MINIMIZING DISSEMINATION ON LARGE GRAPHS

BY LONG T. LE

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Professor Tina Eliassi-Rad

and approved by

_____

_____

_____

New Brunswick, New Jersey

January, 2015

ABSTRACT OF THE THESIS

Minimizing Dissemination on Large Graphs

by Long T. Le

Thesis Director: Professor Tina Eliassi-Rad

Given the topology of a graph G and a budget k, can we quickly find the best k edges to delete that minimize dissemination on G? Stopping dissemination on a graph is important in variety of fields such as epidemic control and network administration. Understanding the *tipping point* is crucial to the success of minimizing dissemination. The tipping point of an entity (e.g, virus or memo) on an arbitrary graph only depends on (i) the topology of graph and (ii) the characteristics of the entity. In this work, we assume that we cannot control the characteristics of the entity such as its strength, death rate, and propagation rate. Thus, we can only modify the topology of the graph. In particular, we consider the problem of removing k edges from the graph. In order to minimize the dissemination, we need to reduce the graph's *connectivity*. The *connectivity* of a graph is determined by the *largest eigenvalue* of its adjacency matrix. Therefore, reducing the leading eigenvalue can minimize the dissemination. In social graphs, the small gap between the largest eigenvalue and the second largest eigenvalue creates a challenge for minimizing the leading eigenvalue. In this work, we propose a scalable algorithm called *MET* (*short for Multiple Eigenvalues Tracking*), which minimize the largest eigenvalue. MET can work well even if the gap between the top eigenvalues is small. We also propose a learning approach called *LearnMelt*, which is useful when the exact topology of graph is not available. We evaluate our algorithms on different types of graphs such as technological autonomous system networks and various social networks.

Acknowledgements

First, I want express my gratitude to my research advisor, Prof. Tina Eliassi-Rad, for her guidance, direction and support. She trained me on how to identify the problems and gave me critical comments about my research. Without her help, I would not have finished this thesis.

I thank my parents, Vinh Le and Kien Nguyen, and my brother, Hoang Le, for their love and encouragement. I also thank my wife, Lan Hoang, for her love, encouragement and support. She always stays besides me to share happiness and difficulties.

I also thank:

- My lab-mates, Sucheta Soundarajan and Priya Govindan, for their discussion about the work,
- Prof. Hanghang Tong, Prof. Christos Faloutsos and Prof. Michalis Faloutsos, for their help and suggestions about my work,
- Prof. Foster Provost, Dr. Lauren Moores, Dr. Prabhakar Krishnamurthy, and Dr. Yun Chi, for mentoring me during my internships at EveryScreenMedia and Yahoo! Lab,
- Prof. Thu Nguyen, my academic advisor, for his guidance and help.

Dedication

To my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Controlling the dissemination of an entity (e.g., a virus or a meme) on a graph is an important problem with a variety of applications in social good, cyber-security, epidemiology, and marketing. In order to control (or manipulate) dissemination, one could add/delete nodes; or alternatively add/delete edges. We are interested in the case of deleting edges in order to minimize dissemination. In a social network, such as the Facebook friendship graph, removing an edge means that we "un-friend" the friendship, while deleting a node means that we need delete a user account.

From past literature [18, 33, 26, 32], we know that the tipping point for an entity's spread on a graph is a function of the connectivity of the graph (which is often approximated by the largest eigenvalue of the adjacency matrix) and the strength of the entity (e.g., the birth and death rates of a virus). In this work, we will assume that we cannot control the intrinsic properties of entities. Therefore, to minimize dissemination on a given graph, we will only focus on ways to modify its connectivity−i.e., which edges to delete.

1.2 Problem Definition

Since we do not have control over the characteristic of the entity, the only possible way to minimize the dissemination on a graph is by modifying its structure. In our work, we consider the problem of removing edges to minimize the largest eigenvalue. Below is the formal definition of our problem:

Problem Definition. Given a graph represented by its adjacency matrix $\mathbf{A}$ and a budget k, select the k edges in $\mathbf{A}$ whose deletions will create the largest drop in the leading eigenvalue of

$\mathbf{A}$.[1]

Finding an optimal solution for the above problem is NP hard [32]. Existing algorithms suffer from some important drawbacks, and so we propose new algorithms called *MET* and *LearnMelt*. For example, existing algorithms such as *NetMelt* [32] perform poorly in social graphs. Figure 1.1 visualizes the list of edges selected by an existing method (*NetMelt*) and our algorithms, *MET* and *LearnMelt*. We see that our proposed methods select edges in various areas of the graph, which lead to higher drop in the leading eigenvalue. In particularly, our algorithms perform well in cases where the gap between the top eigenvalues is small.

The majority of the existing methods require the exact graph topology as their input. Yet, in some scenarios (such as the financial domain), the actual link information might be too sensitive to be accessed by an algorithm. How can we still tell which links to delete without knowing the actual graph topology? An approach that is based on structural features does not directly use the graph's topology. To address this limitation, we also study the following modified problem definition:

