | 0.3h | 0.33h | 0.37h | 0.34h |
| Orkut | 1.97h | 2.15h | 1.2h | 2.16h | 2.26h |

Table 3.10: Running time of FastDSLO on very large datasets as a function of $\alpha$

### 3.4.1 Case Study: Finding a Graph Decomposition

Recent works have focused on finding new graph decompositions, one example being the density-friendly decomposition presented in [93]. The density-friendly decomposition obtains a set of overlapping subgraphs $H_1, H_2, H_3 \dots H_m$ where $H_1 \subset H_2 \subset H_3 \dots H_m$, and the nested subgraphs have an increasing average degree, that is $\rho(H_k) > \rho(H_{k+1}), 1 \leq k < m$. Due to the objective function that is used for obtaining the decomposition, we speculate that the subgraphs induced by $H_1, H_2 \setminus H_1, H_3 \setminus H_2, \dots, H_m \setminus H_{m-1}$ are dense. We investigate if a density-friendly decomposition can be a good heuristic for obtaining the top $k$ dense subgraphs. The advantage of using a graph decomposition would be the removal of the factor $k$ in the running time, which is necessary for an iterative algorithm that computes the top $k$ dense subgraph.

In [93] the authors present two algorithms for computing a density-friendly decomposition: an exact algorithm and a 1/2-approximation. The exact algorithm is based on the max flow algorithm for computing the densest [43] and the approximate version on the 1/2-approximation algorithm [22]. We will use the notations ExactLD and GreedyLD for these two algorithms and we will compare their performance with the one of our algorithms, MinAndRemove and FastDSLO. Due to the diversity in solving methods and objective functions, we can get different numbers of subgraphs for each of the four algorithm on a certain dataset. We deal with this limitation by showing results for the top $k = 1, 10, 20, 30, 40, 50$, when these values exist. For MinAndRemove and FastDSLO, k is an input parameter and for ExactLD and GreedyLD, we get from the decomposition the top k dense subgraphs by sorting in non-increasing order by their density the subgraphs $H_1, H_2 \setminus H_1, H_3 \setminus H_2, \dots, H_m \setminus H_{m-1}$ and selecting the first $k$ subgraphs. For the experiments we set overlap between subgraphs to be the empty set, that is $\alpha = 0$.

| Name | k=1 | k=10 | k=20 | k=30 | k=40 | k=50 |
|---|---|---|---|---|---|---|
| web-Stanford | 59.22 | 225.16 | 273.37 | 307.09 | 333.85 | 357.06 |
| com-Youtube | 45.57 | 75.98 | 84.39 | 91.13 | 96.99 | 102.39 |
| web-Google | 27.17 | 133.98 | 200.56 | 241.24 | 272.54 | 297.53 |
| AS-Skitter | 89.14 | 277.68 | 237.69 | 305.64 | 326.17 | 359.19 |

Table 3.11: GreedyLD, total density for various $k$.

| Name | k=1 | k=10 | k=20 | k=30 | k=40 | k=50 |
|---|---|---|---|---|---|---|
| web-Stanford | 58.96 | 306.33 | 372.38 | 403.81 | 421.70 | 430.30 |
| com-Youtube | 45.56 | 184.09 | 212.62 | 226.07 | 232.85 | 238.73 |
| web-Google | 26.95 | 140.05 | 179.70 | 190.78 | 196.51 | $\infty$ |
| AS-Skitter | 88.70 | 475.25 | 615.52 | 684.364 | 713.64 | 731.91 |

Table 3.12: FastDSLO, total density for various $k$.

| Name | k=1 | k=10 | k=20 | k=30 | k=40 | k=50 |
|---|---|---|---|---|---|---|
| web-Stanford | 59.39 | 381.92 | 619.25 | 804.82 | 964.50 | 1114.09 |
| com-Youtube | 45.59 | 160.46 | 207.23 | 245.77 | 275.90 | 301.16 |
| web-Google | 28.04 | 226.02 | 401.55 | 564.76 | 719.38 | 868.67 |
| AS-Skitter | 89.40 | 461.79 | 678.73 | 839.79 | 980.06 | 1108.22 |

Table 3.13: ExactLD, total density for various $k$.

| Name | k=1 | k=10 | k=20 | k=30 | k=40 | k=50 |
|---|---|---|---|---|---|---|
| web-Stanford | 59.39 | 384.25 | 624.82 | 814.68 | 981.19 | 1138.34 |
| com-Youtube | 45.59 | 204.89 | 288 | 356.32 | 414.91 | 465.65 |
| web-Google | 28.04 | 226.02 | 401.96 | 567.40 | 726.52 | 879.99 |
| AS-Skitter | 89.40 | 514.98 | 800.62 | 1020.77 | 1208.31 | 1376.85 |

Table 3.14: MinAndRemove total density for various $k$.

We can see in Table 3.11 and Table 3.12 that FastDSLO performs better in terms of density, except for the dataset web-Google, where GreedyLD has an overall better density. The performance of the exact decomposition ExactLD is better than the performance of FastDSLO, as we can see in Table 3.12 and Table 3.13. The algorithm MinAndRemove obtains subgraphs with a larger sum of densities than ExactLD as shown in Table 3.13 and Table 3.14, which proves that MinAndRemove is a good heuristic for the top $k$ dense subgraphs problem. However, the total density obtained by the density-friendly algorithm is competitive and could provide an alternative when the running time is more important or when both running time and density are important, as the exact decomposition fairs much better than the fast top $k$ algorithm FastDSLO in term of density and maintains a good running time. The greedy heuristic,GreedyLD, has linear running time and the exact algorithm, ExactLD, runs in $O(n^2m)$, where n is the number of nodes and m the number of edges. The algorithm FastDSLO runs in $O(k(m+n))$ while MinAndRemove in $O(kn^3\log(n))$, where the subroutine for computing the densest uses the min-cut algorithm presented in [43]. The worst case running time for the ExactLD algorithm is better than the worse case running time for MinAndRemove, and in practice, it requires only $124s$ to compute the decomposition for a large dataset as As-Skitter. Another important aspect which remains to be investigated is the gain obtained by allowing overlap. We note that the density-friendly decomposition incorporates the overlap in the objective function, which could be an added advantage. We conclude that the density-friendly graph decomposition can be a useful heuristic for computing the top-$k$ subgraphs with maximum sum of densities.

## 3.5 Conclusions

This chapter studies the problem of finding at most $k$ subgraphs from a large graph given in input such that the total density be maximized while satisfying a constraint on the pairwise Jaccard coefficient between the subgraphs. Although very natural,

this variant of the densest subgraph problem has been surprisingly neglected so far.

After showing the NP-hardness of this problem (even when the subgraphs are disjoint), we develop an algorithm that computes an optimum solution for our problem in the case when there are $k$ disjoint densest subgraphs in the input graph. Moreover, our experimental evaluation on large real-world graphs shows that our algorithm delivers near-optimal solutions, in that, they are very close to an upper bound on any optimum solution. We presented an efficient heuristic for our problem which, albeit less accurate, can handle graphs containing up to 100 million edges.

We will devote our future investigation to adapting the max-flow algorithm [43] to the task of finding a minimal densest subgraph and we will study the advantages of using graph decompositions for the task of finding the top-$k$ dense subgraphs. Another interesting direction is to determine whether finding minimal densest subgraphs can be a valuable tool in finding interesting patterns in social networks and other real-world graphs.

# Chapter 4

# Dense Subgraphs with Size Constraints

## 4.1 Introduction

In the previous chapter, we studied the problem of finding $k$ dense subgraphs with limited overlap and maximum sum of densities. However, for some applications such as event detection, densest subgraphs might be large and difficult to analyze. In order to cope with this limitation, we consider the problem of finding dense subgraphs under size restriction. In particular, given a weighted graph, we wish to find a subgraph with $k$ nodes and maximum total edge weight. This problem is referred in the literature as the heaviest k-subgraph problem (HkS) and also as the weighted version of the densest $k$-subgraph problem. The problem is NP-hard and difficult to approximate. In our work, we leverage the properties of real-world graphs, so as to develop efficient algorithms.

We summarize our contributions as follows.

- We develop an efficient (exact) branch and bound algorithm to solve the heaviest k-subgraph problem. Our algorithm scales to large weighted real-world networks, for $k$ up to 15 or more depending on the structure of the graph. We also develop an approximated version of our algorithm scaling to even larger values of $k$. We show that our algorithms are more effective than state-of-the-art heuristics for the same problem.

- We show that our algorithm is well suited as a subroutine for solving related problems, like finding the top $t$ H$k$S in a graph.

The rest of the chapter is organized as follows. In Section 4.2 we define formally the problems we will attempt to solve, then in Section 4.3, we present our algorithms for solving the problems. We then evaluate the performance of our algorithm in Section 4.4. Finally, we conclude and present future work in Section 4.5.

## 4.2 Problem Definition

In this section, we give a formal definition of the problems we study in our work. We also introduce necessary notations and definitions that will be used in the rest of the chapter. We will assume that we are given an undirected graph $G = (V(G), E(G))$ and a weight function $w : E(G) \to R^+$.

**Problem definition** (*Heaviest-k-Subgraph problem*). Given an undirected weighted graph $G$ and an integer $k > 1$, we wish to find a subgraph containing $k$ nodes and such that the sum of the weights its edges is maximum. For this problem we assume the graph has at least $k$ nodes.

In order to treat the problem for large $k$ we also define an approximate version of the problem.

**Problem definition** (*Heaviest-k-Subgraph $\alpha$-approximation problem*). Given an undirected weighted graph $G$, an integer $k > 1$ and a real number $\alpha \geq 1$, we wish to find a subgraph containing $k$-nodes and such that the sum of the weights on its edges times $\alpha$ is greater or equal to the sum of the weights on the edges of any subgraph of size $k$.

**Problem definition** (*Top $t$ heavy $k$-subgraphs problem*). Given an undirected weighted graph $G$, an integer $k > 1$, and an integer $t > 0$, find at most $t$ disjoint subgraphs containing $k$ nodes such that the sum of the weights on its edges is maximum. For this problem we assume the graph has at least $k \cdot t$ nodes.

Decomposing an unweighted graph into a hierarchical structure via the core decomposition is a standard operation in any modern graph-mining toolkit. This decomposition, is based on a recursive pruning of a vertex of minimum degree and is used as a subroutine in a large variety of algorithms, in particular it is related to the problem of finding densest subgraph (in unweighted graph and without constraints on the number of nodes in the subgraph) as it leads to a 2-approximation algorithms. It can be straightforwardly generalized to the weighted case where we recursive prune a vertex such that the sum of the weights on its adjacent edges is minimum.

**Problem definition** (*Weighted core decomposition problem*). Given an undirected weighted graph $G$, we wish to compute a weighted core decomposition of $G$.

We will show that an efficient heuristic for the Heaviest-k-Subgraph problem can be derived from the Weighted core decomposition. However, this solution has no fixed parameter approximation guaranteed for our setting.

## 4.3 Algorithms

### 4.3.1 Branch and Bound Algorithm for HkS

Branch and bound is a well-known method for solving combinatorial maximization problems. The main idea is to divide the search space into several branches of computation. Intuitively, we can think of the whole process as forming a tree starting from the root which is the set of all possible solutions, while the children of a node

are smaller sets of solutions. Forming the children of a node is called the *branching phase*. Each node of the tree is associated with a lower bound (generally a solution of the problem) and an upper bound, while if the upper bound of a node in the tree is lower than the global lower bound (that is, the maximum of the solutions found so far), the children of the node do not need to be explored as they would lead to a worse solution, so the branch can be pruned. This step is the *the bounding phase*.

In our approach, the branching phase is based on deciding whether to add a specific edge (and thus its endpoints) in the corresponding solution or not. More specifically, if we are looking for the heaviest subgraph of size $k$, our branch and bound algorithm consists of the following.

- We start with the root such that all edges are possibly here or not. The upper bound is the sum of the $\binom{k}{2}$ heaviest edges, while the associated solution is the empty subgraph, the lower bound is thus 0.
- Then at each iteration, we create two children for the node with maximum lower bound (i.e. density of the associated solution). Suppose the node is at depth $i$ in the tree, we keep the decisions made on the first $i-1$ edges and create two children, one where the $i^{th}$ edge is included and one where it is not.

The lower bound is given by the sum of the weights on the edges of the subgraph induced by the edges that are included.

Assume the number of nodes in the subgraph induced by the edges that are included is $s$. Then the upper bound is given by the weights of subgraph plus the sum of the next $\binom{k}{2} - \binom{s}{2}$ highest weight edges. When we add a new edge $(u, v)$ we have to check if both endpoints are part of the current's solution vertex list. If they are not, we add them in the vertex list and we update the weight of this solution by adding the weight of every possible edge between $u$ (respectively $v$) and vertices in $S$. We thus need a subroutine to check efficiently if two nodes are adjacent or not.

**Checking adjacency efficiently.** We initially compute a core ordering (or degeneracy ordering) of the unweighted version of the graph. We then keep for each node in the graph a sorted list of its neighbors having higher core ordering (the maximum size of such a truncated neighborhood list is thus $c$, the core value of the graph). Given 2 nodes $x$ and $y$ (w.l.o.g. we assume that $y$ has a higher core ordering than $x$) we can efficiently check if they are adjacent by checking if $y$ is in the truncated neighborhood list of $x$ by binary search in $O(\log(c))$ time. Note that this step is in practice the bottleneck of our algorithm taking about 80% of the time. We were not able to make this subroutine faster in practice using a hashtable containing all edges.

The worst case total running time of the process of creating a child is thus in $O(k \cdot \log(c))$ (where $c$ is the core value of the graph), which is the time to check if $u$ and $v$ belongs to the neighborhood of the nodes in the solution subgraph (there are at most $k-1$ of them), note that the maximum size of a neighborhood is $n-1$. We keep forming the children of the node with the higher lower bound till the branch is pruned. In addition to the pruning using lower and upper bound, if the number of nodes in the subgraph exceeds $k$ we also prune the branch. When we add a new edge

$(u, v)$, we have to check for both endpoints if they belong in the current solution. If they don't, we add the weight of the edges between the endpoints of the new edge $(u, v)$ and the vertices of the current solution. However, if such an edge $(u, x)$ has a higher ranking than $(u, v)$ (i.e. the weight of $(u, x)$ is greater or equal to the weight of $(u, v)$), then we can prune this solution. Indeed, since we make the decisions on edges in non-increasing order according to their weight it means that this edge $(u, x)$ was excluded from the solution, it thus cannot be added at this step.

A key feature of a branch and bound algorithm is the order by which the method iterates over the nodes of the branch and bound tree. This order can be BFS, DFS, based on the lower bound or on the upper bound of a node. After examining those different order, we found that examining the nodes according to the weight of the candidate solutions was the most efficient one. Hence, every time a child is created by the branch and bound algorithm, it is added to a heap, whose head is always the solution of maximum weight.

**Intuition behind our method.** The intuition behind our method is that the best solution should contain many edges of high weight (that is well ranked in non-increasing order of weight), while possibly containing few edges of low weight. The method should not explore all edges till the ones of low weights as those will be added to the solution through forming the induced subgraph on the edges of high weight.

**Branch and bound on nodes.** During the branching phase where we make decisions upon the edges, one might ask if it would be better to make decisions upon the vertices. After some experimentation with both approaches, we observed that branch and bound with edges is more efficient. Indeed, a good upper bound for the branch and bound with edges is easy to compute, while we couldn't find any simple way to do so for the branch and bound with nodes. In addition, ranking edges in non-increasing order of weight is rather natural and it is more probable that the best solution contains edges of high weight leading to a quicker termination of the algorithm. While ranking nodes is less natural (even though weighted degree ordering or weighted-core ordering are possible) and nodes in the best solutions can be ranked further leading to a slower termination of the algorithm.

**Approximation algorithm for HkS.** In order to obtain an $\alpha$ approximation of the $HkS$ problem, we modify our branch and bound algorithm in the following way. If at some point, the current largest upper bound divided by the current best solution is higher than $\alpha$, we output the solution. As there is no other subgraph of size $k$ such that the sum of the edges is higher than the largest upper bound, we know that our solution is an $\alpha$-approximation.

**Top heavy k-subgraphs.** Due to the hardness of the problem we cannot hope to solve it optimally, so we use a simple greedy heuristic: we find the heaviest $k$-subgraph, we remove all its vertices and their incident edges from the graph and

**Algorithm 7** Branch and bound

**Input**: A graph $G(V, E)$ and an integer $k$
**Output**: The heaviest $k$-subgraph of $G(V, E)$
$edges \leftarrow$ edge list sorted in non-increasing order according to the weight
For a current $BnB$ node, let $S$ be the list of vertices of the subgraph, $w$ its weight, $b$ the upper bound and $i$ the index of the current edge.
Initially $S \leftarrow \varnothing, w \leftarrow 0, i \leftarrow 0, b \leftarrow$ sum of the heavier $\frac{k(k-1)}{2}$edges
HEAP.insert($(S, w, i, b)$)
$bestSolution \leftarrow 0$
**while** HEAP is not empty **do**
  $(S, w, b, i) \leftarrow$ HEAP.$pop()$;
  **if** $b \leq bestSolution$ or $|S| > k$ **then**
    **Prune** current solution
  **else**
    **if** $w > bestSolution$ **then**
      $bestSolution \leftarrow w$
    **end if**
    $(S_1, w_1, b_1, i_1) \leftarrow$ createChild( $(S, w, b, i)$ ,1)
    HEAP.insert($(S_1, w_1, b_1, i_1)$)
    $(S_0, w_0, b_0, i_0) \leftarrow$ createChild( $(S, w, b, i)$ ,0)
    HEAP.insert($(S_0, w_0, b_0, i_0)$)
  **end if**
**end while**
**return** best solution found;

**Algorithm 8** Procedure createChild($(S, w, b, i)$, int c)

```
1: if c = 1 then
2:    e ← edges[i];
3:    Let (u, v) be the endpoints of edge e
4:    if u ∉ S then
5:       for all x ∈ S do
6:          if (u, x) ∈ E then
7:             w ← w + (u, x).weight
8:          end if
9:       end for
10:      S ← S ∪ {u}
11:   end if
12:   if v ∉ S then
13:      for all x ∈ S do
14:         if (v, x) ∈ E then
15:            w ← w + (v, x).weight
16:         end if
17:      end for
18:      S ← S ∪ {v}
19:   end if
20: end if
21: i ← i + 1;
22: Update b;
23: return (S, w, b, i)
```

we iterate on the remaining graph until we have found $t$ subgraphs or the graph is empty. At the end of the computation, we will obtain at most $t$ disjoint subgraphs. Note that this heuristic gives a $\frac{1}{t}$-approximation guarantee as the first found heaviest subgraph is at least as heavy as any subgraph in the optimal solution containing $t$ subgraphs.

### 4.3.2 Heuristic based on weighted core decomposition

In order to have a basis for comparison, we implement the greedy heuristic used in [9]. The algorithm is based on the weighted core decomposition and it consists in repeatedly removing a vertex with the minimum weighted-degree in the currently remaining graph until exactly $k$ vertices are left. We implemented the algorithm using a min-heap as detail in Algorithm 11, so that its asymptotic running time is $O((m+n)\cdot\log(n))$.

**Algorithm 9** Weighted core heuristic for DkS

```
1: for each node u in G do
2:    d_w(u) ← weighted-degree of u
3:    Δ_η(u) ← list of neighbors of u and associated edge weight (v, w_{u,v})
4: end for
5: create a min heap HEAP initialized with (u, d_w(u)) for each node u in G
6: while HEAP has more than k elements do
7:    u ← HEAP.pop()
8:    for each (v, w_{u,v}) ∈ Δ(u) do
9:       d_w(v) ← d_w(v) − w_{u,v}
10:      update HEAP with (v, d_w(v))
11:   end for
12: end while
13: return the k nodes in HEAP
```

One essential drawback of this algorithm is that it does not have any fixed parameter approximation guarantee. As proved in [9], for $k < \frac{n}{3}$ it has a $O(\frac{k}{n})$ approximation guarantee, however for our setting $k$ is small (events are described by a small number of entities, in practice we work with $k$ smaller than 20), while $n$ is large and thus the approximation guarantee can be very bad.

Next, we show that for any $\alpha < 1$, we can find a graph where the ratio between the solution the weighted core heuristic and the optimal one is smaller than $\alpha$. This can be easily seen in Figure 4.1, for $k = 2$, we modify the edge weight between $u$ and $v$ to be an integer $\delta$, satisfying $\delta > \frac{1}{\alpha}$ and we keep a clique of size $(\delta+2)$ where every edge has unitary weight. This can be extended also for larger $k$.

**Improving the solution by local search.** Note that the solution given by Algorithm 11 is a stable local optimum in the sense that adding a node such that the weight of the subgraph of size $k+1$ is maximized and then removing the node such that the size of the subgraph of size $k$ is maximized will lead to a solution having the same value as the initial one. However we can improve the solution by switching a node inside the subgraph of size $k$ and a node outside the subgraph

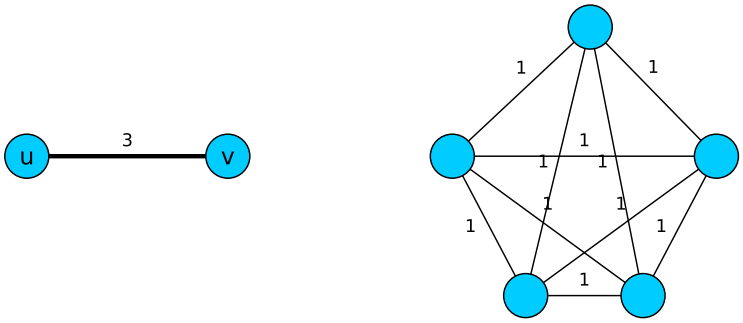


Figure 4.1: On this graph for $k = 2$ the best solution is the subgraph induced by the vertices $u, v$ with total weight 3. However, the weighted core decomposition algorithm will exclude first those two vertices since the contribution of every vertex in the clique is 4 and theirs is 3.

such that the weight is maximized, this can be repeated till the solution becomes a stable local optimum with respect to this switch operation. This can be done efficiently by considering only the nodes inside the subgraph and nodes outside the subgraph having at least one neighbor inside the subgraph. We also add this technical improvement which does not lead to a better approximation but gives, in practice, slightly better results.

## 4.4 Experiments

### 4.4.1 Experimental setup

We collected a set of tweets by means of the Twitter Streaming API during the months of November and December 2015. The sample of tweets contains only English and French tweets (language is automatically detected by the Twitter platform). We note that in this experimental part we will only present quantitative results, such as running time and density and a qualitative evaluation on the task of event detection will be presented in Chapter 6.

We construct graphs from each of the datasets as follows. We extract nouns from the tweets using the Stanford POS Tagger [1] and also hashtags and we construct a weighted undirected graph $G = (V, E, w)$, where the set of nodes consists of the terms extracted in the previous step, while there is an edge between two nodes if the corresponding terms co-occur in at least one tweet, the weight corresponds to the number of co-occurrences.

We obtain four graphs which are detailed Table A.3.

The algorithms were implemented in C and were run on a machine under GNU/Linux with 2.39 GHz clock, while limiting the total amount of memory available to 20 GB.

[1] http://nlp.stanford.edu/software/tagger.shtml

| Name | Nodes | Edges |
|---|---|---|
| French-Nov. | 220 K | 2.9 M |
| French-Dec. | 200 K | 2.1 M |
| English-Nov. | 2.5 M | 4.5 M |
| English-Dec. | 1.8 M | 3.1 M |

Table 4.1: Our set of graphs-of-words extracted from Tweets.

### 4.4.2 Running time and structural comparisons

Table A.4 shows the running time of our branch and bound algorithm as well as the running time of the algorithm based on weighted core decomposition. In order to differentiate between the time to load the dataset in memory and the time to run the heaviest-$k$ dense computation, we added an extra column "Loading time". For values of $k \leq 15$, we can see that both algorithms have a good performance, finishing the computation in a matter of seconds in most of the cases. The time of the algorithm based on weighted core does not vary much with $k$ as it always computes the weighted core and then performs few operations for the local search. However, as we can see in Figure 4.2, the time of the branch and bound algorithm increases exponentially with $k$. This can be explained as when $k$ is large, the probability that the solution contains many edges of high weight and only a few edges of low weight becomes lower. This shows a limitation of our approach which is not efficient when $k$ is too large, however for small $k$ we are able to find the exact solution in a short amount of time. We note that this is enough for a task such as event detection, where we are interested in small values of $k$.

In Table A.5 we present the total weight of HkS for different values of $k$ when using the two techniques. We note that our branch and bound algorithm finds the optimal solution. In order to compare the subgraphs obtained by the two algorithms, we compute the ratio between the solution and the optimal solution. As we can see the approximation of the weighted core heuristic varies between 0.99 and 0.67 on our graphs.

Table A.6 shows the ratio between the number of edges in the subgraphs of size $k$ and the number of edges in a clique of size $k$. As we can see, in all cases except the one of "French-Dec." the solution is a clique. This shows that the structure of those real-world graphs is special and that, even though we are looking for $HkS$ and not cliques, designing an algorithm to find a clique of size $k$ and of maximum weight is a very similar problem on those real-world graphs. As enumerating cliques can be done efficiently in real-world graphs [25,63], a heuristic based on enumerating all cliques of size $k$ and returning the one of maximum weight could be considered.

**Approximation for larger k.** Figure 4.2 (up) shows the running time of our exact branch and bound algorithm as a function of the input k (number of nodes in the subgraph) for our four datasets. We truncated the curves at 1 hour of computation. As we can see, within one hour our algorithm can solve the problem for $k$ = 16 in the dataset French-Dec. while it can go up to $k$ = 36 for English-Dec. In Figure 4.2 we can see in the lower part the same curves, but for our approximation branch and bound algorithm. When using the approximation ratio of $\alpha$ = 1.5 our

| | | Weighted core | B&B DkS | | |
|---|---|---|---|---|---|
| Dataset | Loading time | $k$ = 5, 10 and 20 | $k$ = 5 | $k$ = 10 | $k$ = 15 |
| French-Nov. | 0.8s | 0.7s | 0.5s | 0.9s | 17s |
| French-Dec. | 0.6s | 0.5s | 0.9s | 1.0s | 9m58s |
| English-Nov. | 11s | 8.9s | 17s | 17s | 18s |
| English-Dec. | 12s | 4.8s | 7.5s | 7.0s | 7.0s |

Table 4.2: Running time comparison.

| | Weighted core | | | B&B DkS | | |
|---|---|---|---|---|---|---|
| Dataset | $k$ = 5 | $k$ = 10 | $k$ = 15 | $k$ = 5 | $k$ = 10 | $k$ = 15 |
| French-Nov. | 24669 (0.92) | 70528 (0.96) | 115341 (0.96) | 26824 | 73432 | 120419 |
| French-Dec. | 18158 (0.95) | 22458 (0.67) | 36358 (0.81) | 19090 | 33338 | 45020 |
| English-Nov. | 2942467 | 5836705 (0.99) | 8114183 (0.99) | 2942467 | 5913372 | 8161838 |
| English-Dec. | 2386716 | 5125458 (0.98) | 7830637 | 2386716 | 5235181 | 7830637 |

Table 4.3: Total weight of the HkS for different values of $k$ when using the weighted core and our algorithm.

| | Weighted core | | | B&B DkS | | |
|---|---|---|---|---|---|---|
| Dataset | $k$ = 5 | $k$ = 10 | $k$ = 15 | $k$ = 5 | $k$ = 10 | $k$ = 15 |
| French-Nov. | 1.0 | 1.0 | 0.98 | 1.0 | 1.0 | 0.98 |
| French-Dec. | 0.4 | 0.27 | 0.44 | 1.0 | 0.44 | 0.3 |
| English-Nov. | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| English-Dec. | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

Table 4.4: Edge Density.

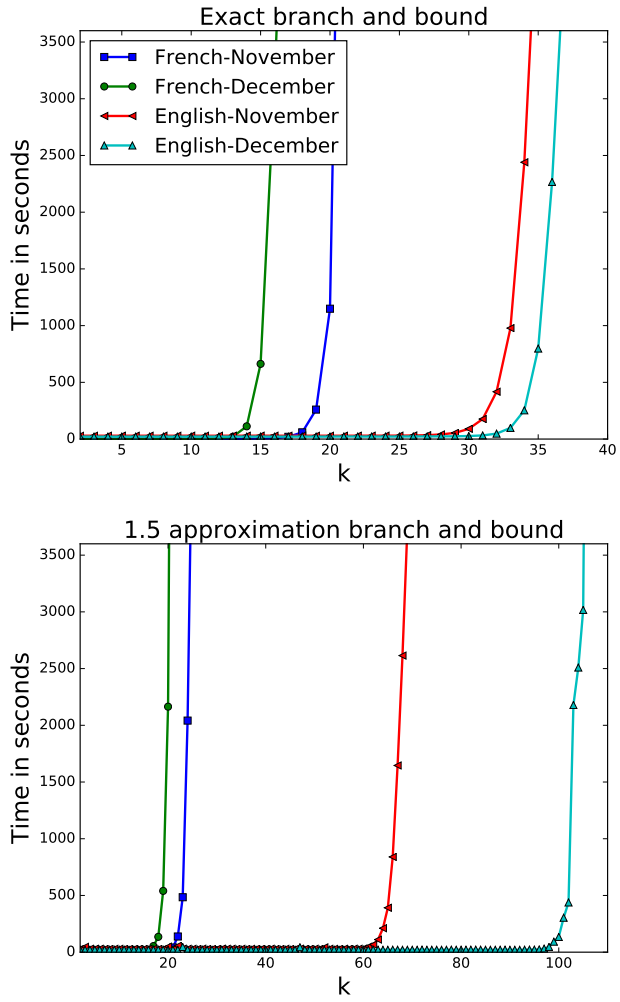


Figure 4.2: Running time of branch and bound algorithm as a function of the input $k$ for our four datasets: (up) exact algorithm, (down) approximation algorithm with approx. ratio to $\alpha = 1.5$.

algorithm is able to solve the problem for larger k, and within one hour we can compute a 1.5 approximation for $k = 20$ for French-Dec. and a 1.5 approximation for $k = 105$ for English-Dec.

