Pattern Detection in Bipartite Temporal Network

Nam Nguyen Hai
Abstract

Pattern Detection in Bipartite Temporal Network

Nam Nguyen Hai

Pattern detection in social networks has been of great interest recently because it helps to reveal insights about how people communicate. In graph mining, this is referred to as a frequent sub-graphs mining problem and it has many variations regarding the structure of the network - i.e. how much information the graph holds. As a part of the evolution of the problem, additional information such as data dimensions or more advanced network structure are incorporated to the input to find more interesting hidden patterns.

This thesis concerns finding patterns in Twitter data modeled as a bipartite network with an additional temporal dimension. After defining three topology-based types of pattern mathematically, three pattern detection methods are developed, implemented and tested on real-world data collected from Twitter. The results reveal the most popular patterns of each type on Twitter data, a decaying tendency in the replying time as the conversation develops and other interesting observations. They also show that using the difference between two consecutive messages in a pattern (time leap) could be a good alternative time constraint for a time window in pattern detection. Despite being run on a small number of test cases, the results successfully demonstrate the potential of studying chain-like patterns separated from the dominant star-like ones in social networks like Twitter.
Acknowledgements

This master thesis was conducted at Uppsala University Information Laboratory (Infolab). I would like to express my very great appreciation to my supervisor and my reviewer, Davide Vega D’aurelio and Prof. Matteo Magnani, who made everything possible for me to write my master thesis at Infolab, closely followed my progress and contributed to the project with valuable comments. Davide was of great help with the data visualization part of my thesis and he also provided support and encouragement when I was demotivated the most. I cannot describe my gratitude to them highly enough.

My special thanks are given to Amin Kaveh, Khalid Mahmood and other researchers working at building 19, Polacksbacken campus for wonderful discussions regarding not only my thesis project but also my personal development. My grateful thanks are also extended to Johannes Henriksson, who worked closely with me on a parallel project, for his great interest in my project and for giving very useful feedback on my work.

I gratefully acknowledge Anders Berglund, my course coordinator, who offered mental support and encouragement to me throughout the course of this project. I also appreciate the feedback offered by Tuan Phong Ngo, Tuan Anh Dao, Thi Lan Vu and Suraj Murali regarding my report writing as well as the thesis presentation.

Special gratitude goes to my family and friends, who have always encouraged me, shared my excitement, and who have been part of my accomplishments throughout my entire education.

Nam Nguyen Hai

November 28th, 2018

Uppsala, Sweden
Thesis summary

The content of this thesis is divided into 6 chapters as follows.

Chapter 1 goes through basic theories about multilayer network and bipartite network, the data structure most suitable for representing social media data. Basic knowledge about pattern detection and network growth models will also be introduced in this chapter.

Chapter 2 presents an overview of related works, including different representation of Twitter data, existing pattern discovery methods and popular network growth models. The chapter also briefly discusses the practical application of this work.

Chapter 3 starts by introducing the bipartite temporal network structure modeling Twitter data, which is based on the background knowledge presented in chapter 2. Three types of patterns are mathematically defined as well as the methodology to count them in bipartite temporal networks.

Chapter 4 documents the experiment procedure and motivates why certain experiments take place. After that, experimental results are analyzed with respect to patterns found of three different types. Interesting observations will be explained and/or reflected on the theoretical part of the thesis.

Chapter 5 introduces five network growth models that aim to generate networks that not only display patterns found in the given Twitter data but also share common characteristics as that of real-life data such as a long-tail degree distribution. This helps to generalize the understanding on patterns found in Twitter data. The generated networks using those five models will be evaluated in term of how closely they resemble inputted datasets.

Chapter 6 concludes the thesis with a summary of findings. Discussions on the shortcomings and potential future directions to extend the project are also covered in this chapter.
Contents

1 Introduction .................................................. 1
  1.1 Bipartite network ........................................ 1
  1.2 Temporal network ........................................ 3
  1.3 Pattern detection ...................................... 4
  1.4 Network growth models ................................. 5
  1.5 Multinet library ...................................... 6

2 Literature review ........................................... 7
  2.1 Modeling Twitter data .................................. 7
  2.2 Pattern definitions .................................... 9
  2.3 Candidate pattern generation ......................... 11
  2.4 Graph isomorphism .................................. 13
  2.5 Summary .............................................. 15

3 Methodology .................................................. 17
  3.1 Network modeling .................................... 17
  3.2 Definition for patterns ................................ 17
    3.2.1 Chain-like patterns .............................. 18
    3.2.2 Star-like patterns .............................. 23
    3.2.3 Hybrid patterns ................................ 26
  3.3 Pattern detection algorithm ......................... 28
    3.3.1 $P_a$ pattern .................................. 28
    3.3.2 $P_b$ pattern .................................. 32
    3.3.3 $P_c$ pattern .................................. 34

4 Pattern detection experiments ............................... 37
  4.1 Experiment settings .................................... 37
    4.1.1 Data pre-processing ............................. 37
    4.1.2 Input data analysis ............................. 38
    4.1.3 Experiment environment ......................... 40
    4.1.4 Experiments procedures ......................... 41
  4.2 $P_a$ patterns found ................................ 41
    4.2.1 Pattern size w.r.t. time windows ............... 42
    4.2.2 Pattern size w.r.t. time leaps ................. 43
    4.2.3 Pattern order w.r.t. time windows and time leaps 45
    4.2.4 Correlation between patterns’ size and order .... 45
    4.2.5 Correlation between time window and time leap ... 48
    4.2.6 Temporal density of patterns .................... 49
  4.3 $P_b$ patterns found ................................ 51
  4.4 $P_c$ patterns found ................................ 51

5 Network growth models .................................... 55
  5.1 Motivation ............................................ 55
  5.2 Random models ....................................... 55
    5.2.1 Random model 1 ................................ 55
    5.2.2 Random model 2 ................................ 56
    5.2.3 Evaluate random models ......................... 56
  5.3 Generic Barabási–Albert model ...................... 59
  5.4 BA model with replying .............................. 61
5.5 BAR model with decaying reply probability .................................. 62

6 Discussion and future works .................................................. 65
  6.1 Conclusion ................................................................................. 65
  6.2 Shortcomings and improvements ........................................... 66
List of Tables

1. Demonstrate how Twitter handles commenting and retweeting 7
2. Information of 14 datasets including the number of nodes and edges 37
3. The default value of parameters in the three algorithm 41
4. Eighteen values of the time constraint up to one hour 42
5. Thirty values of the time constraint up to two hours 44
6. The value of parameters in two random network growth algorithms 57

List of Figures

1. A multilayer network consisting of four actors in two layers, one interlayer edge and two intralayer edges 2
2. A bipartite network consisting of two actors and two messages (four nodes) 3
3. Three types of conversation graphs modeling the series of interaction in Table 1 8
4. Three types of conversation graphs modeling the series of interaction in Table 1 11
5. Two isomorphic representations of the exact same graph 13
6. Eight directed and isomorphic graphs consisting of 3 actors and 3 edges. Any directed graph of the same size and order is topologically similar to one of these eight graphs 15
7. A network that models the Twitter conversation in Table 1 18
8. Five valid chain-like patterns 19
9. Four invalid chain-like patterns 20
10. Chain-like pattern growth cases 21
11. Network flattening process from bipartite temporal networks to directed temporal networks 24
12. All 2-actor and 3-actor, 3-message patterns. The green background highlights the four 2-actor patterns (bottom left) and the grey background highlights the eight triangles. The 24 other patterns are stars. The first edge in each motif is from the green to the orange node. The second edge is the same along each row, and the third edge is the same along each column [1] 24
13. All 3-actor 2-messages patterns having one message sent simultaneously to two different actors. These patterns are indexed by \( p_{ci} \), \( i \in \{1, \ldots, 6\} \). 25
14. Three types of conversation graphs modeling the series of interaction in Table 1 26
15. Degree distribution in four datasets 38
16. Temporal growth of input datasets 39
17. Temporal density of input data 40
18. The support counts of \( P_a \) patterns aggregated by size w.r.t. different time windows up to one hour display two distinct behaviors 42
19. The support counts of \( P_a \) patterns aggregated by size w.r.t. different time leaps up to one hour display two distinct behaviors 43
20. The support counts of \( P_a \) patterns aggregated by size w.r.t. both time windows and time leaps up to two hour in dataset devices 44
21 The support counts of $P_a$ patterns aggregated by order w.r.t. both time windows and time leaps up to one hour.  
22 The correlation between $P_a$ patterns’ size and order w.r.t. time windows up to one hour in dataset smartdevices.  
23 The correlation between $P_a$ patterns’ size and order w.r.t. time windows up to one hour on dataset devices.  
24 The correlation between $P_a$ patterns’ size and order w.r.t. time windows up to one hour in dataset itsecurity.  
25 The correlation between time windows and time leaps up to one hour on $P_a$ patterns aggregated by size, in dataset itsecurity.  
26 The correlation between time windows and time leaps up to one hour on $P_a$ patterns aggregated by size, in dataset devices.  
27 The temporal density of $P_a$ patterns in two datasets.  
28 The support count of $P_b$ patterns w.r.t. four different time windows in two datasets.  
29 The support counts of $P_c$ patterns w.r.t. time windows up to one hour in four datasets.  
30 The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in two generated datasets Random1 and Random2.  
31 The degree distribution of two generated datasets Random1 and Random2.  
32 (Upper) the correlation between $P_a$ patterns’ size and order w.r.t. an one-hour-long time window and (lower) the degree distribution of the three datasets $BA_{\alpha}$, $BA_{\beta}$ and $BA_{\gamma}$.  
33 The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in the generated dataset BAR$_c$(\(\Delta n = 3\), \(p_{reply} = 0.1\)).  
34 The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in four generated datasets BARD$_{\alpha}$, BARD$_{\beta}$, BARD$_{\gamma}$ and BARD$_c$.  


1 Introduction

This work aims to detect patterns and understand the structure of conversations in input datasets from Twitter, one of the most popular online social network on the Internet. The fact that Twitter publishes APIs to collect data attracts a great interest from computer scientists and sociologists to exploit online users’ behaviors. This introduction will establish the foundation to a complex network structure used to model Twitter data in this thesis and explain why pattern detection is an important and meaningful task.

1.1 Bipartite network

Networks have become a fundamental tool to model communication and interactions in a wide range of systems from the Internet to the nervous system. Anything can be mathematically represented by a graph consisting of vertices and edges - pairwise relations between vertices. This representation allows to study the behaviors and characteristics of the system, e.g. the shortest path from one vertex to another.

In an ordinary network, the connection between two nodes is usually represented by a single, static, directed/undirected, unweighted or weighted edge; although the latter gets less support from network algorithms. In most of the cases, this generic data structure simplifies the realistic system that it models to a high level of abstraction (e.g. an email person $A$ sent to person $B$ could be modeled as a directed edge from node $A$ to node $B$). Many characteristics of real-life networks have been found deploying this simple graph framework, such as the small-world property. However, this simplification contributes to an information loss in the modeling process. In other words, conventional network structure fails to model additional properties of vertices and edges as well as more complex relationship between vertices such as different types of connections or the temporal existence of an edge.

Therefore, simple directed/undirected graphs are extended with additional levels of details to them, resulting in graphs with more distinct and powerful features. In road networks, edges are commonly assigned with a weight representing the length of the road or the traveling cost between two destinations. In spatial networks, the positions of vertices are added to illustrate the geometrical relation between them. Similarly, adding time dimension to edges to represent the time when they are active creates a temporal graph [1]. Network structures are also extended by having multiple edges between two actors (known as multiplex), which could later be flattened into a weighted graph. Ultimately, additional layers of actors could be added to the graph whose edges represents how actors in different layers are connected. For example, friendship on Facebook and Twitter could be modeled as either two separated graphs or a set of two graphs in which one Facebook account in one layer is connected to the Twitter account of the same person on the other layer, which retains much more information compared to that of the former. This multilayer(ed) structure is usually referred to as network of network, or multilayer...
network and was heavily inspired by social networks where one person partakes in multiple social network platforms. The goal of this is to deal with complex social phenomena, which can be classified into the two following categories.

- Heterogeneity of relational ties: There are different relations between the same set of actors. In social network, two persons can, at the same time, be spouses and colleagues.

- Heterogeneity of actors: When modeling Twitter data, users and messages are different types of actors. Both modeling each set of actors in a single simple graph and combining both into one graph do not work. Thus, the need for multilayer network emerges.

**Definition of multilayer network**  Given a set of actors and a set of layers respectively denoted as $A$ and $L$, a multilayer network is $M = (A, L, V, E)$ where $(V, E)$ is a single-layer graph and $V \subseteq A \times L$. Since one actor could present in several layers, vertices are coupled with the layer that they are in to form nodes. For example, Peter having an account on both Facebook and Twitter will be modeled as two connected nodes (Peter, Facebook) and (Peter, Twitter) as illustrated in Figure 1. Regarding the set of edges $E$, there are two types of connections in multilayer network namely intralayer and interlayer connection. The edge between two nodes in the earlier example was an interlayer edge between two layers Facebook and Twitter. Intralayer edges are ones connecting nodes in the same layer, i.e. an edge connecting (Peter, Facebook) and (John, Facebook) could model a friendship between Peter and John on Facebook. This generic data structure is extremely flexible and powerful in modeling networks.

Based on the definition of multilayer network, it is possible to define various more specific types of unordinary network structures depending on ways of separating layers. Intuitively, a typical *multiplex* network is a set of layers in which actors (or vertices) are connected to each other in a fashion referred to as pivots when imagining that layers are placed one of top of another. In such networks, the same actor might appear in different layers, thus needs to be represented altogether with the layer, i.e. actor $a$ intralayer edges within each layer belong to one type of relationship and interlayer edges connect the same actors. The aforementioned
example of friendship on Facebook and Twitter serves well as an illustration for this kind of separation.

Another approach to this is having different types of actors in different layers, forming a multimode (also known as multilevel) network. Under the restriction that only interlayer edges present in the network and not the intralayer ones, the network structure is slightly reduced in complexity and is referred to as \( k \)-partite network, where \( k \) is the number of actor types. Bipartite network is one special and the most popular configuration of \( k \)-partite network. It aims to model networks of two types of actors such as users and messages (tweets in Twitter for example) in social network data modeling, which helps to model the action that "one user sends a tweet to two other people".

**Definition of bipartite network**  A bipartite (also known as two-mode or affiliation) network is an inherited multilayer network \( M = (A, L, V, E) \) where the set of layers \( L \) contains exactly two layers \( l_1, l_2 \) and only interlayer edges are present in the network. There are two types of actors in the actor set \( A \), for example authors with joint publication or people and messages they sent, each type of which are held in one layer \( l_1 \) or \( l_2 \). Since bipartite network focuses on the interaction between two actor types, only edges between two actors of different types are present, which are ones going from one layer to another (interlayer). Because the feature of bipartite network aligns with the feature of Twitter data, this network structure will be employed to model input data in this thesis. Figure 2 pictures a bipartite network modeling a conversation between two actors Peter and John in layer \( l_1 \) via two messages in layer \( l_2 \).

### 1.2 Temporal network

As discussed in the previous part, the additional weight in weighted networks is not sufficient to model important features of data such as spatial positions of vertices and temporal information of vertices and edges. In social network data, which is the concern of this thesis, there is a lot of temporal information that could enrich network information, potentially resulting in more interesting insights upon network analysis. Each event in social network data integrates a timestamp, such as creating a new user, creating a message, sending/receiving a message, a user
following another user, to name a few. This information not only enforce the order of which events happen, e.g. a user following another user after sharing her tweet on Twitter; but also allows to investigate users’ behaviors with regards to a time duration, e.g. how often people reply to messages within 10 minutes after receiving one. Networks with temporal information are referred to as temporal networks, and they have been of great use recently to model social network data such as Twitter data.

There are multiple ways to extend a simple network with time, which by itself could be expressed with different notions including a timestamp, a continuous period of time or discontinuous values of temporal slots in duty-cycle sensor network [2]. The most popular approach is to have temporal attribute associated with each edge in the edge set $E$. An edge $e \subset E$ being created at time $t$ (e.g. a text message) is represented by its temporal weight $t$. Alternatively, if $e$ only exists in a time period, which means it is created and terminated after some time (e.g. a phone call), $e$ has either two temporal attributes $t_1$ and $t_2$ or a pair of timestamps $(t_1, t_2)$. One can also model a vertex being created at some point in time (e.g. creating a Facebook account) by having a temporal weight with it. However, this approach is rarely used and not simultaneously with either of the previous approaches, mainly because this timestamp could be implied from the temporal weight(s) of the earliest out-going edge from the vertex.