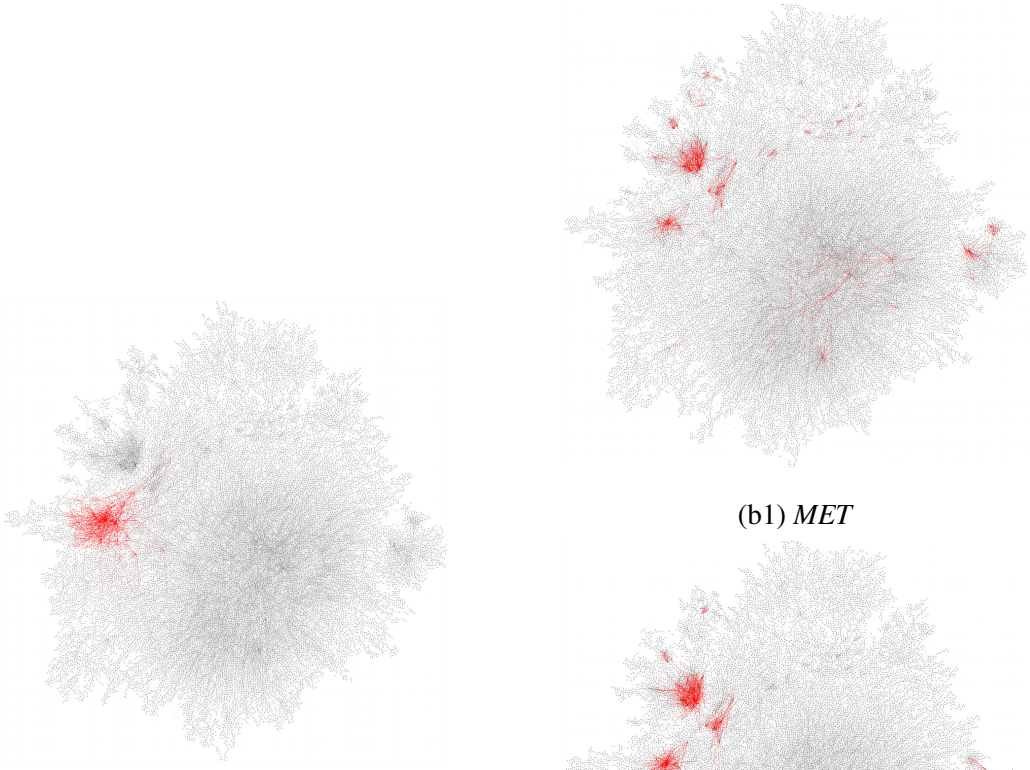
Modified Problem Definition. Given a graph H, can we effectively and efficiently learn a model on H that can predict which k edges to delete on a never-seen-before graph G in order to get the largest drop in the leading eigenvalue of G?

1.3 Thesis Overview

In this thesis, we study an important problem about minimizing the largest eigenvalue of graph. The problem was studied in some previous works (see Chapter 3 for details) but there exists several limitations. We propose novel frameworks which can work well in different types of graphs. Below are our contributions:

- We observe that social graphs have small gaps between their eigenvalues. These small gaps make the problem of estimating the drop in the largest eigenvalue from k edge-deletion difficult.
- We introduce a novel algorithm, called *MET*, which tracks multiple eigenvalues. Our

[1] Both 'largest eigenvalue' and 'leading eigenvalue' refer to the biggest eigenvalue in magnitude.



(b1) *MET*

(b2) *LearnMelt*

(a) An existing method ($NetMelt$) on Yahoo! IM (b) Our algorithms on Yahoo! IM

Figure 1.1: (Best viewed in color.) Comparing the edges selected for deletions (in red color) in Yahoo! IM by an existing method ($NetMelt$ [32]) and our proposed methods (*MET* and *LearnMelt*). $NetMelt$ gets 'stuck' in one area of the graph while *MET* and *LearnMelt* select edges in multiple areas of the graph.

algorithm performs well on different types of graphs, even when the gap between eigenvalues is small.

- We introduce a learning approach, called *LearnMelt*, that learns which edges to delete in a new graph when its exact topology is not available.

- We conduct an extensive empirical study on different real graphs, which shows the *efficacy* and *efficiency* of our methods. The datasets used in our experiment include both technological and social graphs.

The thesis is organized as follows. Chapter 2 describes the background about eigenvalues, eigenvectors and eigen-scores of an adjacency matrix. This chapter also provides the background about structural features of the nodes in a graph. Chapter 3 lists the related works such as information diffusion and applications of structural features. Chapter 4 explains how we can track multiple eigenvalues to minimize the dissemination. We present our learning approach, which uses the structural features of a graph to learn which edges to delete in Chapter 5. In Chapter 6, we evaluate our algorithms and compare with other methods using different datasets. We discuss conclusions and future works in Chapter 7.

Chapter 2

Background

2.1 Notation

Table 2.1 lists the notations used in this thesis. We represent a graph by its adjacency matrix, which is in bold upper-case letter. The bold lower-case letters represent vectors. We use the standard Matlab notation to denote elements in a matrix: $\mathbf{A}(i, j)$ denotes the $(i, j)^{\text{th}}$ element of matrix $\mathbf{A}$; $\mathbf{A}(i, :)$ denotes the i^{th} row of matrix $\mathbf{A}$; and $\mathbf{A}(:, j)$ denotes the j^{th} column of matrix $\mathbf{A}$. We use $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq$ ... to represent the eigenvalues of an adjacency matrix. The left and right eigenvectors of the adjacency matrix are denoted by $\mathbf{u}_i$ and $\mathbf{v}_i$ respectively. We use $\Delta\lambda_{i,e}$ to denote the drop in λ_i due to removal of edge e.

Symbol	**Definition**
$\mathbf{A}$, $\mathbf{B}$, ...	matrices (capital letters in boldface)
$\mathbf{A}(i, j)$	the $(i, j)^{\text{th}}$ element of matrix $\mathbf{A}$
$\mathbf{A}(i, :)$	the i^{th} row of matrix $\mathbf{A}$
$\mathbf{A}(:, j)$	the j^{th} column of matrix $\mathbf{A}$
n	the number of nodes in graph
m	the number of edges in graph
k	the budget (in terms of the number of edges that we can delete)
$\lambda_1, \lambda_2, \lambda_3, ...$	eigenvalues of adjacency matrix: $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq$ ..
$\mathbf{u}_i, \mathbf{v}_i$	the left and right eigenvector corresponding to λ_i
$\Delta\lambda_{i,e}$	the drop in λ_i due to removal of edge e

Table 2.1: Notations used in the thesis.