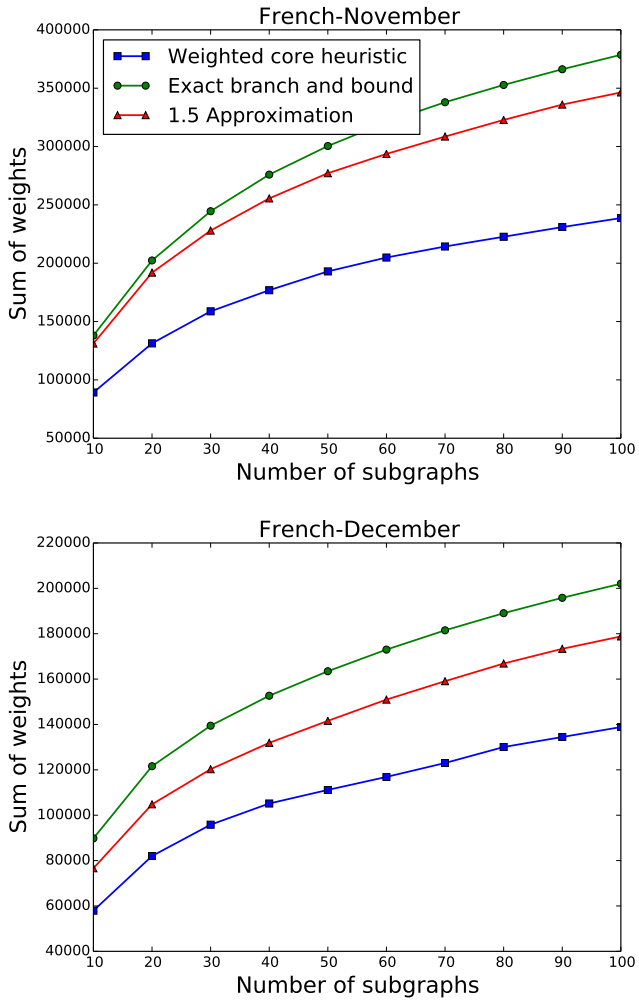


Figure 4.3: Sum of the weights of the top $t$ heavy subgraphs of size 5 as a function of $t$ for our French datasets.

In order to compare how well performs (i) the heuristic based on weighted core decomposition (ii) our exact branch and bound algorithm and (iii) our approximation branch and bound algorithm for finding the heaviest $t$ subgraphs, we compute the top 100 subgraphs having as subroutine one of the three algorithms (for the approximation we use $\alpha = 1.5$. Figure 4.3 shows the sum of weights of the subgraphs as a function of the number of subgraphs for the French datasets. We omit the English datasets as the results are similar. In all datasets our exact algorithm, as well as our approximation algorithm for $\alpha = 1.5$, consistently outperforms the weighted core decomposition algorithm, so we can conclude that it is better suited for solving the top $t$ heavy $k$-subgraphs problem for small values of $k$.

## 4.5 Conclusion

We presented a new branch and bound algorithm for solving the densest k-subgraph problem in weighted graphs. The branching phase is based on deciding whether to include an edge in the subgraph or not, and edges are examined in non-increasing order of weight to maximize the efficiency of the algorithm. The pruning phase is two-fold: (i) based on the size of the subgraph and (ii) on the efficient computation of a tight upper bound.

The algorithm scales to large weighted real-world graphs containing millions of edges for up to $k = 15$ or more depending on the structure of the graph. An approximation version of our algorithm can scale to larger k. We show that our algorithm performs better than a state-of-the-art method based on weighted core decomposition of the graph.

Future work includes improvements to our branch and bound algorithm using parallel computing, as well as its generalization to the detection of other kinds of subgraphs. One possible direction is community detection, where a community is intuitively defined as a set of nodes that are highly connected together but poorly connected to the outside [38].

# Chapter 5

# $k$-clique Listing and $k$-clique Core Decomposition

## 5.1 Introduction

Recent works in the data mining and database community call for efficient algorithms for listing or counting all $k$-cliques in the input graph. In particular, in [95] the author develops an algorithm for finding subgraphs with maximum average number of $k$-cliques, with counting $k$-cliques being an important building block. In [89] an algorithm for organizing cliques into hierarchical structures is presented, which requires to list all $k$-cliques. Efficient algorithms for counting $k$-cliques would allow for the computation of a natural generalization of the well-known core decomposition, the *$k$-clique core decomposition*. Another interesting application could be in event detection, because as we discovered in the previous chapter, a heavy subgraph of a small size will often be a clique.

Motivated by the aforementioned studies, we develop the most efficient parallel algorithms for listing and counting all $k$ cliques of an input graph, with $k$ being an input parameter. Our theoretical analysis shows that even the sequential version of our algorithm outperforms state-of-the-art algorithms for the same problem, while leveraging the sparsity of real-world graphs. Moreover, as opposed to state-of-the-art algorithms, our algorithm is parallel which improves the total running time even further. Our extensive experimental evaluation shows that both the sequential and parallel versions of our algorithm outperform significantly state-of-the-art approaches for the same problem. In particular, our parallel algorithm is able to list all cliques in graphs containing up to tens millions edges, as well as all 10-cliques in graphs containing billions of edges, within a few minutes or a few hours, respectively, while achieving excellent degree of parallelism. We also show that our algorithm can be employed as an effective subroutine for computing a $k$-clique core decomposition in large graphs and an approximation of the $k$-clique densest subgraph [95].

We summarize our contributions as follows:

- We develop a parallel algorithm for listing and counting all $k$-cliques in a graph. Our theoretical analysis shows that our algorithm achieves the best known asymptotic running time leveraging the sparsity of real-world networks.

It requires linear memory in the size of the input graph. In practice, the sequential version of our algorithm is an order of magnitude faster than state-of-the-art algorithms on real-world graphs, while the parallel version achieves an excellent degree of parallelism.

- We develop an efficient algorithm for the $k$-clique core decomposition. Such an algorithm is based on the algorithm for counting $k$-cliques, while requiring asymptotically the same running time;

- We show how our algorithm for counting $k$-cliques can be employed to effectively find approximate $k$-clique densest subgraphs;

The rest of the chapter is organized as follows. In Section 5.2 we define the problems we will attempt to solve, then in Section 5.3, we present our algorithm for listing $k$-cliques and prove its theoretical guarantees. We also adapt such an algorithm so as to (i) compute the $k$-clique core decomposition of a graph, and (ii) compute an approximation of the k-clique densest subgraph. We then evaluate the performance of our algorithm against the the state of the art and present the results obtained for our applications in Section 5.4. Finally we conclude and present future work in Section 5.5.

## 5.2 Problem Definition

In this section, we give a formal definition of the problems we study in our work. We also introduce necessary notations and definitions that shall be used in the rest of the chapter. We shall assume that we are given an undirected connected graph $G = (V(G), E(G))$. If the input graph is not connected, then our algorithms can be executed in each of the connected components, independently. Given an integer $k > 1$, an induced subgraph of $G$ is called a $k$-clique if it contains exactly $k$ nodes each pair of which being connected by an edge in $E(G)$. The following problem generalizes the problem of finding all triangles in a graph.

**Problem definition** (*$k$-clique listing problem*). Given an undirected graph $G = (V(G), E(G))$ and an integer $k > 1$, we wish to list all $k$-cliques in $G$.

As the number of $k$-cliques in a graph can be very large, storing or even producing in output all of them might pose significant issues. Therefore, we also consider a variant of the problem where the objective is to count the number of $k$-cliques in a graph.

**Problem definition** (*$k$-clique counting problem*). Given an undirected graph $G = (V(G), E(G))$ and an integer $k > 1$, we wish to count the number of $k$-cliques in $G$.

The aforementioned problems are as hard as the maximum clique problem which is NP-hard and NP-hard to approximate within a factor $n^{1-\epsilon}$ [46]. Therefore, it is unlikely that efficient algorithms for arbitrary graphs are developed. In our work, we devise efficient algorithms for sparse graphs i.e. with relatively low core and arboricity values.

We generalize the definition of core value of a graph as follows. Given an integer $k > 1$, the *k-clique degree* of a node $v$ in $V(G)$ is defined as the number of $k$-cliques in $G$ which contain $v$. For any node $v$, let $c^k(v)$ be the largest integer such that there is a subgraph $H$ of $G$ with (i) $H$ containing $v$, (ii) any node $u$ in $H$ having $k$-clique degree at least $c^k(v)$. $c^k(v)$ is called the *k-clique core value of* $v$.

**Problem definition** (*k-clique core decomposition*). Given an undirected graph $G = (V(G), E(G))$ and an integer
$k > 1$, compute a $k$-clique core decomposition of $G$.

The previous definition generalizes the well known core decomposition (for $k = 2$).

**Problem definition**: ($k$-clique densest problem). Given an undirected graph $G = (V(G), E(G))$, find a subgraph $H$ of $G$ such that the $k$-clique density is maximized. The $k$-clique density of $H$ is $d_k(H) = \frac{c_k(H)}{|V(H)|}$, where $c_k(H)$ is the number of $k$-cliques induced by $H$.

This problem has been addressed and formalized in [95].

## 5.3 Algorithms

In this section, we present our parallel algorithm for finding all $k$-cliques on large real-world graphs. We borrow ideas from the sequential algorithms presented in [55], [25] respectively, so as to obtain an efficient parallel algorithm for processing large real-world graphs. The most appealing features of our algorithm is that it is parallel (as opposed to state-of-the-art algorithms for the same problem) and that it leverages the sparsity of the input graph leading to the best known asymptotic running time as shown by our theoretical analysis. This allows to find all 10-cliques on real-world graphs containing more than one billion edges, as well as finding all cliques in graphs containing a few million edges, within a few minutes or hours.

The key issue to derive efficient algorithms for listing all $k$-cliques is to effectively prune the search space while limiting redundant operations (such as processing a same clique several times), as well as divide the main task into independent subtasks that can be executed in parallel.

The first part of our algorithm consists of determining an optimal ordering of the nodes in the graph with the following procedure: Starting from the input graph $G$, successively remove a node (and its edges) with smallest degree from the current graph, until the graph becomes empty. This is essentially the same algorithm used for computing a $k$-core decomposition [66]. Let $\eta : V(G) \to \{1, \ldots, n\}$ be a function indicating for each node $v$ the step $t$ in which such a node has been deleted, $1 \leq t \leq n$. For each node $u$, we define the set $\Delta_\eta(u)$ consisting of all the neighbours $v$ of $u$ with $\eta(v) > \eta(u)$. Note that given this definition we have $\max_{u \in V(G)} |\Delta_\eta(u)| = c$ the core number of the graph. We use $\Delta(u)$ to denote all the neighbours of $u$.

The algorithm for finding all 4-cliques can be described as follows. For each edge $(u, v)$ in parallel, the algorithm finds all 4-cliques in the subgraph induced by all the nodes in $\Delta_\eta(u) \cap \Delta_\eta(v)$. More precisely, for each edge $(u, v)$ the set $\Delta_\eta(u, v) = \Delta_\eta(u) \cap \Delta_\eta(v)$ is computed. Then, for each $w$ in $\Delta_\eta(u, v)$ the set

$\Delta_\eta(u,v,w) = \Delta_\eta(w) \cap \Delta_\eta(u,v)$ is computed. Finally, for each $x$ in $\Delta_\eta(u,v,w)$, the 4-clique $\{u,v,w,x\}$ is produced in output. A pseudocode of our algorithm is shown in Algorithm 10.

Note that for each edge $(u,v)$ the set $\Delta_\eta(u,v)$ contains nodes whose core value is not smaller than those of $u$ and $v$. Therefore, each $k$-clique is listed when the edge $(u,v)$ is processed, where $u$ and $v$ are the nodes with smallest core value in the clique. This turns out to be an effective pruning strategy, as it limits significantly the number of neighbors to consider. Moreover, the size of the $\Delta_\eta$'s become smaller and smaller as we dive deeper in the nested for loops, thereby pruning the search space event further.

Our algorithm can be implemented in a few lines of code and our theoretical analysis shows that it is the fastest algorithm for listing $k$-cliques even in the sequential case, while its parallel version performs even faster as shown by our experimental evaluation.

**Algorithm 10** Parallel algorithm for finding $3,4,5$-cliques

```
1: i ← 1;H ← G;
2: while V(H) ≠ ∅ do
3:    Let v be a node with minimum degree in H;
4:    V(H) ← V(H) ∖ {v};
5:    E(H) ← E(H) ∖ Δ(v);
6:    η(v) = i;
7:    i ← i + 1;
8: end while
9: for each node u in G do
10:   Δ_η(u) ← set of neighbors v of u, η(v) > η(u)
11: end for
12: for each edge (u,v) ∈ E(G) parallel loop do
13:   Δ_η(u,v) ← Δ_η(u) ∩ Δ_η(v)
14:   for each w in Δ_η(u,v) do
15:      Output triangle {u,v,w}
16:      Δ_η(u,v,w) ← Δ_η(u,v) ∩ Δ_η(w)
17:      for each x in Δ_η(u,v,w) do
18:         Output 4-clique {u,v,w,x}.
19:         Δ_η(u,v,w,x) ← Δ_η(u,v,w) ∩ Δ_η(x)
20:         for each y in Δ_η(u,v,w,x) do
21:            Output 5-clique {u,v,w,x,y}
22:         end for
23:      end for
24:   end for
25: end for
```

**Subtleties about the parallel for loop:** In Algorithm 10, the parallel loop (line 10) is on the edges, it could be adapted to have this parallel loop on the nodes. However there is a notable difference between the edge and the node parallel

processing. Indeed, if the graph is a clique of size $K > 1$ then, given a $k < K$, there are $\binom{K}{k}$ $k$-cliques in the graph: each node belongs to $\binom{K-1}{k-1}$ $k$-cliques and each edge to $\binom{K-2}{k-2}$ $k$-cliques. This gives, in the case of processing nodes in parallel, a maximum of $\lceil \binom{K}{k} / \binom{K-1}{k-1} \rceil = \lceil \frac{K}{k} \rceil$ threads that can work at the same time, for example for $K = 100$ and $k = 10$ only 10 threads can work at the same time. While in the edge parallel case, a maximum of $\lceil \binom{K}{k} / \binom{K-2}{k-2} \rceil = \lceil \frac{K(K-1)}{k(k-1)} \rceil$ threads can work at the same time, so again if $K = 100$ and $k = 10$, then 110 threads can work at the same time. It is thus better to use the parallel for loop on the edges while working on a graph containing a large clique, as the number of threads that can be used in parallel is less limited. We have encountered this effect in our experiments on real world graphs. Note that the parallelization can be also on the triangles or on higher order $k$-cliques if needed.

**Implementation details:** For each node $u$ we maintain the set $\Delta_\eta(u)$ which contains the neighbors of $u$ with $\eta$ values larger than $u$. $\Delta_\eta(u)$'s are stored in sorted lists according to the $\eta$ values of their nodes, therefore, the intersection between any two sets $\Delta_\eta(u)$'s can be computed in time $O(c)$ (recall that $|\Delta_\eta(u)| \leq c$, for any $u$). This holds also for sets $\Delta_\eta(u, v, w)$'s etc. In order to minimize the required amount of memory we proceed as follows. Observe that we do not need to keep in main memory all $\Delta_\eta$'s. In particular, a given $\Delta_\eta$ created in one of the nested loops in Algorithm 10 is required only in inner loops. Therefore, we need $O(c)$ space for each of the $k$ nested loops, leading to $O(m + k \cdot c)$ total amount of memory for storing the $\Delta_\eta$'s. Observe that if $k > c$, then the number of $k$-cliques is zero which implies we can assume $k = O(c)$. From the fact that $c = O(\sqrt{m})$, it follows that $k \cdot c = O(m)$ which implies that the total amount of memory required by the sequential algorithm is linear in the size of the graph. In the parallel version, all threads share the $\Delta_\eta(u)$'s while each thread requires some space to store all $\Delta_\eta$'s created in the parallel for loop. This leads to $O(m + p \cdot c \cdot k)$ total space, where $p$ is the total number of threads.

**Theorem 5.3.1.** Given an integer $k > 1$, an undirected connected graph $G = (V(G), E(G))$, Algorithm 10 outputs and counts all $k$-cliques in $G$, while requiring $O(c \cdot \sum_{l=2}^{k-1} N_l + N_k)$ total number of operations for counting $k$-cliques, where $c$ and $N_l$ denote the core value of $G$ and the number of $l$-cliques in $G$, respectively. It requires linear amount of memory in the size of the graph.

*Proof.* We prove our theorem for $k = 4$, the generalization to any $k > 1$ is straightforward. First, observe that any clique $C$ is counted/produced in output exactly once when the for loop in Line 10 processes the edge $(u, v)$ with $u$ and $v$ having minimum $\eta$ values among all nodes in $C$. Lines 1-11 require $O(N_2)$ total number of operations. By construction and by definition of core value, $\Delta_\eta(u) \leq c$ for every $u$. $\Delta_\eta(u)$'s are stored in sorted lists (sorting all $\Delta_\eta(u)$'s requires $O(n)$ total number of operations using bucket sort). Therefore, computing $\Delta_\eta(u, v)$ in Line 13 requires $O(c)$ operations. Lines 15,16 are executed $O(N_3)$ times, with Line 16 requiring $O(c \cdot N_3)$ total number of operations. Line 18 is executed $O(N_4)$ number of times in total. It follows that the total number of operations required to count all 4−cliques is $O(c \cdot N_2 + c \cdot N_3 + N_4)$. Our algorithm can be implemented so as to require a linear amount of memory in the size of the graph, as discussed in the paragraph "implementation details". □

**Lemma 5.3.1.** Let $k > 1$ be an integer, it holds that $N_k \leq m \cdot \binom{c-1}{k-2} \leq 2 \cdot m \cdot (\frac{c-1}{2})^{k-2}$.

*Proof.* Let $(u, v)$ be an edge in the input graph. Without loss of generality, let $\eta(v) > \eta(u)$. Let $N_{uv}^k$ be the number of $k$-cliques $C$ such that 1) $C$ contains the nodes $u$ and $v$ 2) for all other nodes $w$ in $C$, $\eta(w) > \eta(v)$. There are at most $c-1$ neighbors $w$ of $u$ with $\eta(w) > \eta(v)$, for otherwise $\Delta_\eta(u) > c$. It follows that $N_{uv}^k \leq \binom{c-1}{k-2}$. Hence, $N_k \leq \sum_{uv \in E(G)} N_{uv}^k \leq m \cdot \binom{c-1}{k-2}$. Since $\binom{c-1}{k-2} \leq \frac{(c-1)^{k-2}}{(k-2)!} \leq 2 \cdot (\frac{c-1}{2})^{k-2}$, the claim follows. □

**Theorem 5.3.2.** Given an integer $k > 1$, an undirected connected graph $G = (V(G), E(G))$, Algorithm 10 outputs and counts all $k$-cliques in $G$, while requiring $O(m \cdot (\frac{c-1}{2})^{k-2})$ number of operations for counting $k$-cliques, if $c > 3$ and $O(m)$ otherwise. It requires linear amount of memory in the size of the graph.

*Proof.* We consider first the case when $c > 3$. From Theorem 5.3.1 and Lemma 5.3.1 it follows that the total number of operations required by the algorithm is $O(m \cdot (c \cdot \sum_{l=0}^{k-3} (\frac{c-1}{2})^l + (\frac{c-1}{2})^{k-2}))$. The claimed bound then follows by observing that $\sum_{l=0}^{k-3} (\frac{c-1}{2})^l = \frac{(\frac{c-1}{2})^{k-2}-1}{(\frac{c-1}{2})-1}$. In the case when $c \leq 3$, we have $\forall l > 4$, $N_l = 0$ and $\forall\ l \leq 4$, $N_l = O(m)$, which together with Theorem 5.3.1 imply the claimed bound of $O(m)$. □

Our algorithm is the most efficient algorithm for counting k-cliques. Prior to our work the best known algorithm was the one developed in [25], with running time of $O(m \cdot a^{k-2})$ (recall that $c \in [\![a, 2a-1]\!]$ [103] and thus $\frac{c-1}{2} < a$). This suggests that our algorithm might be faster for large values of $k$ which is shown in Section 5.4. Another appealing feature of our algorithm is that it is *output sensitive* as shown by Theorem 5.3.1, which means roughly speaking that it is relatively fast when the number of $k$-cliques is relatively small.

### 5.3.1 The k-Clique Core Decomposition

The algorithm for computing the well-known core decomposition of a graph [22, 66] ($k = 2$), proceeds in the following fashion: starting from the input graph, at each step a node of minimum degree and all its edges are removed from the current graph until the graph becomes empty. Building on this approach, we develop an algorithm for any $k \geq 3$.

In the following we describe the algorithm for 5-clique core decomposition, from which we can easily generalize to any $k \geq 3$. The first part of our approach consists in computing the number of 5-clique each node belongs to, that is, the 5-clique degree of each node. We perform the computation by running the Algorithm 10 and increasing the 5-clique degree of a node $u$ each time we find a clique of size 5 containing $u$. We then store the computed degrees in a binary heap in order to keep track of the minimum 5-clique degree. Then, as in the standard core decomposition, we iteratively remove a node of minimum 5-clique degree using the heap and update the 5-clique degree of its neighbours until the graph becomes empty.

The last operation is not trivial, as the node $u$ might belong to a large number of 5-cliques. Storing all such 5-cliques is therefore not feasible. To avoid that, we

do the following. At each iteration when removing a node $u$ with minimum 5-clique degree, we list all 5-cliques containing $u$ (without storing any one of them). Each time we find a clique containing the nodes $\{u, v, w, x, y\}$, we decrement by one the degrees of each node in $\{v, w, x, y\}$.

In order to list all the 5-cliques which contain the node $u$, we can use a simple subroutine similar to Algorithm 10. The main difference is that we consider the whole set of neighbours of $u$, $\Delta(u)$ as opposed to the set $\Delta_\eta(u)$. See Algorithm 11 for a pseudocode of the algorithm for computing a $k$-clique core decomposition. Note that it is also parallel.

**Running time.** In order to maintain the heap structure we need $O((n+m)\log(n))$ operations, as for both removing a node and updating its value we require $O(\log(n))$ operations, while we perform $n$ removals and $m$ updates. Updating the $k$-clique degrees takes at most $O(m \cdot (\frac{c-1}{2})^{k-2})$ operations, which gives the total running time of $O(m \cdot (\frac{c-1}{2})^{k-2} + (n+m)\log(n))$ for Algorithm 11.

**Algorithm 11** k-clique core decomposition for k=5

Use Algorithm 10 to compute the 5-clique degree and the truncated neighbors list of each node:
**for** each node $u$ in $G$ **do**
  $c^5(u) \leftarrow$ 5-clique degree of $u$
  $\Delta_\eta(u) \leftarrow$ set of neighbors $v$ of $u$, $\eta(v) > \eta(u)$
**end for**
**create** a min heap HEAP initialized with $(u, c^5(u))$ for each node $u$ in $G$
$c' \leftarrow 0$
**while** HEAP not empty **do**
  $(u, c) \leftarrow$ HEAP.*pop*()
  $c' \leftarrow \max(c, c')$
  **Output** $(u, c')$
  **for** each edge $(u, v), v \in \Delta(u)$ *parallel loop* **do**
    $\Delta_\eta(u, v) \leftarrow \Delta(u) \cap \Delta_\eta(v)$
    **for** each $w$ in $\Delta_\eta(u, v)$ **do**
      $\Delta_\eta(u, v, w) \leftarrow \Delta_\eta(u, v) \cap \Delta_\eta(w)$
      **for** each $x$ in $\Delta_\eta(u, v, w)$ **do**
        $\Delta_\eta(u, v, w, x) \leftarrow \Delta_\eta(u, v, w) \cap \Delta_\eta(x)$
        **for** each $y$ in $\Delta_\eta(u, v, w, x)$ **do**
          **decrement** $c^5(v), c^5(w), c^5(x), c^5(y)$
        **end for**
      **end for**
    **end for**
  **end for**
  **for** each $v$ in $\Delta(u)$ **do**
    **update** HEAP with $(v, c^5(v))$
  **end for**
  **delete** $u$ from $G$
**end while**

| networks | $n$ | $m$ | $c$ | $k_{max}$ | $n_{k_{max}}$ |
|---|---|---|---|---|---|
| road-CA | 1,965,206 | 2,766,607 | 3 | 4 | 42 |
| Amazon | 334,863 | 925,872 | 7 | 7 | 32 |
| soc-pocket | 1,632,803 | 22,301,964 | 47 | 29 | 6 |
| loc-gowalla | 196,591 | 950,327 | 51 | 29 | 2 |
| Youtube | 1,134,890 | 2,987,624 | 51 | 17 | 2 |
| cit-patents | 3,774,768 | 16,518,947 | 64 | 11 | 2 |
| zhishi-baidu | 2,140,198 | 17,014,946 | 78 | 31 | 4 |
| WikiTalk | 2,394,385 | 4,659,565 | 131 | 26 | 141 |

Table 5.1: Our set of large graphs (for which we are able to count all $k$-cliques). $k_{max}$ is the size of a maximum clique and $n_{k_{max}}$ is the number of maximum cliques.

### 5.3.2 k-Clique Densest Subgraph

A simple 1/2 approximation algorithm for the densest subgraph problem ($k = 2$) [22] proceeds as follows: starting from the input graph we remove at each step a node with minimum degree (and all its edges) from the current graph. We iterate until the graph becomes empty. We then return a subgraph with maximum density among all the subgraphs produced during the execution of the algorithm. The algorithm is similar to the algorithm for computing a core decomposition. In [95], the author shows that the variant of such an algorithm where at each step the node with minimum $k$-clique degree is removed gives a $1/k$ approximation algorithm. Hence, our $k$-clique core decomposition algorithm can be easily modified so as to obtain an $1/k$ approximation algorithm for the $k$-clique densest subgraph problem. The running time is $O(m \cdot (\frac{c-1}{2})^{k-2} + (n+m)\log(n))$. We evaluate such an algorithm in Section 5.4.

## 5.4 Experimental Evaluation

We consider several real-world graphs that we obtained from [58] and [53]. We divide them into two main groups: *large* graphs containing up to tens of millions of edges, for which we are able to count all $k$-cliques, as well as *very large* graphs containing up to billions of edges and being less sparse for which we could only count $k$-cliques of limited size (up to 10). Tables 5.1 and 5.2 summarize the characteristics of the two datasets, respectively. For each of the graph we also report its core number, which according to our theoretical analysis is an important feature for determining the running time of our algorithm.

### 5.4.1 k-Clique Counting

We evaluate separately triangle and $k$-clique counting. We carried our experiments on a linux machine having 2 threads Intel Xeon CPU E7-4870 @ 2.40 GHz with 10 cores split in 2 threads each (a total of 40 threads) and equipped with 64 G of RAM DDR3 1333 MHz. We evaluate our algorithms against the state-of-the-art

| networks | $n$ | $m$ | $c$ |
|---|---|---|---|
| DBLP | 425,957 | 1,049,866 | 113 |
| Wikipedia | 2,080,370 | 42,336,692 | 208 |
| Orkut | 3,072,627 | 117,185,083 | 253 |
| Friendster | 124,836,180 | 1,806,067,135 | 304 |
| LiveJournal | 4,036,538 | 34,681,189 | 360 |

Table 5.2: Our set of very large graphs (for which we are able to count $k$-cliques of limited size).

| | | Algorithms | | | |
|---|---|---|---|---|---|
| networks | # triangles | CF | MACE | Arboricity | FkCE1 |
| DBLP | 2,224,385 | 0.8s | 1.3s | 0.6s | 0.4s |
| Wikipedia | 145,707,846 | 1m07s | 22m22s | 1m04s | 40s |
| Orkut | 627,584,181 | 4m06s | 28m02s | 3m41s | 2m14s |
| Friendster | 4,173,724,142 | 1h50m41s | 5h29m40s | 2h57m21s | 1h05m31s |
| LiveJournal | 177,820,130 | 44s | 6m13s | 37s | 27s |

Table 5.3: Running time for counting triangles on our very large graphs.

algorithms for the corresponding problems while measuring the running time. We consider following approaches:

- **CF:** the compact-forward algorithm for counting triangles of [55], we used the C implementation available at the webpage of the author.
- **MACE:** the algorithm detailed in [63, 91], we used the C implementation available at the webpage of the author.
- **Arboricity:** the algorithm detailed in [25]. Not aware of an existing implementation, we have made an efficient implementation in C.
- **FkCE1:** our algorithm using only one thread (i.e. the sequential version of our algorithm) implemented in C.
- **FkCE10:** the parallel version of our algorithm using 10 threads. We used openMP [31] to make our C code parallel[1].