Since this thesis deals with social network data, utilizing temporal data in modeling Twitter network is of the author’s interest. One of the major differences between temporal networks and the conventional ones is that edges are not transitive, such that two vertices $a$ and $c$ are not connected if the connection between $a$, $b$ and between $b$, $c$ are not simultaneously active at any time. Although temporal information enriches the network, many traditional measures do not cope well (or undefined) in temporal networks. This explains the lack of network measures for temporal networks as a lot of tweaks, adjustments and redefinitions are required to take place.

1.3 Pattern detection

*Patterns* (also known as *network motifs*) are induced subgraphs that occur in significantly higher frequencies in networks modeling real-life data than in random ones. They uncover the structural design principles of networks and reveal behaviors that are not seen by simple network measures. Accordingly, *pattern detection* is the process of discovering popular subgraphs in large graphs, which usually consists of two following subtasks.

1. Discover subgraphs occurred in the input graph.

2. Classify subgraphs into classes that have identical topology, i.e. there is a mapping from one subgraph to another in the same class and count them. The former step is well-known as graph isomorphism problem [3].
Because of the power to exploit underlying network structures, pattern detection has many applications, mostly in event prediction, recommendation, and social behavior analysis.

In cyber security, mask attack detection helps to find correlation between an attack or an intrusion and system malfunctions prior to the event. The main goal is to find early signs of an attack, detecting attempts to distract defense effort and mask the true move. One of the patterns found was "if a host $u$ is involved in a DDoS attack ($P_1$) at some time, then this host is likely to be the victim of an information exfiltration (attack $P_2$) within two minutes" [4].

In online social networks, based on a popular pattern that two Facebook users are likely to know each other in real-life if they have more than 5 mutual friends, the system could suggest two users this potential friendship.

Another common and intuitive pattern could be a picture posted with a lot of comments without being replied. In a broader context, pattern detection can also contribute to the understanding and prediction of personal behavior. For example, people tend to focus on one conversation at a time with regular texts; but will usually have parallel discussion with many people online [1]. This thesis aims to detect conversational patterns in Twitter modeled as bipartite temporal networks.

### 1.4 Network growth models

Network growth model is to generate a network based on some assumptions. There are a lot of applications for network growth models. This is useful to create realistic data that does not hold real information of people (concerning data privacy). It could also be used to construct random data (null model) to test against real data to see if some features found in real data presents in random data or not. If the answer is yes, then the features found are not significant. Otherwise, if the features seen in input data also exist in the random data, those features do describe the network, but they are not strong features. Some studies do not concern this and thus skip this process.

Another practice of network growth model is to testify assumptions about the data. Upon finding some features of the input network, assumptions could be made and by comparing the newly generated network based on those assumptions with the input network. If the artificially generated networks display the nature of inputted network such as similar degree distribution, one can say the assumptions were correct and vice versa. This last use case of network growth model will be of concern in this thesis project as to testify the understanding of patterns found.
1.5 Multinet library

Multinet library is a graph library written by Uppsala Infolab. The library provides means of analyzing and mining multilayer networks, based on the book *Multilayer Social Networks* [5]. Multinet supports two languages C++ and R, and it is still being developed to maximize the flexibility in terms of network structure and types of data. This thesis utilizes multinet library to model input network and the final implementation of this project will be integrated back to the library.
2 Literature review

As discussed earlier, pattern detection (or pattern discovery) is the process of identifying most popular subgraphs. There have been many studies concerning this topic in Twitter data as well as other types of social networks. The interest in pattern detection has shifted from non-temporal to temporal patterns, meaning that temporal information is becoming a crucial factor in pattern detection. By discussing the literatures, this sub-section will not only explain why that is but also motivate the contribution of this thesis.

2.1 Modeling Twitter data

Twitter data is so vast that one can hardly model it in a way that retains all information, and even if such case holds, the complexity of the network would be tremendous. In fact, many studies chose to filter some specific features of the data such as geometric location, temporal information, etc. and analyze the data based on that to bypass the network modeling process [6]. For example, one can analyze the frequency of tweets posted in different time in a day and on different days in a week, which does not require network modeling. Although data analysis in this fashion is relatively cheap, it is difficult to generalize the results. Other studies modeled networks from Twitter data but reduced the interest to only one aspect of the data, essentially simplifying the network complexity. Following the same fashion, this thesis project concerns the conversational aspect of Twitter data, modeling the data as a conversation graph. Three main ways to define a conversation graph, including mention graph, reply graph and user graph are elaborated below [7].

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Tweet created</th>
</tr>
</thead>
<tbody>
<tr>
<td>A tweets</td>
<td>$M_1$: &quot;What a beautiful day! @B&quot;</td>
</tr>
<tr>
<td>B replies to $M_1$</td>
<td>$M_2$: &quot;@A Nah. Its too hot #Ihatesummer&quot;</td>
</tr>
<tr>
<td>C likes $M_1$</td>
<td></td>
</tr>
<tr>
<td>C retweets $M_1$</td>
<td>$M_3$: &quot;@A What a beautiful day!&quot;</td>
</tr>
<tr>
<td>B replies to $M_3$</td>
<td>$M_4$: &quot;@C Let’s go swimming!&quot;</td>
</tr>
<tr>
<td>C replies to $M_4$</td>
<td>$M_5$: &quot;@B Id love that. What do you think? @A&quot;</td>
</tr>
</tbody>
</table>

Before going into three types of conversation graphs, it is important to quickly go through different relationships and ways of interactions between Twitter users. If a user $A$ follows another user $B$, his tweets would be shown in $A$’s home page; note that this is not the only way for $A$ to see $B$’s tweets. $A$ can also post an original content known as a tweet, which could contain mentions (a reference to another user using @ symbol), hashtags or some kinds of media. Besides tweeting, there are four types of interactions that $A$ could do with one tweet $M$, including liking, replying, re-posting (also known as retweeting) and direct messaging. Amongst those, direct messaging is excluded in the modeling phase because of data privacy. Three other interactions are handled differently by Twitter. Except for liking a tweet, both replying and retweeting to $M$ generate new tweets accordingly with
mention to the creator of the original tweet, in this case, @A. Table 1 illustrates a very simple series of interaction between three users A, B and C and how mentions are utilized in the newly created tweets (except for liking a tweet). At this point, we see that mentions are used very commonly in Twitter, representing any interaction between users, including a deliberate mention, a reply or a retweet. In general, a mention resembles a connection from one tweet to another and essentially from one user to another, which is useful for modeling connections between tweets and/or users.

**Mention graph** The first type of conversation graph called mention graph are constructed based on these *mention* connections, which basically resembles any interaction between users. A mention graph is a directed simple graph \( G = (V, E) \) where \( V \) is the set of users and each edge \( e \subset E \) between two users denotes a tweet from the first user mentioning the second one. For example, A posts a tweet mentioning B will be modeled as a graph of two vertices and one directed edge from A to B. The problem with this data representation is that it has to be multiplex, meaning that there are multiple edges between two vertices as one user can mention another user as many times as he wants. The naive solution is to flatten multiple edges between two vertices into one edge with the total number of mentions being the weight (referred to as communication intensity) [8]. For a higher order of simplicity, the modeled network could be non-weighted or undirected. Because it is extremely straightforward to collect data and construct mention graphs, most related studies employed this approach to model Twitter data [6, 7, 8, 9, 1, 10, 11]. An example of a mention graph is shown in Figure 3a, which is constructed based on the series of interaction in Table 1.

**Reply tree** The second method to model conversation, known as conversation tree or reply tree, concerns only tweets and the relationship between them, instead of users’. The main idea is to model a node as a tweet and edges as the *replying* connection between tweets. Because one tweet replies to only other tweet, the constructed graph is essentially a tree. Starting from an original tweet being the root, a conversation tree contains downstream connected components such that each node is a reply to its parent node. To avoid the tree to grow exponentially, a temporal constraint is enforced such that every reply is within \( \delta t \) seconds from the root. The reply trees constructed from the series of interaction in Table 1 are shown in Figure 3b, where there are two conversations started from \( M_1 \) and \( M_3 \). It is seen that the mentions only play a partial role in the construction of reply trees.
This network structure is very similar to that of cascades in blog graphs modeling [12, 13]. The difference is that edges in cascades represent referencing, not replying relationship thus one node representing a blog could have multiple parents. Even though reply trees describe more accurately the conversational aspect of Twitter, the construction of such trees is expensive and requires using Twitter APIs, which enforce a constraint on the data crawling speed [7]. Another apparent weakness of reply trees is that they only resemble the relationship between tweets but not between users. Due to all these weaknesses, this definition of conversation graph is not commonly used.

User graph The third approach to model conversational Twitter network, user graph, is a middle ground between the first two. Similar to reply trees, user graphs model exclusive replies but focus on the relationship using a network structure alike to that of mention graphs. To be specific, a user graph is a graph $G = (V, E)$ where $V$ is the set of users and an edge $e \in E$ represents an explicit reply between two users. This could be thought of as a projection from reply trees such that an edge connecting two tweets in a reply tree becomes one connecting two users in a user graph. Figure 3c illustrates a user graph constructed from the interactions in Table 3c. Compared to the mention graph in Figure 3a, the user graph does not have the edge from C to A, because tweet $M_3$ is not a reply but a retweet to $M_1$. Although this definition overcame one of the weaknesses that reply trees have, the construction of such networks is still time-consuming due to the limitation of Twitter APIs. Therefore, both reply trees and user graphs are not used as commonly as mention graphs.

As it is seen, out of three ways to model conversations in Twitter, using mention graphs is not most popular approach due to its simplicity. However, pattern detections using simple graphs representing mention network structure mostly confirmed some properties of networks such as the small-world phenomenon, fat-tailed distribution and the intuitive nature of social network: bursty and causal [6, 9]. Pattern detection in simple graph seems to have hit a wall. The reason could lie in the two uncommon representation of conversation graphs. As discussed earlier, both reply trees and user graphs study how the conversation evolves but from different aspects: information flows and human interaction. They were always studied separately. Regarding the Twitter modeling process, this thesis aims at combining the simplicity of mention graph with multilayer network structure to represent both users and tweets at the same time. By doing so, both aspects of Twitter conversation can be considered simultaneously, and the network becomes more expressive. In addition, using mention graphs allows the use of naturally collected data from APIs, avoiding the excessive cost for collecting Twitter replies using the reply tree crawler [7].

2.2 Pattern definitions

Since a pattern is defined as a subgraph, there are many topological variations to it. Furthermore, one can add more constraints to the definition of patterns that he/she is looking for, such as textual content of tweets, temporal bound/order, maximality and closedness of patterns, or even the relative relation
between patterns. The following section elaborates different definitions of network patterns.

In conversation graphs concerning the information flow on Twitter (i.e. tweets only) and other social networks such as reply trees and cascade trees, each conversation tree is usually regarded as a pattern and accounts for one appearance of that pattern. The advantage of tree structures over graphs when modeling subgraphs in pattern detection is that tree isomorphism can achieve linear time complexity using prefix and/or suffix trees [1]. While analyzing general social networks (e.g. Facebook, blogs) with tree patterns revealed interesting results about information flow [13, 12, 10], very little insights about the interactions between users was found in Twitter, of which the burstiness feature (i.e. star-like patterns) was the highlight [7]. This downside could easily be overcome by modeling nodes as users instead of tweets; however, the interaction that one user replies to a tweet sent from another user will create a cycle, thus cannot be modeled using trees.

As for graph presentation of Twitter conversation (e.g. mention graphs and user graphs), the simplest and possibly the most general pattern family is defined as 3-node 3-edge directed or undirected graphs and without any special constraints to it [6]. Thanks to its simplicity, this definition of pattern could be applied to any network. When used to analyze Twitter data, it revealed some expected insights such as the probability for a tweet to be replied decreases through time and interactions has low reciprocity (meaning that few tweets get replied).

As the topic matures, the temporal information is also integrated in the patterns. In the beginning, many studies attempted to find temporal patterns (equivalent to rules, not to be confused with ”network patterns” defined as frequent subgraphs). For example, the frequency of tweets sent at different time in a day or on days of the week are exploited [6, 8, 14]. Although these results are merely surface, they illustrated a great potential from temporally enriched data, especially in combination with Twitter APIs’ ability to collect timestamps associated with every interaction of users. This motivated a stronger integration of the temporal dimension to patterns such that the whole pattern should happen within a time window, in other words, being timely bounded. Patterns in this category are defined as connected directed graphs with pairwise $\Delta t$-connected edges, making them connected not only topologically but also temporally [9]. This notion is heavily employed in many studies in the field, of which the results agreed to the domination of star-like pattern [7, 1].

Another interesting finding was that interactions on Twitter and other social networks has a high order of causality. This means that there is an influence upon initiating connection between users. Figure 4 illustrates communication influenced by either a mutual prior communication with another user in Figure 4a or through chain communication in Figure 4b. A study on a large mobile call network found the aforementioned patterns significantly dominating two other patterns that show no causality in Figure 4c and 4d [9].
To exploit more specific aspect of Twitter, slightly more topologically complex and less general patterns such as closed-chain graphs are formulated to show collaborate behaviors between Twitter users. However, these patterns have low frequency counts except for the stars of size 2 and 3, suggesting that Twitter collaboration is limited to small-sized flat networks [15].

Patterns can also be expressed not using graphs but association rules. Graph temporal association rules are proposed as sets of temporal events conveying meaningful rules. For instance, one of the real-world association rules found in the “Panama” dataset stated as follows.

Companies with jurisdiction area “BVI” (British Virgin Islands) that were once inactive become active again with a changed jurisdiction “Panama” in 4 years ($< 1500$ days) [4].

Despite the great potential of this method in finding specific rules with high confidence, the method is extremely data-sensitive, meaning that two datasets from the same domain might reveal conflicting rules. Due to this downside, this definition of patterns is of no interest in this thesis.

2.3 Candidate pattern generation

This task is the first step in the pattern detection process. The goal of this step is to discover subgraphs in the input network, which includes two smaller subtasks: traversing the search space and generating candidate subgraphs. The first subtask reflects the completeness of the search (exact search or inexact search) and it directly implies a NP-hard problem of enumerating subgraphs in an input graph [16]. Full enumeration is the intuitive and most common solution for this step because it does not introduce any parameters, meaning that it is not parameter-sensitive and provides deterministic results. In other words, pattern discovery using full enumeration always yield the same output set of patterns with the same order for one input dataset. However, using exact search with full enumeration is time
consuming, and is not desirable when working with large datasets. Inexact search methods using subgraph sampling resolves this drawback; but they introduce sampling parameters, which was proved to bear some bias, leading to overcounting and undercounting of some patterns.

As for the second subtask, the idea is to generate subgraphs systematically of increasing size from the smallest pattern of size 1 (e.g. one node and no edges) until the pattern size limit is reached. There are many ways to implement this subtask [17].

- Level-wise join: Joining two frequent subgraphs of size $k$ to form a size-$(k+1)$ subgraph. Not every two size-$k$ subgraphs can be merged together, but only the two sharing the same core subgraph of size $(k-1)$. Since a subgraph $s$ of size $k$ could have as many as $k$ subgraphs of size $(k-1)$ by removing one vertex $v$ from $s$ ($v \subset s$), level-wise join generates too many redundant candidates. The solution to this is to pick only distinct size-$(k-1)$ subgraphs, which is essentially the graph isomorphism problem later discussed in chapter 2.4.

- Rightmost path expansion: In this candidate generation strategy, the rightmost path is defined as the path from root to the rightmost leaf. Given a size-$k$ subtree, one new vertex will be added as a right child of any vertex on its rightmost path to form a subtree of size $(k+1)$. By only extending a subgraph to its right-hand side, this strategy overcomes the significant drawback of redundant subgraph generation. In other words, it prevents discovering the same subgraph multiple times, which leads to duplication in candidate generation phase.

- Extension and join: Adding a new vertex to a subtree $s$ rooted from one vertex $v$ in a tree representing a pattern and joining $s$ with one of the siblings of $v$ to create a new tree with lower height.