2.2 Eigenvalues of and Dissemination on Graphs

Minimizing the leading eigenvalue is crucial in minimizing dissemination on a graph. In [6], Chakrabarti et al. showed that the leading eigenvalue of an adjacency matrix is the threshold of an epidemic for more than 25 different virus models [6]. Previous work, such as *NetMelt* [32], relies on a large gap between λ_1 and λ_2, which is not true in many real graphs (especially social graph).

In our work, we propose an algorithm that keeps track of multiple eigenvalues and their corresponding eigen-scores. Our algorithm uses *eigen-scores* of multiple eigenvalues to estimate the effect of removing an edge. The way to compute the eigen-scores of a particular eigenvalue is to use the corresponding left and right eigenvectors of the adjacency matrix. Given an eigenvalue and its corresponding left and right eigenvectors, the *eigen-scores* of an edge (i, j) is equal to the product of the i^{th} and j^{th} element of the left and right eigenvectors. We can use the power iteration method to compute the leading the eigenvalue and its corresponding eigen-scores, which takes $O(m + n)$, where m, n are the number of edges and nodes of the graph, respectively [32]. In order to compute the top-T eigenvalues and their corresponding eigen-scores, we can use the iterative approximate method by Lanczos algorithm [23], which takes $O(mT + nT + nT^2)$.

In social communication graphs (e.g. instant messaging networks), the gap between the largest eigenvalues is often small. Figure 2.1 shows one case of deleting edges based on the eigen-scores of the leading eigenvalue in a Yahoo! IM graph. The drop in the leading eigenvalue is basically constant after a few edges deletions. The figure also shows that the leading eigenvalue drops slowly when λ_1 and λ_2 are equal. When an edge *e* is removed, all eigenvalues are affected. Since the strategy of choosing edges is based on the leading eigenvalue λ_1, the value of λ_1 quickly drops to equal or below λ_2. In this case, the original λ_2 will play the role of the leading eigenvalue. Therefore, the strategy of selecting edges based on the original eigenvalue will not be effective anymore. In social graphs, many eigenvalues could be close to each other, which makes the situation even more complex. In Chapter 6, we compute the top eigenvalues of different graph types, which shows that the gap could be very small. This observation motivates us to develop new algorithms, which can track multiple eigenvalues.

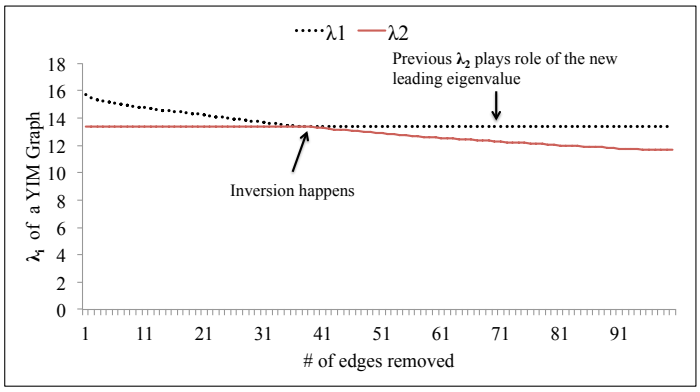


Figure 2.1: The poor performance in minimizing the leading eigenvalue of a social graph (here Yahoo! IM) when using the eigen-scores of the leading eigenvalue in order to pick up the best k edges to delete. Removing an edge will affect the values of all eigenvalues. Selecting the edges based on the highest eigen-scores of the leading eigenvalue makes λ_1 drop quickly while λ_2 drops slowly. When the gap between λ_1 and λ_2 is small, their values may invert as more edges are removed. Therefore, the strategy of selecting the edges based on the original leading eigenvalue will not be effective anymore.

2.3 Structural Features

In some real applications, the exact topology of a graph may be too sensitive to release (such as in the financial domains). In this case, how can pick which edges to delete without knowing the actual graph topology? Our approach, *LearnMelt*, is based on structural features, which does not directly use the graph's topology. Studying the structural features of graph is an emerging field in recent years. For example, structural features are used to measure the network similarity [1], [30]. In this work, we will show that local structural features are useful in our learning approach.

We also evaluate which structural features are important in minimizing dissemination. In particular, degree-based features are more important than clustering-based features. The intuition is that degree-based features have a stronger effect on the number of paths in the graph compared to say clustering coefficient.

Chapter 3

Related Work

To the best of our knowledge, our *MET* is the first algorithm that tracks multiple eigenvalues to minimize dissemination on large graphs, and *LearnMelt* is the first learning approach to minimize the dissemination. In this section, we discuss works related to ours.

3.1 Dissemination and Tipping Point

There has been a lot of research on different cascade models such as Linear Threshold model [19], [20], and Independent Cascade model [11]. One of the most popular models is the SIS model (short for Susceptible-Infected-Susceptible). In the SIS model, the entity can be in either one of two states: *infected* or *susceptible*. The susceptible node will be infected through its infected neighbors at some infection rate. At the same time, an infected node can recover by itself. Due to the popularity of SIS and related models, researchers studied the effect of the topology and found the tipping point of a virus [6], [34], [35], [6]. The epidemic threshold depends on the leading eigenvalue of adjacency matrix. In [27], Prakash et al. proved the epidemic threshold is applicable for 25 different virus models. In [10], Du et al. proposed a randomized algorithm to estimate dissemination in a network. Purohit et al. [29] examined an algorithm to collapse multiple nodes into one single node, which minimizes the change in the leading eigenvalue.