Table A.7 shows the running time of the algorithms for counting triangles. In this case, we consider the sequential version of our algorithm (FkCE1). Such a table shows that even the sequential version of our algorithm always outperforms state-of-the-art algorithms for triangle counting. It also shows that the algorithm presented in [55] is very efficient.

We then consider the $k$-clique counting problem on our collection of large datasets. Table A.8 shows that our algorithm can count all $k$-cliques for any value of $k$ in all those graphs, within a few minutes in most of the cases. It is much faster

[1]Our code is available at `github.com/maxdan94/kcliques`.

| | Algorithms | | |
|---|---|---|---|
| networks | MACE | Arboricity | FkCE1 |
| road-CA | 0.9s | 2.2s | 2.0s |
| Amazon | 1.0s | 0.8s | 0.6s |
| soc-pocket | 14m27s | 11m23s | 1m15s |
| loc-gowalla | 8m46s | 7m52s | 34s |
| Youtube | 1m05s | 1m12s | 3.9s |
| cit-patents | 22s | 24s | 15s |
| zhishi-baidu | 1h00m44s | 32m23s | 3m58s |
| WikiTalk | >24h | >24h | 5h53m36s |

Table 5.4: Time to compute all cliques on our large graphs.

than the other algorithms, up to a factor of 10. Moreover, our algorithm is the only one being able to process, in a reasonable amount of time, WikiTalk which is the dataset with largest core value.

For triangles counting on very large graphs and counting all cliques on large graphs we obtain satisfactory results even with the sequential version of our algorithm. The full potential of our parallel algorithm becomes apparent when counting $k$-cliques for larger $k$ on very large graphs.

Figure 5.1 shows the running time of the algorithms as a function of $k$, when executed on the very large graphs. It shows that our parallel algorithm using 10 threads is the only one that can count all 9-cliques for Friendster in a reasonable amount of time. It also shows that the running time of our algorithm grows more gracefully as a function of $k$.

We then investigate whether we achieve good degree of parallelism. Figure 5.2, shows the running time and the speedup[2] of our algorithm as a function of the number of threads employed. We can see that by employing $t$-threads we gain almost a factor of $t$ in terms of running time (e.g. with 30 threads we gain a factor of 26). This means that our algorithm is able to distribute (almost) uniformly the computational load across the threads. As a result, by employing a larger number of threads we can process larger graphs and deal with larger values of $k$. In particular, using 30 threads we were able to iterate over the 487 trillions 10-cliques of Friendster[3] in approximately 30 hours of computation.

### 5.4.2 k-Clique Core Decomposition and k-Clique Densest Subgraph Computation

In this section, we study the performance of our algorithms for computing a $k$-clique core decomposition as well as an approximation of the $k$-clique densest subgraph.

Figure 5.3 shows the number of nodes and the density of the subgraphs obtained by running the approximation algorithm for the $k$-densest subgraph as a function of

[2]The speedup is defined as the running time using one thread divided by the actual running time.

[3]Friendster has exactly 487,090,833,092,739 10-cliques.

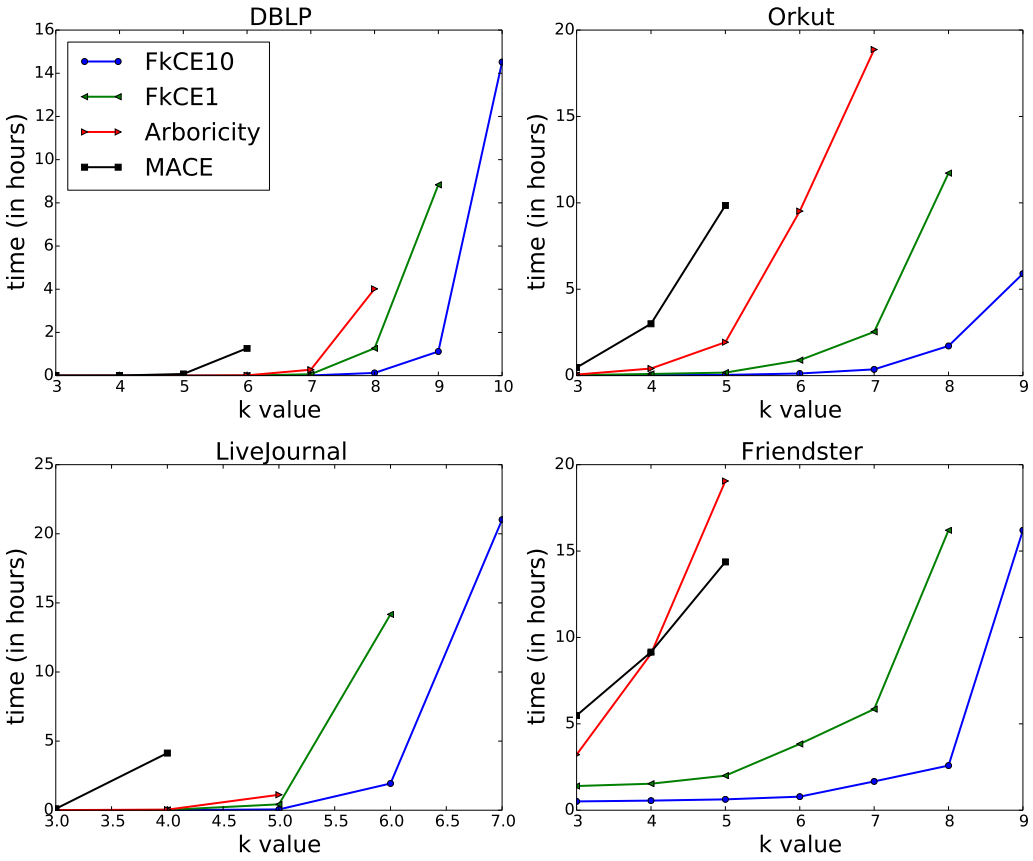


Figure 5.1: Running time of the algorithms as a function of k. We truncated the curves at 24 hours.

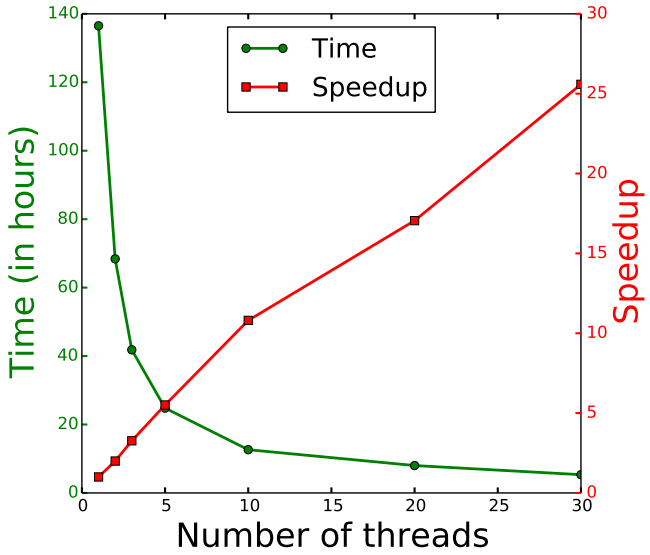


Figure 5.2: Running time and speedup obtained counting the 9 cliques on Friendster with our parallel algorithm as a function of the number of threads.

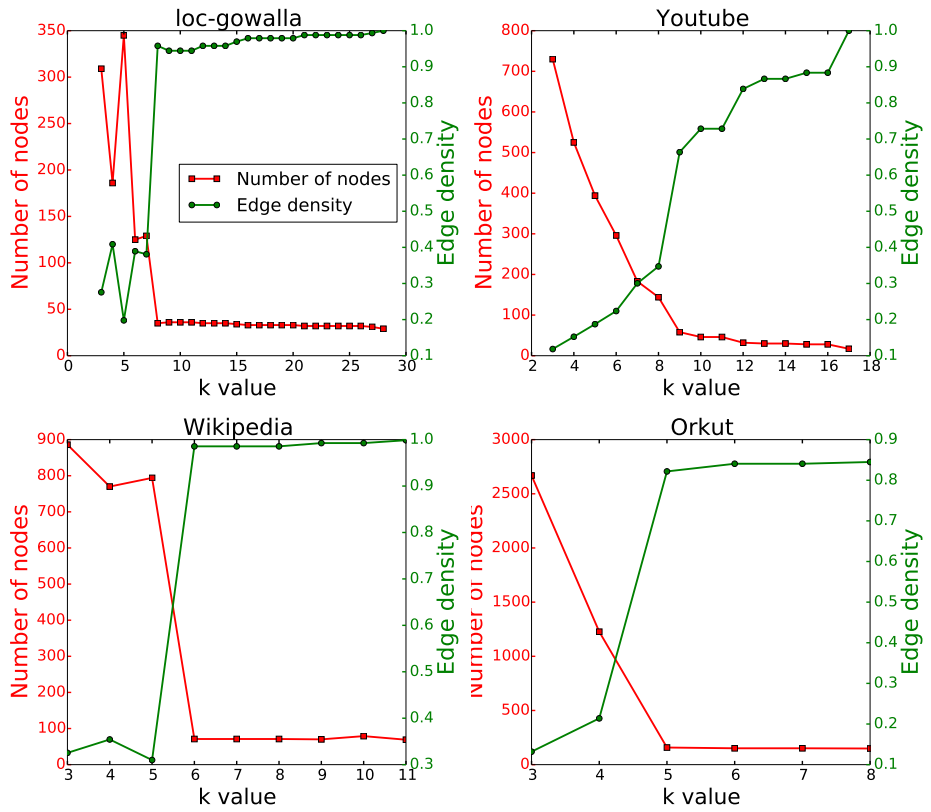


Figure 5.3: Number of nodes and edge density of the k-densest subgraph approximation versus k.

$k$, on two of our large graphs and two of our very large graphs. We can see that the edge density of the subgraphs found by our algorithm quickly converges towards 1, which is the density of a clique. This makes it an effective heuristic to find cliques and quasi cliques in large graphs.

| networks | $k = 3$ | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ | $k = 10$ |
|---|---|---|---|---|---|---|---|---|
| DBLP | 7s | 9s | 15s | 2m | 321m | 337m | >24h | >24h |
| Wikipedia | 2m | 4m | 8m | 9m | 19m | 31m | 86m | 453m |
| Orkut | 6m | 15m | 48m | 237m | 1025m | >24h | >24h | >24h |
| LiveJournal | 1m | 3m | 93m | >24h | >24h | >24h | >24h | >24h |
| Friendster | 186m | 309m | 348m | 583m | >24h | >24h | >24h | >24h |

Table 5.5: Running time of Algorithm 11 (in the sequential version) to compute the k-core decomposition and find a k-approximation of the k-clique densest subgraph as a function of k on our very large datasets.

Our results for the $k$-clique core decomposition are shown in Table 5.5. Here we measure the running time for computing such a decomposition as a function of $k$ on our very large graphs. Those results show that we can compute a 5-clique core decomposition in all graphs within a few hours and a 7-clique core decomposition in most of the cases in a reasonable amount of time.

## 5.5 Conclusion and Future Work

In this chapter, we present a parallel algorithm for listing or counting all $k$-cliques in very large real world graphs. Our algorithm leverages the sparsity of the input graphs and has the best known asymptotic running time, while requiring linear amount of memory in the size of the input. In practice, for large values of $k$, the sequential version of our algorithm is an order of magnitude faster than state-of-the-art algorithms for the same problem, while the parallel version allows for the gain of another order of magnitude achieving an almost optimal degree of parallelism. Our experimental analysis shows that our parallel algorithm is able to list all cliques in graphs containing up to tens of millions of edges, as well as all 10-cliques in graphs containing billions of edges, within a few minutes or a few hours, respectively.

We adapt our algorithm for (i) computing the $k$-clique core decomposition of a graph, and (ii) computing an approximation of the $k$-clique densest subgraph.

In the future, we would like to investigate further whether our algorithm could be successfully employed in graph compression, community and event detection. In particular, our $k$-clique core decomposition seems to be a promising tool for data mining.

# Chapter 6

# Detecting Crisis Events in Social Media

## 6.1 Introduction

Online social media, such as Twitter, represent a valuable source of information, as they might provide important insights into the offline world. In particular, event detection in social media has played a major role in data mining and social network analysis, with several research studies focusing on earthquakes [87], general real-life crisis situations [48], computer security breaks [85] and many other classes of events (see [10, 21, 48] for a survey).

Event detection has been focusing for a long time on traditional news media [5], however, online social media present non-trivial challenges as well as unprecedented opportunities. Tweets, for example, are short and written by non-experts using informal language while containing typos and abbreviations. Moreover, relevant information is often intertwined with noisy or non-interesting content such as spam or posts by so-called twitter bots. Another additional difficulty when analyzing data from Twitter lies on the fact that not all tweets are publicly available, with only a small fraction of them being accessible through the Twitter API.

On the other hand, tweets might provide novel information about well-known events or even about events that have not been covered at all or quickly dismissed by mainstream media. For example, the civil war in Yemen has received little attention in mainstream media [1], while in our collection of tweets we have found many tweets mentioning the situation. Similarly, the Romanian protests in 2013 were largely not covered during the first week of the event [2]. It would be interesting to analyze the tweets (if any) related to such an event. Tweets could also provide timely information about crisis events, the notable example being the airplane crash on the Hudson river in 2009 where eyewitnesses were the first to give news about the event through Twitter.

The American Red Cross (ARC) pointed out the effectiveness of social media and mobile apps[3] during crisis events. ARC opened its Social Media Digital Operations

[1] http://www.acnur.org/t3/fileadmin/Documentos/Publicaciones/2016/10449.pdf?view=1
[2] http://www.dw.com/en/protests-erupt-in-romania-over-gold-mine/a-17068049
[3] http://www.redcross.org/news/press-release/More-Americans-Using-Mobile-Apps-in-

Center for Humanitarian Relief, "demonstrating the growing importance of social media in emergency situations". Unfortunately, acquiring valuable information from social media presents some obstacles. According to a survey by the US Congressional Service, the administrative cost for monitoring multiple social media sources which typically produce large amounts of noisy data is uncertain [61].

Our work aims at shedding light on which crisis events, such as earthquakes, mass shootings, bombings, are covered by social media. In particular, we seek to determine whether events (such as the war in Yemen) that received little or no attention from mainstream media are covered on Twitter. Despite the huge efforts made by the research community (see [10, 21, 48] for a survey), a viable and satisfactory approach for finding events in Twitter has not emerged yet. Moreover, most of those approaches require a non-negligible effort to interpret the results making a large-scale study difficult to perform. To alleviate such a problem and allow a large-scale comparison, we define a variant of the event detection problem where we enforce a strict upper bound on the size of the output produced by the algorithm.

We develop a novel technique for event detection which satisfies those constraints. For example, in our experimental evaluation, we allow our algorithm to produce at most ten results (i.e. events), each one consisting in at most ten keywords describing the event. Our approach is based on the following steps: i) filtering of the tweets by retaining only those containing terms related to crisis events, such as shooting, earthquake, etc. To this end we use the lexicon built in [73]; ii) finding locations whose occurrence in tweets deviate significantly on a given time window from their average frequency; iii) a graph mining approach for providing additional information about the event. We call our approach EviDense, as our graph mining approach is based on finding "dense" regions in the co-occurrence graph.

The main idea behind our approach is to uniquely identify an event by its location and time, and not by the keywords describing it. There are two major advantages of our algorithm: when using a fine granularity there cannot be two important events occurring in the same place and time, and we can precisely describe each event due to the fact we allow the repetition of keywords, as for example "attack" can describe two situations occurring at the same time. A disadvantage comes from the fact that people might not precisely pinpoint the location of an event, and such we might miss events occurring simultaneously because of the use of too general locations, for example, names of cities. Most existing approaches compute similarities between words based on semantic and temporal features and then find clusters [101] or dense subgraphs [8], without taking into account the geographic aspect. Approaches that have considered the location suffer from a lack of redundancy of keywords describing events, due to the fact that location is another feature for clustering, where keywords are grouped together based on a similarity in semantic, temporal and geographic usage [1]. However, as we previously mentioned, keywords can describe more than one event, so any type of clustering in which a keyword represents just one point cannot provide satisfactory results. A method widely used for topic modeling, LDA, has been modified in [78] in order to take into account temporal and geographic

Emergencies

features. The approach in [78] represents a pioneering work, but it suffers from the fact that either the number of topics or the number of events has to be fixed.

We note that event detection is a more complex task than trend detection, which is a major focus on Twitter. The algorithms presented in the whitepaper [4] released by the data science team at the Twitter track every keyword that has an unusual usage, without taking into account if it is related to a real word event and without aggregating related keywords.

We forsee two main use-cases for our algorithm: detecting new crisis events as they occur and analyzing in retrospective the crisis events discussed on social media. In the following, we focus on the latter, first because we can evaluate our performance in comparison with state-of-the-art approaches in a straightforward manner, and second because previously discussed events can help us better understand the advantages of using social media for event detection. Our algorithm is evaluated against other approaches for a similar problem on a collection of tweets covering the period between November 2015 and February 2016. Such an evaluation shows that our approach outperforms the other approaches in terms of precision while delivering short summaries easy to analyze. We then conduct one of the most extensive evaluations for an event detection algorithm over a period of 14 months between November 2015 and December 2016. Our large-scale evaluation confirms the accuracy of the algorithm, while it provides more insights on what kind crisis events are covered on Twitter and how. During our evaluation, we found that part of the events we have found corresponded to *tweet storms*, organized events that focused on bringing awareness of the situation in Yemen. More particularly, only in the month of February 2016, there were five such events. In our future work, we will investigate the communities that form around tweet storms and how successful are this kind of awareness campaign.

We conclude this chapter with a comparison on the task of grouping semantically related terms in a graph-of-words. We evaluate four different algorithms: the densest subgraph, the heaviest-$k$ subgraph, the heaviest-$k$ clique and the heaviest quasi-clique. Our purpose is to show that the latter method, that we will explain in this chapter, is the best suited for the task.

### 6.1.1 Problem Definition

In order to allow a rigorous experimental evaluation, we give a precise definition of the problem we are going to study in our work. We are going to focus on *crisis events*, which are events that cause or might cause "dangerous" situations for a "large" group of people. For our purposes, a group of people is considered large if it includes five or more people. Crisis events include earthquakes, terrorist attacks, mass shootings, airstrikes, etc.

Our main goal is to develop an efficient algorithm for detecting crisis events and to investigate what is the potential of social media by looking at the events that have been previously covered. To allow a large-scale study, we define a novel variant of the well-known event detection problem which enforces a strict upper bound on

[4] https://blog.twitter.com/2015/trend-detection-social-data

the total size of the results. We can state our problem as follows:

**Definition 6.1.1** (Event Detection with Limited Output Size)**.** Given integers, $k, s > 0$ and a collection of documents (e.g. tweets), find $k$ distinct crisis events each one consisting of a set of at most $s$ keywords, including the location and the time of the event.

Although there are approaches that enforce some kind of constraint on the size of the results(e.g. [45]), such a variant of the event detection problem has not been formulated, to the best of our knowledge. In our experiments, we set both $k$ and $s$ to ten. This allows us to conduct a large-scale manual evaluation of the results on a collection of tweets collected over a period of 14 months. In our experimental evaluation, we shall see that even such a limited amount of information allows us to obtain meaningful insights on the coverage of crisis events in our collection of tweets.

## 6.2 Algorithms

Our algorithm consists of the following main steps: 1) collection of tweets containing keywords related to crisis events by means of the Twitter API; 2) recognition and tagging of mentions of locations in the tweets; 3) finding bursts of mentions of locations; 4) mentions of locations are finally complemented with related keywords so as to provide more informative results. Each of these steps is described in the following sections.

### 6.2.1 Collection of Tweets and Preprocessing

Tweets have been collected by means of the Twitter API while specifying a list of keywords related to crisis events, such as *attack*, *flood*, *victims*. To this end, we use the list of keywords provided in [73]. For the recognition and tagging of locations, we use an entity recognition tagger that was trained on Twitter data [84]. Such a tagger focuses on ten different categories: person, location, company, product, facility, tv show, movie, sports team, and band. We retain only location and facility tags while ignoring the others. We will refer to both tags locations and facilities as locations. Our intuition is that bursts in the mentions of a location ( in tweets dealing with crisis events ) might signal the happening of an important event in that location. After the tagging step, we filter the remaining words such that to remove stopwords, URLs and infrequent terms (i.e. terms with an hourly frequency smaller than 5).

### 6.2.2 Finding Bursts of Locations

When an event such as a crisis event occurs, we observe a burst of activity in Twitter with terms pertinent to the event increasing suddenly their frequency in tweets. In our approach (where tweets contain keywords related to crisis events), a burst in the number of occurrences of a location gives us a first signal that a crisis event is unfolding at that location. For each location, we compute a set of intervals in which

the deviation between the frequency of the location and its expected frequency is always above a threshold. Our intuition is that all tweets (dealing with the same location) posted during such an interval refer to the same event. We refer to such intervals as *interesting intervals*. We are interested in finding *maximal* interesting intervals. The expected frequency of a location is computed assuming that location frequencies can be approximated by the binomial distribution.

Our algorithm computes for every location and every maximal interesting interval of that location its frequency and how much it deviates from the expected frequency. Then, all the (location, maximal interesting interval) pairs are ranked according to how much the frequency of a location deviates from its average frequency. A larger deviation corresponds to a higher interest in the event, therefore we retain the top $k$ (location, maximal interesting interval) pairs with larger deviation from the average.

Below we provide more technical details. The expected frequency of a term (a location in our case) is computed as follows. Let $X_w$ be the random variable indicating the number of tweets containing a term $w$ while letting $N$ be the number of tweets. We assume that the probability of having $n$ tweets containing $w$, denoted as $P(X_w = n)$, can be approximated by a binomial distribution:

$$P(X_w = n) = \binom{N}{n} p_w^n (1 - p_w)^{N-n},$$

where $p_w$ is the probability of a tweet containing the term $w$.

Thus, we can compute the expected number of tweets containing a term $w$ and the standard deviation as follows:

$$E[X_w] = Np_w \quad \text{and} \quad \sigma[X_w] = \sqrt{Np_w(1 - p_w)}.$$

Let $f_{l,w}(t)$ be the frequency of a term $w$ in all tweets posted in the time window $[t, t + l]$. We let

$$\beta_{l,w}(t) = \frac{f_{l,w}(t) - E[X_w]}{\sigma[X_w]},$$

which measures how much $f_{l,w}(t)$ deviates from the expected frequency of the term $w$ (similarly to [104]). We say that an interval $\mathcal{I} = [a, b]$ is *interesting* with respect to a term $w$ if for all $t \in \mathcal{I}$, $\beta_{l,w}(t) \geq \alpha$. We say that an interesting interval $\mathcal{I} = [a, b]$ is maximal if for all $\bar{a} < a$ and $\bar{b} > b$, $[a, \bar{b}]$, and $[\bar{a}, b]$ are not interesting.

Given a maximal interesting interval $\mathcal{I}$, we define $\beta_{l,w}^{\max}(\mathcal{I}) = \max_{t \in \mathcal{I}} \beta_{l,w}(t)$, i.e. the maximum deviation from the expected frequency of $w$ in $\mathcal{I}$. When it is clear from the context we use the abbreviation $\beta_{\max}$.

Our goal is to find all the maximal interesting intervals for every term $w$ together with their corresponding $\beta_{max}$'s. We then retain only the top $k$ intervals with maximum $\beta_{max}$'s among all the terms. Algorithm 12 shows a pseudocode for computing all bursts of a given location, while Algorithm 14 shows a pseudocode for computing the top $k$ maximal interesting intervals together with their corresponding locations, as well as other relevant terms (to be discussed next).

**Algorithm 12** FindTermBurst$(\mathcal{C}, w, \alpha, \ell)$

**Input**: A collection of tweets $\mathcal{C}$ with their timestamps, a location $w$, a threshold $\alpha \geq 0$, a window size $\ell$
$S \leftarrow \varnothing, \beta_{max} \leftarrow 0, \mathcal{I} \leftarrow \varnothing$
$t \leftarrow minTimestamp(\mathcal{C})$
**while** $t \leq maxTimestamp(\mathcal{C}) - \ell$ **do**
  **if** $\beta_{\ell,w}(t) \geq \alpha$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup [t, t + \ell]$
    $\beta_{\max} \leftarrow \max(\beta_{\max}, \beta_{\ell,w}(t))$
  **end if**
  **if** $(\beta_{\ell,w}(t) < \alpha$ or $t = maxTimestamp(\mathcal{C}) - \ell)$ and $\mathcal{I} \neq \varnothing$ **then**
    $S \leftarrow S \cup (w, \mathcal{I}, \beta_{\max})$
    $\mathcal{I} \leftarrow \varnothing$, $\beta_{max} \leftarrow 0$
  **end if**
  $t \leftarrow t + 1$
**end while**
Return $S$

### 6.2.3 Finding Quasi-Cliques

In order to complement the set of locations (found with Algorithm 12) with additional information about the corresponding event, we employ a graph mining approach. In particular, for each location and each interesting interval for that location, we wish to find a set of terms which induce a dense region in the co-occurrence graph during that time interval. Given an interesting interval $\mathcal{I}$ and a collection of tweets, we define a weighted undirected graph $G_{\mathcal{I}} = (V_{\mathcal{I}}, E_{\mathcal{I}})$, where $V_{\mathcal{I}}$ consists of the set of terms in the collection of tweets, while there is an edge between two nodes if the corresponding terms co-occur in at least one tweet posted within $\mathcal{I}$. A weight function $c : E \to \mathbb{N}$ represents the number of co-occurrences of terms in tweets posted within $\mathcal{I}$.

Cliques and quasi-cliques are the dense subgraphs par excellence. Several definitions of weighted cliques have been provided in the literature, such as a subgraph of maximum total weight where any two nodes are connected [77], as well as a subgraph with maximum total weight and number of nodes no larger than a threshold provided in input [2]. Observe that both definitions would favor the graph on the left in Figure 6.1, which exhibit weak connections between the set of nodes $\{1, 2\}$ and {3,4}. As a result, those two different parts of the graph might actually refer to two different events. It is more likely that the nodes of the graph on the right in Figure 6.1 refer to the same event, as the edges of the graph have the same weight.

This motivates the following definitions of cliques and quasi-cliques. We say that a graph $H$ is a *weighted clique* if all pairs of nodes in $H$ are connected by an edge with the same weight. Given a parameter $\gamma > 0$, we then define a *weighted quasi-clique* as follows.

**Definition 6.2.1** (Weighted Quasi-Clique)**.** Given an undirected weighted graph $H = (V(H), E(H), w)$, $0 < \gamma \leq 1$, we say that $H$ is a *weighted $\gamma$-quasi-clique* if the

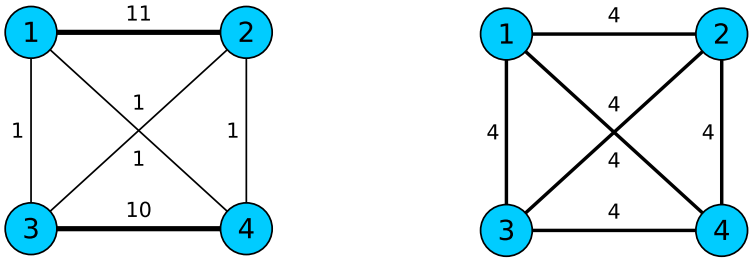


Figure 6.1: The left subgraph has a larger weight than the right one, but the nodes in the right subgraph are much better connected.

following holds:

$$\sum_{e \in E(H)} w(e) \geq \gamma \cdot w_{max}(H) \binom{|V(H)|}{2},$$

where $w_{max}(H) = \max_{e \in E(H)}(w(e))$.

We define the function $q_G : V \to (0, 1]$ ($q$ for short) to be the function which associates to every set $S \subseteq V$ a rational number $\gamma$ such that the subgraph induced by $S$ in $G$ is a $\gamma$-quasi-clique and $\gamma$ is the largest value for which this holds.

Finding quasi-cliques is an NP-hard problem, therefore we resort to the following heuristic for finding quasi-cliques containing a node $v$ and at most $s$ nodes. The algorithm starts with $v$ and adds the edge with maximum weight containing $v$. At any given step, let $S$ be set of current nodes. If $|S| = s$, the algorithm stops. Otherwise, it adds a node $x$ in the neighborhood of $S$ maximizing $q(S \cup \{x\})$ provided that by adding $x$ the resulting subgraph is still a $\gamma$-quasi-clique. Algorithm 13 shows a pseudocode, where $\Delta(u)$ denotes the sets of neighbors of $u$.