- Equivalence class-based extension: Two subtrees of size $k$ in the same equivalence class are joined together to generate a $(k+1)$-subtree. An equivalence class is defined by its prefix encoding and the list of its vertices.

- Right-and-left tree join: Two subtrees of size $k$ are joined if they share the same topology when removing the leftmost and rightmost vertices respectively. In mathematically terms, two subtrees $s_1$ and $s_2$ will be merged together if removing the leftmost vertex $v_{left}$ of $s_1$ and the rightmost vertex $v_{right}$ of $s_2$ produce two topologically similar subtrees, i.e. there exists a mapping between two new subtrees $\exists f : (s_1 \setminus v_{left}) \mapsto (s_2 \setminus v_{right})$.

The spirit of all these joining techniques is to produce a new subgraph that differs from its parent(s) only one vertex. While rightmost path expansion uses only one parent, other techniques use two parent subgraphs that are topologically close enough (e.g. same equivalence class). In such fashion, these methods can be classified into two groups: Apriori-based and pattern growth-based, inspired by algorithms used in association rule mining, Apriori and FP-growth respectively [17, 16].
Apriori-based algorithms explore the input graph using breadth-first search traversal strategy, meaning that they have to finish counting the support of all size-\( k \) subgraphs before proceeding to that of size-(\( k+1 \)) subgraphs. This is referred to as generate-and-test strategy and is necessary because the subgraph merging process is done with two parents of the same size. Since the number of subgraphs grow exponentially with regards to their size, Apriori-based approaches usually use up a lot of memory for storing the subgraphs, but they allow to safely prune non-frequent candidate subgraphs of size \( k-1 \) before generating size-\( k \) candidates, increasing the performance significantly. Meanwhile, pattern growth-based algorithms traverse the network using depth-first search to recursively discover all super-graphs of one subgraph by adding vertices one by one, resulting in a disadvantage of weak pruning compared to Apriori-based approaches. In exchange, pattern growth-based approaches consume much less memory and is more suitable for datasets of large size. Level-wise join and rightmost path expansion are the most popular method for Apriori-based and pattern growth-based approaches respectively.

2.4 Graph isomorphism

After the first step, a number of candidate subgraphs are generated. At this point, counting the occurrence of these subgraphs essentially involves determining which subgraphs are topologically identical to each other, classifying them into buckets of topologically distinct subgraphs and simply counting their occurrence as the size of the buckets. The key challenge of this step is the classification mentioned above, which is commonly known as graph isomorphism or exact graph matching. In graph theory, graph isomorphism is the problem of finding a bijection between two graphs \( g_1 \) and \( g_2 \) such that they share the same structure or topology. This is a computationally expensive task and is not known to be P or NP-complete, which suggests its NP-intermediate complexity [17]. Figure 5 illustrates two representations of one graph that look nothing similar to each other.

For a general graph isomorphism problem, the theoretical lower bound is \( 2^{O(\sqrt{n \log n})} \) for graphs with \( n \) vertices [16]. Some important special cases of the graph isomor-
phism problem have been efficiently solved in polynomial runtime such as trees and permutation graphs. In fact, the best solution for subtree isomorphism has a complexity of $O\left(\frac{k^{1.5} \log k}{n}\right)$ and much research effort was put on how to efficiently generate subgraph candidates in form of trees [16]. This is the reason why tree representation of pattern is dominant in pattern detection methods. Other more challenging categories of graph isomorphism correspond to other representations of graphs that have additional properties or restrictions, including labeled graph, finite automata and multigraphs.

The general idea of graph isomorphism is to generate an unique code for each graph such that two isomorphic graphs will share the same code, which is referred to as labeling. For instance, a graph isomorphism algorithm will generate the same label for two graphs in Figure 5 even if the second consists of different vertices. For inputs of small size, e.g. size-3 directed graphs, the isomorphism problem can be seen as mapping process from all size-3 subgraphs in the given network to one of the eight pre-calculated isomorphic size-3 graphs illustrated in Figure 6.

Given two subgraphs $s_1, s_2$ mapped to two codes $c_1, c_2$; a surjective mapping function can guarantee that $s_1$ and $s_2$ are non-isomorphic if $c_1$ is different to $c_2$, but the opposite case when $c_1$ and $c_2$ are similar does not hold. The best mapping function is a bijective one, which guarantees the opposite case mentioned earlier also holds. However, a bijective mapping has much higher computational cost compared to that of a surjective mapping function. A wide variety of mapping functions have been introduced to fit different requirements of the graph isomorphism problem.

The most popular mapping is canonical labeling and it is a bijective map. The idea behind it is to flatten the adjacent matrix of a graph into a string, or a code which might later be shortened or trimmed to optimize the performance. The downside of canonical labeling is its expensive cost. On top of that, adjacent matrix does not support cycles in the graphs, making canonical labeling most suitable for acyclic graphs.

The other end of the spectrum is graph signatures, whose idea is to encode a graph by its essential statistics. For example, one can use a combination of the number of nodes, edges, in-degree and out-degree as an effective graph signature [14]. Since this is a surjective map, two non-isomorphic graphs could share the same signature. Therefore, this is usually used as a coarse-grained isomorphism check on big graphs to reduce cost.

Lying in between two extremes there are various methods to tackle graph isomorphism with different complexity levels and costs such as suffix and prefix trees, hashing, edge coloring, or a combination of different methods [1, 9, 14]. This thesis will introduce a labeling method that is inspired by both canonical labeling and signature by encoding only important information in the graph.
Figure 6: Eight directed and isomorphic graphs consisting of 3 actors and 3 edges. Any directed graph of the same size and order is topologically similar to one of these eight graphs.

2.5 Summary

Pattern detection is a computationally-heavy task involving enumeration through the network multiple times. Therefore, various methods to lower the problem’s complexity by constraining patterns or not focusing on every pattern but only some distinctive forms. Counting only cliques, networks where everyone is connected to everyone else, is computationally cheap (due to the possibility to solve clique isomorphism quickly) and produces a relatively small number of patterns in the network [17]. Maximal and closed patterns are introduced to limit the number of patterns in Apriori-based pattern detection methods. A frequent pattern is maximal if none of its super-pattern is frequent, and closed if its super-patterns all have lower support [9, 14, 18]. Finding only edge-disjoint patterns or non-overlapping patterns is another way to lower the complexity, even though this method could undercount some patterns [19, 20, 16]. Finally, sampling is an effective technique to not fully enumerate the network but still provide close enough approximated counts of patterns. The only problem with sampling is that it is parameter-sensitive and could be bias to some families of patterns while undercount the rest [20, 21].

Ultimately, there are tools and libraries implementing these pattern detection algorithms to help scientists not in computer science field to utilize its ability. Two most general-purpose libraries supporting this are MFINDER [22], FANMOD [23, 21]. While they both provide exact and approximate count of patterns, the first one’s sampling algorithm over-samples triangles in the network and the second one is claimed to have fixed that. Being written for general-purpose usage, these two libraries only support simple directed/undirected graphs without weights or
any other data attributes. In contrast, both MFINDER and FANMOD can handle very large graphs and can find all topological patterns up to an order of 8, thus are used widely by biologists to find patterns in protein networks [24]. Besides MFINDER and FANMOD which solely focus on pattern detections, some other graph libraries also incorporate pattern detection as one of their features. SNAP is one of those libraries and it provides an efficient pattern detection for size-3 patterns in directed network [1].

In conclusion, pattern detection is somewhat a matured topic in data mining, in a sense that there are well-developed tools and libraries to detect patterns as long as one can model his/her data in a simple graph. There are many ways to define a pattern, and studies have found some insights about Twitter conversation that aligned with the expected nature of online social networks. In general, the conclusion drawn about Twitter data and that of other online social networks are summarized below.

- Low reciprocity: Sent tweets have a low probability of being replied, which decreases through time with different decay factors in different types of social networks.

- High burstiness: One person sends tweets to multiple people in a short period of time. Combined with the previous feature, one can say with high confidence that it is common to send out a lot of tweets, very few of which is replied.

- Causality: This relates mainly to the order of events in patterns. Time-inconsistent patterns appears with very low support compared to the time-consistent ones. For example, the chain of events “one person talks to a friend and his friend talks to another friend of him” is more popular than the same one with reversed order.

However, there are weaknesses in existing methods that motivate this thesis project. First, according to earlier discussions about which patterns have been found, the role of temporal dimension has not been fully exploited and further exploration was explicitly suggested in some studies [7, 15]. Second, all definitions of patterns failed to capture both aspects of Twitter conversation, information flows and users’ behaviors, respectively represented by tweets and users. Third, the agreement in results of many studies in Twitter pattern detection that patterns of small sizes mostly showed bursty behaviors, which suggests that star-like patterns overshadow patterns of other forms and they could be filtered out to reveal other frequent but not dominant structures. To overcome these drawbacks, this thesis aims to model both users and tweets on Twitter in a multilayer network and find patterns with focus on further exploitation of the temporal dimension.
3 Methodology

3.1 Network modeling

This section will discuss several options for modeling Twitter data and motivates the author’s choice of network structure. It is important that the network model used can not only encode necessary information that the pattern detection algorithms need but also accommodate future extensions to the definitions of patterns. In general, the communication patterns of the author’s interest hold information about tweets sent by users with the time they were sent. In the future, the content of the tweets could be considered in the patterns, for example the sentimental changes in Twitter conversations.

As discussed in section 2.1, this thesis aims at modeling Twitter data as a mention graph with multilayer network structure to represent both users and tweets at the same time to maximize the expressiveness. To be specific, the input Twitter data will be modelled as a bipartite temporal network, which is a bipartite network that accommodate temporal information. Both of bipartite network and temporal network are explained in section 1.1. Hence, input Twitter data will be modelled as follows.

**Bipartite temporal network** A bipartite temporal network is a multilayer network \( M = (A, L, V, E) \) where \( A, L, V \) and \( E \) represent the set of actors, layers, vertices and edges. There are only two layers in a bipartite temporal network \( (L = \{l_1, l_2\}) \) and edges can only go from one layer to another (interlayer edges). There are two types of actors in the actor set \( A \), human actors and messages representing Twitter users and tweets, the former is in \( l_1 \) and the latter is in \( l_2 \). Each actor and the layer it stays on make up one vertex in the set of vertices \( V \). Consider \( v_a = (a, l_1) \) and \( v_m = (m, l_2) \), an edge going from \( v_a \) to \( v_m \) represents actor \( a \) writing message \( m \). The set \( E \) stores all the edges in the network. Each edge \( e \in E \) associates with a temporal attribute \( t_e \) representing the timestamp where that edge takes place denoted as \( t_e: \) edge \( e \) connecting \( v_a \) and \( v_m \) means that \( a \) tweets \( m \) at time \( t_e \). The notion ”a tweet (or message) \( m \) is sent at time \( t \)” will later be referred to as ”an edge \( e \) happens at time \( t \)” . In fact, by utilizing multinet library, each edge is supported with an arbitrary number of attributes, making it possible to extend the network with additional attributes besides time such as a text attribute containing textual content of the tweets. Because of its versatility and extendibility, this network structure will be used to model input Twitter data in this project and potentially future works. Figure 7 illustrates a network modeled from the Twitter conversation in Table 1.

3.2 Definition for patterns

This sub-section defines the patterns that are of interest in this thesis, which severs as the prerequisite for the pattern detection algorithms presented in the next section.
As discussed in section 2, social network data, especially Twitter data, usually has high burstiness, low reciprocity and illustrates some signs of causality. The common weakness in existing pattern detection methods done in this type of data is that they look at both star-like and chain-like patterns, of which the former is much more dominant than the latter. Because of this, chain-like patterns that reveal information flow and homophily behaviors are overlooked. This thesis project aims to study these two types of patterns separately to obtain more in-depth understanding about communication patterns on Twitter.

3.2.1 Chain-like patterns

A chain-like pattern is defined as a timely connected subgraph of the input network introduced in sub-section 3.1. Because it is a subgraph of the input network, it uses the same network architecture as that of the original network: a bipartite temporal network. Denote a pattern (graph) \( g \), which consists of \( N \) actors and \( M \) messages stored in two lists. There is one root actor (or the first actor) denoted \( n_0 \) that initiates the chain and each connection in the chain is done by the actor \( n_i \) sending a message \( m_k \) to another actor \( n_j \) (\( k < M, i \neq j \) and \( i, j < N \)). While the list of actors is ordered by the order of their first appearance in the pattern, the list of messages is ordered ascendingly in time (\( \forall 0 < k < l < M : m_k < m_l \)); this will be elaborated later in the pattern growth part. The smallest chain-like is a subgraph \( g \) containing two actors \( n_0, n_1 \) and one message \( m_0 \) that is sent from \( n_0 \) to \( n_1 \), which effectively represents an edge. This allows the subgraph to grow, such that new actors and messages could be added to the pattern as long as there is a connection (a tweet sent) from the last chain to the new actor. As the chain grows, the new connection has to happen later in time compared to all existing edges. The notion of being ”timely connected” (or temporally connected) means that between any node in the graph and the root, there exists a path consisting of edges of increasing timestamps. Some example of valid and invalid chain-like patterns are illustrated in Figure 8 and Figure 9.
There are some terminologies that will be used extensively in the rest of this thesis report regarding some features of these chain-like patterns.

- The size of a pattern represents the number of connections (or messages) to go from the first actor to the last actor in that pattern, effectively showing how long the pattern lasts topologically. For example, two patterns in Figure 8e and 8d both have 5 messages, which makes them greater in size than the other two patterns in Figure 8a whose size is 2.

- The order of a pattern is regarded as the number of actors partaking in that pattern. For example, Figure 8 shows both pattern 8e and 8b having an order of 2 because there are only 2 actors involved in the loop.

- The temporal relationship between two messages \( m_i < m_j \) denotes \( m_i \) being sent before \( m_j \). Similarly, \( m_i > m_j \) means the opposite temporal order between them and \( m_i = m_j \) means two messages are sent at the same time.

- Another expression that will be used extensively is a “connection between two actors \( a, b \) at time \( t \)”, in which the connection is regarded as a tweet (message) sent from \( a \) to \( b \) at a timestamp \( t \).

Notice that a chain could also go back to the root actor, effectively creating a closed-chained, or loop-like, pattern which represents information going back to where it originates from. Figure 8c illustrates a size-3 order-3 loop that ends at actor \( A \) and could potentially grow bigger from \( A \).
Figure 9: Four invalid chain-like patterns

According to the previous definition, chain-like patterns have some distinct properties and Figure 9 showcases some chain-like patterns that violate these properties. Pattern 9a is smaller in order compared to the smallest chain-like pattern according to the definition of chain-like patterns. In pattern 9b, there is no temporally-connected paths going from the root $A$ to actor $C$, making it temporally inconsistent. The other two patterns 9d and 9c are invalid because they violate the chain-like continuity as the chain branches out at $A$ for the former and $B$ for the latter.

The motivation for defining patterns in this fashion is that these patterns are extendable. New actors and messages can be added to a pattern to increase its size and order as long as the previously formulated definition is not violated. Patterns of small sizes have been the focus in existing works because it is simpler to find them and the cost for graph isomorphism algorithm is cheap, but they do not reveal much insights as discussed in section 2. In this project, the chain-like patterns are designated in a way that they can model subgraphs of large sizes and orders while allowing an effective and low-cost isomorphism detection algorithm.

**Pattern growth**  Given a pattern $g$ consisting of a list of $N$ actors and another list of $M$ messages, such that actor $n_0$ initiated the chain and the chain ended at an actor $n_i$, $i < N$. There are two scenarios that $g$ can be extended are as follow.

1. Adding a new connection to an existing actor: A new connection at time $t$ from the last actor $n_i$ to another actor $n_k$ that already exists in the list of actors in $g$ ($i \neq j$ and $i, j < N$) can be added to $g$ if the new connection happens no earlier than the last message in the pattern: $m_t \geq m_{M-1}$. The list of messages in $g$ will then be extended with one more message, $m_t$, which becomes the last message in $g$ from that point on and every new connection added to $g$ needs to happen no earlier than $t$. This extension increases only the size of the pattern as $g$ now has $N$ actors and $M + 1$ messages.