3.2 Extracting Features from Graphs

Extracting features at both the micro- and macro-level of graphs is an important graph mining task. Henderson et al. [16] presented a scalable framework for analysis of volatile graphs by extracting features at multiple levels and multiple resolutions of the graph. Berlingerio et

al. [1] proposed an efficient algorithm which uses a small number structural features to capture similarity between graphs.

3.3 The Importance of Graph Structure to Dissemination

The topology of a graph affects the dissemination in that graph. There are two ways to minimize dissemination in graphs (i) deleting nodes and (ii) deleting edges. Removing a node means that we need to shut down an infected machine in the network or close a user account in a social network. In both cases, the leading eigenvalue of the graph is reduced.

Many previous studies have considered the problem of removing nodes from the graph. Tong et al. [33] selected the best nodes to delete in order to minimize the leading eigenvalue of the adjacency matrix. Chan et al. [7] developed different strategies to measure and minimize the robustness of a graph. In [27], [24], the authors identified the nodes from which the infections started to spread. In [14], Goyal et al. proposed Simpath, an algorithm to maximize dissemination under the linear threshold model. Many previous works studied the importance of nodes in controlling dissemination or removing important nodes [9], [12], [19], [18].

Recently, researchers began studying the problem of modifying edge connections in a graph. In [32], the authors used eigen-scores of the leading eigenvalue to select the best edges to delete, but this method performs poorly in social graphs where the gap between top eigenvalues is small. In [2] and [13], authors learnt important links by studying the propagation logs in the past. The important links are the ones that most likely can explain propagation in a social network. These works examined the influence probabilities from available propagation events since the social network connections are not associated with probabilities. The influence probability is dynamic and can change over time. Our work uses the topology of the graph, rather than the propagation logs. Kuhlman et al. [21] studied the way to block simple and complex contagions in ratcheted dynamical systems, where an infected node cannot recover. In simple contagion, a node is infected by contacting with another infected node, while complex contagion requires interaction with at least two infected nodes. They showed that simple contagion is an easier task in general.

3.4 Spectra of Graphs

Spectra of a graph is an important subject which studies the relation between the topology and the eigenvalues of a graph. Spectra of a graph gives information about the graph such as its connectivity, robustness, and randomness [8], [4]. Stewart et al. [31] studied the perturbation matrix to see the behavior of a particular function when the matrix changes. One emerging field is analyzing the spectra of graph by using only the local structural information [37], [28].

3.5 Transfer Learning

Effective transfer learning between graphs is important. Pan et al. [25] presented a detailed survey of current progress on transfer learning for classification, regression and clustering problems. Transfer learning is an active field and is used widely in graph mining in recent years. Role transfer is adopted to determine similarity between graphs [17]. Wang et al. [36] measured matrix similarity to transfer across networks. Cao et al. [5] have shown that transfer learning across graphs achieves high accuracy in collective link prediction. Transfer learning shows a lot of potential for solving challenging problems in graph mining.

Chapter 4

Multiple Eigenvalues Tracking (MET)

4.1 Proposed Method

In social graphs, the gaps between the top eigenvalues are normally small. In order to minimize the largest eigenvalue, we should track several top eigenvalues at the same time. In this chapter, we introduce the Multiple Eigenvalues Tracking (*MET*) algorithm.

As mentioned before, all eigenvalues are affected when an edge is removed from a graph. When the graph has small eigen-gaps,[1] algorithms that estimate *only* the leading eigenvalue can exhibit significant performance degradations. One can alleviate these shortcomings in two ways: (1) track/estimate more than one eigenvalue, and (2) recompute eigenscores of the remaining edges periodically. In the former, the question becomes how many eigenvalues should one track. In the latter, the question becomes how often should one recompute eigenscores. Bad answers to these questions can adversely affect the efficiency and efficacy of one's algorithm.

Algorithm 1 describes our *MET* algorithm. *MET* provides satisfactory answers to both of the aforementioned questions. **Tracking more than one eigenvalue.** *MET* determines the number of eigenvalues T to track *adaptively*. It starts by tracking the top-2 eigenvalues (i.e., $T = 2$) and modifies T only when the gap between λ_1 and λ_T changes. If the gap decreases, then *MET* increases T by 1; if the gap increases, then *MET* decreases T by 1; otherwise, T does not change. In *MET*, the minimum value for T is 2. Note that a large T value can cause unnecessary tracking of too many eigenvalues and lead to increased runtime. **Recomputing eigenscores periodically.** *MET* uses λ_T as a threshold for deciding when to recompute eigenscores. Specifically, when all estimated λ values (i.e., $\lambda_1, \ldots, \lambda_{T-1}$) fall below the threshold λ_T, then *MET* recomputes the eigenscores. Since a small T value can cause

[1] Figure 6.3 shows that the eigen-gap between the top-21 eigenvalues can be as small as 0.05 in a social graph (such as Facebook user-postings).

unnecessary recomputation of eigenscores (leading to a longer runtime), *MET* uses a slack variable ϵ, whereby *MET* recomputes the eigenscores only if all estimated λ values are less than $\lambda_T - \epsilon$. A nonzero ϵ is useful when all the top T eigenvalues are very close to each other. We evaluate the effects of various ϵ values in Chapter 6.

4.2 Complexity

Next, we analyze the runtime complexity and space cost of our *MET* algorithm. Recall n and m are the number of nodes and edges in the graph, respectively.