**Algorithm 13** FindQuasiClique($G, v, \gamma$, s)

**Input**: A graph $G = (V, E)$, a vertex $v$, $0 < \gamma \leq 1$
**Output**: A weighted quasi-clique containing $v$
$u \leftarrow \arg\max_{u:uv \in E} w(uv)$
$S \leftarrow \{u, v\}$
**while** true **do**
  if $|S| = s$ return $S$
  $\Delta(S) \leftarrow \cup_{u \in S} \Delta(u) \setminus S$
  **if** $\Delta(S) = \varnothing$ **then**
    return $S$
  **end if**
  $x \leftarrow \arg\max_{u \in \Delta(S)} q(S \cup \{u\})$
  **if** $q(S \cup \{x\}) < \gamma$ **then**
    return $S$
  **end if**
  $S \leftarrow S \cup \{x\}$
**end while**
Return $S$

### 6.2.4 EviDense and Analysis of its Running Time

Our algorithm for event detection can then be recapped as follows. The first step consists of collecting tweets from Twitter specifying as a filter a list of terms related to crisis events. Then, mentions of locations in tweets are recognized and tagged. A list of the top $k$ bursty locations is computed, each of them is then complemented with additional terms related to the same event by finding quasi-cliques in the co-occurrence graph. Algorithm 14 shows a pseudocode for our main algorithm. An additional step which is not detailed in the algorithm is checking for duplicate events, that is events that cover a large area, for example a tornado. In this case, the quasi clique contains more than one location and we mark all the locations as seen in that period of time, as to dismiss duplicate quasi-cliques.

**Algorithm 14** EviDense($\mathcal{C}, \gamma, \alpha, \ell, k, s$)

1: **Input**: A collection $\mathcal{C}$ of tweets each one containing at least one term related to crisis events (lexicon provided in [73]), $0 < \gamma \leq 1$, an integer $\alpha$, a window size $\ell$, integer $k, s$
2: **Output**: The list of the top $k$ events in $\mathcal{C}$
3: $S \leftarrow \varnothing$
4: Recognize and tag locations in $\mathcal{C}$ with [84]
5: **for** $w \in \mathit{Locations}(\mathcal{C})$ **do**
6: $\quad S \leftarrow S \cup$ FindTermBurst$(\mathcal{C}, w, \alpha, \ell, k)$
7: **end for**
8: Let $S_k$ be the top $k$ triples in $S$ with maximum $\beta_{\max}$'s
9: **for** $(w, \mathcal{I}, \beta) \in S_k$ **do**
10: $\quad$ Compute $G_{\mathcal{I}}$
11: $\quad$ **output** (FindQuasiClique ( $G_{\mathcal{I}}, w, \gamma, s$), $\mathcal{I}$)
12: **end for**

To ease the presentation, Algorithm 12 and Algorithm 14 show a simple pseudocode of the main steps of our algorithm. However, our algorithm can be efficiently implemented so that it requires a few constant passes over the collection of tweets. For each location we store its frequency in the current time window (i.e. $f_{l,w}(t)$) together with the corresponding $\mathcal{I}, \beta_{\max}$. Every time we process a new tweet, we verify for every location in that tweet the conditions from Algorithm 12, lines 5-10. Let $c$ be the maximum number of terms in a tweet. We can then claim that the total running time of Algorithm 12 is $O(c|\mathcal{C}|)$.

Let $n, m$ be the number of nodes and the number of edges in $G$, respectively (i.e. the input of Algorithm 13). For line 3 we require $O(n)$ operations, for line 7 we need at each iteration $O(n)$ operations, while the computation at line 10 requires in the worst case to process all the edges in $G$. Thus, the overall running time of Algorithm 13 is $O(s(m+n))$. We now evaluate the running time of Algorithm 14, lines 5-10, that is, excluding the tagger. Line 7 is implemented using a heap of size $k$, requiring a number of operation of $O(|\mathcal{C}|c \log k)$. Any graph constructed from a collection of at most $|\mathcal{C}|$ tweets will have at most $c^2|\mathcal{C}|$ edges and $c|\mathcal{C}|$ nodes. For computing a graph $G_{\mathcal{I}}$, we need at most $O(c^2|\mathcal{C}|)$ operations. Then, the running

time for lines 8-10 is $O(ksc^2|\mathcal{C}|)$, which implies a total running time of Algorithm 14 of $O(T + ksc^2|\mathcal{C}|)$, where $T$ is the running time required by the tagger. Notice that in practice $k, s, c$ are small constants.

## 6.3 Experimental Evaluation

### 6.3.1 Experimental Settings

**Corpora.** We collected tweets posted over a period of 14 months between November 2015 and December 2016. We use the Twitter Streaming API while filtering the tweets so that they contain at least one term related to crisis events. To this purpose, we specify in the filter settings of the Twitter API the lexicon provided in [73] and the English language. We obtained 10M tweets in total, which we divide in fourteen datasets (one per month). We then use an entity recognition tagger [84] for recognizing and tagging the mentions of locations in the tweets. The first four months in our dataset are used to evaluate our approach against other approaches for event detection. A more extensive evaluation of our approach is then conducted over the period of 14 months.

**Related approaches.** The literature on event detection is vast, surveys can be found in [10, 48, 62, 71]. In our experimental evaluation, we considered those approaches that can be used for a large-scale analysis over a long period of time. Supervised or semi-supervised approaches requires a significant amount of labeled data, as the topics discussed in Twitter might change quickly over time. Therefore, we focus on unsupervised approaches. In particular, we focus on those approaches that enforce or allow to enforce some kind of constraint on the output size. We consider the following three methods: MABED [45], StaticDens [8, 59], and EDCoW [101]. MABED allows to specify a bound on the number of results produced by the algorithm, StaticDens is a graph mining algorithm (similarly to our approach), while EDCoW is the algorithm that performed best in a recent evaluation [100] (it has not been compared against the previous two methods).

We run the four algorithms on each of the first four months of our dataset and evaluate the top 10 as well as top 20 results for each such dataset. StaticDens and EDCoW provide a measure of relevance for their results which we use to determine the top results. The code for [8] is not available, however, we could obtain the code for [59] which is similar in spirit and can be used as a proxy. We refer to such algorithm as StaticDens as it can be seen as a static version of [8].

Observe that not all the approaches included in our evaluation are originally designed for detecting crisis events. We ensure a fair evaluation by running them on the same input dataset. EviDense and MABED produce a list of terms as well as a time interval specifying when a given event takes place, while StaticDens and EDCoWproduce in output only a list of terms. In the latter case, we assume the time interval to be the month corresponding to the input dataset.

**Parameter settings.** We run MABED using the implementation provided by the authors [5] and the setting specified in the original paper [45], that is $p = 10, \theta = 0.7$

[5] https://github.com/AdrienGuille/MABED

and $\sigma = 0.5$. The parameter $\sigma$ controls the similarity between the events produced in the results. Small values of sigma correspond to more diverse results. We observe that determining the right value for such a parameter is crucial in the MABED approach, as illustrated in the section containing the results of the approaches. In particular, if $\sigma$ is too small one may not get enough relevant results, while with large values of $\sigma$ the algorithm might return many duplicates of the same "popular" event.

We run StaticDens using as a measure of density the total edge weight of the subgraph, whose number of nodes has been limited to $s = 10$. We compute the co-occurrence graph over the entire input dataset and retrieve the top $k = 10$ disjoint subgraphs with large weight.

For the EDCoW algorithm we use the implementation of [100] and we set the parameters as follows: the size of first level of intervals is $s = 100s$, while we take $\Delta = 32$, setting a size of $3200s$ for the second-level intervals and, as in [100], we set $\gamma = 1$. As EDCoW does not enforce any constraint on the size of the output, we order the results according to $\epsilon$ (as defined in the original paper), which measures the relevance of the results and we retain only the top $k$ results.

We run our algorithm with parameters $s, k$ set to ten. In our approach, we set $\ell$ to three hours, $\alpha = 8$ and $\gamma = 0.4$. We discuss how to choose the values of the parameters of EviDense in Section 6.3.2.

**Evaluation.** We evaluate the precision at $k$ (which is widely used in information retrieval) for all four methods on the first four months of our dataset. Let $A_k$ be the set of distinct events in the top $k$ results of the algorithms while letting $\mathcal{E}$ be the set of distinct events in the collection $\mathcal{C}$ of tweets. Precision at $k$, denoted with $P@k$ (or precision for short), is defined as follows:

$$P@k := \frac{|A_k \cap \mathcal{E}|}{k}.$$

In order to understand if a set of terms refers to a crisis event, we perform a manual evaluation with the help of *news.google.com*, the Google search engine for news. The results of a given method consist of a list of terms and an associated time interval which are included in a Google query. We then check whether the event occurs on the first page of the results of google news and it is a crisis event. If this is the case and at least 50% of the query keywords are related to the event, the result is considered accurate. In order to provide an evaluation which is as objective as possible, we also include the output of our algorithm in Table 6.5.

**Running time.** We run our experiments on a Linux machine equipped with 4 processors Intel Xeon CPU E7-4870 @ 2.40 GHz as well as 10 cores split into 2 threads each (a total of 80 threads) and 64 G of RAM DDR3 1333 MHz. Annotating the entities in the tweets using the entity recognition tagger [84] takes on average 0.02s/tweet. The average running time for the algorithms (without taking into account the annotating phase) is the following: for MABED is around 90s, for StaticDens is around 21s, for EDCoW is 52min and around 26s for EviDense.

### 6.3.2 Parameter Settings for EviDense

Our algorithm has four input parameters: $k, \ell, \alpha$ and $\gamma$. For the experiments we used the values $k = 10$ or $k = 20$, we set $\ell$ to three hours, $\alpha = 8$ and $\gamma = 0.4$. The first parameter $k$ is used to limit the output size. The size of the time-window, $\ell$, has two functionalities: firstly, it has a smoothing effect over the number of occurrences of a term and can alleviate the difference between active periods and inactive periods, such as day and night. Secondly, because of the smoothing effect, it can decrease the importance of small events which have short-lived peaks and increase the importance of events which are discussed for longer periods but without having important peaks. We experiment with different values of $\ell$ in order to balance between the segmentation of a single event and the order of magnitude of an event. In order to assess what is the best value for the time window, we compute the percentage of duplicate events between the events retrieved, the *DeRate*. We only include an evaluation on the top 20 events in the month of December 2015, as the results are similar on the other datasets. In Figure 6.2, we can notice that the number of duplicate events decreases as the time window increases, which is intuitive, as a larger time window decreases the effect of an inactive period. We settled for three hours, as choosing larger time windows would likely merge different events that occur in the same location. Moreover, larger time windows would also increase the running time of the algorithm, as shown in Figure 6.3. This is due to the fact that the search for weighted quasi-cliques is performed in a larger graph.

The parameter $\alpha$ is used as a threshold for determining which location has a frequency which deviates significantly from the average frequency. We use the binomial distribution as a simple model that gives an estimate of a mean and standard deviation, and we try several values for $\alpha$, as shown in Figure 6.4. When varying the parameter $\alpha$ the average length of interesting intervals decreases (Figure 6.5) and given that our goal is to minimize the length of interesting intervals without reporting the same event twice, we select the value $\alpha = 8$. The last parameter, $\gamma$, is used to provide different levels of detail concerning an event. We have obtained good results for a large range of values of $\gamma$. The best is to avoid values smaller than 0.3 which might add unrelated terms to an event, and values larger than 0.8 as the description of the events could be not informative.

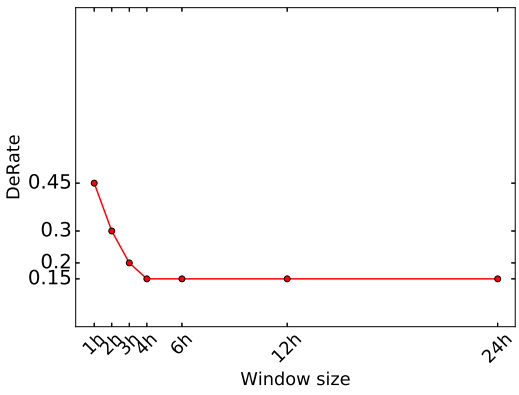


Figure 6.2: DeRate vs $\ell$

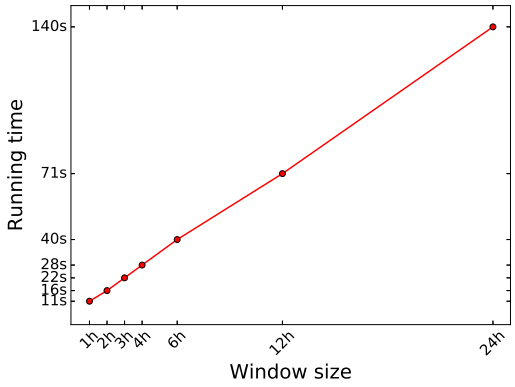


Figure 6.3: Running time vs $\ell$

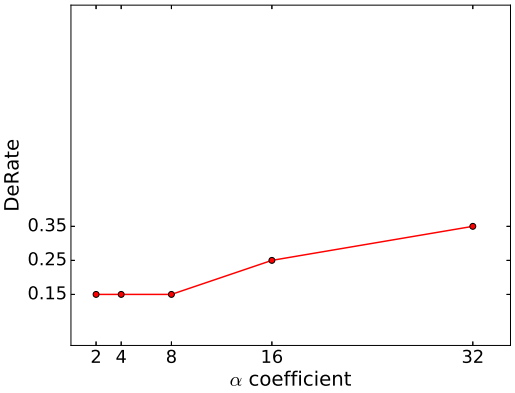


Figure 6.4: DeRate vs $\alpha$

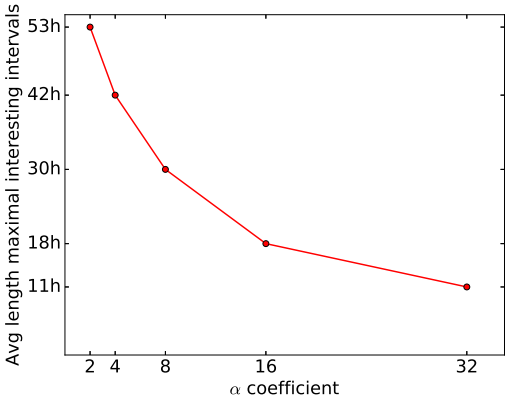


Figure 6.5: Avg length of maximal interesting intervals vs $\alpha$

### 6.3.3 Evaluation of the Algorithms

In Table A.9 and Table A.10 we present precision at 10 and precision at 20 for all the algorithms, respectively. We observe that EviDense performs much better than

the other three algorithms in terms of precision on all datasets for both top-10 and top-20 results.

| | EviDense | MABED | StaticDens | EDCoW |
|---|---|---|---|---|
| November 2015 | **0.400** | 0.300 | **0.400** | 0.200 |
| December 2015 | **0.600** | 0.200 | 0.400 | 0.200 |
| January 2016 | **0.700** | 0.300 | 0.300 | 0.300 |
| February 2016 | **0.800** | 0.400 | 0.200 | 0.000 |

Table 6.1: Precision on datasets for top 10. We highlight the best results.

| | EviDense | MABED | StaticDens | EDCoW |
|---|---|---|---|---|
| November 2015 | **0.350** | 0.250 | 0.250 | 0.150 |
| December 2015 | **0.600** | 0.250 | 0.300 | 0.200 |
| January 2016 | **0.650** | 0.250 | 0.300 | 0.150 |
| February 2016 | **0.500** | 0.400 | 0.100 | 0.150 |

Table 6.2: Precision on our datasets for top 20. We highlight the best results.

In order to provide an objective experimental evaluation and to get a better understanding of our results, we report the top 10 results for the algorithms we evaluated. We first report the results provided by EviDense and MABED for the November 2015 dataset on Table 6.3 and Table 6.4. While both algorithms report the terrorist attack several times, we can see that in the case of EviDense two of the mentions refer to different subevents (Disneyland was closed, the Eiffel Tower was lighted in the colors of the French flag), and the other two mentions occur because people use different levels of granularity for location (Paris vs France). In the case of MABED, there isn't a clear distinction between the six events related to the attacks. This happens because MABED finds several "event terms", however, the similarity function cannot correctly identify duplicate events. The authors use the correlation coefficient [36] of the time series of term occurrences in order to compute the similarity and we suspect that because two keywords might be connected to the same event but also to other different events, their correlation is very low. The results confirm our intuition that the best identifiers of an event are the location and time-window, and not a particular keyword or set of keywords.

We report the results for EviDense and MABED for the dataset December-2015 in Table 6.5 and Table 6.6 for the sake of completeness. In this dataset EviDense finds only a duplicate event, again because of the different granularity levels used in the tweets. However, MABED finds the San Bernardino event four times, without a reference to subevents connected to the shooting. In this example, we can see that both approaches found the tornado sweeping the region of Dallas, Texas as being the same as the one that hit Rowlett. In the case of EviDense if one quasi-clique contains another location apart from the original one, we remove the quasi-clique

| Time interval (UTC) | Event keywords | Description |
|---|---|---|
| Nov 13 22:06 , Nov 28 10:53 | **Paris**, attacks, breaking, concert, dead, french, hall, hostages, killed | The November 13th terrorist attacks in Paris. |
| Nov 13 22:10 , Nov 21 04:12 | **France**, amp, attack, attacks, bbcbreaking, closes, declares, emergency | The November 13th terrorist attacks in Paris. |
| Nov 14 01:15 , Nov 15 11:12 | **Eiffel Tower**, attacks, dark, deadly, lights, memory | The November 13th terrorist attacks in Paris. |
| Nov 14 01:15 , Nov 15 11:12 | **Lebanon**, attack, Baghdad, Beirut, Japan, Mexico, entire, Halsey, matter | Not a clear event. |
| Nov 20 09:49 , Nov 23 19:31 | **Mali**, 170, attack, gunmen, hostage, hostages, hotel | Gunmen take 170 hostages in a Mali hotel |
| Nov 08 19:11 , Nov 09 02:26 | **Wembley**, ashton5sos, bbcr1, blast, brilliant, closing, energy, full, show | Not a clear event. |
| Nov 14 12:09 , Nov 15 09:05 | **Disneyland Paris**, 1992, attacks, closes, daily, girlposts, horrendous, opened, time | The November 13th terrorist attacks in Paris. |
| Nov 17 19:27 , Nov 18 06:22 | **Hannover**, Germany, bomb, breaking, cancelled, game, soccer, stadium. | Game cancelled in Hannover due to bomb threat. |
| Nov 27 20:56 , Nov 28 11:38 | **Colorado Springs**, active, Colorado, contained, injured, planned, shooter | Man kills several people at a Colorado clinic. |
| Nov 24 17:11, Nov 28 00:27 | **Chicago**, Burger King, Laquan Mcdonald, deleted, footage, murder, police, publicly, stated | Not a crisis event, Chicago police is suspected of having destroyed evidence accusing some policemen of murder. |

Table 6.3: Top 10 events discovered in November 2015 by EviDense. The event is centered around the location given in bold.

| Time interval (UTC) | Event keywords | Description |
|---|---|---|
| Nov 13 20:30, Nov 15 14:00 | **religion**, terrorism, terrorists, muslims, aren't, terrorist, pass | Not a clear event. |
| Nov 13 20:00, Nov 15 16:00 | **families**, paris, prayers, thoughts, attacks, prayforparis, tragedy, terrorist, france, praying | The November 13th terrorist attacks in Paris. |
| Nov 13 20:00, Nov 14 17:30 | **safe**, paris, stay, safety, prayers, heart, thoughts, amp | The November 13th terrorist attacks in Paris. |
| Nov 13 20:00, Nov 14 09:30 | **concert**, paris, hall, police, 100, bataclan, killed, breaking, dead, hostages, attackers, venue | The November 13th terrorist attacks in Paris. |
| Nov 13 20:00, Nov 14 21:00 | **sad**, paris, prayers, thoughts, situation, involved, evening, occurred, niallofficial | The November 13th terrorist attacks in Paris. |
| Nov 20 07:00, Nov 22 11:00 | **mali**, hotel, gunmen, hostages, bamako, radisson, hostage, 170 | Gunmen take 170 hostages in a Mali hotel. |
| Nov 13 20:00, Nov 16 05:30 | **pray**, prayforparis, victims, attacks, amp | The November 13th terrorist attacks in Paris. |
| Nov 13 19:00, Nov 14 16:30. | **heard**, paris, prayers, prayforparis, show, happened, justinbieber, france | The November 13th terrorist attacks in Paris. |
| Nov 13 20:00, Nov 16 06:30 | **lebanon**, paris, isis, media | Not a clear event. |
| Nov 27 15:00, Nov 30 06:00 | **planned**, parenthood, shooting, colorado | Man kills several people at a Colorado clinic. |

Table 6.4: Top 10 events discovered in November 2015 by the MABED. The event is centered around the term given in bold.

containing that second location, if the time intervals overlap. This allows us to account for correlated events occurring in different locations as if the events are correlated than it is very likely the two locations will be connected by a heavy edge in the graph corresponding to the event timeframe.

| Time interval (UTC) | Event keywords | Description |
|---|---|---|
| Dec 03 02:20 , Dec 07 08:50 | **San Bernardino**, dead, female, #sanbernardino, killed, male, police, shooting | The San Bernardino terrorist attack. |
| Dec 07 03:24 , Dec 10 21:11 | **Chennai**, damaged, floods, fresh, issue, lost, passport, psks, sushmaswaraj | The Chennai floods: "passport" occurs because the Minister of External Affairs announced that Chennai residents whose passports were damaged during the floods could get a new one. |
| Dec 27 01:04 , Dec 28 03:37 | **Dallas**, Rowlett, Texas, area, large, tornado | Many tornadoes in the Dallas, Texas area, including close to Rowlett. |
| Dec 04 09:12 , Dec 04 23:23 | **Cairo**, attack, firebomb, killed, nightclub, people, restaurant | People killed in Cairo at a restaurant that operated also as a nightclub. |
| Dec 28 20:15 , Dec 29 09:21 | **Cleveland**, 12-year-old, Tamir Rice, charged, death, grand, indict, jury, police, prosecutor | Not a crisis event. |
| Dec 03 02:33 , Dec 05 08:02 | **California**, #sanbernardino, killed, people, police, shooting, suspect | The San Bernardino terrorist attack. |
| Dec 01 13:32, Dec 06 23:03 | **Chennai**, find, floods, girl, lost, parents, pls, share, shrutihaasan | Not a crisis event or a duplicate event. |
| Dec 21 04:56, Dec 22 00:57 | **Las Vegas**, crash, critical, dozens, driver, injured, pedestrians, people, strip | Driver in Las Vegas deliberately attacked pedestrians. |
| Dec 18 16:40 , Dec 19 02:32 | **republican**, agrabah, aladdin, bombing, country, #nottheoni, nationally, ppppolls, primary | Not a crisis event, also we can notice a mislabelling of the tagger. |
| Dec 07 13:31, Dec 08 08:59 | **Pearl Harbor**, attack, #pearlharbor, honor, lives, lost, remember, today, years | Commemoration of the attack on Pearl Harbor 74 years ago today. |

Table 6.5: Top 10 events discovered in December 2015 by EviDense. The event is centered around the location given in bold.

In Table 6.7 we report the results for StaticDens. The first impression is that using only entities to identify an event makes it impossible to know the na-

| Time interval (UTC) | Event keywords | Description |
|---|---|---|
| Dec 02 17:30 , | **san**, bernardino, shooting, sanbernardino, victims, police, 14, dead, suspects | The San Bernardino terrorist attack. |
| Dec 02 18:00, Dec 04 07:00 | **thoughts**, prayers, igorvolsky, nra, gun, violence | Not a crisis event. |
| Dec 26 16:30, Dec 28 04:30 | **tornado**, texas, dallas, rowlett | Many tornadoes in the Dallas, Texas area, including close to Rowlett. |
| Dec 02 17:30, Dec 04 14:30 | **mass**, shooting, shootings, prayers, dead | Not a crisis event. |
| Dec 02 18:00, Dec 04 05:30 | **sanbernadino**, prayers, police, thoughts | The San Bernardino terrorist attack. |
| Dec 07 20:00, Dec 11 03:30 | **trump**, donald, eagle, bald. | Not a crisis event. |
| Dec 02 18:00, Dec 04, 07:00 | **sanbernardino**, shooting, police, 14, dead, suspects, injured | The San Bernardino terrorist attack. |
| Dec 16 15:00, Dec 17 09:00. | **magicbus**, magicbusindia, faasos, hardwell, help, rs, donate | Not a crisis event. |
| Dec 15 21:00, Dec 16 09:30 | **gopdebate**, rt, terrorism, terrorists, military, trump, terrorist, attack | Not a crisis event. |
| Dec 02 17:30, Dec 04 06:30 | **california**, shooting, police, san, bernardino | The San Bernardino terrorist attack. |

Table 6.6: Top 10 events discovered in December 2015 by the MABED. The event is centered around the term given in bold.

ture of the event. While in this evaluation we don't allow any overlap, in the original approach dense subgraphs are considered different if there is at least one node that is not shared. While this can be helpful for finding similar events occurring in different locations, it can also lead to many duplicate events as it is not clear what uniquely identifies an event in their approach. For example, if four entities $e_1, e_2, e_3$ participate in an event, StaticDens could find all the subgraphs $\{e_1, e_2\}, \{e_2, e_3\}, \{e_1, e_3\}, \{e_1, e_2, e_3\}$ as representing different events. Another possible issue is the that two events can be merged together if we search only for dense subgraphs with a size equal to $k$, as will show in Section 6.4.

| Event keywords | Description |
|---|---|
| France, Paris, ISIS, Syria | France starts a massive bombing of Syria following the attacks on Paris. |
| Beirut, Lebanon, Baghdad, Japan, Mexico | Not a clear event. |
| Belgium, Brussels, Paris, Obama | The Belgium justice minister announces that a number of arrests were made in Brussels in connection with the Paris attacks. |
| Vladimir Putin, Taiwan, China, God | Not a clear event. |
| Egypt, Russia, Russian, Sinai, Turkey | Russian plane crash in Sinai, Egypt. |
| Afghanistan, Iraq, Pakistan, India, Libya | Not a clear event. |
| Nigeria, Facebook, Bokoharam, Britain, Twitter | Not a clear event. |
| Gunmen, Mali, Bamako, Radisson Blu, Radisson Hotel | Gunmen take 170 hostages in a Mali hotel |
| University, Missouri, CA, Justin Bieber, Los Angeles | Not a clear event. |
| Burger King, Colorado Springs, Colorado, Chicago, Planned Parenthood | Man kills several people at a Colorado clinic. |

Table 6.7: Top 10 events discovered in November 2015 by StaticDens. We consider an event to be a subgraph where at least 50% of the nodes refer to the occurring.

Finally, we present the results of algorithm EDCoW on the dataset November-2015. The implementation that we used differs slightly from the original, as after the clustering of terms, clusters containing less than five keywords are increased with co-occurring terms in order to be more descriptive, which explains why some keywords appear more than once. While the events are more descriptive than the ones found by StaticDens however, because of the lack of entities it is difficult to pinpoint the exact event a set of keywords refers to. The EDCoW algorithm has difficulties also in detecting duplicate events as the Paris attack appears two times in the top 10. Although a sophisticated technique, this algorithm computes the similarity between keywords only based on their frequency in time and has as unique identifier for an event a set of keywords. As a consequence of the first feature, keywords that have a comparable pattern of use but represent different events will be grouped together, while because of the second feature, similar but unrelated events will be represented

as a single event.

| Event keywords | Description |
|---|---|
| arrested, son, man, police, news | Not a clear event. |
| childs, nationaliloveyouday, life, donate, toysfortots | Not a clear event. |
| checked, dozens, automatically, unfollowed | Not a clear event. |
| affected, words, paris, thoughts, prayers | The November 13th terrorist attacks in Paris. |
| died, heartbreaking, paris, attacks, attack | The November 13th terrorist attacks in Paris. |
| attacks, mali, watch, paris, terror | Gunmen take 170 hostages in a Mali hotel. |
| injured, tips, killed, shooting, police | Not a clear event. |
| google, police, news, android, stream | Not a clear event. |
| military, seconds, live, listen, family | Not a clear event. |
| armed, heart, police, forces, robbery | Not a clear event. |

Table 6.8: Top 10 events discovered in November 2015 by EDCoW. We consider an event to be a subgraph where at least 50% of the nodes refer to the occurring.

### 6.3.4 Large-Scale Analysis and Findings

We evaluate our approach on a period of 14 months of Twitter data, from November 2015 to December 2016, resulting in one of the most extensive evaluations for an event detection algorithm. We also conduct an analysis of the results, providing interesting insights on the coverage of crisis events in social media.

We divide our dataset in 14 smaller datasets, each one corresponding to a month in our timeframe. For each month we compute the top-10 results, resulting in a total of 140 events, which are analyzed so as to evaluate the precision of our method. We then categorize the events as follows:

- *natural disaster*: "any event or force of nature that has catastrophic consequences, such as avalanche, earthquake, flood, forest fire, hurricane, lightning, tornado, tsunami, and volcanic eruption" [6];
- *armed conflict*: "a contested incompatibility which concerns government and/or territory where the use of armed force between two parties, of which at least one is the government of a state" [7];
- *terrorist attack*: "a surprise attack involving the deliberate use of violence against civilians in the hope of attaining political or religious aims" [8];

[6] www.dictionary.com
[7] www.pcr.uu.se
[8] www.vocabulary.com

- *violent attack*: used here to denote the violent actions of an individual that is not motivated by political or religious gains, but by racial hatred or psychological instability;

- *man-made disaster*: "a disastrous event caused directly and principally by one or more identifiable deliberate or negligent human actions." [9].