2. Adding a new connection to a new actor: In a similar scenario as the previous case except that $n_k$ is not in the list of actors in $g$, which requires adding $n_k$ to the list. This extension increases both the size and the order of the pattern as $g$ now has $N + 1$ actors and $M + 1$ messages.
Figure 10 illustrates the growth of chain-like patterns. Given a pattern in Figure 13d, adding a new message to an existing actor results in pattern 13e and adding a new connection to a new actor gives pattern 13f.

In theory, a pattern could grow as big as it covers all the actors and messages in the input network. In other words, the whole input network is the biggest pattern, given that it is fully connected. However, large patterns have obscure meaning and are not of interest in this project. There are three constraints to impose on the patterns to limit how large they can be.

1. Size constraint: there is a maximum number of edges that a pattern can have.

2. Order constraint: there is a maximum number of actors that a pattern can have.

3. Time constraint: there are two ways to impose a time constraint on a pattern.
   
   (a) The whole pattern exist within a time interval, meaning that the difference between the earliest and the latest messages in the pattern is smaller than a given time duration. This duration is referred to as a time window of a pattern, and this constraint is referred to as a time window constraint.

   (b) Each edge in the pattern happens within a time interval from the latest edge prior to that. This means that the difference between every two consecutive edges in the pattern is smaller than a given time duration. This duration is referred to as a time leap. Notice that a pattern with time leap constraint imposed does not bear under any time window constraint - the time window of the pattern can grow very large as long as the time leap constraint is satisfied.

Although the time window constraint has been commonly applied on pattern detection in existing works, the time leap constraint is a more natural and flexible constraint for conversations in real life. In a conversation where two people are talking back and forth, the assumption that the conversation stops after exceeding a certain time window is not always true. This could be overcome using time leap constraint. On the other hand, there are various factors deciding the replying time of messages depending on the context of the conversation or how long the conversation has been [17]. For example, my friend reply to my first tweet after
five minutes but it might take her two minutes to reply my question following up her reply. If she is not replying to my fourth follow up tweet within one minute, it might be safe to call the end for our conversation. Using a dynamic time leap constraint that changes depending on the context and the length of conversation could provide more accurate assessment to whether a conversation has stopped or not. There are a lot of possibilities for time leap constraints, but this thesis project will only examine the fixed time leap constraint. Both of these two time constraints will be put in the test in the experiments in section 4.

Graph isomorphism Since patterns can grow in size and order, the graph isomorphism method needs to also accommodate this growth. A method similar in spirit to many other graph isomorphism algorithms presented in section 2.4 such as canonical labels or signatures of graphs is utilized. Patterns will be encoded to a string in a way that two patterns are only topologically equivalent if they share the same code. This method bypasses the expensive computational cost for comparing two patterns vertex-by-vertex.

Given the pattern in form of a graph $g$ consisting of $N$ actors and $M$ messages, $g$ will be encoded to a string $s_g$ of length $M \times 2$ that can reconstruct the original graph. In that string, each character denotes the actor of corresponding index in the actor list, and every two consecutive characters, say $i$ and $j$, form a pair representing a connection from $n_i$ to $n_j$ via $m_k$. The first pair of characters is always "01" since the first connection is always from $n_0$ to $n_1$ via $m_0$. Figure 8 shows the code of some valid chain-like patterns. As the pattern grows, adding a new connection by either of the two pattern growth scenarios have the code extended to the end. Given a pattern with a code "01...ij" of length $M \times 2$ ($i \neq j$ and $i,j < N$), an extension to the pattern has the code extended as follow.

1. Adding a new connection to an existing actor: The new code will be appended with "jk" ($j \neq k$ and $j,k < N$), making a string of length $(M+1)\times 2$. For example, Figure 10 illustrates the pattern 0110 be extended to 011001 in this fashion.

2. Adding a new connection to a new actor: In a similar scenario as the previous case except that $n_k$ is not in the list of actors in $g$, $n_k$ is then added to the list. This extension increases both the size and the order of the pattern as $g$ now has $N+1$ actors and $M+1$ messages. The result is also a string of length $(M+1) \times 2$. For instance, Figure 13f shows pattern 011002 grown from pattern 0110 in 13d.

The temporal order of messages in $g$ decides the order of all the pairs of two consecutive characters in $s_g$, such the first pair of characters denotes a connection that happens no later than that denoted by the second pair. Because of this, reconstructing the $g$ from $s_g$ is as simple as going through every pair of two consecutive characters and extending a graph with a new edge and possibly a new node that each pair represents. It is worth noticing that this encoding scheme abstracts away the labels of the nodes and keeps only the core topology of the graph; in which, the temporal difference between edges are not accounted for but only the temporal order between them is imposed.
The significant benefit of this encoding is that two graphs that share the same code are topologically similar. In contrast, if the codes of two graphs are different, they are not isomorphic. For example, graphs $X$ and $Y$ have the same code and are isomorphic even though their topology may appear vastly different from each other such as two graphs in Figure 5. In other words, this encoding is a bijective map. Although this is not supported by mathematical proofs, the author has not encountered any case where two non-isomorphic graphs share the same code.

To sum up this part, a chain-like pattern is defined as a connected temporal bipartite network whose edges are temporally connected. As for solving graph isomorphism, subgraphs are encoded to strings of size of double the number of edges in them and comparing these codes reveals if two subgraphs are topologically similar, meaning that they belong to one pattern. For example, two graphs $g_1$ and $g_2$ belong to the same pattern $p$ if $s_{g_1}$ is similar to $s_{g_2}$. According to the two steps of the pattern detection process, this section establishes the foundation to solve the second step with the encoding of patterns. The rest of the process include enumerating through all the possible chain-like patterns, encoding them to strings that correspond to isomorphic patterns and counting the number of times those patterns appear in the network. The pattern detection process will be elaborated in section 3.3.1.

### 3.2.2 Star-like patterns

Similarly to chain-like patterns, star-like ones are also defined as a subgraph of the input temporal bipartite network, in which edges are temporally connected. As elaborated in section 2, existing work on detecting star-like patterns has come to the conclusion about the burstiness nature of social networks. Therefore, the main motivation for studying star-like patterns in this project is to verify the burstiness characteristic of the input Twitter networks. For that purpose, all small patterns of order and size up to 3 (consisting of up to 3 actors and 3 messages) are chosen to study because this allows to utilize SNAP library to quickly enumerate the number of patterns of each type and determining whether star-like patterns dominate as seen in related works [1]. There are two types of pattern families that are of interest.

The first type is all 2-actor and 3-actor 3-message patterns, some of which are star-like while the rest is not. The role of non-star-like patterns is to serve as an anchor to assess the popularity of star-like patterns, which helps to validate the burstiness characteristic of social network as discussed earlier. Since SNAP library supports counting these patterns in traditional directed graphs, the task is now minimized to transforming the input from a temporal bipartite network to a traditional directed network.
Network flattening is the process of simplifying a complex network to a directed one by keeping only the essential information. In this case, the information being kept include actors and the connections between them with sending timestamp. Figure 11 illustrates two 3-actor 3-message patterns and their flattened representation. There are exhaustively 36 patterns that consist of up to 3 actors and 3 messages and Figure 12 shows the flattened representation of those 36 patterns indexed by $M_{x,y}$ with $x, y \in \{1, ..., 6\}$ (6 rows and 6 columns).
Unlike in chain-like patterns where the focus is on patterns of big scale, when it comes to patterns of small scale like these, edges that happen simultaneously cannot be overlooked. It is impossible to model this kind of behavior using single-layer network. For instance, to model a tweet that makes up the edge between two actors $a$ and $b$ that is also sent to $c$, simply adding the same timestamp to two edges is not sufficient to tell $b$ and $c$ receive the same tweet at the same time by $a$. Parameters need to be added to two edges from $a$ to $b$ and from $a$ to $c$ to indicate that they are sent simultaneously by one person, and those parameters add an extra layer to the network. On the other hand, bipartite network naturally supports this kind of behavior - $a$ sends a message $m$ and there are two edges connecting $m$ to both $b$ and $c$. The network architecture makes it possible to study the relationship between edges that happen at the same time and other messages.

To be specific, the question is if there is prior communication between two people receiving the same message before or after that message. Unfortunately, this cannot be done using the existing method applied in directed network [1] because directed network cannot properly represent a three-way relationship between three actors. The algorithm would put two simultaneous edges at an arbitrary order - one has to go before the other. Therefore, this analysis has to be done on the input data in form of a temporal bipartite network and there are 6 patterns of this kind that has 3 actors and 2 messages as shown in Figure 13. However, a problem appears: graph isomorphism still needs to take place and the encoding of patterns cannot account for edges happening at the same time as discussed in the previous section. The simplest solution is to extend the edge representation from a pair to a triplet of characters, which lengthens the patterns’ code from 6 to 9 characters. Since the patterns are of fixed size and order, this extension to the encoding scheme does not scarify performance and will be selected for star-like
Figure 14: Three types of conversation graphs modeling the series of interaction in Table 1.

patterns with edges that happen simultaneously.

The encoding of star-like patterns that involves multiple receivers tweets is similar to that of chain-like patterns except that edges that are sent at the same time are enclosed in parentheses. Given a graph $g$ consisting of $N$ actors and $M$ messages, its code is a string $s_g$ of length $(M \times 2 + 2)$ with the increment in length caused by the parentheses. This allows to encode the six patterns having two edges that happen at the same time as captioned in Figure 13. Similar to the original encoding, two graphs that share the same code are topologically equivalent. Therefore, the task is now broken down to enumerating through all the possible subgraphs of the previously defined form, encoding them to one of six patterns and counting how many times each pattern (string) appears. The algorithm that detects all patterns of this type is elaborated in section 3.3.3.

To sum up this section, two kinds of star-like patterns have been proposed. One type of pattern validates the burstiness of the input networks and it will be analyzed using an existing method, requiring that the input network is flattened into a directed network [1]. The other type reveals what happens before and/or after a tweet is sent out to two different people and requires enumerating subgraphs in the input networks.

3.2.3 Hybrid patterns

It is possible to combine star-like and chain-like patterns to form hybrid patterns that showcase both types of behaviors. There are various theoretical problems and technical challenges that prevent this kind of patterns from being examined in this project. This section explains why hybrid patterns could be interesting for future work but not this project.
In general, hybrid patterns include instances of both star-like and chain-like behaviors, which make them grow bigger in size and more complicated to interpret compared to two separate pattern types. Some examples of hybrid patterns are seen in Figure 14.

Most hybrid patterns could be broken down to smaller chain-like and star-like patterns. For example, Figure 14b describes a dominating bursty behavior combined with a small chain pattern and Figure 14d is a combination of a star and a loop.

As for generalizing the meaning of a big enough hybrid patterns, the overall behavior becomes more important than the relationship between edges within that pattern. For example, Figure 14a and 14c are two very different graphs in term of topology but they can be seen as one pattern since they convey the same meaning: a person tweets to many people but only engage in conversation with one receiver. To account for this, pattern families need to be defined. However, this is beyond the scope of this thesis project and better be examined in future works.

As hybrid patterns are of generally large size, there is more randomness to the topology of these patterns, which introduces more noise [18]. As a consequence, the difference between support counts of these patterns become less significant, meaning that it is less reliable to say if one pattern is more popular than another of the same size and order. Defining pattern family in this case is even more complicated, because multiple noisy patterns that appear very infrequently could significantly increase the support count of a pattern family.

Enumerating subgraphs to find hybrid patterns is expensive. Most existing works approached this by sampling the input network which introduces sampling bias [17, 21]. An algorithm implementing full enumeration was proved in the latest related work, but this only works for directed graph and linear runtime is only achieved with patterns of size and order up to 3. Finding an effective algorithm to enumerate all subgraphs of arbitrary size and order is neither the purpose nor within the scope of this project.

Overall, the author believes that analyzing hybrid patterns is not the best way to study social network data. They have obscure meaning, low reliability and most importantly, they can be broken down to star-like and chain-like patterns that are well-defined. Therefore, hybrid patterns are not the focus of this thesis project.

To conclude section 3.2, three types of patterns have been defined, namely growing chain-like patterns, 2-actor and 3-actor 3-message patterns and patterns involving multiple-receiver tweets. In the later part of the thesis, these three pattern types will be referred to as pattern $P_a$, $P_b$ and $P_c$ accordingly. As recalled in section 2.2, pattern detection includes two steps: subgraph enumeration and graph isomorphism. Since the graph isomorphism problem is tied to the definition of
their structures, the solution for each kind of pattern type has also been elaborated in this section. The next section will present the method used to enumerate subgraphs in the input network.

3.3 Pattern detection algorithm

This section presents the algorithms to enumerate three types of pattern $P_a$, $P_b$, and $P_c$ in the input networks. While the pattern detection algorithm for $P_a$ and $P_c$ are implemented using Multinet library, pattern $P_b$ is assessed using SNAP library which provides an effective method to count patterns of this specific type [1].

3.3.1 $P_a$ pattern

The first type of patterns has been defined in a way that allows them to grow in size and order by adding edges to the patterns one by one, which is the main idea behind the enumeration algorithm. Since each pattern comes from a root actor, the process of enumeration would start from each actor in the actor set to make sure the enumeration is exhaustive. In general, there will be three parameters to the algorithms: the size limit, the order limit and the time duration for either time window or time leap.

Before going into the algorithm, a data structure to hold necessary information of a pattern is defined as C++ struct called Subgraph having the following.

- A list of all the actors in the pattern called actors. New actors added to the pattern will be appended to the end of this list. Unlike ordinary lists, the most common operator on this list is not to access an actor at a given index but to retrieve the index of the actor instead. Therefore, the list is organized as a dictionary to make retrieving actors’ index most accessible. For example, the index of an actor $a$ in the dictionary is actors[\text{actors}]$.

- A list of all edges in the pattern called edges. New edges added to the pattern will be appended to the end of this list. Maintaining an up-to-date list of edges is essential to avoid adding duplicates.

- The beginning timestamp telling when the pattern started called $t_{first}$. This is useful for imposing time window constraint.

- The timestamp of the latest edge added to the pattern called $t_{last}$, telling when the last activity was registered in the pattern. This is useful for imposing time leap constraint.

- A code representing the pattern that the current graph belongs to called code, which is built as explained in section 3.2.1. Every time the graph expands with an edge, two new characters representing the indices of two actors forming that edge are appended to the end of this string.

To reconstruct the original graph from this struct, one starts with an empty graph $g$ and gradually adding to it edges corresponding to pairs of characters in
the code. In each iteration, two actors \( n_i \) and \( n_j \) corresponding to two characters \( i, j \) taken out of the beginning of the code are added to \( g \) along with the edges between them. This process is repeated until the string becomes empty. This again emphasizes the convertible relationship between a graph and a pattern.

Algorithm 1 is the pseudo code for the subgraph enumeration algorithm with two major steps represented by two main loops. \( GetLayer() \) is a Multinet library built-in function that returns the layer that the actor is on. Both actors and messages are in \( A \) but actors is on layer \( l_1 \) and messages (tweets) are on layer \( l_2 \).

The algorithm starts with an empty stack of subgraphs denoted \( G \) and an empty dictionary \( D \) that holds information about the support count of all patterns. The first step is to pick one actor \( a \) in the actor set \( A \) and populate \( G \) with all the graphs rooted from \( a \). These graphs would all have a code of length two "01" but with different actor lists. The second step is to pop out one graph in \( G \) and process it until \( G \) becomes empty. In each iteration, the head of \( G \) denoted \( g_0 \) is popped out and its code will be registered, meaning that the pattern \( p_{g_0} \) has its support count increased by one in the dictionary \( D \). After that, \( g_0 \) will be extended with all out-going tweets from its last actor whose index correspond to the last character in \( g_0 \)’s code. Let the last actor in \( g_0 \) be \( a' \), all tweets sending out from \( a' \) will be added to \( g_0 \) including even the tweets going back to \( a \). The graph extension would fail under the following circumstances:

- The adding edge(s) already exists in the graph.
- If a new actor is added which makes the number of actors exceed the maximum number of actors allowed in one pattern (order limit).
- The adding edges make the number of edges exceed the maximum number of edges allowed in one pattern (size limit).