Lemma 4.2.1. *The runtime complexity for* MET *is* $O(k_1\bar{T} \times (m + n\bar{T}) + k \times (m + \bar{T}))$. *The space cost for* MET *is* $O(m \times (1 + \bar{T}))$. n *and* m *are the number of nodes and edges in the graph, respectively;* k *is the budget for edge deletions;* k_1 *is the number of times that* MET *recomputes eigenvalues (*$k_1 < k$*); and* $\bar{T}$ *is the average number of eigenvalues that* MET *tracks.*

Proof. In Algorithm 1, the following lines do not take $O(1)$. The $while$ loop on Line 4 is executed on average k_1 times, which is also the number of times that *MET* recomputes eigenvalues and eigenscores. Line 5 computes the average $\bar{T}$ eigenvalues and eigenvectors using Lanczos algorithm [23]; this takes $O(m\bar{T} + n\bar{T}^2)$. Line 6 computes eigenscores, which takes $O(m\bar{T})$. Line 9 takes $O(\bar{T})$ to find the maximum λ in the estimated eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_{\bar{T}}\}$. Let k_2 denote the average number of iterations of the $while$ loop on Line 10. (Note that the edge-deletion budget k is equal to $k_1 \times k_2$.) Line 11 takes $O(m)$ to find the edge with the highest eigenscore. The for loop on Lines 13 through 16 takes $O(\bar{T})$. Similar to Line 9, Line 17 takes $O(\bar{T})$ to find the maximum λ. Thus, the $while$ loop covering Lines 10 through 18 takes $O(k_2 \times (m + \bar{T} + \bar{T})) \approx O(k_2 \times (m + \bar{T}))$. Putting all of this together, we get that the total runtime of *MET* is $O(k_1 \times (m\bar{T} + n\bar{T}^2 + m\bar{T} + \bar{T} + k_2 \times (m + \bar{T}))) \approx O(k_1 m\bar{T} + k_1 n\bar{T}^2 + k_1 m\bar{T} + k_1\bar{T} + k \times (m + \bar{T})) \approx O(k_1\bar{T} \times (m + n\bar{T} + m + 1) + k \times (m + \bar{T})) \approx O(k_1\bar{T} \times (m + n\bar{T}) + k \times (m + \bar{T}))$.

MET needs $O(m)$ to store the graph and $O(k)$ to store the deleted edges. On average, *MET* tracks $\bar{T}$ eigenvalues and their corresponding eigenscores, this requires $O(m\bar{T})$ space. Thus,

Algorithm 1 MET Algorithm

Input: A graph represented by its adjacency matrix **A** and a budget k

Output: The list of k edges to be deleted from **A**

$E \leftarrow \{\}$

$T \leftarrow 2$

$previousGap \leftarrow \infty$

while ($|E| < k$) **do**

  Compute the top-T eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_T$; and the corresponding left $\mathbf{u_i}$ and right $\mathbf{v_i}$ eigenvectors for $i = 1, 2, \ldots, T-1$.

  $Eigenscores[e, i] \leftarrow getEigenscore(e, \lambda_i, \mathbf{u_i}, \mathbf{v_i})$,

  for each edge e and for $i = 1, 2, \ldots, T-1$; see Chapter 2.

  $currentGap \leftarrow \lambda_1 - \lambda_T$

  $threshold \leftarrow \lambda_T - epsilon$

  $\lambda_{max} \leftarrow max\{\lambda_1, \lambda_2, \ldots, \lambda_{T-1}\}$

  while ($\lambda_{max} > threshold$ **and** $|E| < k$) **do**

    $e_{max} \leftarrow$ edge with the maximum eigenscore under λ_{max} and its corresponding left and right eigenvectors.

    $E \leftarrow E \cup e_{max}$; and remove e_{max} from **A**.

    for $i = 1, 2, \ldots, T-1$ **do**

      $\lambda_i \leftarrow \lambda_i - Eigenscores[e_{max}, i]$ // Update λ_i

      $Eigenscores[e_{max}, i] \leftarrow -1$ // Mark e_{max} as deleted

    end for

    $\lambda_{max} \leftarrow max\{\lambda_1, \lambda_2, \ldots, \lambda_{T-1}\}$

  end while

  if ($currentGap < previousGap$) **then**

    $T \leftarrow T + 1$

  else if ($currentGap > previousGap$ **and** $T > 2$) **then**

    $T \leftarrow T - 1$

  end if

  $previousGap \leftarrow currentGap$

end while

return E

MET requires $O(m + k + m\bar{T})$ storage. Since $k << n$, the total storage cost for *MET* is $O(m \times (1 + \bar{T}))$.

□

In the runtime complexity of *MET*, there are terms $\bar{T}^2$ and k_1. Table 6.2 shows that $\bar{T}$ and k_1 are very small. In addition, $k << m$. Thus, the runtime of *MET* is linear with the size of graph.

MET-Naive. If one happens to know how many eigenvalues T to track *a priori*, then *MET* does not need to adaptively track multiple eigenvalues. We have a naive version of *MET*, called *MET-naive*, where k edges are sequentially deleted based on the largest estimated λ_i in $i = 1, 2, \ldots, T$. Similar to *MET*, all estimated λ values are updated after each edge deletion (Line 15 in Algorithm 1) and the eigenscore of the deleted edge is marked so that edge is not selected again (Line 16 in Algorithm 1).

MET-Naive tracks a constant number of eigenvalues T and does not recompute eigenscores. Its runtime cost is $O(T \times (m + nT) + k \times (m + T))$. For brevity, we have omitted the proof for it here. Similar to *MET*, the space cost for *MET-Naive* is $O(m \times (1 + T))$.

See Appendix A for a discussion of how far *MET*'s results are from the upper-bound on the optimal solution that minimizes the largest eigenvalue of a graph.

Chapter 5

Learning to Minimize Dissemination (LearnMelt)

5.1 Proposed Method

Most of the existing methods need the exact topology of a graph to find the k edges to delete. This might not be feasible in some domains, such as finance. We develop a method, called *LearnMelt* that only uses structural features of a graph to learn which edges to delete in a never-before-seen graph. Structural features are useful in measuring the similarity of graphs [1]. In our work, we want to understand whether structural features are useful in learning dissemination or not. Furthermore, we also want to measure which features are more important in our learning approach.