We obtain the following results: 82 out of 140 results refer to distinct crisis events, giving a precision of 0.585 for our method. From the 82 events, 27 are about terrorist attacks ( 33%), 21 natural disasters( 26%), 19 violent attacks ( 23%), 8 man-made disasters ( 9%), and 7 armed conflicts ( 8%). We note that a large ratio of events is about the actions of an individual or a group of individuals against unarmed civilians (terrorist attacks, violent attacks). Although the results obtained in those months are influenced by the actual events happening around the world, we can see an imbalanced in the armed conflict events which can be due to the fact we analyzed English tweets, while the areas involved in wars are not English speaking.

Perhaps our most interesting finding is the coverage of the conflict in Yemen. The conflict in Yemen has been widely overlooked by the international media, according to a report available at the website of the UN Refugee Agency [10]: "The international media and political discourse have widely overlooked the human narrative and widespread and suffering in Yemen and there has been little political resolve to stop the violence and improve humanitarian access..". After inspecting the tweets related to those events, it seems that the very fact that such a conflict has not received enough attention by the international media is the main motive driving Twitter users to post about Yemen. At a closer look, we discovered that much of the activity on Twitter related to Yemen was due to tweet storms, organized events that occur at a certain date and time and have as purpose bringing awareness on certain topics. For example, we found five tweetstorms in the month of February 2016, which correspond to the peaks represented in Figure 6.6. Each tweet storm was announced with some time in advance, as shown in Table 6.9. This is not the case for all months, as for example in December 2015 a peak in activity corresponds to the killing of the governor of Yemen's city of Aden. We show in Table 6.10, samples of tweets written during a tweet storm and tweets written in the aftermaths of an event. In our future work, we will investigate how communities are formed around tweet storms, how fast new users participate in storms and for how long they remain interested in the topic.

[9] www.businessdictionary.com

[10] http://www.acnur.org/t3/fileadmin/Documentos/Publicaciones/2016/10449.pdf?view=1

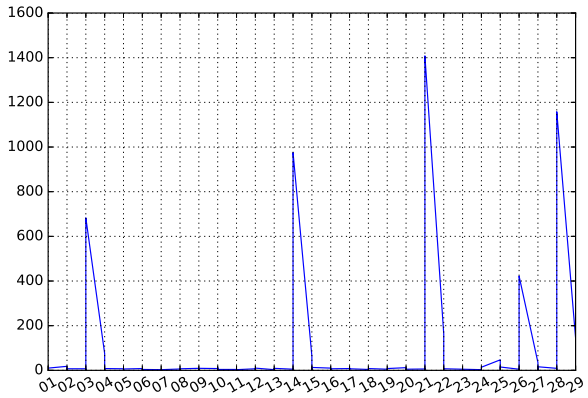


Figure 6.6: Mentions of "Yemen" during February 2016. Each peak corresponds to a tweet storm.

| Tweet timestamp (UTC) | Tweet text |
|---|---|
| 31 Jan 2016, 13:07 | 10 months of #US backed #Saudi-led massacres in #Yemen, break the silence #tweetstorm February 3rd at 17.00 UTC. |
| 13 Feb 2016, 18:43 | #TWEETSTORM for #Yemen, February 14th at 17.00 UTC/GMT. Join us to break the silence. |
| 18 Feb 2016, 09:35 | TWEETSTORM for #Yemen, February 21th at 17.00 UTC/GMT. Join us to break the silence. |
| 25 Feb 2016, 18:57 | TWEETSTORM for #Yemen, February 26th at 17.00 UTC/GMT. Join us to break the silence. |
| 27 Feb 2016, 18:13 | TWEETSTORM for #Yemen, February 28th at 17.00 UTC/GMT. Join us to break the silence. |

Table 6.9: Early tweets announcing the tweetstorms. We notice that the tweets tend to follow the same format.

| Event tweets | Storm tweets |
|---|---|
| Car Bomb Attack Kills Governor Of Yemen's Aden | How many victims does the world need to admit that Yemenis are humans and need our support ? #Yemen #USAkillYemenis |
| What a mess — Yemen's Aden governor killed in RPG attack | Any media out there? #Yemen under US-backed Saudi-led attack for 10 months yet world silence @UN-Geneva #YemenCrisis #USAkillYemenis |
| RT @TheArabSource: 16 civilians killed by militant group at Yemeni nursing home | In #Sana alone +13 major factories/companies destroyed in US-backed Saudi-led airstrikes #USAkillYemenis |
| Isis blamed for Yemen care home attack, Pope 'shocked' | STOP bombing civilian@lassecgen @SaudiEmbassyUSA @SaudiEmbassyUK Stop the War on #YEMEN @UN #10MonthsOfSaudiWar |
| 4 nuns, 12 others killed in Yemen elderly home attack | #Yemen is now the world's worst humanitarian crisis |

Table 6.10: A sample of tweets written in the aftermath of an event and a sample of tweets written during a tweetstorm.

## 6.4 Case Study: Dense Subgraphs for Event Description

In this section, we analyze several types of dense subgraphs which satisfy slightly different properties. The goal is to show that the quasi-clique notion that we introduced in this chapter is the best suited for event detection. The first type of dense subgraph we consider is a *densest subgraph* which we introduced in Chapter 3 for unweighted graphs. Next, we examine the *heaviest-k-subgraph* presented in Chapter 4, that is a dense subgraph with $k$ nodes. In Chapter 5 we studied $k$-cliques and we will be concerned in this evaluation with the *heaviest k-clique*. Finally, we look at the heaviest *quasi-clique* as defined in Section 6.2.3. We recall some basic notions and the definitions of all the types of subgraphs:

**Definition 6.4.1** (Average Degree Density)**.** Given an undirected weighted graph $G = (V, E, w)$, we define the *average degree density* $\rho(G)$ to be

$$\rho(G) = \frac{\sum_{e \in E} w(e)}{|V|}$$

**Definition 6.4.2** (Edge density)**.** Given an undirected weighted graph $G = (V, E, w)$, we define the *edge density* of $G$ to be

$$\rho_e(G) = \frac{2|E|}{|V|(|V|-1)}$$

**Definition 6.4.3** (Weighted Quasi-Clique)**.** Given an undirected weighted graph $G = (V, E, w)$, a rational number $\gamma$, $0 < \gamma \leq 1$, $G$ is a *weighted $\gamma$-quasi-clique* if

$$\sum_{e \in E} w(e) \geq \gamma \cdot w_{max}(G) \binom{|V|}{2},$$

where $w_{max}(G) = \max_{e \in E}(w(e))$.

We define the function $q_G : V \rightarrow (0, 1]$ ($q$ for short) to be the function which associates to every set $S \subseteq V$ a rational number $\gamma$ such that the subgraph induced by $S$ in $G$ is a $\gamma$-quasi-clique and $\gamma$ is the largest value for which this holds.

**Definition 6.4.4** (Heaviest Subgraph Problem)**.** Given an undirected weighted graph $G = (V, E, w)$, find an induced subgraph $H = (V(H), E(H))$ of $G$ such that

$$\rho(H) = \max_{\forall S \subseteq G} \{\rho(S)\}$$

**Definition 6.4.5** (Heaviest $k$-Subgraph Problem)**.** Given an undirected weighted graph $G = (V, E, w)$ and an integer $k$, find an induced subgraph $H = (V(H), E(H))$ of $G$, such that $H$ has $k$ nodes and maximum sum of edge weights.

**Definition 6.4.6** (Heaviest $k$-clique Problem)**.** Given an undirected weighted graph $G = (V, E, w)$ and an integer $k$, find an induced subgraph $H = (V(H), E(H))$ of $G$, such that $H$ is a $k$-clique and has maximum sum of edge weights.

**Definition 6.4.7** (Heaviest Quasi-Clique Problem)**.** Given an undirected weighted graph $G = (V, E, w)$, a rational number $\gamma$, $0 < \gamma \leq 1$, and an integer $k$, find an induced subgraph $H = (V(H), E(H))$ of $G$ such that $H$ is a $\gamma$-quasi-clique and has maximum sum of edge weights. For the sake of brevity, we will refer to a $\gamma$-quasi-clique as quasi-clique.

### 6.4.1 Algorithms

The exact and the approximated algorithm for computing the densest subgraph presented in Chapter 3 can be adapted to the heaviest subgraph. In Chapter 4 we presented an exact and an approximated algorithm for the heaviest $k$-subgraph. In order to compute the heaviest $k$-clique and the heaviest quasi-clique, we modify the branch and bound algorithm presented in Chapter 4. We shortly remind the approach: in the branching phase a new edge is added to a subgraph while in the bounding phase we update the upper bound of the subgraph and we decide if this branch can lead to the optimum solution. For the heaviest $k$-clique problem, we add the constraint that each subgraph that we keep has to be a clique, which is intuitive as every induced subgraph of the heaviest $k$-clique is a clique. We note however that the branch and bound algorithm is not guaranteed to find a solution containing $k$ nodes. This happens because of the bounding phase in which we eliminate subgraphs that could possibly lead to a clique of size $k$, but with a smaller weight. However, this can be an advantage for the graph mining task that we consider so we preserve this approach. To adapt the algorithm for the heaviest quasi-clique problem, we add the constraint that each new best solution found is a quasi-clique. The modifications are straightforward and can be added both to the exact and approximate algorithm.

### 6.4.2 Evaluation

In this evaluation we will compute for each subgraph an 1/2 approximation. For the heavy $k$-subgraph, heavy $k$-clique and heavy quasi-clique we set $k = 10$. For

the quasi-clique we used $\gamma = 0.4$ as in the previous experimental evaluation. In order to compare the results, we compute the top 10 heaviest subgraphs in a graph by computing a dense subgraph, removing the dense subgraph from the graph and restarting the iteration step until we have found $k$ subgraphs or the graph is empty. We use the same datasets as in the previous section, more precisely four graphs representing tweets written in the period November 2015 - February 2016.

In Table 6.11 we first evaluate the quality of the results using the total sum of weights of the subgraphs, the minimum edge density of a subgraph, and the minimum value of $q_G$ for the four datasets for $k = 10$. As expected the top 10 heaviest subgraphs have the largest sum of weights for all datasets. In terms of edge density, we see less connected subgraphs when computing the heaviest subgraph. In terms of the minimum value of $q_G$ , the heavy $k$-subgraph approach performs similar with the heavy $k$-clique, while again the heaviest subgraph performs the worst. We notice that the heaviest quasi-clique solution has a lower total weight than the heavy $k$-subgraph and the heavy $k$-clique solutions.

| Heaviest Subgraphs | | | Heavy $k$-Subgraphs | | | Heavy $k$-cliques | | | Heavy quasi cliques | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sum \rho$ | $\rho_E$ | $q_G$ | $\sum \rho$ | $\rho_E$ | $q_G$ | $\sum \rho$ | $\rho_E$ | $q_G$ | $\sum \rho$ | $\rho_E$ | $q_G$ |
| 2073216 | 0.38 | 0.004 | 441858 | 0.77 | 0.09 | 462930 | 1.0 | 0.09 | 360962 | 0.85 | 0.40 |
| 1404949 | 0.28 | 0.003 | 157973 | 0.66 | 0.08 | 173074 | 1.0 | 0.08 | 134427 | 0.68 | 0.40 |
| 2797305 | 0.27 | 0.0009 | 334593 | 0.57 | 0.14 | 415268 | 1.0 | 0.11 | 307653 | 0.71 | 0.40 |
| 1701296 | 0.45 | 0.002 | 221829 | 0.40 | 0.12 | 221615 | 1.0 | 0.11 | 187815 | 0.80 | 0.40 |

Table 6.11: Results for the four datasets representing the period November 2015 - February 2016

In Table 6.12 we present the top 10 heaviest subgraphs retrieved from the dataset November 2015. We notice that the first event corresponds to the Paris attacks, however, the second event is hard to label as it contains in total 394 terms. The terms refer to several events, such as the attack at the Mali hotel, the Paris attacks, a show of Justin Bieber and a match of Germany. We can clearly see from this example that heaviest subgraphs can have a large number of nodes, and although the average degree is high we have the intuition that not all the nodes are well connected. A first attempt to alleviate the problem would be to enforce a size of the subgraphs, so we investigate the results for heaviest-$k$ subgraphs in Table 6.13. We note that the solutions have improved, however, the first subgraph is still a mixture of two events, the Paris attacks, and climate change. We use the intuition that if the unweighted subgraph is a clique then it is unlikely two events will be grouped in the same subgraph. For this, we compute the top heaviest $k$-cliques and we present the result in Table 6.14. The first subgraph is still a mixture of two events although now we are sure all the nodes in the subgraph are connected. This suggests there are a few tweets mentioning together the events, but the number should be small and because of this, the measure of density we propose in Section 6.2.3 should be able to correctly group semantically connected keywords. We present the results in Table 6.15. We find an example in which an unrelated term (prayforparis) is added to a Justin Bieber event, *justinbieber purpose streams album spotify prayforparis billion week* , however in the correspond heavy $k$-subgraph we find several unrelated

terms, *free streaming justinbieber purpose streams album spotify prayforparis praying world*. Overall the quasi-clique approach performs better than the previous methods.

| Event keywords |
|---|
| paris, attacks, prayers, thoughts, people |
| prayforparis, planned, parenthood, affected, mali, hotel, show, justinbieber, great, happened, heard, events, minds, discuss, lost, lives, refugees, civilians, match, germany, + **other 374 words** |
| sad, situation, involved, evening, occurred, niallofficial, stop, kill, biafrans, supporting, christian, buhari, emekagift, britain, biafranslivesmatters |
| change, climate |
| unseen, footage, tonight, showed, house, calaurand, stayed, stay, strong, safe |
| disneylandparis, closes, horrendous, daily, 1992, opened |
| largest, database, access, favorite, instantly, unlimited, mobile, download, legally, tablet, ipob, nnamdikanu, unarmed, protesters, peaceful, biafra, crush, orders |
| object, nearest, dramatically, princesses, taught, throw, disney |
| haters, selenagomez, embraces, extra, shuts, lbs, champ |
| brighter, retweet, toysfortots, nationaliloveyouday, dogs, bowl, tweetforbowls, starving, pedigreeus, donates, secs, positivepuzzle |

Table 6.12: Top 10 heaviest subgraphs in dataset November 2015

| Event keywords |
|---|
| attacks, paris, prayers, people, thoughts, attack, victims, *climate, change*, killed |
| stream, live, muslims, terrorists, movies, online, findallmovies, religion, terrorism, watch |
| strong, stay, safe, breaking, news, police, dead, amp, syrian, refugees |
| year, life, situation, sad, involved, evening, occurred, niallofficial, summer, mtvstars |
| stop, terrorist, christian, kill, emekagift, buhari, biafrans, supporting, man, death |
| lives, lost, minds, discuss, bomber, suicide, bombing, syria, isis, france |
| unseen, footage, tonight, showed, house, calaurand, stayed, plane, crash, show |
| state, emergency, hall, concert, islamic, french, parisattacks, bataclan, war, terror |
| free, streaming, justinbieber, purpose, streams, album, spotify, prayforparis, praying, world |
| time, today, daily, opened, disneylandparis, closes, horrendous, fetty, eye, 1992 |

Table 6.13: Top 10 heavy $k$-subgraphs November 2015

## 6.5 Conclusions

Motivated by the richness of content available on social media and the need of information during crisis situations, we have developed an event detection algorithm which has been evaluated on a collection of tweets against state-of-the-art approaches

| Event keywords |
|---|
| attacks, paris, prayers, people, thoughts, attack, victims, *climate, change*, killed |
| movies, online, findallmovies, watch, streaming, stream, free, download, full, easy |
| muslims, terrorists, religion, terrorism, strong, stay, safe, breaking, news, police |
| year, life, heart, walking, dead, amp, death, love, shot, terrorist |
| situation, sad, involved, evening, occurred, niallofficial, time, today, events, day |
| emekagift, buhari, kill, stop, biafrans, supporting, christian, world, terror, isis |
| unseen, footage, tonight, showed, house, calaurand, stayed, show |
| bomber, suicide, state, emergency, french, parisattacks, france, syria, concert, 100 |
| lives, lost, rip, yesterday, lebanon, media, covered, donate, support, prayforparis |
| justinbieber, purpose, listen, live, streams, album, spotify, 2015, heard, happened |

Table 6.14: Top 10 heavy $k$-cliques November 2015

| Event keywords |
|---|
| prayers, paris, people, thoughts, victims |
| stream, live, movies, online, findallmovies, watch, streaming |
| terror, attacks, terrorist, france, isis, attack, amp |
| french, police, breaking, dead, hall, concert, 100, attackers |
| life, death, situation, sad, involved, evening, occurred, niallofficial |
| emekagift, buhari, kill, stop, biafrans, supporting, christian, enforcement, law, britain |
| muslims, terrorists, religion, terrorism, islam |
| unseen, footage, tonight, showed, house, calaurand, stayed, hostages, killed, show |
| justinbieber, purpose, streams, album, spotify, prayforparis, billion, week |
| time, today, events, praying, world, affected, stay, love, news |

Table 6.15: Top 10 heavy quasi-cliques November 2015

for the same problem. Our experimental evaluation shows that our algorithm outperforms the other approaches in terms of precision. Additionally, we investigate whether our new measure of density performs better than existing definitions and we find encouraging results. We provided one of the largest-scale evaluation of an event detection algorithm while offering some insights on the coverage of crisis events in social media. Our work shows that many major events are discussed on Twitter, including events that are poorly covered by the mainstream media, such as the conflict in Yemen.

# Chapter 7

# Conclusion

Due to the rise of the Internet a plethora of data is available for the scientific community. A large part of this data can be classified in information and social networks, which can be represented using graphs. In this thesis, we have studied several important problems in graph and data mining. In the following, we provide an overview of our contributions and we highlight future research directions.

## 7.1 Contribution

$k$ DENSE SUBGRAPHS WITH LIMITED OVERLAP. In Chapter 3 we first introduce the problem of finding $k$ dense subgraphs with a limited overlap and then we show it is NP-hard. As a first step in finding an efficient heuristic, we highlight the importance of minimal densest subgraphs. We devise an algorithm that finds an optimum solution for our problem for a particular input case, and solutions close the optimum in the general case, without however having an approximation guarantee. We demonstrate the efficiency of our algorithm through an extensive experimental evaluation. Finally, we present an analysis in which we highlight the similarity between our problem and recent research work on graph decompositions.

DENSE SUBGRAPHS WITH RESTRICTED SIZE. In Chapter 4, we develop an exact and an approximate algorithm for the problem of finding the heaviest $k$-subgraph. Even if the heaviest $k$-subgraph problem is NP-hard, we show in our evaluation that we are able to compute exact solutions for $k$ up to 15 on graphs with millions of edges. Our approximate algorithm scales even further, finding heaviest subgraphs for $k = 105$. We note that both the exact and approximation algorithm have exponential running time, but due to the sparseness of real networks we are able to compute efficiently solutions for small values of $k$. The branch-and-bound algorithm presented in this chapter can be adapted for finding other types of subgraphs, such as the heaviest quasi-clique.

K-CLIQUE LISTING AND K-CLIQUE DECOMPOSITION. In Chapter 5 we present a parallel algorithm for listing or counting all $k$-cliques in very large real-world graphs. We prove that our algorithm has the best known asymptotic running time while requiring only a linear amount of memory. Our experimental analysis shows

that we are able to list all cliques in graphs of tens of millions of edges, as well as all cliques for $k \leq 10$ in graphs containing billions of edges. We solve efficiently two related problems, computing the $k$-clique core decomposition of a graph, and computing an approximation of the $k$-clique densest subgraph. We have the intuition that the $k$-clique core decomposition could be a useful tool in data mining.

EVENT DETECTION. In Chapter 6 we study the problem of finding crisis-related events in social media. We propose a natural approach, that is based on the intuition that important events can be uniquely identified by their location and time frame. Additionally, we propose a new definition of density that better captures the structure of a strongly connected weighted subgraph. We evaluate this definition and the event detection algorithm against previous approaches and we show they have a better performance. We conclude with a large scale study that confirms the performance of our event detection algorithm and highlights differences between events covered in the mainstream media and events covered in social media.

## 7.2 Future work

SCALING ALGORITHMS TO LARGE REAL-WORLD GRAPHS. In Chapter 3, we have compared two exact algorithms for computing the densest subgraphs, a max-flow based algorithm, and a linear programming algorithm. In our experimental evaluation, we have found that the max-flow based algorithm performs better than the linear programming solver in terms of running time. However, for the task of finding minimal densest subgraphs, we currently have only a linear programming solution. As future work, we want to adapt the max-flow algorithm for the minimal densest subgraph problem. Another interesting line of research is in adapting existing approaches or developing new algorithms for efficiently solving the problem of $k$ dense subgraph with limited overlap. We have the intuition that a graph decomposition, such as the one presented in [93] might alleviate the cost of the $k$ factor needed in our heuristics. We will investigate how allowing overlap will improve the quality of the solution, and because the density-friendly decomposition [93] incorporates overlap in the objective function we suspect we will obtain very good results.

DYNAMIC TOP-$k$ DENSE SUBGRAPHS. In particular, when dealing with a dynamic graph, the problem of maintaining $k$ dense subgraphs can be very challenging as any change in the structure of one of the dense subgraphs might affect the structure of the remaining dense subgraphs. We have preliminary results for a generalization of the algorithm of [34], but we are not able to obtain a fast update time for an edge addition or an edge removal. In the future, we will investigate if a density-friendly decomposition [93] can be efficiently maintained in a dynamic graph. If this approach gives good results, we will have also a heuristic for maintaining the top-$k$ dense subgraph, without depending on the parameter $k$.

GENERALYSING PROBLEMS TO HYPERGRAPHS. Many types of data, such as collaboration networks or word co-occurrence networks have a natural representation

as hypergraphs. However, there is little work on generalizing algorithms for the densest subgraph or decompositions on hypergraphs. There are three important aspects that should be addressed: devising new algorithms appropriate for dealing with real-word hypergraphs, analyzing from a structural and semantic perspective the results of the techniques, and, finally, comparing these results with the results retrieved from the traditional graph representation of the data. We have developed an algorithm for computing the densest subgraph in a hypergraph using a max-flow based technique. We will apply it to real datasets and compare the results with the ones obtained by computing the densest in the graph representation. We will continue with algorithms for finding minimal dense subgraph and for density friendly hypergraph decomposition.

ANALYSING TERM FREQUENCY TIME SERIES IN SOCIAL MEDIA. In Chapter 6 we have emphasized the importance of finding points in a time series that deviate a lot from the expected value. However, finding the expected value and normal intervals of deviation is not straightforward, as term frequency cannot be correctly modeled using distributions such as the normal distribution or the Poisson distribution. In our future work, we will try to explore related fields, such as anomaly detection, to find better-suited algorithms. A very important goal is to eliminate parameters, or if not possible to learn those parameters from the data.

WEB APPLICATION FOR SOCIAL MEDIA AND MAINSTREAM MEDIA EVENTS. Our methods can deal with the large volume of data produced in the social media, but could also extract events from mainstream media. The important difference is of the increased length of the text of news in traditional media, but in our preliminary work we found that the titles of news or summaries obtained using extractive summarizing techniques could be used to reduce the size of the text and eliminate less relevant content. We are currently working on developing a web application that will present to a user top events in social media and top events in traditional media. Through our application, we hope to provide users with a broader perspective of events happening in the world.

CHRONOLOGY OF TWEETSTORMS AND STORMS COMMUNITIES. In our work we came across a very interesting phenomenon, Twitter storms. Twitter storms are organized events that occur at a certain time and aim at bringing awareness on a topic by taking advantage of the popular trends feature of Twitter. A group of users will post tweets containing certain hashtags and keywords that because of the sudden usage in the stream of tweets, they will be featured on the Twitter page as trends. We aim at studying the timeline of tweet storms, more particularly, what is the correlation between the patterns of tweet storms and events in the real world, and how much time does it take for users to become interested in a cause and actively participate in tweets storm. When several tweets storms are organized for the same cause, there might be variation in the participation of users; studying how the community evolves over time could also give an intuition over the success of the tweet storms in bringing awareness.

# Bibliography

[1] Hamed Abdelhaq, Christian Sengstock, and Michael Gertz. Eventweet: Online localized event detection from twitter. *Proc. VLDB Endow.*, 6(12):1326–1329, August 2013.

[2] Bahram Alidaee, Fred Glover, Gary Kochenberger, and Haibo Wang. Solving the maximum edge weight clique problem via unconstrained quadratic programming. *European Journal of Operational Research*, 181(2):592 – 597, 2007.

[3] James Allan, editor. *Topic Detection and Tracking: Event-based Information Organization*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[4] James Allan, Jaime Carbonell, George Doddington, Jonathan P. Yamron, and Yiming Yang. Topic detection and tracking pilot study: Final report. *Broadcast news transcription and understanding workshop*, 1998:194–218, 1998.

[5] James Allan, Victor Lavrenko, Daniella Malin, and Russell Swan. Detections, Bounds, and Timelines: UMass and TDT-3. *Information Retrieval*, pages 167–174, 2000.

[6] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[7] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5427 LNCS:25–37, 2009.

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

[9] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203 – 221, 2000.

[10] Farzindar Atefeh and Wael Khreich. A Survey of Techniques for Event Detection in Twitter. *Comput. Intell.*, pages 132–164, February 2015.

[11] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 33–42, New York, NY, USA, 2012. ACM.

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

[13] Oana Denisa Balalau, Francesco Bonchi, T-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. Finding Subgraphs with Maximum Total Density and Limited Overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, 2015.

[14] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 16–24, 2008.

[15] Hila Becker, Mor Naaman, and Luis Gravano. Beyond Trending Topics: Real-World Event Identification on Twitter. *Icwsm*, pages 1–17, 2011.

[16] James Benhardus and Jugal Kalita. Streaming trend detection in twitter. *International Journal of Web Based Communities*, pages 122–139, 2013.

[17] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, and Uriel Feige. for Densest k -Subgraph. *Organization*, (873):201–210.

[18] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *Automata, Languages, and Programming*, pages 223–234. Springer, 2014.

[19] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *KDD*, 2014.

[20] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[21] Carlos Castillo. *Big Crisis Data: Social Media in Disasters and Time-Critical Situations*. Cambridge University Press, Cambridge, 007 2016.

[22] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In Klaus Jansen and Samir Khuller, editors, *APPROX*. Springer, 2000.

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

[24] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the*

*18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1240–1248. ACM, 2012.

[25] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[26] Flavio Chierichetti, Jon Kleinberg, Ravi Kumar, Mohammad Mahdian, and Sandeep Pandey. Event Detection via Communication Pattern Analysis. pages 51–60, 2014.

[27] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[28] Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, and Riccardo Torlone. Finding All Maximal Cliques in Very Large Social Networks Categories. 2016.

[29] Mário Cordeiro. Twitter event detection: combining wavelet analysis and topic inference summarization. *Proceedings of Doctoral Symposium on Informatics Engineering*, 2012.

[30] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. Online search of overlapping communities. In *SIGMOD*, 2013.

[31] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[32] Maximilien Danisch, T-H. Hubert Chan, and Mauro Sozio. Large scale density-friendly graph decomposition via convex programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[33] Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E. Lee, and John H. Thornton, Jr. Migration motif: a spatial - temporal pattern mining approach for financial markets. In *KDD*, 2009.

[34] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient Densest Subgraph Computation in Evolving Graphs. *Www*, 2015.

[35] David Eppstein, Maarten Löffler, and Darren Strash. *Listing all maximal cliques in sparse graphs in near-optimal time*. Springer, 2010.

[36] Orhan Erdem, Elvan Ceyhan, and Yusuf Varli. A new correlation coefficient for bivariate time-series data. *Physica A: Statistical Mechanics and its Applications*, 414:274 – 284, 2014.

[37] Uriel Feige. The Dense k -Subgraph Problem 1 Introduction. 1999.

[38] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[39] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. MotifCut: regulatory motifs finding with maximum density subgraphs. In *ISMB*, 2006.

[40] Esther Galbrun, Aristides Gionis, and Nikolaj Tatti. Top-k overlapping densest subgraphs. *Data Mining and Knowledge Discovery*, 30(5):1134–1165, 2016.

[41] Jie Gao and Li Zhang. Tradeoffs between stretch factor and load balancing ratio in routing on growth restricted graphs. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 189–196, New York, NY, USA, 2004. ACM.

[42] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. *International Conference on Very Large Data Bases (VLDB)*, pages 721—-732, 2005.

[43] Av V Goldberg. Finding a maximum density subgraph, 1984.

[44] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[45] Adrien Guille and Cecile Favre. Mention-anomaly-based Event Detection and Tracking in Twitter. *ASONAM 2014 - IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, (Asonam):375–382, 2014.

[46] Johan Håstad. Clique is hard to approximate within n 1-&epsiv. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 627–636. IEEE, 1996.

[47] Haiyan Hu, Xifeng Yan, Yu Huang 0003, Jiawei Han, and Xianghong Jasmine Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. In *ISMB (Supplement of Bioinformatics)*, pages 213–221, 2005.

[48] Muhammad Imran, Carlos Castillo, Ji Lucas, Patrick Meier, and Sarah Vieweg. AIDR: Artificial intelligence for disaster response. *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, (October):159–162, 2014.

[49] Samir Khuller and Barna Saha. On finding dense subgraphs. *Icalp*, 5555:597–608, 2009.

[50] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170, New York, NY, USA, 2000. ACM.

[51] Tamara G Kolda, Ali Pinar, Todd Plantenga, C Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[52] April Kontostathis, Leon M. Galitsky, William M. Pottenger, Soma Roy, and Daniel J. Phelps. *A Survey of Emerging Trend Detection in Textual Data Mining*, pages 185–224. Springer New York, New York, NY, 2004.