Algorithm 1 Chain-like pattern detection algorithm

Input A bipartite network \( M = (A, L, V, E) \), \( L = \{l_1, l_2\} \)

Output A dictionary \( D \) of key-value pairs \((s, c)\)

\( s \) is a string encoding a pattern and \( c \) is its support count

1: \( G \leftarrow \emptyset \)
2: \( D \leftarrow \emptyset \)
3: for each \( a \in A \) and \( M.GetLayer(a) = l_1 \) do
4: \( G \leftarrow \text{InitPattern}(a, M) \)
5: repeat
6: \( g_0 \leftarrow G.Pop() \)
7: \( D[g_0.code] \leftarrow D[g_0.code] + 1 \)
8: var \( G' \leftarrow \text{GrowPattern}(g_0, M) \)
9: for each \( g' \in G' \) do
10: \( G.Push(g') \)
11: until \( G = \emptyset \)
12: return \( D \)
The time constraint (time window or time leap) imposed on the graph is violated.

All the graphs that survive this extension are now bigger than $g_0$ in size and/or order, which illustrates the idea of patterns’ growth. These graphs will be added back to the stack $G$ and a new iteration starts. By the end of the second step, all possible chain-like patterns rooted from $a$ has been registered. Another actor is selected from the actor set $A$ and the first step is repeated, following by the second step being repeated. The whole process ends when all the actors in $A$ have been processed. The result is a dictionary of all the codes of processed subgraphs representing all patterns found with their support counts.

Let take a closer look at the two procedures used in the pattern detection algorithm for chain-like patterns. First, $InitPattern(a, M)$, whose pseudo code is shown in Function 1, initializes all 2-actor 1-edge graphs rooted from $a$ in $M$. As a quick reminder, in the input multilayer network $M$, actors are located on layer $l_1$ while messages are in layer $l_2$. Thus, $a$ is on layer $l_1$ and there is no direct links from $a$ to other actors but indirect links through some messages on $l_2$. In order to find out-going tweets from $a$ to other actor, the list of all messages in $l_2$ sent from $a$ denoted messages needs to be retrieved. For each message $m$ in messages that is not a self-tweet (a person tweeting himself), there is a link between each receiver $a_m$ of $m$ and $a$ ($m$ may have multiple receivers). The time constraint still needs to be imposed on both edges from $a$ to $m$ and from $m$ to $a_m$, otherwise the graph is invalid. In the pseudo code for $InitPattern(a, M)$ below, Neighbor() and GetEdge() are Multinet library built-in methods for getting neighbors having an out-going edge from a given node and getting an edge between two given nodes in a bipartite network; SetTfirst() and AddEdge() are two functions to respectively set the beginning time of the subGraph struct and to add a new connection between $a$ and $a_m$ to the struct.

**Function 1 InitPattern($a, M$)**

Input: An actor $a$ and a bipartite network $M$

Output: List of all 2-actor 1-edge subgraphs in $M$ rooted from $a$

1: $G \leftarrow \emptyset$
2: messages $\leftarrow M.NeighborOut(a)$
3: for each $m \in$ messages do
4: var $e_1 \leftarrow M.GetEdge(a, m)$
5: var $t_{first} \leftarrow t_e$
6: receivers $\leftarrow M.NeighborOut(m)$
7: for each $a_m \in$ receivers do
8: var $e_2 \leftarrow M.GetEdge(m, a_m)$
9: var $t_{last} \leftarrow t_e$
10: if $t_{last} \geq t_{first}$
11: var $g \leftarrow$ new Subgraph()
12: $g.SetTfirst(t_{first})$
13: $g.AddEdge(a, a_m, e_1, e_2, t_{last})$
14: $G.Add(g)$
15: return $G$
Function 2 \textit{GrowPattern}(a, M)

\begin{itemize}
  \item \textbf{Input} An actor \(a\) and a bipartite network \(M\)
  \item \textbf{Output} List of all subgraphs grown from \(g_0\)
\end{itemize}

\begin{enumerate}
  \item \(G \leftarrow \emptyset\)
  \item \(a \leftarrow g_0.\text{GetLastActor}()\)
  \item \(t_{\text{Last}} \leftarrow g_0.\text{GetTlast}()\)
  \item \(\text{messages} \leftarrow M.\text{NeighborOut}(a)\)
  \item \textbf{for each} \(m \in \text{messages} \) \textbf{do}
    \begin{enumerate}
      \item \(e_1 \leftarrow M.\text{GetEdge}(a, m)\)
      \item \(t_1 \leftarrow t_{e_1}\)
      \item \textbf{if} \(t_1 \leq t_{\text{Last}}\) \textbf{continue}
    \end{enumerate}
  \item \(\text{receivers} \leftarrow M.\text{NeighborOut}(m)\)
  \item \textbf{for each} \(a_m \in \text{receivers} \) \textbf{do}
    \begin{enumerate}
      \item \(e_2 \leftarrow M.\text{GetEdge}(m, a_m)\)
      \item \(t_2 \leftarrow t_{e_2}\)
      \item \textbf{if} \(t_2 \geq t_1\)
        \begin{enumerate}
          \item \(g \leftarrow g_0.\text{AddEdge}(a, a_m, e_1, e_2, t_2)\)
          \item \textbf{if} \(g \neq \text{NULL}\)
            \begin{enumerate}
              \item \(G.\text{Add}(g)\)
            \end{enumerate}
        \end{enumerate}
    \end{enumerate}
  \item \textbf{return} \(G\)
\end{enumerate}

Similarly, as illustrated in Function 2, \textit{GrowPattern}(\(g_0, M\)) finds the possible connections from the last actor in \(g_0\) and extend \(g_0\) with one of those connections. The pseudo code for this function is shown below, in which \textit{GetLastActor}() and \textit{GetTLast}() return the last actor in \(g_0\) and the timestamp of the last edge in the subgraph. It is worth noticing that \textit{AddEdge}() function here attempts to extend \(g_0\) with the new actors and edges and returns the new subgraph. If the extension fails due to violating any of the three constraints (size, order, time) or the new edge(s) already exist in \(g_0\), the return subgraph will be \(\text{NULL}\).

Finally going one level deeper, let take a look at the \textit{AddEdge}() with the pseudo code illustrated in Function 3. What this function does is to attempt lengthening pattern \(g_0\) with new edges. The extension will be tested against all the constraints, as well as the new edges are checked against the list of edges that the subgraph already contained. The function returns the new subgraph extended from \(g_0\) if the extension succeeds and \(\text{NULL}\) otherwise. In the pseudo code for this function below, there are some constant variables that needs to be explained. \textit{MAXSIZE} and \textit{MAXORDER} are the size and order limit of subgraphs. \textit{TIMECONSTRAINT} is an enum variable specifying whether the time constraint is \textit{TIMEWINDOW} or \textit{TIMELEAP}. \textit{MAXTIMEWINDOW} and \textit{MAXTIMELEAP} are respectively the limit for two types of time constraint. If any of these limits are violated, or if the new edge(s) already exists in \(g_0\), the extension fails and the function returns \(\text{NULL}\).
Function 3 AddEdge($a, a_m, e_1, e_2, tLast$)

**Input** A subgraph $g_0$
- two actors $a$ and $a_m$
- two edges $e_1$ and $e_2$
- the timestamp of the latest new edge $t_{last}$

**Output** A new subgraph grown from $g_0$ if the extension succeeds
- NULL otherwise

```java
1: $g \leftarrow$ new Subgraph($g_0$)
2: if $g$.edges.Contains($e_1$) or $g$.edges.Contains($e_2$)
   3: return NULL
4: if $g$.edges.Size() ≥ MAXSIZE
   5: return NULL
6: $g$.edges.Add($e_1$)
7: $g$.edges.Add($e_2$)
8: if $g$.actors.Size() ≥ MAXORDER
   9: return NULL
10: if $!g$.actors.Contains($a$)
11: $g$.actors[$a$] = $g$.actors.Size()
12: if $!g$.actors.Contains($a_m$)
13: $g$.actors[$a_m$] = $g$.actors.Size()
14: if TIMECONSTRAINT = TIMEWINDOW
15: if $t_{Last} - g$.tFirst ≥ MAXTIMEWINDOW
16: return NULL
17: else if TIMECONSTRAINT = TIMELEAP
18: if $t_{Last} - g$.tLast ≥ MAXTIMELEAP
19: return NULL
20: $g$.tLast ← $t_{Last}$
21: $g$.code ← $g$.code + (char)$g$.actors[$a$] + (char)$g$.actors[$a_m$]
22: return $g$
```

3.3.2 $P_3$ pattern

The second type of pattern include all the patterns in Figure 12 that have up to 3 actors and 3 messages. An effective algorithm to count these patterns in directed graph has been introduced as a part of a graph library called SNAP, and this project aims at utilizing this algorithm to examine these 3-actor 3-message patterns.

The main concern when using SNAP library is that it only works with directed graph, while the data in this project is modelled as a bipartite network due to multiple reasons elaborated in section 3.1. The biggest difference between two network architecture is that there is no direct links between actors. A subgraph with 3 messages will have 3 edges in a directed network but 6 edges in a bipartite network. Therefore, it is necessary to flatten the network from a temporal bipartite network to a temporal directed network.
This flattening process is fairly straightforward. It starts with an empty temporal directed graph $DG$ and a temporal bipartite graph $M = (A, L, V, E)$, $L = \{l_1, l_2\}$ to be flattened. In each iteration, one actor $a$ selected from the actor set $A$ is added to $DG$. All tweets sent from $a$ will be retrieved in a list called $messages$. For each message $m$ in $messages$, the receiver $a_m$ of the tweet is seen to have a connection with $a$ and will be added to $DG$ along with an edge $e$ between $a$ and $a_m$. The question is, what the timestamp for $e$ would be, since there are two edges $e_1, e_2$ in $M$ contributing to the connection between $a$ and $a_m$ via $m$, each has a different timestamp $t_1$ and $t_2$ (presumably). Because it is more accurate to say the message $m$ arrives or is read by $a_m$ at the receiving time $t_2$ than at the sending time $t_1$, the timestamp of $e$ is set to $t_2$. Adding all actors who receive tweets from $a$ and the edge connecting them with the timestamp of the receiving time marks the end of an iteration. Another actor is selected from $A$ to continue the flattening process and the algorithm stops when all actors in $A$ has been processed.

The pseudo code for the temporal bipartite network flattening algorithm is presented in Algorithm 2, in which $HasActor()$, $AddActor()$ and $AddEdge()$ are functions checking if an actor presents in a network, adding an actor and adding an edge to the temporal directed network $DG$.

**Algorithm 2** Temporal bipartite network flattening algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$DG \leftarrow$ new temporal directed graph</td>
</tr>
<tr>
<td>2</td>
<td>for each $a \in A$ do</td>
</tr>
<tr>
<td>3</td>
<td>$DG.AddActor(a)$</td>
</tr>
<tr>
<td>4</td>
<td>$messages \leftarrow M.NeighborOut(a)$</td>
</tr>
<tr>
<td>5</td>
<td>for each $m \in messages$ do</td>
</tr>
<tr>
<td>6</td>
<td>var $e_1 \leftarrow M.GetEdge(a, m)$</td>
</tr>
<tr>
<td>7</td>
<td>var $t_1 \leftarrow t_{e_1}$</td>
</tr>
<tr>
<td>8</td>
<td>receivers $\leftarrow M.NeighborOut(m)$</td>
</tr>
<tr>
<td>9</td>
<td>for each $a_m \in receivers$ do</td>
</tr>
<tr>
<td>10</td>
<td>var $e_2 \leftarrow M.GetEdge(m, a_m)$</td>
</tr>
<tr>
<td>11</td>
<td>var $t_2 \leftarrow t_{e_2}$</td>
</tr>
<tr>
<td>12</td>
<td>if !$DG.HasActor(a_m)$</td>
</tr>
<tr>
<td>13</td>
<td>$DG.AddActor(a_m)$</td>
</tr>
<tr>
<td>14</td>
<td>$DG.AddEdge(a, a_m, t_2)$</td>
</tr>
<tr>
<td>15</td>
<td>return $DG$</td>
</tr>
</tbody>
</table>

The input network is now converted to a temporal directed network. To count all patterns having up to 3 actors and 3 messages in this network, a built-in function in SNAP library is used. The algorithm essentially enumerates all subgraphs of this size in the graph and performs a graph isomorphism check using prefix and suffix tables. SNAP library also only supports the $TIMEWINDOW$ constraint, but there should be no significant difference compared to when imposing $TIMELEAP$ constraint given the fact that the latter works more effectively for chain-like patterns. The results for this analysis is presented in section 4.3.
3.3.3 $P_c$ pattern

The last type of pattern are ones that involve tweets sent to multiple people. As discussed in section 3.2.2, these patterns stay at a fixed size and do not grow like $P_a$ patterns so the enumeration process for this kind of pattern is very straightforward. The patterns at their core include a tweet that is sent from one person to two other people, which already fulfills 3 actors and 2 messages in the subgraphs. There are three ways that an extra tweet can be fit in the subgraph, creating 3 possible patterns. Because this tweet can happen either before or after the timestamp of the multiple-receiver tweet, the possibility of patterns is doubled. Therefore, there are six different 3-actor 2-messages $P_c$ patterns as shown earlier in Figure 13. Since there are solely two distinct timestamp in one subgraph, there is no difference between $TIMEWINDOW$ and $TIMELEAP$ constraints and the former is chosen to enforce in $P_c$ patterns.

The main idea of the pattern detection algorithm for $P_c$ patterns is to find a multiple-receiver tweet and try to extend it with another connection between two of the three actors. The algorithm starts with a temporal bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$ and an empty dictionary of pairs of (key, value) in which key is the code of a pattern and value is its support count. The first step is to form a list of multiple-receiver tweets called $tweets$ by going through all messages in layer $l_2$ and picking out ones that have more than one out-going neighbors. Each message in $tweets$ has one $sender$ and more than one $receivers$.

The second step is to pop out each $tweet$ in $tweets$ and extend it with edges that fall into one of the three following categories.

1. An edge going from $sender$ to one of the $receivers$.
2. An edge going from one of the $receivers$ to the $sender$.
3. An edge going from one receiver to another, which are both in $receivers$.

In each category, the timestamp of the additional edge is compared to the timestamp of the sending time of tweet, the result of which decides the pattern code of the subgraph. These six cases are handled manually as can be seen in the pseudo code Algorithm 3, and the code is then registered to the dictionary. Notice that in the first step in the pseudo code, instead of picking out tweets with multiple receivers, we simply filter out all tweets with no more than one receiver. After this whole procedure, all $P_c$ patterns involving $tweet$ are found.