Given a graph G, *LearnMelt* uses its trained model to predict which k edges to delete for the maximum drop in the leading eigenvalue. So, the key to *LearnMelt*'s success is how well it can generalize across various graphs. This section is divided into four parts: (1) obtaining training data; (2) structural features used in *LearnMelt*; (3) learning an eigen-drop model on structural features (i.e., the *LearnMelt* model); (4) inference on the learnt eigen-drop model; and (5) computational complexity of *LearnMelt*.

5.1.1 Obtaining Training Data

Like all supervised learning models, *LearnMelt* needs labeled data to train a model. *LearnMelt*'s labeled data is of the form $\langle X, y \rangle$, where each instance in X represents an edge e. X is a $1 \times 2p$ dimensional vector, which encodes the features of the two endpoints of e. In *LearnMelt*, p corresponds to the seven local structural features extracted by the $NetSimile$ algorithm [1]. We can easily add other features into our framework. We describe these features in detail below.

The target (a.k.a. response variable) y is a real number. For an edge e in the labeled set, y is

set to the actual eigen-drop for e. We ran experiments with y being set to the edge's eigen-score, but those response variables were too noisy on social graphs.

Given a training graph H and a training set of size t, we use $NetMelt^+$ to select $\frac{t}{2}$ edges from H. These edges correspond to 'good' examples of edges with large eigen-drops. For the remaining $\frac{t}{2}$ edges, we randomly select an edge and compute its eigen-drop. When a randomly selected edge is added to the training set, it is not removed from the graph. These random edges provide diversity to the training set. The cost of obtaining labeled data is a one-time cost. As we show in Section 6.3.4, *LearnMelt* performs well with relatively small train sets (e.g., 1000 edges).

5.1.2 LearnMelt's Structural Features

We represent an edge $e : (u, v)$ with the structural features of its endpoints (i.e., features of u, and v). In particular, we utilize these seven local structural features of an endpoint node i:

- Number of neighbors (i.e., degree) of node i
- Average degree of node i's neighbors
- Clustering coefficient of node i[1]
- Average clustering coefficient of node i's neighbors
- Number of edges in node i's egonet (i.e., its 1-hop induced subgraph)
- Number of edges leaving from node i's egonet
- Number of neighbors of node i's egonet

Berlingerio et al. [1] showed that these seven features correspond to four commonly used social theories−namely, Social Capital, Social Exchange, Structural Holes, and Balance. *LearnMelt* can easily incorporate additional features. In Section 6.3.4, we discuss results of our feature selection experiments.

[1]We define the clustering coefficient of a node i as the number of triangles connected to i divided by the number of the connected triples centered at i.

5.1.3 Learning LearnMelt's Eigen-drop Model

Any regression model can be used for *LearnMelt* (since the target variable y is numeric). We use an ensemble approach−namely, random forests [3] with regression trees. Random forests have many advantages. First, they use *bagging*, which is known to alleviate overfitting (by reducing variance). Second, they conduct feature selection as part of the training process. Third, they are fast in both training and inference.

5.1.4 Inference on LearnMelt

Algorithm 2 provides the pseudo-code for inference on *LearnMelt*. The inputs to this inference algorithm are (1) a never-before-seen graph, (2) the number of edges k to be deleted, and (3) the learnt random-forest model (described in Section 5.1.3). *LearnMelt* iterates over all the edges in the given graph and predicts the drop in $\Delta\lambda$ for each edge deletion. It returns the k edges with the largest $\Delta\lambda$.

Algorithm 2 The *LearnMelt* Inference Algorithm

Input:

- A graph represented by adjacency matrix **A**
- Budget k
- The learnt random-forest model *M* (see Section 5.1.3)

Output: The list of k edges to be deleted from **A**

1: **F** ← structural features extracted from **A**
2: **for** (each edge $e : (i, j)$ in **A**) **do**
3: $f_e \leftarrow$ [**F**(i, :), **F**(j,:)] //feature vector for each e
4: $\Delta\lambda_e \leftarrow M(f_e)$ //Use model M to predict $\Delta\lambda$ for e
5: **end for**
6: $E \leftarrow$ the top-k edges having highest predicted $\Delta\lambda_e$ //$\Delta\lambda_e$ is predicted in Line 4
7: **return** E: list of edges to be removed

5.2 Complexity

Step 1 of Algorithm 2 computes the structural features for the nodes. This is an *almost* linear time algorithm on the number of edges, taking *almost* $O(n \log n)$, where n is the number of nodes [1]. Computing $\Delta\lambda$ in Step 2 takes $O(m)$, where m is the number of edges. Finding k edges with the highest predicted values in Step 3 takes $O(mk)$. Therefore, the total computational cost for *LearnMelt* is *almost* $O(n \log n + mk)$−i.e., *LearnMelt*'s inference is *almost* linear on the number of edges.

Chapter 6

Evaluation

In this chapter, we evaluate our algorithms and compare them with existing methods. First, we present the datasets used in our experiments. Next, we describe the experiment settings, our results, and a discussion.

6.1 Datasets

Table 6.1 lists the graphs used in our experiments. Our graphs are undirected and unweighted. We have four sets of real graphs, which include:

- **Oregon Autonomous System (AS):**[1] We have four technological graphs, each corresponding to a week's worth of data. A node here is an autonomous system. An edge is a connection inferred from the Oregon route-views.[2]

- **Yahoo! Instant Messenger (YIM):**[3] We have three social graphs, each corresponding to a day's worth of data. The data was collected in April 2008. Our graphs are from the 1^{st}, 11^{th}, and 21^{st} of the month. A node is a Yahoo! IM user. An edge is a communication between two users during the selected day.