[53] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 1343–1350. International World Wide Web Conferences Steering Committee, 2013.

[54] Michael A. Langston and et al. A combinatorial approach to the analysis of differential gene expression data: The use of graph algorithms for disease prediction and screening. In *Methods of Microarray Data Analysis IV*. 2005.

[55] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458–473, 2008.

[56] Ryong Lee and Kazutoshi Sumiya. Measuring geographical regularities of crowd behaviors for Twitter-based geo-social event detection. *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Location Based Social Networks*, pages 1–10, 2010.

[57] 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*. 2010.

[58] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[59] Matthaios Letsios, Oana Denisa Balalau, Maximilien Danisch, Emmanuel Orsini, and Mauro Sozio. Finding heaviest k-subgraphs and events in social media. In *The Sixth IEEE ICDM Workshop on Data Mining in Networks (DaMNet 2016), Barcelona Spain, 12*, 2016.

[60] Chenliang Li, Aixin Sun, and a Datta. Twevent: Segment-based Event Detection from Tweets. *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 155–164, 2012.

[61] Bruce R Lindsay. Social media and disasters: Current uses, future options, and policy considerations, 2011.

[62] Amina Madani and Omar Boussaid. What's Happening : A Survey of Tweets Event Detection. *INNOV 2014 : The Third International Conference on Communications, Computation, Networks and Technologies*, pages 16–22, 2014.

[63] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian Workshop on Algorithm Theory*, pages 260–272. Springer, 2004.

[64] Kamran Massoudi, Manos Tsagkias, Maarten de Rijke, and Wouter Weerkamp. Incorporating query expansion and quality indicators in searching microblog posts. *ECIR'11 Proceedings of the 33rd European conference on Advances in information retrieval*, pages 362–367, 2011.

[65] Michael Mathioudakis and Nick Koudas. Twittermonitor: Trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1155–1158, New York, NY, USA, 2010. ACM.

[66] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[67] Andrew J. McMinn, Yashar Moshfeghi, and Joemon M. Jose. Building a large-scale corpus for evaluating event detection on twitter. *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management - CIKM '13*, pages 409–418, 2013.

[68] Polykarpos Meladianos, Giannis Nikolentzos, Francois Rousseau, Yannis Stavrakas, and Michalis Vazirgiannis. Degeneracy-based real-time sub-event detection in twitter stream. 2015.

[69] Nicholas R. Miller. Graph-theoretical approaches to the theory of voting. *American Journal of Political Science*, 21(4):769–803, 1977.

[70] Muhammad Anis Uddin Nasir, Aristides Gionis, Gianmarco De Francisci Morales, and Sarunas Girdzijauskas. Top-k densest subgraphs in sliding-window graph streams. *CoRR*, abs/1610.05897, 2016.

[71] A Nurwidyantoro and E Winarko. Event detection in social media: A survey. *ICT for Smart Society (ICISS), 2013 International Conference on*, pages 1–5, 2013.

[72] Alexandra Olteanu, Carlos Castillo, Nicholas Diakopoulos, and Karl Aberer. Comparing events coverage in online news and social media: The case of climate change. 2015.

[73] Alexandra Olteanu, Carlos Castillo, Fernando Diaz, and Sarah Vieweg. CrisisLex: A Lexicon for Collecting and Filtering Microblogged Communications in Crises. In *In Proceedings of the 8th International AAAI Conference on Weblogs and Social Media (ICWSM'14)*, 2014.

[74] Miles Osborne and Mark Dredze. Facebook , Twitter and Google Plus for Breaking News: Is There a Winner ? *Proceedings of the 8th International AAAI Conference on Weblogs and Social Media*, pages 611–614, 2014.

[75] Miles Osborne and Victor Lavrenko. Streaming First Story Detection with application to Twitter. *Computational Linguistics*, (June):181–189, 2010.

[76] Miles Osborne, S Petrovic, and R McCreadie. Bieber no more: First Story Detection using Twitter and Wikipedia. *Redirect.Subscribe.Ru*, 2012.

[77] Patric R J Östergård. A New Algorithm for the Maximum-Weight Clique Problem. *Electronic Notes in Discrete Mathematics*, 3:153–156, 1999.

[78] Chi-Chun Pan and Prasenjit Mitra. Event detection with spatial latent dirichlet allocation. In *Proceedings of the 11th Annual International ACM/IEEE Joint Conference on Digital Libraries*, JCDL '11, pages 349–358, New York, NY, USA, 2011. ACM.

[79] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3), 1991.

[80] Saša Petrović, Miles Osborne, and Victor Lavrenko. Streaming first story detection with application to twitter. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 181–189, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[81] Swit Phuvipadawat and Tsuyoshi Murata. Breaking news detection and tracking in Twitter. *Proceedings - 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Workshops, WI-IAT 2010*, pages 120–123, 2010.

[82] Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982.

[83] Ana-maria Popescu and Marco Pennacchiotti. Detecting controversial events from twitter. *Proceedings of the 19th ACM international conference on Information and knowledge management - CIKM '10*, page 1873, 2010.

[84] Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. Named entity recognition in tweets: An experimental study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 1524–1534, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[85] Alan Ritter, Evan Wright, William Casey, and Tom Mitchell. Weakly Supervised Extraction of Computer Security Events from Twitter. *WWW 2015*, i, 2015.

[86] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao Ning Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6044 LNBI:456–472, 2010.

[87] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 851–860, 2010.

[88] Jagan Sankaranarayanan, Benjamin E Teitler, Hanan Samet, Michael D Lieberman, and Jon Sperling. TwitterStand : News in Tweets. *Information Storage and Retrieval*, 156:42–51, 2009.

[89] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 927–937, 2015.

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

[91] UNO Takeaki. Implementation issues of clique enumeration algorithm. *Special issue: Theoretical computer science and discrete mathematics, Progress in Informatics*, (9):25–30, 2012.

[92] Nikolaj Tatti and Aristides Gionis. Discovering nested communities. In *ECML/PKDD (2)*, 2013.

[93] Nikolaj Tatti and Aristides Gionis. Density-friendly graph decomposition. *Www*, 2015.

[94] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[95] Charalampos Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1122–1132, New York, NY, USA, 2015. ACM.

[96] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. Denser Than the Densest Subgraph: Extracting Optimal Quasi-cliques with Quality Guarantees. In *KDD*, pages 104–112, 2013.

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

[98] Andreas Weiler, Michael Grossniklaus, and Mh Scholl. Event Identification and Tracking in Social Media Streaming Data. *Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference*, pages 282–287, 2014.

[99] Andreas Weiler, Marc H. Scholl, Franz Wanner, and Christian Rohrdantz. Event identification for local areas using social media streaming data. *Proceedings of the ACM SIGMOD Workshop on Databases and Social Networks - DBSocial '13*, pages 1–6, 2013.

[100] Mark Weiler, Andreas, Grossniklaus, Michael and Scholl. Evaluation Measures for Event Detection Techniques on Twitter Data Streams. *Bicod*, pages 1–157, 2015.

[101] Jianshu Weng and Bu-Sung Lee. Event Detection in Twitter. In *International Conference on Weblogs and Social Media*, 2011.

[102] Wayne Xin Zhao, Jing Jiang, Jianshu Weng, Jing He, Ee-Peng Lim, Hongfei Yan, and Xiaoming Li. *Comparing Twitter and Traditional Media Using Topic Models*, pages 338–349. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[103] Xiao Zhou and Takao Nishizeki. Edge-coloring and f-coloring for various classes of graphs. *MATCH Commun. Math. Comput. Chem*, 51:111–118, 1999.

[104] Yunyue Zhu and Dennis Shasha. Efficient Elastic Burst Detection in Data Streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 336–345, New York, NY, USA, 2003. ACM.

# Appendices

# Appendix A

# Résumé en français

## A.1 Introduction

Un graphe peut s'avérer être une puissante abstraction pour représenter des données issues du monde réel, et ce malgré l'apparence simple de ce graphe. Depuis que Leonard Euler a formalisé le problème de l'existence d'un chemin visitant chaque arête du graphe exactement une fois, les applications de la théorie des graphes ont donné des résultats remarquables dans de nombreux domaines de recherche. Ainsi ces dernières années les graphes sont devenus des outils incontournables dans *l'exploration de données*, mettant en évidence le potentiel du domaine de *l'exploration de graphes*, c'est-à-dire le domaine qui cherche à trouver des modèles intéressants dans les graphes. Les graphes peuvent modéliser de nombreux types de données réelles, telles que des données representant des réseaux routiers, des réseaux sociaux, des réseaux cérébraux et bien d'autres encore. Par exemple les graphes ont été utilisés pour étudier la théorie des votes [69], et pour étudier le phénomène du petit monde de manière formelle [50] ainsi que par une vaste évaluation expérimentale sur le réseau social Facebook [11]. Exceptées les applications dans les réseaux sociaux, les graphes peuvent également aider à comprendre et modéliser les réseaux d'informations, tels que le réseau formé par les liens hypertextes du Web [12] ou le réseau formé par les cooccurences de termes utilisés dans des collections de tweets [8].

L'une des tâches essentielles dans *l'exploration de graphes* est de trouver des sous-graphes denses, car la densité peut être utilisée pour mesurer l'importance et la cohésion du sous-graphe. Les sous-graphes denses ont d'autres propriétés en plus d'une densité importante, telles qu'un petit diamètre. Dans les graphes des réseaux de communication sans fil et des réseaux peer-to-peer, le diamètre du graphe est important dans la conception des algorithmes de routage [41]. Les communautés sont par définition des sous-graphes denses et des résultats prometteurs dans [23] montrent une nette séparation entre les communautés de blogs libéraux et celles de conservateurs dans le réseau de citations de blog. Dans le graphe du Web, les sous-graphes denses pourraient être une indication de sites qui essaient d'augmenter leur pagerank afin de modifier les résultats d'une requête dans un moteur de recherche [42]. Dans [47] les sous-graphes denses fréquents dans les réseaux biologiques sont utilisés pour prédire les fonctions inconnues de certains gênes. Dans un graphe de mots représentant un flux de tweets, un sous-graphe dense pourrait indiquer la présence

d'événements dans le monde réel, comme indiqué dans [8]. En raison de la diversité et de la pertinence des applications, il est important de développer des algorithmes qui sont bien adaptés aux réseaux réels. Dans cette thèse nous allons développer de nouveaux algorithmes pour trouver des sous-graphes denses et nous allons nous concentrer sur une application particulière, la détection d'événements dans les médias sociaux.

Internet a constamment remis en cause et bouleversé la façon dont nous interagissons avec le monde depuis son adoption par le grand public au milieu des années 1990. Aujourd'hui, les nouvelles ne sont pas souvent lues depuis leur support traditionnel, le papier, mais en ligne, sur des sites d'information ou encore de médias sociaux. Les médias sociaux diffèrent des médias traditionnels par une caractéristique importante : chaque utilisateur peut contribuer à l'histoire avec sa propre expérience et son propre point de vue. Cette richesse du contenu peut aider dans de nombreuses situations, par exemple dans les situations de crise [21] dans laquelle les autorités peuvent avoir accès à beaucoup plus d'informations sur les zones touchées, dans des cas de conflits d'intérêts entre les médias traditionnels et les acteurs principaux d'un événement, ou à la couverture de petits événements qui pourraient être négligés dans les médias traditionnels. En conséquence, les gens se tournent souvent vers les médias sociaux.

Cependant, parce que les médias sociaux ont été conçus comme une plateforme ou les utilisateurs partagent des histoires personnelles, et non pas des événements de grande ou petite échelle, une grande partie des données disponibles n'ont d'intérêt que pour une petite partie des utilisateurs, et dans la plupart des cas exclusivement auprès des followers d'une personne. En plus des histoires personnelles, les médias sociaux sont confrontés au problème des spams, de la publicité, et des fausses nouvelles [1]. Même lorsque les utilisateurs signalent des événements, à cause de la brièveté des messages, les phrases pourraient contenir des mots mal orthographiés, abréviations et structures syntaxiques hors du commun, ce qui augmente la difficulté des tâches telles que la reconnaissance d'entités. Un autre problème important est le volume du contenu disponible sur les médias sociaux, partagé par $1,8$ milliards d'utilisateurs sur Facebook et 317 millions d'utilisateurs sur Twitter [2]. Afin de mieux comprendre les informations partagées par les utilisateurs, Osborne et al. [74] ont comparé Twitter, Facebook et Google Plus pour voir quel site couvrait le plus de nouvelles, et où sont les nouvelles rapportées en premier. Les auteurs ont constaté que pour la couverture des nouvelles les trois fonctionnent aussi bien, mais Twitter a été le premier à signaler les dernières nouvelles. Pour cette raison, nous allons nous concentrer sur la détection d'événements dans le flux de tweets. Avant d'aborder la tâche de détection d'événements, nous résolvons des problèmes qui aideront à comprendre la structure de graphes de mots, graphes qui sont construits a partir de tweets. Dans la section suivante, nous détaillons nos contributions.

[1] https://www.theguardian.com/technology/2016/dec/15/facebook-flag-fake-news-fact-check

[2] http://www.smartinsights.com/social-media-marketing/social-media-stratégie/nouvelle-mondiale-médiassociaux-recherche/

## A.2 Contributions

Dans ce travail, nous nous efforçons de fournir de nouvelles définitions et une meilleure compréhension des problèmes liés aux graphes, de concevoir des algorithmes efficaces qui peuvent traiter des réseaux réels et résoudre des problèmes concrets d'exploration de données. Notre intérêt de recherche suit deux directions principales:

- l'amélioration des algorithmes existants pour l'exploration de graphes en tenant compte des propriétés des réseaux réels;
- la détection d'événements par une approche graphe.

Dans l'exploration de graphes on se préoccupe d'extraire des motifs qui satisfont certaines propriétés à partir de graphes. Nous étudions le problème de trouver des sous-graphes denses tout en explorant différentes définitions de la densité.

La détection d'événements est la tâche d'exploration de données concernant la recherche d'anomalies dans un flux de données et, dans notre cas, un flux de tweets. Nous abordons le problème de l'identification unique d'un événement et la présentation d'une description courte mais instructive de l'événement sous la forme d'un sous-graphe.

Dans ce qui suit, nous fournissons un aperçu des contributions de cette thèse.

*Trouver k sous-graphes dense avec une somme totale maximale des densités.*

Trouver le sous-graphe le plus dense a des applications dans de nombreux domaines, comme dans la détection de spam et la détection des communautés. Le plus souvent, nous sommes intéressés à trouver plus d'une communauté, ce qui pourrait correspondre à la recherche de plusieurs sous-graphes denses. Nous définissons le problème de trouver $k$ sous-graphes dense de chevauchement limité et de densité totale maximale et nous présentons plusieurs heuristiques pour résoudre efficacement ce problème NP-difficile. Une direction intéressante est de rechercher des sous-graphes minimaux, c'est-à-dire les sous-graphes les plus denses qui ne contiennent pas un sous-graphe induit de densité égale. Afin de calculer ce sous-graphe, nous proposons une solution efficace basée sur la formulation de programmation linéaire pour calculer les sous-graphes les plus denses [22]. En outre, nous améliorons le calcul du sous-graphe le plus dense en réduisant l'espace de recherche et nous concevons des heuristiques qui imposent efficacement le chevauchement maximal entre les sous-graphes. Nous avons mené une évaluation approfondie pour prouver la qualité des solutions de nos algorithmes. Nos heuristiques trouvent des sous-graphes denses dans les sous-graphes soigneusement choisis du graphe d'entrée initial, mais pour les travaux futurs, nous aborderons des approches holistiques, telles que des décompositions de graphe. Nous montrons dans une étude de cas des résultats prometteurs pour résoudre le problème de top $k$ sous-graphes denses en utilisant la décomposition sensible à la densité [93].

Une partie de ce travail s'est fait en collaboration avec Yahoo! et l'Université de Hong Kong, et est présentée dans l'article: Oana Denisa Balalau, Francesco Bonchi,

T-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. Finding Subgraphs with Maximum Total Density and Limited Overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, 2015.

*Le sous-graphe le plus lourd ayant k sommets: pouvons-nous trouver des solutions exactes dans les graphes du monde réel?*

Les sous-graphes denses peuvent fournir des résultats insatisfaisants lorsqu'ils sont appliqués à certaines tâches d'exploration de données, telles que la détection d'événements. Il y a deux problèmes principaux qui apparaissent : d'abord, le sous-graphe en sortie est habituellement trop volumineux pour que l'utilisateur comprenne sa signification, et ensuite, la mesure de degré moyenne tend à regrouper les sous-graphes correspondant à différents événements. Afin d'atténuer ces problèmes, nous étudions une variante du problème du sous-graphe le plus dense dans un graphe pondéré, le sous-graphe le plus lourd ayant $k$ sommets. Bien qu'étant un problème NP-difficile, les solutions exactes pour de petites valeurs $k$ peuvent être calculées efficacement sur de grands graphes du monde réel. Nous construisons sur une stratégie de séparation et d'évaluation et nous utilisons l'intuition selon laquelle les sous-graphes lourds devraient contenir des liens lourds. Afin d'accélérer notre algorithme, nous observons davantage comment calculer efficacement les limites supérieures et éviter les solutions en double. Cet algorithme peut être facilement modifié afin de trouver d'autres types de sous-graphes, comme la plus lourde clique de taille $k$, comme nous l'indiquerons dans ce travail.

Ce travail est présenté dans l'article: Matthaios Letsios, Oana Denisa Balalau, Maximilien Danisch, Emmanuel Orsini, and Mauro Sozio. Finding heaviest k-subgraphs and events in social media. In *The Sixth IEEE ICDM Workshop on Data Mining in Networks (DaMNet 2016), Barcelona Spain, 12*, 2016.

*Trouver des cliques dans des graphes du monde réel: énumérer les k-cliques et la k-clique décomposition*

Les cliques sont des sous-graphes denses parfaits. Le problème d'énumérer les $k$-cliques est un problème NP-difficile, mais des algorithmes efficaces qui tiennent compte de la structure du réseau peuvent traiter de gros graphes. Ce problème standard dans l'exploration de graphes a été négligé dans les travaux de recherche récents en faveur de l'énumération de triangles ou de la recherche de la clique maximale. Nous proposons un algorithme basé sur le travail de [25], pour lequel nous montrons de meilleures garanties et une méthode de parallélisation directe. Nous étendons notre contribution en concevant un algorithme pour calculer la $k$-clique décomposition et une approximation du sous-graphe avec le plus grand nombre moyen de $k$-cliques par noeud. Nous montrons la performance de nos algorithmes par rapport à l'état de l'art avec une vaste évaluation expérimentale.

Ce travail de recherche s'est fait en collaboration avec Maximilien Danisch et Mauro Sozio.

*Détection d'événements dans les médias sociaux*

La détection d'événements est un sujet qui a reçu beaucoup d'attention dans

le domaine de l'exploration de données. Il existe plusieurs avantages à trouver des événements dans les médias sociaux: nous pouvons détecter certains événements beaucoup plus tôt que les médias traditionnels, nous pouvons trouver des événements qui ont reçu peu d'attention dans les médias et nous avons accès à une importante source d'informations dans des situations qui affectent de nombreuses personnes. Nous avons commencé notre travail en requêtant intensivement la plateforme sociale Twitter, qui permet d'accéder à 1% des tweets grâce à une API dédiée. Nous avons collecté plusieurs jeux de données, chacun correspondant à un ensemble de mots clefs ou à un lieu géographique. À partir de ces données, nous pouvons créer des graphes de cooccurrences de mots, dans lesquels chaque tweet correspond à une clique, en s'appuyant sur l'intuition qu'un événement contiendra un ensemble de mots clefs importants qui vont induire un sous-graphe dense. Nous expérimentons avec différents types de sous-graphes denses et nous proposons notre propre définition de densité que nous trouvons mieux adaptée à notre tâche. Nous concevons un algorithme de détection d'événement qui exploite les fonctionnalités clés qui décrivent un événement: l'lieu et la période de temps. Finalement, nous présentons une étude de 14 mois qui confirme l'efficacité de notre approche et, enfin, nous mettons en évidence des résultats intéressants concernant des campagnes visant à sensibiliser le public sur la guerre civile au Yémen.

Ce travail de recherche s'est fait en collaboration avec Mauro Sozio.

## A.3 Trouver des sous-graphes avec une densité totale maximale et un chevauchement limité

Un problème qui se pose naturellement dans la détection d'événements est la découverte de plusieurs événements qui se produisent à un certain moment sous l'hypothèse que les événements peuvent être corrélés, ayant un ou plusieurs facteurs communs. Le problème correspondant consiste à trouver de multiples sous-graphes denses pouvant se chevaucher.

Nous fournissons des résultats théoriques et pratiques pour ce problème dans un graphe non orienté dans l'article "Trouver des sous-graphes avec une densité totale maximale et un chevauchement limité" [13], accepté à WSDM 2015. Dans ce qui suit, je présente un bref résumé de nos contributions.

Étant donné un graphe non orienté et non pondéré $G = (V, E)$, sa densité $\rho(G)$ est définie comme $|E|/|V|$. Cette densité est souvent utilisée dans la littérature et elle s'appelle *densité moyenne de degré*. Le problème de trouver un sous-graphe qui maximise cette métrique est appelé *problème de sous-graphe le plus dense*. Notre problème est de trouver au plus $k$ sous-graphes denses de sorte que la somme totale des densités soit maximisée. Nous autorisons un chevauchement entre les sous-graphes, mais nous souhaitons également une certaine diversité; dans ce but, nous utilisons le coefficient Jaccard entre les paires d'ensembles.

**Définition du problème**.

Étant donné un graphe non-orienté et non pondéré $G = (V, E)$, un nombre entier

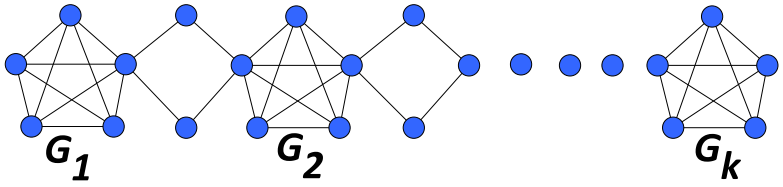


Figure A.1: $G$ est un sous-graphe dense non-minimal.

$k > 0$ et un nombre rationnel $\alpha \in [0, 1]$, le but est de trouver un ensemble d'ensembles de sommets $\mathcal{S} = \{S_1, \ldots, S_{\bar{k}}\}$ avec $\bar{k} \leq k$, et $S_i \subseteq V, \forall S_i \in \mathcal{S}$, tel que

$$\sum_{i=1}^{\bar{k}} \rho(G(S_i))) \quad \text{est minimum et}$$

$$\frac{|S_i \cap S_j|}{|S_i \cup S_j|} \leq \alpha \quad \forall S_i, S_j \in \mathcal{S}. \tag{A.1}$$

Nous dénoterons ce problème $(k, \alpha)$ - Sous-Graphes Denses avec Chevauchement Limité ($(k, \alpha)$ - SGDCL).

Ce problème est NP-difficile (preuve dans [13]) et nous nous concentrons sur la conception de bonnes heuristiques. Une heuristique simple pour notre problème dans le cas $\alpha = 0$ serait de trouver le sous-graphe le plus dense, enlever tous ses noeuds et ses arêtes, calculer le sous-graphe le plus dense dans le graphe restant et ainsi de suite jusqu'à ce que nous ayons au plus $k$ sous-graphes. Cependant, considérez le cas illustré Figure A.1. Ici, le graphe entier et chaque clique ont la même densité. Si nos algorithmes trouvent d'abord le graphe entier et le suppriment, nous offrons une solution sous-optimale, car nous aurions pu le décomposer en des sous-graphes plus petits et aussi denses (rappelez-vous que notre but était d'obtenir un ensemble de sous-graphes dont la somme des densités est maximale).

Pour faire face à cela, nous présentons la notion de sous-graphe minimal plus dense.

**Definition A.3.1.** (Sous-graphe dense minimal) Un graphe non orienté $G$ avec densité $\rho(G)$ est un *graphe minimal dense* si pour n'importe quel sous-graphe induit $H$ de $G$, $\rho(H) < \rho(G)$ $G$ est un sous-graphe minimal dense s'il est minimal et a une densité maximale.

**Algorithmes.**

Voici un algorithme naturel. À chaque étape, trouvez un sous-graphe dense minimal, enlevez les noeuds et les arêtes et itérez jusqu'à ce que le graphe soit vide ou que nous ayons trouvé $k$ sous-graphes. Cet algorithme garantit qu'il trouvera les $k$ sous-graphes disjoints les plus denses, si ceux-ci existent.

Afin de calculer un sous-graphe minimal dense, nous utilisons les propriétés de la formulation linéaire pour le problème de sous-graphe le plus dense (introduit dans [22]). Les auteurs ont montré que la solution optimale pour la fonction objective est égale à la densité du sous-graphe le plus dense. À partir de cette approche, nous développons une technique efficace pour calculer un sous-graphe minimal dense. Nous démontrons empiriquement que nous pouvons obtenir de bons

résultats en utilisant cette approche. Dans ce qui suit, je présente deux algorithmes: MinAndRemove et FastDSLO. Le premier algorithme, MinAndRemove, calcule à chaque étape un sous-graphe minimal dense, supprime une fraction $(1-\alpha)$ de ses noeuds ($\alpha$ donné en entrée) afin d'appliquer la contrainte Jaccard et ensuite itère jusqu'à ce que le graphe soit vide ou que nous ayons trouvé $k$ sous-graphes. Comme le calcul d'un sous-graphe dense est lent pour des ensembles de données plus larges, dans FastDSLO nous remplaçons l'algorithme exact avec un algorithme d'approximation rapide.

**Algorithm 15** MinAndRemove$(G, k, \alpha)$

**Input:** Un graphe $G = (V, E)$, un nombre entier $k > 0$, $\alpha \in [0, 1]$
**Output:** Une liste $L$ d'au plus $k$ sous-graphes de $G$, $G_i = (V_i, E_i)$, t.q. la contrainte sur le coefficient Jaccard par paire sur les $V_i$ n'est pas violée (Equation (A.1)).
$L := \varnothing$
**while** $< k$ sous-graphes sont trouvés et $G$ n'est pas vide **do**
    Trouver un sous-graphe minimal dense $G_i = (V_i, E_i)$ de $G$
    $L := L \cup \{G_i\}$
    Pour chaque nœud $v$ dans $V_i$, soit $\delta_G(v)$ l'ensemble de voisins de $v$ dans $G$.
    Supprimer $\lceil (1-\alpha)|V_i| \rceil$ nœuds avec la valeur minimale $|\delta_H(v) \setminus V_i|$ et toutes leurs arêtes dans $G$.
**end while**
**return** L

**Algorithm 16** FastDSLO$(G, k, \alpha)$

**Input:** Un graphe $G = (V, E)$, un nombre entier $k > 0$, $\alpha \in [0, 1]$
**Output:** Une liste $L$ d'au plus $k$ sous-graphes de $G$, $G_i = (V_i, E_i)$, t.q. la contrainte sur le coefficient Jaccard par paire sur $V_i$ s n'est pas violée (Equation (A.1)).
$L := \varnothing$
**while** $< k$ sous-graphes sont trouvés et $G$ n'est pas vide **do**
    Trouvez une 1/2-approximation $G_i = (V_i, E_i)$ pour le problème du plus dense sous-graphe en exécutant l'algorithme glouton de [22].
    $L := L \cup \{G_i\}$
    Pour chaque nœud $v$ dans $V_i$, soit $\delta_G(v)$ l'ensemble de voisins de $v$ dans $G$.
    Supprimer $\lceil (1-\alpha)|V_i| \rceil$ nœuds avec la valeur minimale $|\delta_H(v) \setminus V_i|$ et toutes leurs arêtes dans $G$.
**end while**
**return** L

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| web-Stanford | 71% | 73% | 76% | 79% | 81% |
| com-Youtube | 48% | 52% | 51% | 61% | 62% |
| web-Google | 80% | 80% | 80% | 80% | 80% |
| Youtube-growth | 44% | 46% | 53% | 59% | 57% |
| As-Skitter | 58% | 59% | 59% | 62% | 64% |

Table A.1: Ratio entre la densité de MinAndRemove et notre limite supérieure

| $k = 10$ | $\alpha = 0.1$ | $\alpha = 0.2$ | $\alpha = 0.3$ | $\alpha = 0.4$ | $\alpha = 0.5$ |
|---|---|---|---|---|---|
| LiveJournal | 24% | 24% | 25% | 28% | 27% |
| Hollywood-2009 | 18% | 19% | 19% | 21% | 23% |
| Orkut | 18% | 20% | 21% | 25% | 27% |

Table A.2: Ratio entre la densité de FastDSLO et notre limite supérieure

**Evaluation.**

Nous avons effectué une évaluation des algorithmes sur 8 jeux de données, qui peuvent être divisés en deux groupes selon leur taille:

- 5 ensembles de données avec entre 2M et 11M arrêtes
- 3 ensembles de données avec entre 43M et 117M arrêtes

Nous avons étudié la performance des algorithmes en fonction de la valeur de la fonction objectif (somme des densités correspondant aux sous-graphes trouvés). La valeur maximale que la fonction objectif peut prendre est $k \cdot \rho_{max}$, où $\rho_{max}$ est la densité du sous-graphe le plus dense. Nous utiliserons cette valeur maximale possible comme limite supérieure pour l'optimum.