In conclusion to chapter 3, the network architecture that models input data and three pattern types that are of interest in this project have been defined mathematically. The algorithms to detect those patterns given an inputted bipartite network are also well-elaborated and they will be experimented in the next chapter.
Algorithm 3 Pattern detection for $P_c$ patterns

**Input** A temporal bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$

**Output** A dictionary $D$ of key-value pairs $(s, c)$

$s$ is a string encoding a pattern and $c$ is its support count

1: for each $m \in A$ and $M.GetLayer(m) = l_2$ do
   2: sender ← $M.NeighborIn(m)$
   3: receivers ← $M.NeighborOut(m)$
   4: var $e_m$ ← $M.GetEdge(sender, m)$
   5: var $t_m$ ← time
   6: if receivers.Size() <= 1
      7: continue
   8: // Case 1: an edge going from sender to one of the receivers
   9: for each $m_1 \in M.NeighborOut(sender) \setminus m$ do
      10: for each $a_1 \in M.NeighborOut(m_1)$ do
         11: var $e_1$ ← $M.GetEdge(m_1, a_1)$
         12: var $t_1$ ← $t_{e_1}$
         13: if $a_1 \in$ receivers and $|t_1 - t_m| \leq MAXTIMEWINDOW$
            14: if $t_1 > t_m$
               15: $D[["01(0102)01"]'] ← D[["01(0102)01"]'] + 1$
            else
               16: $D[["01(0102)02"]'] ← D[["01(0102)02"]'] + 1$
   17: // Case 2: an edge going from one of the receivers to the sender
   18: for each receiver $\in$ receivers do
      19: for each $m_2 \in M.NeighborOut(receiver)$ do
         20: for each $a_2 \in M.NeighborOut(m_2)$ do
            21: var $e_2$ ← $M.GetEdge(m_2, a_2)$
            22: var $t_2$ ← $t_{e_2}$
            23: if $a_2 ==$ sender and $|t_2 - t_m| \leq MAXTIMEWINDOW$
               24: if $t_2 > t_m$
                  25: $D[["01(0102)10"]'] ← D[["01(0102)10"]'] + 1$
               else
                  26: $D[["01(0102)10"]'] ← D[["01(0102)10"]'] + 1$
   27: // Case 3: an edge going from one receiver to another receiver
   28: for each receiver $\in$ receivers do
      29: for each $m_3 \in M.NeighborOut(receiver)$ do
         30: for each $a_3 \in M.NeighborOut(m_3)$ do
            31: var $e_3$ ← $M.GetEdge(m_3, a_3)$
            32: var $t_3$ ← $t_{e_3}$
            33: if $a_3 \in$ receivers and $|t_3 - t_m| \leq MAXTIMEWINDOW$
               34: if $t_3 > t_m$
                  35: $D[["01(0102)12"]'] ← D[["01(0102)12"]'] + 1$
               else
                  36: $D[["01(0102)12"]'] ← D[["01(0102)12"]'] + 1$
   37: return $D$
Table 2: Information of 14 datasets including the number of nodes and edges

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>smarttech</td>
<td>1 019</td>
<td>1 232</td>
<td>predictiveanalytics</td>
<td>7 379</td>
<td>10 738</td>
</tr>
<tr>
<td>smartdevices</td>
<td>2 581</td>
<td>3 278</td>
<td>datamining</td>
<td>7 453</td>
<td>10 748</td>
</tr>
<tr>
<td>hardware</td>
<td>3 125</td>
<td>4 573</td>
<td>smartphone</td>
<td>8 948</td>
<td>11 250</td>
</tr>
<tr>
<td>smartmobility</td>
<td>3 877</td>
<td>6 344</td>
<td>arduino</td>
<td>7 588</td>
<td>10 590</td>
</tr>
<tr>
<td>devices</td>
<td>4 976</td>
<td>6 558</td>
<td>itsecurity</td>
<td>8 671</td>
<td>13 652</td>
</tr>
<tr>
<td>smartgrid</td>
<td>4 936</td>
<td>6 586</td>
<td>artificial</td>
<td>9 087</td>
<td>15 288</td>
</tr>
<tr>
<td>smarthomes</td>
<td>6 525</td>
<td>8 282</td>
<td>raspberrypi</td>
<td>10 475</td>
<td>15 946</td>
</tr>
</tbody>
</table>

4 Pattern detection experiments

4.1 Experiment settings

4.1.1 Data pre-processing

This sub-section discusses about the input data, including where they are from, how they are organized and pre-processed prior to the experiments as well as basic statistics about them such as degree distribution and the temporal density of tweets.

Fourteen Twitter datasets are provided by Infolab for the experiments, containing from about 1 000 actors to about 10 000 actors. These datasets are collected using Twitter APIs, containing information of tweets filtered by a certain hashtag that the datasets are named after. The information of tweets includes a sender, a receiver (or some receivers), the sending and receiving timestamps. To anonymize the data, sender and receiver(s) of tweets are mapped to a unique ID. Each dataset having \( n \) nodes (as a reminder, a node can either be a user or a tweet) and \( m \) edges is stored in a file of \( m \) lines, each line is formatted as one of the two following ways.

- A user ID, a tweet ID and a timestamp, meaning that the user sends the tweet at the given timestamp.
- A tweet ID, a user ID and a timestamp, meaning that the tweet is received by the user at the given timestamp.

It is worth noting that the data given by Infolab has already been pre-processed such that there are no tweets that have no receivers. However, there is still a portion of self-tweets in the data, meaning that someone tweets something to himself. These tweets are filtered out during the pre-processing process. The information of fourteen datasets that the pattern detection algorithms were tested on are shown in Table 2, including the size of the networks in terms of the number of actors and the number of edges.
4.1.2 Input data analysis

After the pre-processing step, these datasets will be modeled as bipartite networks whose structure is described in section 3.1. The following section analyzes some features of input networks using built-in and custom functions in Multinet library as well as provides some insights about the datasets.

Degree distribution  Figure 15 illustrates the linear log-log degree distribution of nodes in four representative datasets. It is observed that the majority of nodes has fewer than 10 edges (both in- and out-edges). Meanwhile, there exist several nodes in each dataset that have very high degree, going up to over 1 000 edges. This showcases the strong presence of hub nodes in the input network, suggesting that there will be a lot of patterns around these hubs and they are potential bottlenecks in the pattern detection process.

Temporal growth of input data  To study the temporal growth in given data, important statistics in each dataset are documented at every timestamp, plotting a graph that shows the increment of each of those stats. The stats that are of interest in this analysis includes the number of actors with in-edges, the number of actors with out-edges, the number of actors (with either in-edges or out-edges), the number of messages and the total number of edges. Figure 16 illustrates such growth in all 14 datasets.

It is observed that actors with out-edges dominates ones with in-edges in all datasets, in which an out-edge symbolizes a person publishing a tweet and an in-edge represents a person reading a published tweet. This can be explained as people generating more original content than replying to others’ tweets. Figure 16 also marks the timestamp of four hours, which is seen as covering most of the
Figure 16: Temporal growth of input datasets
Figure 17: Temporal density of input data

content in all datasets. Although there is minor increment in the stats near the end of a 10-day time window, it is safe to conclude that in the given datasets, conversations revolving a certain topic (the hashtag that a dataset is named after) lasted for as long as four hours. This will be the one of the inputs for the recreation of networks in section 5 based on patterns found in this section.

**Temporal density of input data** In an attempt to further study input data, the difference between every two temporally closest messages, i.e. one message and another with the earliest timestamp after the first’s, is documented and Figure 17 plots the distribution of this time difference for each dataset. In theory, this measure could reflect an order of *activeness* in the data, such that an *active* dataset has a very low temporal density in average and vice versa. However, little distinction between the temporal density of testing datasets is seen and all datasets appear very *active*.

### 4.1.3 Experiment environment

Experiments were carried on in a Linux machine Intel(R) Xeon(R) CPU E5520 @ 2.27GHz machine having 8 physical cores and 16 logical (multi-threading) cores.
Table 3: The default value of parameters in the three algorithm

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMECONSTRAINT</td>
<td>TIMEWINDOW</td>
</tr>
<tr>
<td>MAXTIMEWINDOW</td>
<td>60 minutes</td>
</tr>
<tr>
<td>MAXTIMELEAP</td>
<td>60 minutes</td>
</tr>
<tr>
<td>MAXSIZE</td>
<td>10</td>
</tr>
<tr>
<td>MAXORDER</td>
<td>10</td>
</tr>
</tbody>
</table>

The default parameters of pattern detection algorithms are shown in Table 3, except for when a certain parameter is exclusively tested on, which will be explicitly mentioned.

4.1.4 Experiments procedures

The experiments happen in a two-step procedure. In the first step, patterns of each type is explored and counted using the corresponding pattern detection algorithm. From the technical aspect, this step outputs a dictionary $D$ of key-value pairs $(s, c)$, where $s$ is a string encoding a pattern and $c$ is its support count. Note that patterns of size one is essentially an edge in the network, whose support count is independent of the time constraint imposed on it. Because of that, the experiments will exclude the size-one pattern to prevent its overpowering support count compared to other patterns under small time window or time leap.

Since algorithm 1 produces exhaustively all the chain-like patterns in the input data, the number of patterns found is relatively large and not all patterns are interesting. Therefore, the second step is to apply aggregative measurements to extract the important features from the patterns found of type $P_a$.

- Aggregation by pattern size: Sum up the support count of all patterns of the same size.
- Aggregation by pattern order: Sum up the support count of all patterns of the same order.

Experiments studying the values of parameters in algorithm 1 are also carried out in this step. Both aforementioned steps will be presented in sub-section 4.2. As for $P_b$ and $P_c$, there are respectively 36 and 6 patterns, thus it is possible to analyze them directly from the output data of the first step. The analysis of these two pattern types are in sub-section 4.3 and 4.4.

4.2 $P_a$ patterns found

In this part, detected patterns will be aggregated by their size and/or order, such that all patterns of size four are grouped together for example. The purpose of the experiments includes:

- Determine which patterns are popular.
Figure 18: The support counts of $P_a$ patterns aggregated by size w.r.t. different time windows up to one hour display two distinct behaviors.

Table 4: Eighteen values of the time constraint up to one hour

<table>
<thead>
<tr>
<th>Time window</th>
<th>1 second</th>
<th>5 seconds</th>
<th>10 seconds</th>
<th>30 seconds</th>
<th>1 minute</th>
<th>2 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 minutes</td>
<td>10 minutes</td>
<td>15 minutes</td>
<td>20 minutes</td>
<td>25 minutes</td>
<td>30 minutes</td>
<td></td>
</tr>
<tr>
<td>35 minutes</td>
<td>40 minutes</td>
<td>45 minutes</td>
<td>50 minutes</td>
<td>55 minutes</td>
<td>60 minutes</td>
<td></td>
</tr>
</tbody>
</table>

- Determine if which time constraint, time window or time leap, better capture conversations.
- Determine a good value for the selected time constraint.
- Study parameters in correlation with each other, including patterns’ size and order as well as time window and time leap.

Notice that some of the results are sensitive to the input data, thus interesting cases observed from the dataset will be showcased and not all output from all datasets are presented.

4.2.1 Pattern size w.r.t. time windows

This experiment concerns the occurrence of the patterns found, aggregated in size, with respect to different time windows. As a quick reminder, the size of a pattern is the number of messages in a conversation, directly representing how lengthy a conversation is. A size-$k$ pattern is a $P_a$ pattern that consists of $k$ messages. For each dataset, eighteen different values of time windows as shown in Table 4 are tested.

Figure 18 shows the examples of two distinct behaviors observed from the experiment. As seen in dataset smarthomes, the support counts stay in the same order in all the time constraints such that patterns of size 2 is always more abundant than patterns of size 3, which in turn is more abundant than patterns of size 4 and so on. It appears that in some datasets, long conversation as less popular than short chats, which is an expected outcome. This behavior is seen common in other input datasets.
However, some other datasets display a different behavior such that the support count of patterns aggregated by size changes as the time window changes. As the time window increases, the support counts of patterns of certain sizes drastically increases. This behavior is most significant in dataset \textit{devices} where the support count of patterns of size 5 to 9 exceed that of size-2 patterns at around 20 minutes, after which size-7 patterns drastically grow in support count and become the most popular amongst all size-\textit{k} patterns. This shows that long conversations take time to develop which depends on input datasets and for this particular dataset, it takes about 20 minutes to develop long conversations.

In both cases, it is observed that the patterns’ support count increase gradually as the time window increases and stabilize as the time window approaches one hour. It is concluded from this observation that conversations are fully formed after about an hour under \textit{TIMEWINDOW} time constraint.

4.2.2 Pattern size w.r.t. time leaps

This experiment is essentially similar to the previous one with the substitution of time leap for time window. Eighteen different values of time leaps as shown in Table 4 are tested for each dataset. The increment in support count of all size-\textit{k} patterns are documented in Figure 19.

Both behaviors observed when using \textit{TIMEWINDOW} time constraint still display in this experiment when using \textit{TIMELEAP}, and the effect appears to be more apparent. On the left-hand side of Figure 4, the drastic rise in support count of patterns of size from 5 to 9 is seen at a time leap of around 10 minutes, meaning that it takes 10 minutes for people to reply in a long conversation.

Unlike when using time window, notice that for dataset \textit{devices}, the support count does not seem to stabilize when the time leap approaches the one hour mark. This motivates a more extensive experiment on this particular dataset with respect
Table 5: Thirty values of the time constraint up to two hours

<table>
<thead>
<tr>
<th>1 second</th>
<th>5 seconds</th>
<th>10 seconds</th>
<th>30 seconds</th>
<th>1 minute</th>
<th>2 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 minutes</td>
<td>10 minutes</td>
<td>15 minutes</td>
<td>20 minutes</td>
<td>25 minutes</td>
<td>30 minutes</td>
</tr>
<tr>
<td>35 minutes</td>
<td>40 minutes</td>
<td>45 minutes</td>
<td>50 minutes</td>
<td>55 minutes</td>
<td>60 minutes</td>
</tr>
<tr>
<td>65 minutes</td>
<td>70 minutes</td>
<td>75 minutes</td>
<td>80 minutes</td>
<td>85 minutes</td>
<td>90 minutes</td>
</tr>
<tr>
<td>95 minutes</td>
<td>100 minutes</td>
<td>105 minutes</td>
<td>110 minutes</td>
<td>115 minutes</td>
<td>120 minutes</td>
</tr>
</tbody>
</table>

Figure 20: The support counts of $P_a$ patterns aggregated by size w.r.t. both time windows and time leaps up to two hour in dataset devices.

...to both time windows and time leaps to see if the support counts stabilize given more time. The time constraint value is increased up to a maximum of two hours as shown in Table 5.

The results for this extended experiment are visualized in Figure 20. It is seen that to maximize the development of chain-like patterns and stabilize the support count, a time window of about 70 minutes or a time leap of about 50 minutes is required. After these two marks under the respective type of time constraint, the support counts for long conversations slowly stabilize and level out. These are referred to as saturation points and the figures clearly show two of those for dataset devices under any time constraint. However, while the first saturation point leads to drastic changes in the order of the support count of patterns, the second one does not contribute to this change but only magnifies it. For that reason, the rest of the experiments will be done with a default time window/ time leap of one hour instead of two hours.
4.2.3 Pattern order w.r.t. time windows and time leaps

This experiment concerns the occurrence of the patterns found, aggregated by order, with respect to different time windows and different time leaps. As a quick reminder, the order of a pattern is the number of actors participating in a conversation, directly representing how large the group of people involving in a conversation is. An order-$k$ pattern is a $P_a$ pattern that consists of $k$ actors. For each dataset, eighteen different values of time windows/time leaps as shown in Table 4 are tested and the results are visualized in figure 21.

The development of the patterns’ support count aggregated by order follows a similar trend as seen in the first two experiments. Order-2 patterns are the most abundant in most cases and the order in support count of order-$k$ pattern is preserved as the time constraint increases.

The most notable observation from this experiment is that patterns of small orders are more popular than larger ones. In datasets datamining and smalldevices, order-2 patterns greatly dominate order-3 patterns and patterns of order greater than 3 show no presence. In more active datasets such as itsecurity, patterns of orders 3 and 4 are about three times more popular than order-5 patterns and patterns of order higher than 6 are not recorded in the results.

4.2.4 Correlation between patterns’ size and order

The first three experiments deduced interesting characteristics of patterns that are popular. The first two experiments suggest that depending on the datasets, patterns of large sizes could dominate the patterns of small sizes in term of support
count. Meanwhile, the third experiment did not showcase this behavior - only patterns of small orders are popular. This raises a hypothesis that depending on datasets, patterns of large sizes and small orders could potentially be the most abundant.

To verify the aforementioned hypothesis, this experiment takes a look at the correlation between patterns’ size and order with respect to different time windows. In other words, this experiment narrows down the popularity of patterns based on both their size and order. For each dataset, eighteen different values of time windows as shown in Table 4 are tested. The support count of detected patterns is aggregated by both size and order and the results are shown in form of heat maps. Three different behaviors are seen in the experimental results.

The dataset smartdevices is a representative for inactive datasets, where most patterns found are of very small sizes and orders. This suggests that there are few replies in the network, resulting in an absence of long conversations. In these networks, small chains are the most popular. In case of this particular dataset, Figure 22 order-3 size-2 patterns have the highest aggregated support count.

In contrast, the results obtained from dataset devices showcase a different behavior. Figure 23 illustrates that size-7 order-3 patterns are the most popular patterns in dataset devices. These are small conversations involving three people and sending back and forth about 7 messages between them, meaning that at least one person in the conversation get replied twice for his/her prior tweet. Looking at a bigger picture, the most popular patterns are the one of big sizes (from 5 to 9) and small orders (2 or 3). This result suggests that discussion between a few people can develop to a longer conversation than that of a larger group of people. One can also say that conversations involving a small group of people draw greater interest from each participant. Looking from a different angle, the mutual interest
that fuels the conversation between a small group of people could be seen as a way of clustering people. Investigating this possibility is beyond the scope of this project but it is nonetheless a potential premise.