- **Facebook user-postings (FB):**[4] We have five social graphs, each corresponding to a day's worth of data. The data was collected in March 2013. We used the 10^{th} to the 14^{th} of the month in our study. A node is a Facebook user. An edge is a 'posting' event (where

[1] `http://topology.eecs.umich.edu/data.html`

[2] In [32], AS graphs were treated as directed graphs. Thus, the number of edges reported there is twice the number reported here.

[3] `http://webscope.sandbox.yahoo.com/`

[4] Proprietary data given to us by a collaborator.

$user_i$ generates a post and $user_j$ sees that post) in the selected day. For our experiments, we convert this directed graph into an undirected one.

- **Twitter re-tweet (TT)**: We have four social graphs, each corresponding to a month's worth of data. The data was collected from May to August in 2009. A node represents a twitter user. There is an edge between two users if there is a re-tweet event between them. A detail description of this dataset is available in [38].

Dataset	Time Span	# of Nodes (n)	# of Edges (m)	Avg. Degree	# of ConComps	% of Nodes in LCC	Avg Dis in LCC
Oregon-1	7 days	5,296	10,097	3.81	1	100%	3.5
Oregon-2	7 days	7,352	15,665	4.26	1	100%	3.5
Oregon-3	7 days	10,860	23,409	4.31	1	100%	3.6
Oregon-4	7 days	13,947	30,584	4.39	1	100%	3.6
YIM-1	1 day	50,492	79,219	3.14	2802	54.5%	16.2
YIM-2	1 day	56,454	89,269	3.16	2650	61.2%	15.9
YIM-3	1 day	31,420	39,072	2.49	2820	62.9%	13
FB-1	1 day	27,165	26,231	1.93	2081	50.4%	9.4
FB-2	1 day	29,556	29,497	1.99	1786	57.8%	9.3
FB-3	1 day	29,702	29,384	1.98	1902	55.6%	9.4
FB-4	1 day	29,442	29,505	2.00	1746	58.1%	8.9
FB-5	1 day	29,553	29,892	2.02	1624	59.8%	8.8
Twitter-1	1 month	25,799	16,410	2.18	2899	63.6%	7.5
Twitter-2	1 month	39,477	45,149	2.29	3827	68.9%	7.3
Twitter-3	1 month	57,235	68,010	2.38	4828	73.4%	7.5
Twitter-4	1 month	77,589	96,980	2.50	5665	77.1%	7.6

Table 6.1: Graphs used in our experiments. LCC stands for the largest connected component and ConComps stands for connected components. "Avg Dis" is the average number of hops between two randomly selected nodes.

6.2 Experiemental Setup

We compare our algorithms with the following eight methods:

1. *NetMelt*: edges are selected based on eigen-scores of the leading eigenvalue [32].
2. *NetMelt+*: an improved version of *NetMelt*; it re-computes the eigen-scores after each edge deletion.
3. *Rand*: edges are selected randomly.
4. *Rich-Rich*: edges are selected based on the highest $d_{src}.d_{dst}$, where d_{src} and d_{dst} are the degrees of the source and destination nodes [7].
5. *Rich-Poor*: edges are selected based on the highest $|d_{src} - d_{dst}|$.
6. *Poor-Poor*: edges are selected base on the lowest $d_{src}.d_{dst}$.
7. *EBC*: edges are selected based on the highest edge betweenness centrality. The EBC of an edge is the number of shortest paths that pass through that edge.
8. *Miobi*: edges are selected to minimize the robustness of graph [7]. In [7], the robustness of graph is defined as the "average" of the top K eigenvalues of the adjacency matrix.

In all graphs, *Rich-Poor* and *Poor-Poor* perform poorly compared with *Rich-Rich*. Thus, we only present the results of *Rich-Rich* for brevity.

6.2.1 The Settings of *MET*

The default parameter of $epsilon$ is 0.5 in *MET* algorithms. In *MET-Naive*, the default value of T is 10 when we do not mention the value of T explicitly. As we mentioned, MET does not need the parameter T to be specified, because it adaptively selects T. We also evaluate the effect of selecting different values of $epsilon$ in Figure 6.4.

6.2.2 The Settings of *LearnMelt*

Our *LearnMelt* does not compute eigen-scores on every graph. Instead, *LearnMelt* uses labeled data from one graph in order to learn a model that predicts the eigen-drop of a set of edges in

never-seen-before graphs. For the experiments reported here, the training and test data come from the same set of graphs. Recall that we have four sets of graphs: Oregon (with 4 graphs), YIM (with 3 graphs), FB (with 5 graphs), and TT (with 4 graphs).

As described in Section 5.1.1, our labeled dataset is of the form $\langle X, y \rangle$, where each instance in X represents an edge by the structural features of its endpoints. The variable $y \in \mathbb{R}$ is the target (a.k.a. response) variable, which is the actual eigen-drop of the edge. For our experiments, we used 2K labeled edges: 1K of which were selected by $NetMelt^{+}$ and the other 1K were selected at random. *LearnMelt* is effective with a much lower number of labeled edges. For example, in Figure 6.5 we show results with only 1K edges in the labeled set.

LearnMelt uses an ensemble approach−namely, a random forest of 100 regression trees. The number of randomly selected features is $log_2 14$ (where 14 is number of structural features of each edge), and the minimum number of samples required to split an internal node is 2. With a random forest of regression trees, we get (1) fast training and inference runtimes, (2) nonlinear decision boundaries, (3) built-in feature selection, and (4) ease overfitting.

6.2.3 Evaluation Criteria

One of our evaluation criteria is *efficacy*, which is the relative drop of the leading eigenvalue: $\%drop = 100\frac{|\bar{\lambda_1} - \lambda_1|}{\lambda_1}$, where λ_1 is the leading eigenvalue of the original graph and $\bar{\lambda_1}$ is the leading eigenvalue of the graph after removing k edges. The higher the percentage drop in the leading eigenvalue, the better the algorithm's performance.