Dans le tableau A.1, nous mesurons la densité totale calculée par notre algorithme principal en pourcentage de notre limite supérieure, lorsque $k = 10$. Le premier aspect que nous pouvons constater est que nos algorithmes fonctionnent très bien en terme de fonction objective: la solution est d'au moins 44% de la valeur de la limite supérieure et peut atteindre un pourcentage de 80%. Le deuxième aspect est que la qualité de la solution s'améliore en fonction de $\alpha$, de sorte qu'en permettant un chevauchement entre les sous-graphes, nous pouvons améliorer notre fonction objective. Nous n'avons qu'un exemple, l'ensemble de données $web-Google$, ou cela ne se produisait pas, mais étant donné la proximité de la limite supérieure, il se pourrait que nous ayons déjà trouvé la solution optimale.

Lorsque nous évaluons le temps nécessaire pour MinAndRemove, nous obtenons entre 15min(le plus petit ensemble de données) et 3h (le plus grand ensemble de données, 11M) pour trouver 10 sous-graphes.

Dans le tableau A.2, nous avons fixé $k = 10$ et nous mesurons la densité de FastDSLO en pourcentage de notre limite supérieure vers une solution optimale. Nous modifions également le paramètre $\alpha$. Bien que cette heuristique soit moins précise que MinAndRemove, elle a l'avantage de pouvoir passer à l'échelle. Nous pouvons remarquer que nous améliorons la fonction objective lorsque nous augmentons $\alpha$.

Le temps nécessaire pour executer FastDSLO varie entre 30min et au plus 2h20' (117M arêtes) pour 10 sous-graphes.

## A.4 Sous-graphes denses avec contraintes de taille

Dans la section précédente, nous avons étudié le problème de trouver des sous-graphes denses avec un chevauchement limité et une somme maximale de densités. Cependant, pour certaines applications telles que la détection d'événements, les sous-graphes les plus denses peuvent être gros et difficiles à analyser. Pour faire face à cette limitation, nous considérons le problème de trouver des sous-graphes denses avec des restrictions sur la taille des sous-graphes. En particulier, dans un graphe pondéré, nous souhaitons trouver un des sous-graphes avec $k$ nœuds dont la somme des poids de ses arêtes est maximale. Le problème est NP-difficile et difficile à approximer. Dans notre travail, nous exploitons les propriétés des graphes du monde réel afin de développer des algorithmes efficaces.

Nous résumons nos contributions de la manière suivante:

- Nous concevons un algorithme efficace et exacte puis un algorithme d'approximation pour résoudre le problème du $k$-sous-graphe le plus lourd. Notre algorithme exacte s'adapte à de grands réseaux réels pondérés, pour $k$ jusqu'à 15 ou plus selon la structure du graphe. La version approximative de notre algorithme s'adapte à des valeurs encore plus grandes de $k$. Nous montrons que nos algorithmes sont plus efficaces que l'état de l'art pour le même problème.

- Nous montrons que notre algorithme est bien adapté pour résoudre des problèmes connexes, comme trouver les top $t$ $k$-sous-graphes lourds dans un graphe.

**Définition du problème.**

*Le $k$-sous-graphe le plus lourd.* Étant donné un graphe pondéré non orienté $G$ et un nombre entier $k > 1$, nous souhaitons trouver un sous-graphe contenant $k$ nœuds dont la somme des poids de ses arêtes est maximale. Pour ce problème, nous supposons que le graphe contient au moins $k$ noeuds.

*Top $t$ $k$-sous-graphes lourds.* Étant donné un graphe pondéré non orienté $G$, et deux nombres entiers $k > 1$ et $t > 0$, trouvez au plus $t$ sous-graphes disjoints contenant $k$ nœuds de sorte que la somme des poids de leurs arêtes soit maximale. Pour ce problème, nous supposons que le graphe contient au moins $k \cdot t$ nodes.

**Algorithmes.**

La stratégie de séparation et d'évaluation est une méthode bien connue pour résoudre des problèmes de maximisation combinatoire. L'idée principale est de diviser l'espace de recherche en plusieurs branches de calcul. Intuitivement, on peut penser que le calcul construit un arbre de recherche où chaque noeud correspond à un ensemble de solution. La racine correspond à l'ensemble de toutes les solutions

possibles tandis que les successeurs d'un noeud sont des sous-ensembles (idéalement disjoint). La formation des succésseurs d'un noeud s'appelle la *phase de séparation*. Chaque noeud de l'arbre possède une borne inférieure (généralement une solution du problème) et une borne supérieure sur la valeur des solutions qu'il contient. Si la borne supérieure d'un noeud dans l'arbre est inférieure à la borne inférieure globale (c'est-à-dire au maximum des solutions trouvées jusqu'à présent), tout le sous arbre du noeud n'a pas besoin d'être exploré car il ne peut contenir que des solutions pire que la meilleure trouvée, et donc il peut être taillé (c'est à dire retiré du calcul). Cette étape est la *phase d'évaluation*.

Dans notre approche, la phase de séparation est basée sur la décision d'ajouter ou non une arête spécifique (et donc ses extrémités) dans la solution correspondante. Plus précisément, si nous recherchons le sous-graphe le plus lourd de $k$ nœuds, notre algorithme peut se résumer de la façon suivant:

- Au départ, l'arbre de recherche est réduit à sa racine pour laquelle nous n'avons pas encore choisi d'arête. La borne supérieure est la somme des $\binom{k}{2}$ arêtes plus lourdes, tandis que la solution associée est le sous-graphe vide et donc la borne inférieure est 0.

- Ensuite, à chaque itération, nous créons deux successeurs pour le nœud avec la borne inférieure maximale (c'est-à-dire la densité de la solution associée). Supposons que le nœud soit en profondeur $i$ dans l'arbre, nous gardons les décisions prises sur les premiers $i-1$ arêtes et nous créons deux successeurs, l'un où la $i^{th}$ arête est incluse et l'autre où elle est exclue.

La borne inférieure d'un noeud est donnée par la somme des poids des arêtes du sous graphe induit par les arêtes incluses.

En supposant que le nombre de nœuds dans le sous-graphe induit par les arêtes est $s$, la borne supérieure est donnée par les poids des arêtes du sous-graphe plus la somme des $\binom{k}{2} - \binom{s}{2}$ arêtes suivantes dans l'ordre. Lorsque nous ajoutons une nouvelle arête $(u, v)$, nous devons vérifier si les deux extrémités font partie de la liste de nœuds de la solution actuelle. S'ils ne le font pas partie, nous les ajoutons dans la liste de nœuds et nous mettons à jour le poids de cette solution en ajoutant le poids de chaque arête entre $u$ (respectivement $v$) et les sommets en $S$. Nous avons donc besoin d'un algorithme pour vérifier efficacement si deux nœuds sont adjacents ou non.

Nous calculons initialement un ordre de base (ou un ordre de dégénérescence) de la version non pondérée du graphe. Nous conservons pour chaque nœud dans le graphe une liste triée de ses voisins ayant un rang plus élevé (la taille maximale d'une liste de voisinage tronqué est donc $c$, la dégénérescence du graphe). Étant donné 2 nœuds $x$ et $y$ (sans perte de généralité nous supposons que $y$ a un rang plus élevé que $x$), nous pouvons vérifier efficacement s'ils sont adjacents en vérifiant si $y$ est dans la liste de voisinage tronqué de $x$ par recherche binaire en temps $O(\log(c))$. Il est à noter que cette étape est en pratique le goulot d'étranglement de notre algorithme prenant environ 80 % du temps.

Le temps de calcul de la création d'un successeur est de $O(k \cdot \log(c))$ (où $c$ est la dégénérescence du graphe), car cela correspond à vérifier si $u$ et $v$ appartiennent au

voisinage des noeuds du sous-graphe de la solution (sachant qu'il y a au plus $k-1$ noeuds). Il est à noter que la taille maximale d'un voisinage est de $n-1$. Nous continuons à former les successeurs du nœud avec la borne inférieure la plus grande jusqu'à ce que la branche soit taillée. En plus de l'élagage en utilisant la borne inférieure et supérieure, si le nombre de noeuds dans le sous-graphe dépasse $k$ on élimine également la branche. Lorsque nous ajoutons une nouvelle arête $(u, v)$, nous devons vérifier si les deux extrémités appartiennent à la solution actuelle. Dans le cas négatif, nous ajoutons les poids des arêtes entre les extrémités de la nouvelle arête $(u, v)$ et les sommets de la solution actuelle. Cependant, si une telle arête $(u, x)$ a un classement supérieur à $(u, v)$ (c'est-à-dire que le poids de $(u, x)$ est supérieur ou égal au poids de $(u, v)$), alors nous pouvons tailler cette solution. En effet, puisque nous prenons les décisions sur les arêtes en ordre croissant en fonction de leur poids, cela signifie que cette arête $(u, x)$ a été exclue de la solution, elle ne peut donc pas être ajouté à cette étape.

Une caractéristique clé de la stratégie de séparation et d'évaluation est l'ordre avec lequel la méthode itère sur les noeuds l'arbre des solutions. L'ordre peut être BFS, DFS, ou en fonction de la borne inférieure ou supérieure d'un noeud. Après avoir examiné ces différents ordres, nous avons constaté que l'examination des nœuds selon les poids des solutions candidates est la plus efficace. Par conséquent, chaque fois qu'un successeur est créé par l'algorithme, il est ajouté à un tas, dont la racine est toujours la solution de poids maximal.

L'intuition derrière notre méthode est que la meilleure solution contient de nombreuses arêtes de poids élevé, tout en contenant éventuellement quelques arêtes de poids faibles. La méthode ne devrait pas à avoir à explorer tous les arêtes, car celles de poids faible devrait être ajoutées à la solution en formant le sous-graphe induit par les arêtes de poids élevé sans que l'on aie besoin de les ajouter explicitement.

**Evaluation.**

Nous avons collecté un ensemble de tweets avec l'API Twitter Streaming entre les mois de novembre et décembre 2015 et nous avons obtenu quatre graphes qui sont détaillés en Table A.3.

| Données | Noeuds | Arêtes |
|---|---|---|
| Francais-Nov. | 220 K | 2.9 M |
| Francais-Dec. | 200 K | 2.1 M |
| Englais-Nov. | 2.5 M | 4.5 M |
| Englais-Dec. | 1.8 M | 3.1 M |

Table A.3: L'ensemble de graphes de mots extraits de Tweets.

Afin d'avoir une base de comparaison, nous avons implémenté l'heuristique gloutone utilisée dans [9]. L'algorithme est basé sur l'ordre de dégénérescence pondérée et consiste à retirer à plusieurs reprises un sommet avec le degré pondéré minimum dans le graphe restant jusqu'à qu'il ne reste que $k$ sommets.

Le tableau A.4 montre le temps de fonctionnement de notre algorithme et le temps de fonctionnement de l'algorithme gluton. Afin de différencier le temps néces-

saire pour charger l'ensemble de données en mémoire et le temps nécessaire pour exécuter le calcul de le $k$-sous-graphe le plus lourd, nous avons ajouté une colonne supplémentaire "Temps de chargement". Pour les valeurs de $k \leq 15$, nous pouvons voir que les deux algorithmes ont une bonne performance, terminant le calcul en quelques secondes dans la plupart des cas. Le temps de l'algorithme basé sur l'ordre de dégénérescence ne varie pas beaucoup avec $k$.

Dans le tableau A.5, nous présentons le poids total du $k$-sous-graphe le plus lourd pour différentes valeurs de $k$ lorsque nous utilisons les deux techniques. Nous notons que notre algorithme trouve la solution optimale. Afin de comparer les sous-graphes obtenus par les deux algorithmes, nous calculons le rapport entre leur solution et la solution optimale. Comme on peut le voir, l'approximation de l'heuristique gloutone varie entre 0.99 et $0,67$ sur nos graphes.

Le tableau A.6 montre le rapport entre le nombre d'arêtes dans les sous-graphes de taille $k$ et le nombre d'arêtes dans une clique de taille $k$. Comme on peut le voir, dans tous les cas sauf celui de "Francais-Dec", la solution est une clique. Cela montre que la structure de ces graphes du monde réel est spéciale et que même si nous cherchons le $k$-sous-graphe le plus lourd et non des cliques, concevoir un algorithme pour trouver une clique de taille $k$ et de poids maximal est un problème très similaire sur ces graphes du monde réel. Comme l'énumération des cliques peut être effectuée efficacement dans les graphes du monde réel [25, 63], une heuristique basée sur l'énumération de toutes les cliques de taille $k$ pourrait être envisagée.

| | | Algorithme gluton | Notre algorithme | | |
|---|---|---|---|---|---|
| Données | Temps de chargement | $k$ = 5, 10 and 20 | $k$ = 5 | $k$ = 10 | $k$ = 15 |
| Francais-Nov. | 0,8s | 0,7s | 0,5s | 0,9s | 17s |
| Francais-Dec. | 0,6s | 0,5s | 0,9s | 1,0s | 9m58s |
| Englais-Nov. | 11s | 8,9s | 17s | 17s | 18s |
| Englais-Dec. | 12s | 4,8s | 7,5s | 7,0s | 7,0s |

Table A.4: Comparaison de temps d'execution.

| | Algorithme gluton | | | Notre algorithme | | |
|---|---|---|---|---|---|---|
| Données | $k$ = 5 | $k$ = 10 | $k$ = 15 | $k$ = 5 | $k$ = 10 | $k$ = 15 |
| Francais-Nov. | 24669 (0,92) | 70528 (0,96) | 115341 (0,96) | 26824 | 73432 | 120419 |
| Francais-Dec. | 18158 (0,95) | 22458 (0,67) | 36358 (0,81) | 19090 | 33338 | 45020 |
| Anglais-Nov. | 2942467 | 5836705 (0,99) | 8114183 (0,99) | 2942467 | 5913372 | 8161838 |
| Anglais-Dec. | 2386716 | 5125458 (0,98) | 7830637 | 2386716 | 5235181 | 7830637 |

Table A.5: Poids total du $k$-sous-graphe le plus lourd pour différentes valeurs de $k$ lorsque nous utilisons l'algorithme gluton et notre algorithme.

Nous avons présenté un nouvel algorithme basé sur la stratégie de séparation et d'évaluation pour résoudre le problème du $k$-sous-graphe le plus lourd dans les graphes pondérés. La phase de séparation est basée sur la décision d'inclure une arête dans la solution ou non, et les arêtes sont examinés dans l'ordre de poids non croissant pour maximiser l'efficacité de l'algorithme. La phase d'évaluation est

| | Algorithme gluton | | | Notre algorithme | | |
|---|---|---|---|---|---|---|
| Données | $k = 5$ | $k = 10$ | $k = 15$ | $k = 5$ | $k = 10$ | $k = 15$ |
| Francais-Nov. | 1,0 | 1,0 | 0,98 | 1,0 | 1,0 | 0,98 |
| Francais-Dec. | 0,4 | 0,27 | 0,44 | 1,0 | 0,44 | 0,3 |
| Anglais-Nov. | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 |
| Anglais-Dec. | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 |

Table A.6: La densité, où la densité 0 correspond à des sommets isolés, et la densité 1 au graphe complet.

double: (i) en fonction de la taille du sous-graphique et (ii) sur le calcul efficace d'une borne supérieure aussi proche du plus grand élément que possible.

L'algorithme passe à l'échelle pour de grands graphes du monde réel pondérés contenant des millions d'arêtes pour une valuer maximale de $k = 15$ ou plus selon la structure du graphe. Une version d'approximation de notre algorithme peut passer à une plus grande échelle. Nous montrons que notre algorithme fonctionne mieux qu'une méthode de l'état de l'art basée sur l'ordre de dégénérescence pondérée.

Les futurs travaux incluent des améliorations à notre algorithme en utilisant la programation parallèle, ainsi que sa généralisation à la détection d'autres types de sous-graphes. Une direction possible est la détection de communautés, où une communauté est définie intuitivement comme un ensemble de noeuds fortement connectés entre eux mais faiblement connectés à l'extérieur [38].

## A.5 Lister ou compter les $k$ cliques et trouver la $k$-clique dégénérescence

Les travaux récents dans la communauté de bases de données et d'exploration de données exigent des algorithmes efficaces pour lister ou compter toutes les $k$-cliques dans le graphe d'entrée. En particulier, dans [95], l'auteur développe un algorithme pour trouver des sous-graphes avec un nombre moyen de $k$-cliques maximal, le comptage de $k$-cliques étant un élément important. Dans [89] est présenté un algorithme pour organiser des cliques dans des structures hiérarchiques, ce qui nécessite d'énumérer toutes les $k$-cliqus. Des algorithmes efficaces pour le comptage de $k$-cliques permettent le calcul d'une généralisation naturelle de la dégénérescence, la *$k$-clique* dégénérescence. Une autre application intéressante peut être dans la détection d'événements, car, comme nous l'avons découvert dans la section précédente, un sous-graphe d'une petite taille tres lourd sera souvent une clique.

Motivé par les études susmentionnées, nous développons les algorithmes parallèles les plus efficaces pour lister et compter tous les $k$-cliques d'un graphe d'entrée, avec $k$ étant un paramètre d'entrée. Notre analyse théorique montre que même la version séquentielle de notre algorithme est supérieure à celle des algorithmes existentes pour le même problème. En outre, par opposition aux algorithmes existentes, notre algorithme est parallèle, ce qui améliore encore le temps de fonctionnement total. Notre vaste évaluation expérimentale montre que les versions séquentielles et parallèles de notre algorithme dépassent les approches exisentes. En particulier,

notre algorithme parallèle est capable d'énumérer toutes les cliques dans des graphes contenant des dizaines de millions d'arêtes, ainsi que toutes les 10-cliques dans les graphes contenant des milliards d'arêtes, en quelques minutes ou quelques heures, respectivement, tout en obtenant un excellent degré de parallélisme. Nous montrons également que notre algorithme peut être utilisé comme sous-routine efficace pour calculer la $k$-clique dégénérescence dans de grands graphes et une approximation du $k$-clique plus dense sous-graphe [95].

Nous résumons nos contributions de la manière suivante:

- Nous développons un algorithme parallèle pour lister et compter toutes les $k$-cliques dans un graphe. Notre analyse théorique montre que notre algorithme réalise le meilleur temps de fonctionnement asymptotique, tirant parti de la faible densité de réseaux réels. Il nécessite une mémoire linéaire dans la taille du graphe d'entrée. En pratique, la version séquentielle de notre algorithme est un ordre de grandeur plus rapide que les algorithmes existenes sur les graphes du monde réel, tandis que la version parallèle réalise un excellent degré de parallélisme.

- Nous développons un algorithme efficace pour calculer la $k$-clique dégénérescence. Un tel algorithme est basé sur l'algorithme pour lister les $k$-cliques, tout en exigeant asymptotiquement le même temps de fonctionnement;

- Nous montrons comment notre algorithme pour lister les $k$-cliques peut être utilisé pour trouver efficacement les sous-graphes les $k$-clique plus dense sous-graphes.

**Définition du problème.**

Nous supposerons que nous recevons un graphe non orienté et connecté $G = (V(G), E(G))$. Si le graphe d'entrée n'est pas connecté, nos algorithmes peuvent être exécutés dans chacun des composants connectés, indépendamment. Etant donné un numbre entier $k > 1$, un sous-graphe induit de $G$ est appelé $k$-clique s'il contient exactement $k$ noeuds dont chaque paire est connectée par une arête appartenant à $E(G)$. Le problème suivant généralise le problème de trouver tous les triangles dans un graphe.

**Définition du problème** (*problème de listage des $k$-cliques*). Étant donné un graphe non orienté $G = (V(G), E(G))$ et un nombre entier $k > 1$, nous souhaitons énumérer toutes les $k$-cliques dans $G$.

Étant donné que le nombre de $k$-cliques dans un graphe peut être très grand, le stockage ou même la production en sortie, peuvent poser des problèmes importants. Par conséquent, nous considérons également une variante du problème où l'objectif est de compter le nombre de $k$-cliques dans un graphe.

**Définition du problème** (*problème de comptage des $k$-cliques*). Étant donné un graphe non orienté $G = (V(G), E(G))$ et un nombre entier $k > 1$, nous souhaitons compter le nombre de $k$-cliques en $G$.

Les problèmes susmentionnés sont aussi difficiles que le problème de la clique maximale qui est NP-difficile et difficile à approximer dans un facteur $n^{1-\epsilon}$ [46]. Par conséquent, il est peu probable que des algorithmes efficaces pour des graphes arbitraires soient développés. Dans notre travail, nous concevons des algorithmes efficaces pour des graphes peu denses, c'est-à-dire avec des valeurs de la dégénérescence et de l'arboricité relativement faibles.

Nous généralisons la définition de la dégénérescence d'un graphe de la manière suivante. Étant donné un nombre entier de $k > 1$, le *k-clique degré* d'un noeud $v$ en $V(G)$ est défini comme le nombre de $k$-cliques en $G$ qui contiennent $v$. Pour n'importe quel noeud $v$, soit $c^k(v)$ le plus grand entier de sorte qu'il existe un sous-graphe $H$ de $G$ avec (i) $H$ contenant $v$, (ii) un noeud $u$ dans $H$ ayant $k$-clique degré au moins $c^k(v)$.

**Définition du problème** (problème de *k-clique dégénérescence*). Étant donné un graphe non orienté $G = (V(G), E(G))$ et un nombre entier $k > 1$, calculer la $k$-clique dégénérescence de $G$.

La définition précédente généralise la notion bien connue de dégénérescence (pour $k = 2$).

**Problem definition**: (problème de $k$-clique plus dense sous-graphe). Étant donné un graphe non orienté $G = (V(G), E(G))$, trouvez un sous-graphe $H$ de $G$ de sorte que la $k$-clique densité est maximisée. La $k$-clique densité de $H$ est $d_k(H) = \frac{c_k(H)}{|V(H)|}$, où $c_k(H)$ est le nombre de $k$-cliques induites par $H$.

Ce problème a été abordé et formalisé dans [95].

**Algorithmes.**

**Lister les $k$-cliques.** Nous présentons notre algorithme parallèle pour trouver toutes les $k$ cliques sur de grands graphes du monde réel. Nous empruntons des idées des algorithmes séquentiels présentés dans [55] et [25], afin d'obtenir un algorithme parallèle efficace pour traiter de grands graphes du monde réel. Les caractéristiques les plus importants de notre algorithme sont qu'il est parallèle (par opposition aux algorithmes existentes pour le même problème) et qu'il tire parti de la sparsité du graphe d'entrée qui mène au meilleur temps d'exécution asymptotique comme le montre notre analyse théorique. Cela permet de trouver toutes les 10-cliques sur des graphes du monde réel contenant plus d'un milliard d'arêtes, ainsi que rechercher toutes les cliques dans des graphes contenant quelques millions d'arêtes, en quelques minutes ou quelques heures.

Le problème clé a resoudre pour créer des algorithmes efficaces pour lister toutes les $k$-cliques consiste à tailler efficacement l'espace de recherche tout en limitant les opérations redondantes (comme le traitement d'une même clique plusieurs fois), ainsi que diviser la tâche principale en sous-tâches indépendantes qui peuvent être exécuté en parallèle.

La première partie de notre algorithme consiste à déterminer une ordre optimale des noeuds dans le graphe avec la procédure suivante: à partir du graphe d'entrée $G$, supprimez successivement un noeud (et ses arêtes) ayant le plus petit degré dans le graphe actuel, jusqu'à ce que le graphe devient vide. Il s'agit essentiellement

du même algorithme utilisé pour calculer la dégénérescence [66]. Soit $\eta : V(G) \to \{1, \ldots, n\}$ soit une fonction indiquant pour chaque noeud $v$ l'étape $t$ dans laquelle un tel noeud a été supprimé, $1 \leq t \leq n$. Pour chaque noeud $u$, nous définissons l'ensemble $\Delta_\eta(u)$ composé de tous les voisins $v$ de $u$ avec $\eta(v) > \eta(u)$. Notez que, avec cette définition, nous avons $\max_{u \in V(G)} |\Delta_\eta(u)| = c$ le nombre de marquage du graphe. Nous utilisons $\Delta(u)$ pour désigner tous les voisins de $u$.

L'algorithme pour trouver toutes les 4 cliqus peut être décrit de la manière suivante. Pour chaque arête $(u, v)$ en parallèle, l'algorithme trouve toutes les 4-cliques dans le sous-graphe induit par tous les noeuds dans $\Delta_\eta(u) \cap \Delta_\eta(v)$. Plus précisément, pour chaque arête $(u, v)$ l'ensemble $\Delta_\eta(u, v) = \Delta_\eta(u) \cap \Delta_\eta(v)$ is calculé. Ensuite, pour chaque $w$ dans $\Delta_\eta(u, v)$ l'ensemble $\Delta_\eta(u, v, w) = \Delta_\eta(w) \cap \Delta_\eta(u, v)$ est calculé. Enfin, pour chaque $x$ dans $\Delta_\eta(u, v, w)$, le 4-clique $\{u, v, w, x\}$ est produit en sortie.

Notez que pour chaque arête $(u, v)$ l'ensemble $\Delta_\eta(u, v)$ contient des noeuds dont le nombre de marquage n'est pas inférieure à celle de $u$ et $v$. Par conséquent, chaque $k$-clique est listé lorsque l'arête $(u, v)$ est traitée, où $u$ et $v$ sont les noeuds ayant le plus petit nombre de marquage dans la clique. Cela s'avère être une stratégie d'élagage efficace, car elle limite considérablement le nombre de voisins à considérer. En outre, la taille des $\Delta_\eta$ devient de plus en plus petite lorsque nous plongeons plus profondément dans les boucles imbriquées, ainsi réduisant l'espace de recherche.

Notre algorithme peut être implémenté dans quelques lignes de code et notre analyse théorique montre qu'il s'agit de l'algorithme le plus rapide pour lister des $k$-cliques même dans le cas séquentiel, alors que sa version parallèle est encore plus rapide, comme le témoigne notre évaluation expérimentale.

**La $k$-clique dégénérescence.**

L'algorithme pour calculer la dégénérescence d'un graphe [22, 66] se déroule de la manière suivante: à partir du graphe d'entrée, à chaque étape, un noeud de degré minimum et tous ses arêtes sont suprimmés jusqu'à ce que le graphe soit vide. En s'appuyant sur cette approche, nous développons un algorithme pour n'importe quel $k \geq 3$.

Dans ce qui suit, nous décrivons l'algorithme pour calculer la 5-clique dégénérescence, à partir de laquelle nous pouvons généraliser facilement à n'importe quel $k \geq 3$. La première partie de notre approche consiste à calculer le nombre de 5-cliques auquel chaque noeud appartient, c'est-à-dire le 5-clique degré de chaque noeud. Nous effectuons le calcul en exécutant l'algorithme qui liste les clique et en augmentant le 5-clique de degré de nœud $u$ chaque fois que nous trouvons une clique de taille 5 contenant $u$. Nous stockons ensuite les degrés calculés dans un tas binaire afin de suivre le minimum 5-clique degré. Ensuite, comme dans la dégénérescence standard, nous supprimes itérativement un noeud d'un 5-clique degré minimale en utilisant le tas et mettons à jour le 5-clique degré de ses voisins jusqu'à ce que le graphe devienne vide.

La dernière opération n'est pas banale, car le noeud $u$ pourrait appartenir à un grand nombre de 5-cliques. Le stockage de toutes ces 5-cliques n'est donc pas réalisable. Pour éviter cela, nous procédons de la manière suivante. À chaque itération lors de la suppression d'un nœud $u$ avec le 5-clique degré minimale, nous

énumérons toutes les 5-cliques contenant $u$ (sans les stocker). Chaque fois que nous trouvons une clique contenant les nœuds $\{u, v, w, x, y\}$, nous décrémentons par 1 les degrés de chaque nœud dans $\{v, w, x, y\}$.

**Le $k$-clique plus dense sous-graphe.**

Un simple 1/2 algorithme d'approximation pour le problème de sous-graphe le plus dense ($k$ = 2) [22] se déroule dans la maniere suivante: à partir du graphe d'entrée, on élimine à chaque étape un noeud avec un degré minimum (et tous ses arêtes). Nous itérons jusqu'à ce que le graphe soit vide. Nous renvoyons ensuite un sous-graphe avec une densité maximale parmi tous les sous-graphes produits lors de l'exécution de l'algorithme. L'algorithme est similaire à l'algorithme de calcul de la dégénérescence. Dans [95], l'auteur montre une variante d'un tel algorithme ou, à chaque étape, le noeud avec le minimum $k$-clique degré est supprimé, donne un algorithme d'approximation $1/k$. Par conséquent, notre algorithme pour calculer la 5-clique dégénérescence peut être facilement modifié afin d'obtenir un algorithme de $1/k$ approximation pour le problème de $k$-clique plus dense sous-graphe.