In some other datasets, although there present some conversations, they are dominated by patterns on the diagonal of the heat map as seen in Figure 24. These order-\(k\) size-\(2(k-1)\) \(P_a\) patterns of small \(k\) (\(k < 4\)) symbolize chain-like communication between \(k\) people such that the first person sends a tweet to the second person, whom sends another tweet to the third person and so on. An example for patterns of this type is pattern 011223, which is a chain-like conversation between

![Figure 23: The correlation between \(P_a\) patterns’ size and order w.r.t. time windows up to one hour on dataset devices.](image)

![Figure 24: The correlation between \(P_a\) patterns’ size and order w.r.t. time windows up to one hour in dataset itsecurity.](image)
4 people via 3 different tweets. This dominance is seen across all different time windows up to an hour as the order of patterns’ support count is preserved when the time window increases.

These three different behaviors reveal the rivalry between chains and conversations in Twitter data. Some datasets display a great dominance of small chains over the other while the rest of the datasets showcases the opposite or combined behavior.

4.2.5 Correlation between time window and time leap

This experiment concerns the relationship between two types of time constraints \textsc{timewindow} and \textsc{timeleap}. While using time window bounds the total duration of the patterns, the time leap constraint only restricts the time between two consecutive tweets, posing no limitation to how long pattern last. For example, a size-10 pattern satisfying a time leap restriction of 30 minutes could last for 2.5 hours if each of the 10 tweets besides the first one is sent 15 minutes after the one prior to it. Previous experiments have imposed these two types of time constraints separately but not together. In this experiment, both the time window and the time leap will be restricted.

To conduct this experiment, each dataset will be imposed by one time window restriction and another time leap restriction. Violating either \textsc{timeleap} or \textsc{timewindow} constraint in a pattern causes it to stop growing. Both time restrictions will go from one minute to one hour with the increment of one minute, making up for 60 different time windows and 60 different time leaps and a total of 3600 runs for each dataset. Due to the nature of the two constraints, a time leap restriction of \( t_2 \) will be naturally bounded by a time window restriction of \( t_1 \) if \( t_2 \leq t_1 \). Because of this, it is possible to halve the amount of runs with the time leap restriction greater or equal to the time window restriction, leaving 1800 runs for each dataset. Due to the number of runs required in this experiment, only five representative datasets are tested on, namely \texttt{smartdevices}, \texttt{devices}, \texttt{smarthomes}, \texttt{datamining} and \texttt{itsecurity}.

Figure 25 illustrates the support count of patterns aggregated by size with respect to 60 time windows and 60 time leaps both from one minute to one hour on one representative dataset. The results indicate a strong correlation between time leap and time window constraints. Notice the support count of patterns of size 7 in dataset \texttt{itsecurity} under a time window of one hour (the top row) in the heat map for patterns of size 7, the support count drastically increases as the time leap increases and it reaches the maximum at a time leap of about 25 minutes. This means that there is at least one amongst 7 messages in one-hour-long patterns that is sent 25 minutes after the one prior to it. The same behavior is seen for time windows greater than 50 minutes (the top 10 rows). Meanwhile, as for the same time leap of 20 minutes, increasing the time window even up to an hour is not enough to let size-7 patterns fully grow. This suggests that 50-minute time window and 25-minute time leap lower bounds are required for size-7 patterns to
fully develop. In addition, a small enough time leap, i.e. less than 10 minutes or a small enough time window, i.e. less than 40 minutes strongly prohibits the growth of patterns as the support count is insignificant.

This behavior is seen across all datasets and all patterns’ size; the bigger the patterns are in size, the more apparent the effect is. According to the results in dataset devices in Figure 26 c), there seems to be an inversely proportional connection between the size of the patterns and the lower bound for the time leap constraint such that the time leap lower bound is decreases as the patterns’ size increases. However, the author failed to identify this mathematical relationship within the time frame of this project.

To sum up this experiment, there exists a strong correlation between the two types of time constraints and using them in combination requires more fine-tuning as there are both time window and time leap lower bounds for patterns to fully grow in size.

### 4.2.6 Temporal density of patterns

This experiment studies the temporal density in the $P_a$ patterns found. Given a pattern $p$ of size $s_p$, all the temporal difference between $n^{th}$ and $(n+1)^{th}$ messages, $1 \leq n < s_p$ give a sense of temporal density of $p$. For better convenience, these temporal differences will be referred to as gaps, such that the $n^{th}$ gap is the gap
Figure 26: The correlation between time windows and time leaps up to one hour on $P_a$ patterns aggregated by size, in dataset devices.

Figure 27: The temporal density of $P_a$ patterns in two datasets.

between $n^{th}$ and $(n+1)^{th}$ messages. Figure 27 plots all such temporal differences in all patterns found in dataset devices under time leap constraints and classified by the numerical order of the gaps.

This result showcases a decreasing trend in the temporal difference between two consecutive messages as the patterns grow longer. The first and second gaps are averagely 20 minutes and 10 minutes for dataset datamining and 10 minutes and 8 minutes for dataset devices. This suggests that once the conversation is elaborated well enough and the mutual interest between the participants is strong enough, the time it takes to reply to a prior message gradually decreases. This knowledge could be helpful for specifying a dynamic time leap constraint that is not constant for every pair of messages but changes accordingly to how deep the conversation is. Exploiting this possibility is beyond the scope of this project.
Figure 28: The support count of $P_b$ patterns w.r.t. four different time windows in two datasets.

4.3 $P_b$ patterns found

$P_b$ patterns are all 2-actor and 3-actor 3-message patterns that could be counted in the input datasets using built-in function in SNAP library. Before proceeding to this step, the input datasets are flattened from temporal bipartite networks to temporal directed networks using algorithm 2. The results are shown in Figure 28, which record the support count of 36 2-actor and 3-actor 3-message patterns previously described in section 3.3.2.

The results for different datasets are very similar. Three star-like patterns at the top-right, bottom-right and bottom-left corners of the heat maps (correspondingly to the patterns at the same position illustrated in Figure 12) significantly dominate the rest of the patterns, although the specific support counts are different in different datasets. Consequently, the support counts for non-star-like patterns are too small compared to that of star-like patterns that they are indistinguishable in the figures. A strong non-blocking style of communication is seen from the results, indicating that individuals send out several tweets in a row without replying to one in between [1]. In general, this also re-strengthens the motivation for studying chain-like patterns separately from star-like patterns. Moreover, the result confirms the bursty nature of Twitter which generalizes to other types of social networks, agreeing to what existing works on the same topic (refer to section 2) have concluded.

4.4 $P_c$ patterns found

$P_c$ patterns are patterns that cannot be modeled in directed network but only appear in bipartite networks and they are also not chain-like like $P_a$ patterns.
Therefore, none of the previously presented methods works for $P_c$ patterns. As proposed in section 3.3.3, algorithm 3 will be run on the temporal bipartite networks modeled from input data to count the support of all six $P_c$ patterns denoted $pc_i, i \in \{1, ..., 6\}$ as shown in Figure 13.

The experimental results are shown in Figure 29, displaying two opposite behaviors. On datasets datamining and itsecurity, the pattern $pc_3$ starts out as the most popular one with respect to time windows from one second to 2 minutes and when the time window is increased beyond 2 minutes, pattern $pc_3$ is dominated by two other patterns $pc_1$ and $pc_4$. Meanwhile, the same pattern dominates across all time windows in the other two datasets.

What makes this observation interesting is that pattern $pc_3$ represents a message sent from someone to two different people simultaneously, followed by a message sent between the two receivers. In other words, pattern $pc_3$ shows an order of causality in Twitter communication. Meanwhile, two patterns $pc_1$ and $pc_4$ purely showcase the burstiness characteristic of the communication. Based on the fact that the specific support count of patterns $pc_3$ and $pc_6$ are almost similar for all 5 datasets (around 1 500 and 1 000 respectively under a time window of one hour), it is possible to deduce that causality involving multiple-receiver tweets exists in all datasets but it is masked out by the overpowering dominance of bursty patterns.
When comparing $P_c$ patterns to $P_b$ patterns in terms of support count, the former ones are greatly dominated by the latter ones when the time window goes beyond one minute similarly to what is observed in this experiment. Hence, the conclusion about causality involving multiple-receiver tweets in Twitter is only valuable with respect to small time windows, such that the effect wears out after a short amount of time. Keep in mind that this exclude other forms of causality that do not involve sending tweets to multiple people, e.g. $a$ talks to $b$ and $b$ talks to $c$ leading to $a$ talking to $c$. 


5 Network growth models

5.1 Motivation

The results of the pattern detection phase presented in section 4 show that long conversation are mostly between two or three people and there is a decaying tendency of the reply time as the conversation progresses, which agrees to previous studies in other types of data [6]. This means that once a user starts concentrating on a conversation, the time it takes for he/she to reply decreases as the conversation grows. The distribution of replying time is useful to generate a realistic network modeling conversation on Twitter.

Based on that, this section presents five network growth models to generate a bipartite network that shares the pattern-based characteristic observed in input datasets such as the distribution of patterns’ size and order. The generated networks will then be analyzed in term of degree distribution to see if they resemble real-life data. Successful generation of new networks that feature the patterns found in input datasets serves for multiple purposes. First, it strengthens and validates the characteristics deduced from observing the detected patterns. Second, it has potential benefits for automating the data collection process or randomizing collected data.

The content of this section is organized as follows. Subsection 5.2 introduces the two random network growth models and points out their weaknesses. After that, a Barabási–Albert model is introduced in subsection 5.3, followed by two extended models on top of that in subsections 5.4 and 5.5.

5.2 Random models

This section presents two random models for network growth and analyzes their pros and cons. Based on the assumption that the generated network has only one connected component, these network growth models are aimed to generate a network consisting of $N$ weakly connected nodes (including both actors and messages) with $N$ being the given size of the network.

5.2.1 Random model 1

To embrace the simplicity of the generation model, a very basic random model is proposed. This first random model starts with a network consisted of $n_0$ actors and no messages. In each step, a new connection between two actors $a_1$ and $a_2$ will be added to the network until the number of nodes exceeds $N$. Note that a connection involves one message and two edges connect the message to two actors. The direction of the connection is randomly chosen with a possibility of $p_{dir} = 50\%$ to maintain a balance between connections going in and out of the network. There is a possibility of $p_{extend} (0 < p_{extend} < 1)$ that $a_1$ is a new actor that is not present in the current network, in which case a new actor will be added to the
network before the new connection is added. This allows the network to stay connected before and after each iteration. Every time new edges are added, their timestamp will be determined by a random value between 0 and a time window $T$, which represents the difference between the earliest and the latest message in the network and from now is referred to as the duration of the network. Algorithm 4 generates a bipartite network using this random model.

**Algorithm 4 Random network growth model 1**

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of nodes $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time duration of the network $T$</td>
</tr>
<tr>
<td></td>
<td>Probability of picking a new node $p_{extend}$</td>
</tr>
<tr>
<td></td>
<td>Initial size $n_0$</td>
</tr>
</tbody>
</table>

**Output** A bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$

1: $M \leftarrow$ new temporal bipartite network of $n_0$ actors
2: while $(M.NodeSize() < N)$
3:   if Random$(0, 1) < p_{extend}$
4:     $a_1 \leftarrow M.AddNewActor()$
5:   else
6:     $a_1 \leftarrow M.GetRandomActor()$
7:     $a_2 \leftarrow M.GetRandomActor()$
8:     $m \leftarrow M.AddNewMessage()$
9:     $t \leftarrow$ Random$(0, T)$
10:    if Random$(0, 1) < 0.5$
11:     $M.AddConnection(a_1, m, a_2, t)$
12:    else
13:     $M.AddConnection(a_2, m, a_1, t)$
14: return $M$

**5.2.2 Random model 2**

The second random model is basically similar to the first random model except the generation of messages’ timestamp. In the first model, the timestamp is selected randomly between 0 and $T$, the duration of the network. However, this model sets the timestamp as an increasing variable. At each iteration, a small time duration is added to the last timestamp and the sum is set as the newly added edges’ timestamp. This method of generating timestamp for edges allows continuity in information flow. Algorithm 5 generates a bipartite network using this random model, in which Random log() is the random function with lognormal distribution, favoring values near zero.

**5.2.3 Evaluate random models**

The two Algorithms 4 and 5 are set up with the parameters’ value shown in Table 6, selected based on the analysis of input data presented in section 4.1.2. The two generated networks are respectively named Random1 and Random2.
Algorithm 5 Random network growth model 2

Input

- Number of nodes $N$
- Max value of time duration $\Delta T$
- Probability of picking a new node $p_{\text{extend}}$
- Initial size $n_0$

Output

A bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$

1: $M \leftarrow$ new temporal bipartite network of $n_0$ actors
2: $\text{lastTimestamp} \leftarrow 0$
3: while $(M.\text{NodeSize}() < N)$
4:   if $\text{Random}(0, 1) < p_{\text{extend}}$
5:     $a_1 \leftarrow M.\text{AddNewActor}()$
6:   else
7:     $a_1 \leftarrow M.\text{GetRandomActor}()$
8:     $a_2 \leftarrow M.\text{GetRandomActor}()$
9:     $m \leftarrow M.\text{AddNewMessage}()$
10:    $\Delta t \leftarrow \text{Random}_{\log}(0, \Delta T)$
11:    $\text{lastTimestamp} \leftarrow \text{lastTimestamp} + \Delta t$
12:    if $\text{Random}(0, 1) < 0.5$
13:      $M.\text{AddConnection}(a_1, m, a_2, \text{lastTimestamp})$
14:    else
15:      $M.\text{AddConnection}(a_2, m, a_1, \text{lastTimestamp})$
16: return $M$

Table 6: The value of parameters in two random network growth algorithms

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>5 000</td>
</tr>
<tr>
<td>$T$</td>
<td>4 hours</td>
</tr>
<tr>
<td>$\Delta T$</td>
<td>30 minutes</td>
</tr>
<tr>
<td>$n_0$</td>
<td>5</td>
</tr>
<tr>
<td>$r$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

An experiment will be carried out to study the chain-like patterns in the two generated networks using Algorithm 1 whose parameters are set up similarly to those in Table 3. After that, the two generated networks will be examined in term of degree distribution to see if they resemble real-life data.

The first experiment studies the correlation between patterns’ size and order in two generated networks and compares the results with the observed behaviors in inputted network presented in section 4.2.4. The same experiment procedure as in section 4.2.4 is repeated and the results are shown in Figure 30. It is apparent that both models fail to generate long conversations between a small group of people, illustrated by low support counts for patterns of big size and small order. In contrast, the size-$n$ order-$(n+1)$ patterns are the most abundant, which agrees to one of the two behaviors seen in experiment 4.2.4.

The second experiment studies the degree distribution in random1 and random2. According to the experimental results in Figure 31, the log-log degree distributions
Figure 30: The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in two generated datasets Random1 and Random2.

Figure 31: The degree distribution of two generated datasets Random1 and Random2.
are not linear and the tails appear very fat and short. This is greatly different from the degree distribution of input datasets.

According to two previous experiments, simple random models are able to recreate chain-like size-\(n\) order-(\(n+1\)) patterns which are seen abundant in inputted networks while also capturing the correlation between time window and time leap. However, they fail to generate patterns of big sizes and small orders in input datasets and instead favor patterns of big sizes and big orders, illustrating shallow conversations between a large group of people. Moreover, the log-log degree distribution in the networks generated by two random models are also non-linear, showing little resemblance to that of inputted networks. Therefore, three BA model is employed in the next section to overcome these shortcomings, re-using the time generation strategy in random network growth model 2 (algorithm 5) since it can recreate some loop-like patterns that the two random models are incapable of.

5.3 Generic Barabási–Albert model

Barabási–Albert (BA) model is a random scale-free network growth model that generates networks using preferential attachment mechanism. The idea behind preferential attachment is that popular nodes have higher chance of gaining more popularity and the more popular a node is, the greater chance it has new connections. In other words, this reflects "the rich get richer" philosophy.

The differences between this generic BA model and the random model 2 are in the initialization, actor selection process and the number of connections formed in each iteration. This BA model starts with a set of \(n_0\) connected actors such that actor \(a_i\) is connected to \(a_{i+1}\), \(0 < i < n_0 - 1\). This setup is to make sure each actor in the initial network have a positive degree. In each iteration, a new actor \(a_{\text{new}}\) is added to the network, from which \(\Delta n\) connections will be formed with existing actors. The connection formation is similar to that of random model 2 such that two actors \(a_1, a_2\) are selected and the connection between them is either from \(a_1\) to \(a_2\) or the opposite one with a 50% chance. The first actor in this case is \(a_{\text{new}}\) and \(a_2\) is selected from the existing actor set with a probability proportionally to the actors’ degree.