**Évaluation.**

Nous considérons plusieurs graphes du monde réel que nous avons obtenus de [58] et [53]. Nous les divisons en deux groupes principaux: *grands* graphes contenant jusqu'à des dizaines de millions d'arêtes, pour lesquels nous pouvons compter toutes les $k$-cliques, ainsi que des graphes *très grands* contenant jusqu'à des milliards d'arêtes pour lesquelles nous ne pouvions compter que des cliques de taille limitée (jusqu'à 10). Nous évaluons séparement l'algorithme pour les triangles et le $k$-clique comptage. Nous avons porté nos expériences sur une machine linux ayant 2 fils Intel Xeon CPU E7-4870 à 2,40 GHz avec 10 noyaux divisés en 2 fils chacun (un total de 40 fils) et équipés de 64 G de RAM DDR3 1333 MHz. Nous évaluons nos algorithmes contre les algorithmes de l'état de l'art pour les problèmes correspondants tout en mesurant le temps d'execution. Nous considérons les approches suivantes:

- **CF:** l'algorithme compact-forward pour le comptage de triangles de [55], nous avons utilisé l'implémentation C disponible sur la page Web de l'auteur.
- **MACE:** l'algorithme détaillé dans [63,91], nous avons utilisé l'implémentation C disponible sur la page Web de l'auteur.
- **Arboricité:** l'algorithme décrit dans [25]. Pas au courant d'une implémentation existante, nous avons effectué une implémentation efficace dans C.
- **FkCE1:** notre algorithme en utilisant un seul thread (c'est-à-dire la version séquentielle de notre algorithme) implémenté dans C.
- **FkCE10:** la version parallèle de notre algorithme en utilisant 10 threads. Nous avons utilisé openMP [31] pour rendre notre code C parallèle

Le tableau A.7 montre le temps de fonctionnement des algorithmes pour compter les triangles. Dans ce cas, nous considérons la version séquentielle de notre algorithme (FkCE1). Une telle table montre que même la version séquentielle de notre

| | | Algorithmes | | | |
|---|---|---|---|---|---|
| Données | # triangles | CF | MACE | Arboricité | FkCE1 |
| DBLP | 2,224,385 | 0.8s | 1.3s | 0.6s | 0.4s |
| Wikipedia | 145,707,846 | 1m07s | 22m22s | 1m04s | 40s |
| Orkut | 627,584,181 | 4m06s | 28m02s | 3m41s | 2m14s |
| Friendster | 4,173,724,142 | 1h50m41s | 5h29m40s | 2h57m21s | 1h05m31s |
| LiveJournal | 177,820,130 | 44s | 6m13s | 37s | 27s |

Table A.7: Le temps d'execution pour compter les triangles sur nos très grands graphes.

| | Algorithmes | | |
|---|---|---|---|
| Données | MACE | Arboricité | FkCE1 |
| road-CA | 0.9s | 2.2s | 2.0s |
| Amazon | 1.0s | 0.8s | 0.6s |
| soc-pocket | 14m27s | 11m23s | 1m15s |
| loc-gowalla | 8m46s | 7m52s | 34s |
| Youtube | 1m05s | 1m12s | 3.9s |
| cit-patents | 22s | 24s | 15s |
| zhishi-baidu | 1h00m44s | 32m23s | 3m58s |
| WikiTalk | >24h | >24h | 5h53m36s |

Table A.8: Le temps d'execution pour compter toutes les cliques sur nos grands graphes.

algorithme est toujours supérieure à celle des algorithmes existantes pour le comptage des triangles. Il montre également que l'algorithme présenté dans [55] est très efficace.

Nous considérons alors le problème de comptage de $k$-cliques sur notre collection de grands ensembles de données. Le tableau A.8 montre que notre algorithme peut compter toutes les $k$-cliques pour n'importe quelle valeur de $k$ dans tous ces graphes, dans quelques minutes dans la plupart des cas. C'est beaucoup plus rapide que les autres algorithmes, jusqu'à concurrence de 10. En outre, notre algorithme est le seul capable de traiter, dans un delai de temps raisonnable, WikiTalk qui est l'ensemble de données avec le plus grand nombre de marquage.

Pour compter les triangles sur de très grands graphes et pour compter les cliques sur de grands graphes, nous obtenons des résultats satisfaisants même avec la version séquentielle de notre algorithme. Le potentiel complet de notre algorithme parallèle devient évident lors du comptage de $k$-cliques pour un plus grand $k$ sur des graphes très grands.

Dans cette section, nous présentons un algorithme parallèle pour lister ou compter toutes les $k$-cliques dans de très grands graphes du monde réel. Notre algorithme s'appuie sur la sparsité des graphes d'entrée et possède le meilleur temps asymptotique, tout en nécessitant une quantité linéaire de mémoire dans la taille de

l'entrée. En pratique, pour les grandes valeurs de $k$, la version séquentielle de notre algorithme est d'un ordre de grandeur plus rapide que les algorithmes de l'état de l'art pour le même problème, tandis que la version parallèle permet le gain d'un autre ordre de grandeur atteignant un degré de parallélisme presque optimal. Notre analyse expérimentale montre que notre algorithme parallèle permet d'énumérer les cliques dans des graphes contenant jusqu'à des dizaines de millions d'arêtes, ainsi que les 10-cliques dans des graphes contenant des milliards d'arêtes, en quelques minutes ou quelques heures, respectivement.

## A.6 Détection des événements de crise dans les médias sociaux

Les médias sociaux, tels que Twitter, représentent une source précieuse d'informations, car ils peuvent fournir des informations importantes sur le monde réel. En particulier, la détection des événements dans les médias sociaux a joué un rôle majeur dans l'exploration de données et l'analyse des réseaux sociaux, avec plusieurs études de recherche centrées sur les tremblements de terre [87], sur les situations générales de crise de la vie réelle [48], sur les pannes de sécurité informatique [85] et sur beaucoup d'autres classes d'événements (voir [10, 21, 48]).

La détection des événements se concentre depuis longtemps sur les médias traditionnels [5], mais les médias sociaux présentent des défis non triviaux ainsi que des opportunités sans précédent. Les tweets, par exemple, sont courts et écrits par des non-experts dans un langage informel émaillé de fautes d'orthographe et des abréviations. En outre, les informations pertinentes sont souvent mélangées avec des contenus bruyants ou inintéressants, tels que des spams ou des messages rédigés par des robots. Une autre difficulté supplémentaire lors de l'analyse des données de Twitter réside dans le fait que tous les tweets ne sont pas accessibles au public: seule une petite fraction d'entre eux est accessible via l'API Twitter.

D'autre part, les tweets pourraient fournir de nouvelles informations sur des événements bien connus ou même sur des événements qui n'ont pas été couverts du tout ou rapidement oubliés par les médias traditionnels. Par exemple, la guerre civile au Yémen a reçu peu d'attention dans les médias traditionnels [3], alors que de nombreux tweets de notre collection la mentionnaient. De même, les manifestations roumaines en 2013 n'ont pas été couvertes pendant la première semaine de l'événement [4]. Il serait intéressant d'analyser les tweets liés à un tel événement alors qu'ils existent. Ces tweets peuvent également fournir des informations en temps utile sur les événements de crise, un exemple notable étant l'accident d'avion sur la rivière Hudson en 2009 où les témoins ont été les premiers à donner des nouvelles de l'événement par Twitter.

Notre travail vise à éclairer quels événements de crise, tels que les tremblements de terre, les fusillades de masse, les bombardements, sont couverts par les médias sociaux. En particulier, nous cherchons à déterminer si Twitter couvre les événements

[3] http://www.acnur.org/t3/fileadmin/Documentos/Publicaciones/2016/10449.pdf?view=1
[4] http://www.dw.com/en/protests-erupt-in-romania-over-gold-mine/a-17068049

(tels que la guerre au Yémen) qui ont reçu peu ou pas d'attention des médias traditionnels. Malgré les énormes efforts déployés par les chercheurs de cette communauté (voir [10, 21, 48]), une approche viable et satisfaisante pour trouver des événements sur Twitter n'a pas encore été proposée. En outre, la plupart de ces approches nécessitent un effort non négligeable pour interpréter les résultats, ce qui complique la réalisation d'une étude à grande échelle difficile. Pour contourner le problème et permettre une comparaison à grande échelle, nous définissons une variante du problème de détection d'événements où nous appliquons une limite supérieure stricte sur la taille de la sortie produite par l'algorithme.

Nous développons une nouvelle technique de détection d'événements qui satisfait ces contraintes. Par exemple, dans notre évaluation expérimentale, nous permettons à notre algorithme de produire au plus dix résultats (c'est-à-dire des événements), chacun comprenant au plus dix mots-clés décrivant l'événement. Notre approche repose sur les étapes suivantes: i) un filtrage des tweets qui conserve uniquement ceux qui contiennent des termes liés à des événements de crise, en utilisant le lexique construit dans [73]; ii) trouver des lieux dont l'occurrence dans les tweets s'écarte de manière significative sur une fenêtre de temps donnée de leur fréquence moyenne; iii) une approche à base de graphes pour fournir des informations supplémentaires sur l'événement. Nous appelons notre approche EviDense, car notre approche graphe est basée sur la recherche de régions "denses" dans le graphe de co-occurrences.

**Définition du problème.**

Afin de permettre une évaluation expérimentale rigoureuse, nous donnons une définition précise du problème que nous allons étudier dans notre travail. Nous allons nous concentrer sur les *événements de crise*, qui sont des événements qui causent ou pourraient causer des situations "dangereuses" pour un groupe de personnes "important". Pour nos besoins, un groupe de personnes est considéré comme grand s'il comprend cinq personnes ou plus. Des examples d'événements de crise sont les tremblements de terre, des attentats terroristes, des fusillades de masse, des attaques aériennes, etc.

Notre objectif principal est de développer un algorithme efficace pour détecter les événements de crise et d'évaluer quel est le potentiel des réseaux sociaux en examinant les événements précédemment couverts. Pour permettre une étude à grande échelle, nous définissons une nouvelle variante du problème de détection d'événements habituel, en imposant une limite supérieure stricte sur la taille totale des résultats. Nous pouvons definir notre problème comme suit:

**Definition A.6.1** (Détection d'événements avec taille de sortie limitée)**.** Étant donné des entiers $k, s > 0$ et une collection de documents (p.ex.tweets), trouver $k$ événements de crise distincts, chacun comprenant un ensemble de $s$ mots-clés y compris le lieu et l'heure de l'événement.

Il existe déjà des approches qui imposent une certaine contrainte sur la taille des résultats (p. ex. [45]), mais une telle variante du problème de détection d'événements n'a pas été formulée, à notre connaissance. Dans nos expériences, nous avons fixé $k$ et $s$ à 10. Cela nous permet de procéder à une évaluation manuelle à grande

échelle des résultats sur une collection de tweets collectés sur une période de 14 mois. Dans notre évaluation expérimentale, nous verrons cette information, même si elle est limitée, nous permet d'obtenir des informations utiles sur la couverture des événements de crise dans notre collection de tweets.

**Algorithmes.**

Notre algorithme comprend les étapes principales suivantes: 1) collecte de tweets contenant des mots clés liés aux événements de crise; 2) reconnaissance et marquage des mentions de lieux dans les tweets; 3) identification des pics de mentions de lieux; 4) les mentions de lieux sont finalement complétées par des mots clés connexes afin de fournir des résultats plus informatifs.

Les tweets ont été collectés avec l'API Twitter tout en spécifiant une liste de mots-clés liés aux événements de crise, tels que *attaque*, *victimes*. À cette fin, nous utilisons la liste des mots-clés fournis dans [73]. Pour la reconnaissance et le marquage des endroits, nous utilisons un marqueur de reconnaissance d'entités qui a été formé sur les données Twitter [84]. Un tel marqueur se concentre sur dix catégories différentes: personne, endroit, société, produit, établissement, émission de télévision, film, équipe sportive et groupe de musique. Nous ne conservons que les étiquettes d'endroits et d'établissement tout en ignorant les autres. Nous nous référerons à la fois aux lieux et aux installations en tant que lieux. Notre intuition est que les pics dans les mentions d'un endroit (dans les tweets traitant des événements de crise) pourraient signaler qu'un événement important a eu lieu dans cet endroit. Après l'étape de marquage, nous filtrons les mots restants de manière à supprimer les mots de liaison, les URL et les termes peu fréquents (c'est-à-dire les termes avec une fréquence horaire inférieure à 5).

**Recherche des pics dans les mentions des endroits.**

Lorsqu'un évènement de crise se produit, nous observons un pic d'activité sur Twitter avec une augmentation soudaine de la fréquence des termes. Dans notre approche (où les tweets contiennent des mots-clés liés aux événements de crise), un pic dans le nombre d'occurrences d'un lieu nous donne un premier signal qu'un événement de crise se déroule à cet endroit. Pour chaque lieu, nous calculons un ensemble d'intervalles dans lesquels l'écart entre la fréquence du lieu et sa fréquence attendue reste supérieur à un seuil. Notre intuition est que tous les tweets (traitant du même lieu) qui sont publiés pendant un tel intervalle se réfèrent au même événement. Nous appelons *intervalles intéressants* de tels intervalles. Nous avons étudié la recherche d'intervalles intéressants *maximaux*. La fréquence attendue d'un lieu est calculée en supposant que les fréquences de localisation peuvent être approximées par la distribution binomiale.

Notre algorithme calcule, pour chaque lieu et chaque intervalle intéressant maximal de cet lieu, quelle est la fréquence de l'intervalle et combien elle s'écarte de la fréquence attendue. Ensuite, toutes les paires (le lieu, l'intervalle intéressant maximal) sont classées en fonction de l'écart entre la fréquence d'une localisation et sa fréquence moyenne. Un écart plus important correspond à un intérêt plus élevé dans l'événement, donc nous conservons les $k$ meilleurs paires dont l'écart est superieur à la moyenne.

**Quasi-cliques pondérées.**

Afin de compléter l'ensemble des lieux avec des informations supplémentaires sur l'événement correspondant, nous utilisons une approche à base des graphes. En particulier, pour chaque lieu et chaque intervalle intéressant pour cet lieu, nous souhaitons trouver un ensemble de termes qui induisent une région dense dans le graphe de co-occurrences pendant cet intervalle de temps. Formallement, étant donné un intervalle intéressant $\mathcal{I}$ et une collection de tweets, nous définissons un graphe non orienté pondéré $G_{\mathcal{I}} = (V_{\mathcal{I}}, E_{\mathcal{I}})$, où $V_{\mathcal{I}}$ se compose de l'ensemble des termes dans la collecte des tweets, et où deux termes sont reliés par une arête lors qu'ils apparaissent ensemble dans au moins un tweet de $\mathcal{I}$. Nous utilisons une fonction de poids $c : E \to \mathbb{N}$ représente le nombre de co-occurrences de termes dans les tweets affichés dans $\mathcal{I}$.

Un graphe $H$ est appelé une *clique pondérée* si toutes les paires de noeuds de $H$ sont connectées par une arête avec le même poids. Étant donné un paramètre $\gamma > 0$, nous définissons une *quasi-clique pondérée* comme suit.

**Definition A.6.2** (Quasi-clique pondérée)**.** Un graphe non orienté pondéré $H = (V(H), E(H), w)$ est une *$\gamma$-quasi-clique pondérée* pour $0 < \gamma \leq 1$ si l'inegalité suivante est vérifiée:

$$\sum_{e \in E(H)} w(e) \geq \gamma \cdot w_{max}(H) \binom{|V(H)|}{2},$$

où $w_{max}(H) = \max_{e \in E(H)}(w(e))$.

Nous définissons la fonction $q_G : V \to (0, 1]$ qui associe à chaque ensemble $S \subseteq V$ la plus grande valeur de $\gamma \in \mathcal{Q}$ telle que le sous-graphe induit par $S$ en $G$ est une $\gamma$-quasi-clique.

La recherche de quasi-cliques est un problème NP-difficile. Pour cette raison nous avons eu recours à l'heuristique suivante pour trouver des quasi-cliques contenant un noeud $v$ et au plus $s$ nodes. L'algorithme commence avec le noeud $v$ et ajoute l'arête avec un poids maximal contenant $v$. On appelle $S$ l'ensemble des noeuds qui ont été ajoutés par l'algorithme à un point donné de l'execution. Si $|S| = s$, l'algorithme s'arrête. Sinon, il ajoute un noeud $x$ dans le voisinage de $S$ en maximisant $q_G(S \cup \{x\})$ à condition que, en ajoutant $x$, le sous-graphe résultant reste une $\gamma$-quasi-clique.

Notre algorithme pour la détection d'événements peut alors être résumé comme suit. La première étape consiste à collecter des tweets en spécifiant comme filtre une liste de termes liés aux événements de crise. Ensuite, les mentions d'lieux dans les tweets sont reconnues et étiquetées. Une liste des $k$ plus grands pics dans les mentions des lieux est calculée, et chaque lieu est ensuite complété par des termes supplémentaires liés au même événement, en trouvant les quasi-cliques dans le graphe de co-occurrences.

**Évaluation.**

Nous avons collecté des tweets sur une période de 14 mois entre novembre 2015 et décembre 2016. Nous utilisons l'API Streaming de Twitter tout en filtrant les tweets afin qu'ils contiennent au moins un terme lié aux événements de crise. À cette fin, nous spécifions dans les paramètres de filtrage de l'API de Twitter le lexique fourni

dans [73], et ne referons que les tweets en langue anglaise. Nous avons obtenu 10M tweets au total, que nous divisons en quatorze ensembles de données (un par mois). Nous utilisons ensuite un identifiant d'entité [84] pour reconnaître et marquer les mentions d'lieux dans les tweets. Les quatre premiers mois de notre ensemble de données servent à évaluer notre approche par rapport à d'autres approches pour la détection d'événements. Une évaluation plus approfondie de notre approche est alors menée sur une période de 14 mois.

L'état de l'art sur la détection des événements est vaste, et est résumé dans [10,48,62,71]. Dans notre évaluation expérimentale, nous avons examiné les approches qui peuvent être utilisées pour une analyse à grande échelle sur une longue période de temps. Les approches supervisées ou semi-supervisées nécessitent une quantité importante de données marquées, car les sujets abordés sur Twitter pourraient changer rapidement avec le temps. Par conséquent, nous nous concentrons sur des approches non supervisées. En particulier, nous nous concentrons sur les approches qui imposent ou permettent d'imposer certaines contraintes sur la taille de la sortie. Nous considérons les trois méthodes suivantes: MABED [45], StaticDens [8, 59], and EDCoW [101]. MABED permet de spécifier une limite sur le nombre de résultats produits par l'algorithme, StaticDens est un algorithme d'exploration de graphes (de la même manière que notre approche), tandis que EDCoW est l'algorithme qui a le mieux performé dans une évaluation récente [100] (mais il n'a pas été comparé aux deux méthodes précédentes).

Nous exécutons les quatre algorithmes sur chacun des quatre premiers mois de notre ensemble de données, et évaluons le top 10 ainsi que les top 20 premiers résultats pour chaque ensemble de données. StaticDens et EDCoW fournissent une mesure de pertinence pour leurs résultats que nous utilisons pour déterminer les meilleurs résultats. Le code de [8] n'est pas disponible, mais nous avons obtenu le code pour [59] qui est analague et que nous utilisons comme approximation. Nous nous référons à un tel algorithme comme StaticDens parce quil peut être vu comme une version statique de [8].

Dans le tableau A.9 et le tableau A.10, nous présentons la précision à 10 et la précision à 20 pour tous les algorithmes. Nous observons que EviDense fonctionne beaucoup mieux que les trois autres algorithmes en termes de précision sur tous les ensembles de données pour les résultats top 10 et top 20.

| Données | EviDense | MABED | StaticDens | EDCoW |
|---|---|---|---|---|
| Novembre 2015 | **0,400** | 0,300 | **0,400** | 0,200 |
| Decembre 2015 | **0,600** | 0,200 | 0,400 | 0,200 |
| Janvier 2016 | **0,700** | 0,300 | 0,300 | 0,300 |
| Fevrier 2016 | **0,800** | 0,400 | 0,200 | 0,000 |

Table A.9: Précision sur les jeux de données pour le top 10. Les meilleurs résultats sont en gras.

| Données | EviDense | MABED | StaticDens | EDCoW |
|---|---|---|---|---|
| Novembre 2015 | **0,350** | 0,250 | 0,250 | 0,150 |
| Decembre 2015 | **0,600** | 0,250 | 0,300 | 0,200 |
| Janvier 2016 | **0,650** | 0,250 | 0,300 | 0,150 |
| Fevrier 2016 | **0,500** | 0,400 | 0,100 | 0,150 |

Table A.10: Précision sur les jeux de données pour le top 20. Les meilleurs résultats sont en gras.

**Analyse à grande échelle et constatations.**

Nous évaluons notre approche sur une période de 14 mois de données Twitter, de novembre 2015 à décembre 2016, ce qui contitue l'une des évaluations les plus approfondies d'un algorithme de détection d'événements. Nous effectuons également une analyse des résultats, qui nous permet d'aboutir à des informations intéressantes sur la couverture des événements de crise dans les médias sociaux.

Nous divisons notre ensemble de données en 14 jeux de données plus petits, chacun correspondant à un mois dans notre calendrier. Pour chaque mois, nous calculons les 10 premiers résultats, ce qui se traduit par 140 événements, qui sont analysés afin d'évaluer la précision de notre méthode. Nous classons ensuite les événements comme suit:

- *catastrophe naturelle* : « tout événement ou force de la nature qui a des conséquences catastrophiques, comme une avalanche, un tremblement de terre, une inondation, un feu de forêt, un ouragan, un éclair, une tornade, un tsunami et une éruption volcanique » [5];

- *conflit armé* : « une incompatibilité contestée qui concerne le gouvernement et/ou le territoire et l'utilisation de la force armée entre deux parties, dont au moins un est le gouvernement d'un état » [6];

- *attaque terroriste* : « une attaque surprise impliquant l'utilisation délibérée de la violence contre les civils afin d'atteindre des objectifs politiques ou religieux »[7];

- *attaque violente*: utilisé ici pour désigner les actions violentes d'un individu qui n'est pas motivé par des gains politiques ou religieux, mais par la haine raciale ou l'instabilité psychologique;

- *catastrophe causée par l'homme* : « un événement désastreux causé directement et principalement par une ou plusieurs actions humaines délibérées ou négligentes. »[8].

[5]www.dictionary.com
[6]www.pcr.uu.se
[7]www.vocabulary.com
[8]www.businessdictionary.com

Nous obtenons les résultats suivants: 82 résultats sur 140 se rapportent à des événements de crise distincts, donnant une précision de $0,585$ pour notre méthode. Sur les 82 événements nous dénombrons 27 attentats terroristes ( 33%), 21 catastrophes naturelles ( 26%), 19 attaques violentes ( 23%), 8 catastrophes causées par l'homme ( 9%), et 7 conflits armés ( 8%). Nous notons qu'un grand nombre d'événements concerne les actions d'un individu ou d'un groupe d'individus contre des civils non armés (attaques terroristes, attaques violentes). Bien que les résultats obtenus au cours de ces mois soient influencés par les événements réels qui se produisent dans le monde entier, nous avons peut-être sous-estimé la fréquence des conflits armés qui peut être dû au fait que nous avons analysé les tweets anglais, alors que les personnnes dans les zones impliquées dans les guerres ne parlent pas anglais .

La plus intéressante découverte porte peut-être sur la couverture du conflit au Yémen. Après avoir inspecté les tweets liés à ces événements, il semble que le fait même qu'un tel conflit n'ait pas fait l'objet d'une attention suffisante par les médias internationaux est le principal motif qui pousse les utilisateurs de Twitter a publier sur le Yémen. En procédent à un examen plus approfondi, nous avons découvert qu'une grande partie de l'activité sur Twitter concernant le Yémen était due à des *tweetstorms*, des événements organisés qui se produisent à une certaine date et heure et qui ont pour but de sensibiliser les utilisateurs sur certains sujets. Dans nos travaux futurs, nous étudierons comment les communautés se forment autour des tweetstorms, à quelle vitesse les nouveaux utilisateurs participent aux tweetstorms, et pour combien de temps ils restent intéressés par le sujet.

Motivés par la richesse du contenu disponible sur les réseaux sociaux et par le besoin d'information en situation de crise, nous avons développé un algorithme de détection d'événements qui a été évalué sur une collection de tweets et comparé aux approches existentes pour ce même problème. Notre évaluation expérimentale montre que notre algorithme dépasse les autres approches en termes de précision. Nous avons fourni l'une des évaluations les plus completes d'un algorithme de détection d'événements, tout en offrant des informations sur la couverture des événements de crise dans les médias sociaux. Notre travail montre que de nombreux événements majeurs sont discutés sur Twitter, y compris des événements qui sont mal couverts par les grands médias traditionnels, comme le conflit au Yémen.

## A.7 Travail futur

Avec le développement d'Internet, une surabondance de données est disponible pour la communauté scientifique. Une grande partie de ces données peut être classée dans des réseaux sociaux et réseaux d'informations, et peuvent être représentés par des graphes. Dans cette thèse, nous avons étudié plusieurs problèmes importants dans l'exploration de graphes et de données. Dans ce qui suit, nous présentons nos perspectives de recherche futures.

TOP $k$ SOUS-GRAPHES DENSE DANS UN GRAPH DYNAMIQUE. En particulier

lorsqu'il s'agit d'un graphe dynamique, le problème de l'entretien des $k$ sous-graphes denses peut être très difficile car toute modification de la structure d'un des sous-graphes denses pourrait affecter la structure des sous-graphes denses restants. Nous avons des résultats préliminaires pour une généralisation de l'algorithme de [34], mais nous ne sommes pas en mesure d'obtenir un temps de mise à jour rapide pour un ajout ou une suppression d'arête. À l'avenir, nous étudierons si une décomposition sensible à la densité [93] peut être maintenue efficacement dans un graphe dynamique. Si cette approche donne de bons résultats, nous aurons également une heuristique pour maintenir le top $k$ sous-graphe dense, sans dépendre du paramètre $k$.

APPLICATION WEB POUR LES MÉDIAS SOCIAUX ET LES MÉDIAS TRADITIONELLES. Nos méthodes peuvent traiter de grands volumes de données produites dans les médias sociaux, mais peuvent aussi extraire les événements des médias traditionnels. La différence la plus importante est la longueur du texte des nouvelles dans les médias traditionnels, mais dans notre travail préliminaire, nous avons constaté que les titres d'actualités ou les résumés obtenus à l'aide de techniques de synthèse extractrice pourraient être utilisés pour réduire la taille du texte et éliminer le contenu moins important. Nous travaillons actuellement sur le développement d'une application Web qui présentera aux utilisateurs les principaux événements dans les médias sociaux et les principaux événements dans les médias traditionnels. Grâce à notre application, nous espérons offrir aux utilisateurs une perspective plus large des événements qui se déroulent dans le monde.

CHRONOLOGIE DES COMMUNAUTÉS DE TWEETSTORMS. Dans notre travail, nous avons rencontré un phénomène très intéressant, les tweetstorms. Les tweetstorms sont des événements organisés qui se produisent à un certain moment et visent à sensibiliser les utilisateurs sur un sujet en profitant de la fonctionnalité tendance de Twitter. Un groupe d'utilisateurs écrit des tweets contenant certains hashtags et mots-clefs qui, en raison de l'utilisation soudaine dans le flux de tweets, seront présentés sur la page Twitter en tant que tendances. Nous projetons à étudier le calendrier des tweetstorms, plus particulièrement, quelle est la corrélation entre les tweetstorms et les événements dans le monde réel, et combien de temps prend-il pour que les utilisateurs s'intéressent à une cause et participent activement aux tweetstorms. Lorsque plusieurs tweetstorms sont organisés pour la même cause, il peut y avoir une variation de la participation des utilisateurs; l'étude de la façon dont la communauté évolue au fil du temps pourrait également donner une idée de la réussite des tweetstorms dans la sensibilisation des utilisateurs.

# Recherche de sous-graphes denses et d'événements dans les médias sociaux

**Oana-Denisa Bălălău**

**RESUME:** La détection d'événements dans les réseaux sociaux consiste à retrouver les traces d'événements réels dans des flux d'informations en ligne. L'ambition de notre travail est double : trouver, dans un premier temps, des événements que les principaux médias ne traitent pas et, dans un second temps, étudier l'intérêt que les utilisateurs ont pour certains types d'événements.

Pour résoudre notre problème nous commençons par une caractérisation des données basée sur un graphe dans lequel les noeuds sont des mots et les arêtes représentent le nombre de cooccurrences. La densité est une très bonne mesure de l'importance et de la cohésion dans les graphes. En prenant en compte les propriétés caractéristiques des réseaux du réel, nous pouvons développer des algorithmes capables de résoudre efficacement des problèmes complexes.

Les contributions de cette thèse sont : concevoir des algorithmes efficaces pour calculer différents types de sous-graphes denses dans des graphes réels, et fournir un algorithme orienté graphe efficace pour la détection d'événements.

**MOTS-CLEFS:** sous-graphes denses, détection d'événements

**ABSTRACT:** Event detection in social media is the task of finding mentions of real-world events in collections of posts. The motivation behind our work is two-folded: first, finding events that are not covered by mainstream media and second, studying the interest that users show for certain types of events.

In order to solve our problem, we start from a graph based characterization of the data in which nodes represent words and edges count word co-occurrences. Density is a very good measure of importance and cohesiveness in graphs. Taking into account the special properties of real-word networks, we can develop algorithms that efficiently solve hard problems.

The contributions of this thesis are: devising efficient algorithms for computing different types of dense subgraphs in real-world graphs, presenting a novel dense subgraph definition and providing an efficient graph-based algorithm for event detection.

**KEY-WORDS:** dense subgraphs, event detection