Algorithm 6 illustrates the implementation of this model, in which the actor selection function \texttt{GetActorBasedOnDegree()} is done using a selection wheel consisting of pieces and each actor \(a\) of degree \(d_a\) owns \(d_a\) pieces on the wheel.

Three datasets are generated with respect to three different values of \(\Delta n\) and they are denoted \(\text{BA}_\alpha(\Delta n = 1)\), \(\text{BA}_\beta(\Delta n = 2)\) and \(\text{BA}_\gamma(\Delta n = 3)\). Repeating the experiments studying the degree distribution and correlation between patterns’ size and order in section 5.2.3 on three networks generated by this generic BA model yields the results illustrated in Figure 32. According to the results, the BA model produces networks having a more linear log-log degree distribution compared to
Algorithm 6 Generic BA model

**Input** Number of nodes $N$
- Max value of time duration $\Delta T$
- Probability for out-going connection $p_{dir}$
- Initial size $n_0$
- The number of connections added every iteration $\Delta n$

**Output** A bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$

```
1: $M \leftarrow$ new temporal bipartite network of $n_0$ connected actors
2: $lastTimestamp \leftarrow 0$
3: while ($M$.NodeSize() < $N$)
4:     $a_1 \leftarrow M$.AddNewActor()
5:     for $i \leftarrow 1$ to $\Delta n$
6:         $a_2 \leftarrow M$.GetActorBasedOnDegree()
7:         $m \leftarrow M$.AddNewMessage()
8:         $\Delta t \leftarrow \text{Random}_{\log}(0, \Delta T)$
9:         $lastTimestamp \leftarrow lastTimestamp + \Delta t$
10:        if Random$(0, 1) < p_{dir}$
11:            $M$.AddConnection($a_1, m, a_2, lastTimestamp$)
12:        else
13:            $M$.AddConnection($a_2, m, a_1, lastTimestamp$)
14:    return $M$
```

Figure 32: (Upper) the correlation between $P_a$ patterns’ size and order w.r.t. an one-hour-long time window and (lower) the degree distribution of the three datasets BA$\alpha$, BA$\beta$ and BA$\gamma$
that of random models. However, the model has yet successfully recreated deep conversations within a small group of people as expected. The appearance of patterns of large sizes and large orders in $BA_{\gamma}(\Delta n = 3)$ suggests that messages are not getting replied quick enough, thus not allowing conversations to develop. Based on that observation, two extensions of the BA model are proposed.

### 5.4 BA model with replying

This BA model with replying (BAR) is developed on top of the generic BA model such that before each iteration ends, there is a possibility of $p_{\text{reply}}$ (0 < $p_{\text{reply}}$ < 1) that a random message in the network is replied. If there is a reply, the sender and receiver of a random message $m_1$ is extracted and a new connection via message $m_2$ between them in the opposite direction, i.e. from the receiver to the sender, is added. The timestamp of the reply $m_2$ is determined by the timestamp of $m_1$ offset by a small time duration.

Algorithm 7 illustrates the implementation of this model, in which $t_{m_1}$ and $t_{m_2}$ are the timestamps of $m_1$ and $m_2$ respectively and $m_1\.sender$, $m_1\.receiver$ are the sender and receiver of message $m_1$. Note that BAR model simplifies to the generic BA model when $p_{\text{reply}} = 0$.

**Algorithm 7** BAR model

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of nodes $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max value of time duration $\Delta T$</td>
</tr>
<tr>
<td></td>
<td>Probability for out-going connection $p_{\text{dir}}$</td>
</tr>
<tr>
<td></td>
<td>Initial size $n_0$</td>
</tr>
<tr>
<td></td>
<td>The number of connections added every iteration $\Delta n$</td>
</tr>
<tr>
<td></td>
<td>Probability for replying $p_{\text{reply}}$</td>
</tr>
</tbody>
</table>

| Output | A bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$ |

1: $M \leftarrow$ new temporal bipartite network of $n_0$ connected actors
2: $\text{lastTimestamp} \leftarrow 0$
3: while $(M\.NodeSize() < N)$
4:   $a_1 \leftarrow M\.AddNewActor()$
5:   for $i \leftarrow 1$ to $\Delta n$
6:     $a_2 \leftarrow M\.GetActorBasedOnDegree()$
7:     $m \leftarrow M\.AddNewMessage()$
8:     $\Delta t \leftarrow \text{Random}.\log(0, \Delta T)$
9:     $\text{lastTimestamp} \leftarrow \text{lastTimestamp} + \Delta t$
10:    if Random$(0, 1) < p_{\text{dir}}$
11:      $M\.AddConnection(a_1, m, a_2, \text{lastTimestamp})$
12:    else
13:      $M\.AddConnection(a_2, m, a_1, \text{lastTimestamp})$
14:    if Random$(0, 1) < p_{\text{reply}}$
15:      $m_1 \leftarrow M\.GetRandomMessage()$
16:      $m_2 \leftarrow M\.AddNewMessage()$
17:      $t_{m_2} \leftarrow t_{m_1} + \text{Random}(0, \Delta T)$
18:      $M\.AddConnection(m_1\.receiver, m_2, m_1\.sender, t_{m_2})$
19: return $M$
Figure 33: The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in the generated dataset $\text{BAR}_\zeta(\Delta n = 3, \ p_{\text{reply}} = 0.1)$.

Figure 33 shows the correlations between size and order of $P_a$ patterns found in the dataset generated using BAR model with $\Delta n = 3$ and $p_{\text{reply}} = 0.1$. It is seen that the addition of $p_{\text{reply}}$ allows a lot more replies to take place while maintaining the dominance of chains. This is illustrated by a much higher coverage in the heat map compared to that of BA model and high support counts of patterns on the diagonal.

On the down side, the BAR model still inherits weaknesses from BA model and two random models. First, the model fails to generate patterns of large sizes and small orders. Second, the chains grow too long compared to the ones seen in input data.

5.5 BAR model with decaying reply probability

This BAR model with decaying reply probability (BARD) is developed on top of the BAR model to allow multiple messages replied back and forth between two actors instead of only one reply in BAR model. Every time a random message $m_1$ is replied with $m_2$, there is a probability of $p_{\text{reply}} \times d$ that this new message is replied again with $m_3$, in which $d$ is the decay factor ($0 < d < 1$). The sender and receiver will be alternated between replies to propagate patterns of big size and small order. The decay factor helps to reduce the reply probability as the conversation develops. According to the observation in section 4.2.6 that the reply time shortens as the conversation develops, the maximum value of time duration between replies $\text{deltaTime}$ also decays for every reply. This configuration promotes long conversation between two people and the decay factor serves to control how long the conversations are expected to last.

Algorithm 8 illustrates the implementation of this model. Note that BARD model simplifies to the BAR model when $d = 0$ and further simplifies to the generic BA model when $p_{\text{reply}} = 0$. 

62
Algorithm 8 BARD model

Input
- Number of nodes $N$
- Max value of time duration $\Delta T$
- Probability for out-going connection $p_{\text{dir}}$
- Initial size $n_0$
- The number of connections added every iteration $\Delta n$
- Probability for replying $p_{\text{reply}}$
- Decay factor for the reply probability $d$

Output
- A bipartite network $M = (A, L, V, E)$, $L = \{l_1, l_2\}$

\begin{algorithm}
1: $M \leftarrow$ new temporal bipartite network of $n_0$ connected actors
2: $\text{lastTimestamp} \leftarrow 0$
3: while $(M.\text{NodeSize}() < N)$
4: \hspace{1em} $a_1 \leftarrow M.\text{AddNewActor}()$
5: \hspace{1em} for $i \leftarrow 1 \text{ to } \Delta n$
6: \hspace{2em} $a_2 \leftarrow M.\text{GetActorBasedOnDegree}()$
7: \hspace{2em} $m \leftarrow M.\text{AddNewMessage}()$
8: \hspace{2em} $\Delta t \leftarrow \text{Random}_\log(0, \Delta T)$
9: \hspace{2em} $\text{lastTimestamp} \leftarrow \text{lastTimestamp} + \Delta t$
10: \hspace{2em} while $\text{Random}(0, 1) < p_{\text{dir}}$
11: \hspace{3em} $M.\text{AddConnection}(a_1, m, a_2, \text{lastTimestamp})$
12: \hspace{2em} else
13: \hspace{3em} $M.\text{AddConnection}(a_2, m, a_1, \text{lastTimestamp})$
14: \hspace{1em} if $\text{Random}(0, 1) < p_{\text{reply}}$
15: \hspace{2em} $m_1 \leftarrow M.\text{GetRandomMessage}()$
16: \hspace{2em} $p_{\text{decay}} \leftarrow 1$
17: \hspace{2em} $\text{sender} \leftarrow m_1.\text{sender}$
18: \hspace{2em} $\text{receiver} \leftarrow m_1.\text{receiver}$
19: \hspace{2em} $\text{sendingTime} \leftarrow t_{m_1}$
20: \hspace{2em} $\text{deltaTime} \leftarrow \Delta T$
21: \hspace{2em} while $(\text{Random}(0, 1) \leq p_{\text{decay}})$
22: \hspace{3em} $m_2 \leftarrow M.\text{AddNewMessage}()$
23: \hspace{3em} $\text{sendingTime} \leftarrow \text{sendingTime} + \text{Random}(0, \text{deltaTime})$
24: \hspace{3em} $M.\text{AddConnection}(\text{receiver}, m_2, \text{sender}, \text{sendingTime})$
25: \hspace{3em} $(\text{sender}, \text{receiver}) \leftarrow (\text{receiver}, \text{sender})$ // swapping
26: \hspace{2em} $p_{\text{decay}} \leftarrow p_{\text{decay}} \times d$
27: \hspace{2em} $\text{deltaTime} \leftarrow \text{deltaTime} \times d$
28: return $M$
\end{algorithm}

Four datasets are generated with respect to three different values of the decay factor and they are denoted $\text{BARD}_\alpha(\Delta n = 3, p_{\text{reply}} = 0.1, d = 0.1)$, $\text{BARD}_\beta(\Delta n = 3, p_{\text{reply}} = 0.1, d = 0.2)$, $\text{BARD}_\gamma(\Delta n = 3, p_{\text{reply}} = 0.1, d = 0.3)$ and $\text{BARD}_\zeta(\Delta n = 3, p_{\text{reply}} = 0.1, d = 0.5)$. Repeating the experiments studying the degree distribution and correlation between $P_a$ patterns’ size and order in section 5.2.3 on three networks generated by this generic BA model yields the results illustrated in Figure 34.

According to the results, under certain setting of parameter, the recursive replying mechanism allows BARD model to overcome the weaknesses persisted from
Figure 34: The support counts of $P_a$ patterns w.r.t. an one-hour-long time window in four generated datasets BARD$_\alpha$, BARD$_\beta$, BARD$_\gamma$ and BARD$_\zeta$.

The random models to the BAR model. To be specific, BARD$_\gamma$ and BARD$_\zeta$ showcase many patterns of large sizes and small orders. Moreover, the two datasets display only chains up to a size of 6 and short chains of size 2 are the most popular patterns.

BARD model is seen as the most successful attempt in recreating Twitter datasets that resemble the popular patterns found in inputted networks while maintaining the degree distribution observed in testing datasets.
6 Discussion and future works

6.1 Conclusion

Pattern detection in social networks such as Twitter and Facebook reveals important insights about how people communicate online. Existing studies have found that communication patterns on social networks are extremely bursty to a point star-like patterns overshadow the rest of the patterns. To add to that, different types of patterns symbolizes different aspect of human behaviors. While stars showcase the strong presence of network hubs, chains reveal collaborative characteristics of human behaviors (e.g. causality, homophily) and loops indicate mutual interest of people on the topic of discussion. This observation suggests a separation in studying patterns of different types.

Therefore, this thesis categorizes patterns to three different types of topology families and studies them in separation. Three types of patterns $P_a$, $P_b$, $P_c$ are mathematically defined, accordingly describing chain-like communication, star-like communication and communication patterns involving multiple-receiver tweets. Putting an emphasis on $P_a$ patterns, this thesis studies the relationship between two types of time constraints namely time window and time leap as well as the correlation between the size and order of $P_a$ patterns. Three methods to detect three types of patterns are proposed, one of which utilizes SNAP library and the other two utilize built-in functions in Multinet library.

As for $P_a$ patterns, the results indicate a lower boundary of both time window and time leap constraints for the patterns to fully develop, e.g. about 50 minutes and 25 minutes respectively for dataset devices. Experimental results also showcase a decaying tendency of the reply time as the conversation progresses. This means that once a user starts concentrating on a conversation, the time it takes for he/she to reply decreases as the conversation grows. The distribution of replying time is useful to generate a realistic network modeling conversation on Twitter.

Regarding which $P_a$ patterns are the most popular, two different behaviors observed in different datasets. For some datasets, short chains are the most popular and patterns involving replies are rarely seen. Other cases observe a high popularity from patterns of large size and small order, which indicate long conversation between a small group of people. The author fails to explain the cause of this input-sensitive difference in this thesis, but some hypotheses are discussed in the future work section.

The analysis on $P_b$ patterns found does not provide new insights about the datasets besides confirming the bursty nature of Twitter data and its non-blocking style of communication. This again motivates the topology-based categorization of patterns in this project. As for $P_c$ patterns, analyzing the patterns’ support count reveal an order of causality such that a message sent from someone to two different people motivates follow up communication between two receivers. This effect is
the most apparent within a small time window, e.g. about 2 minutes on some datasets, and is also dominated by bursty patterns after that time window.

To generalize the understanding on patterns found with regards to Twitter data, five pattern-based network growth models are proposed. These models aim at generating networks that not only display patterns found in the given Twitter data but also share common characteristics as that of real-life data such as long-tail degree distribution. The two random models are the simplest ones. However, their generated networks have short-tail degree distribution and only display long chains. The generic BA model improves the degree distribution, but the patterns seen remain inorganic with a lot of long chains. Further extensions on the BA model namely BAR and BARD models prove significant improvements. The BARD model is the best one such that it displays the most similar behavior of chain-like patterns as seen in inputted data while maintaining the realistic degree distribution as seen in BA and BAR models.

6.2 Shortcomings and improvements

Given more time on this topic, there are many potential directions for future improvements. On one hand, this project has some significant shortcomings that could use more time to overcome and/or optimize. On the other hand, there are some promising results that could be further exploited in a more narrowed scope. This section discusses matters in both aspects.

One of the most notable concerns is the unknown cause for two different behaviors seen with $P_a$ patterns in different datasets. As a hypothesis, this matter could be explained by the activeness of the datasets, which can be indicated by different measures as long as it symbolizes how frequent people reply in the data. The author attempted to represent the activeness by the temporal density of the dataset as presented in section 4.1.2 but failed to see significant distinction between that of inputted data. There might be better ways to measure this activeness parameter of datasets.

Another weakness of the project is the high complexity of patterns detection algorithms. The reason for this is a combination between unoptimized algorithm design and the complicated underlying structure of the Multinet library that the algorithms are built on. This lead to poor scalability of the algorithms and this is further exaggerated in experiments where a great number of runs is required such as the one in section 4.2.5 with 1 800 runs per dataset. Because of this poor scalability, datasets of larger sizes are not tested due to being timed out, which lowers the generality of the results. This complexity problem will be solved by optimizing the algorithms with better utilization of the built-in functions in Multinet library.

As for the network growth models, the experiments are not fully elaborated such that parameters of the models are not fully studied and chosen with strong
motivations due to the lack of time for the project. The best network growth model, BARD, is very sensitive to parameter, i.e. the decay factor, thus more experiment and analysis should be carried out.

Last but not least, the thesis comes up with some interesting conclusions that are worth further investigations.

- Most studies in pattern detection used the time window constraint and this thesis shows that using time leap constraint could be a better alternative.

- The decay in replying time as the conversation grows could potentially appear in other types of social networks such as emails, which might be useful for email reminder.

- Patterns of large sizes and small orders symbolize strong connections between the people in the conversation, which could be a potential input in network clustering. The mutual interest between participants in a conversation could also add more weight to the connection between them during the network flattening process.
References