We also evaluate the *efficiency* of our algorithms by measuring the wall clock time. Obviously, lower runtime is better. All experiments were conducted on a Macbook Pro with CPU 2.66 GHz, Intel Core i7, RAM 8 GB DDR3, hard drive 500 GB SSD, and OS X 10.8.

6.3 Results and Discussion

6.3.1 Comparing the *Efficacy* of Different Methods

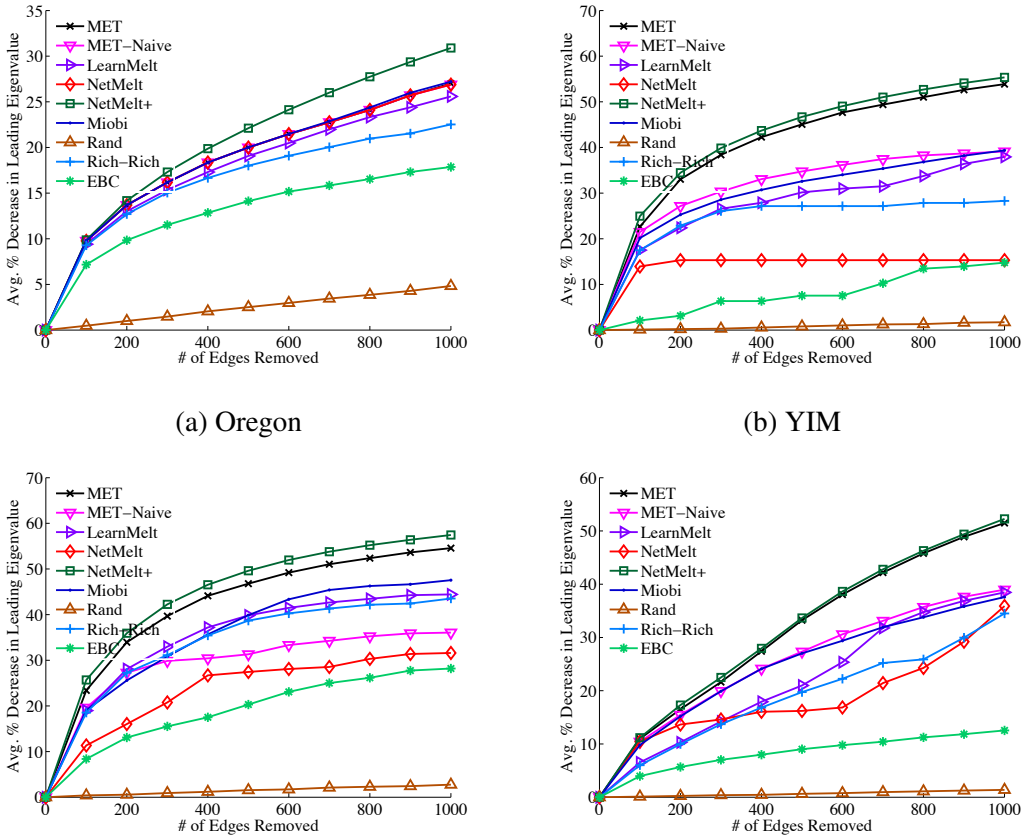
Figure 6.1 compares the *efficacy* of the different methods when we change the budget k from 0 to 1000 edges. *MET-Naive* outperforms *NetMelt* but it is still below *NetMelt+*. *NetMelt+* always creates the highest eigen-drop but it is not scalable (we will discuss the trade-off between *efficacy* and *runtime* in Section 6.3.2). *MET* performs well on all graphs and achieves similar *efficacy* to *NetMelt+*. *NetMelt+* achieves the highest *efficacy* because it updates the topology and the eigens-cores after each deletion. Thus, it can select the next best edges to delete. Even though it is intractable to calculate the optimal solution, we expect that *NetMelt+* and *MET* are very close to optimal solution. We see that *NetMelt* performs very well in Oregon graph due to the large gap between eigenvalues. Our algorithms, *MET-Naive* and *MET*, also perform well in these technological graphs and have a *runtime* similar to *NetMelt*. When the exact topology of graph is not available, *LearnMelt* achieves the highest *efficacy*. The other methods which do not use the full topology are *Rich-Rich*, *Rich-Poor*, *Poor-Poor*. [5]

6.3.2 Trade-off Between *Efficacy* vs. *Efficiency*

Figure 6.2 examines the trade-off between *efficacy* and *efficiency* (in terms of *runtime*). *NetMelt* is a fast algorithm, but it performs poorly in social graphs. *NetMelt+* is very slow. *MET* achieves similar *efficacy* to *NetMelt+* while having a low *runtime*. For example, *MET* is 108 times faster than *NetMelt+* in the Yahoo! IM graphs, while preserving 99% of the *efficacy*. In the FB graphs, *MET* is 12 times faster than *NetMelt+* while preserving 96% of the *efficacy*. In the TT graph, *MET* is 30 times faster than *NetMelt+* while preserving 99.3% of the *efficacy*.

In Figure 6.2, there are two methods which are applicable when the exact topology of graph is not available. They are *LearnMelt* and *Rich-Rich*. Both method are scalable. Our method is a learning approach while *Rich-Rich* deletes the edges based on the degrees of the edges' endpoints. We see that our method always achieves higher drop of the leading eigenvalue than *Rich-Rich*. Notice that the runtime of *LearnMelt* does not depend on the the budget k, since

[5] *Rich-Poor*, and *Poor-Poor* always achieve lower *efficacy* than *Rich-Rich*. We omitted for brevity.



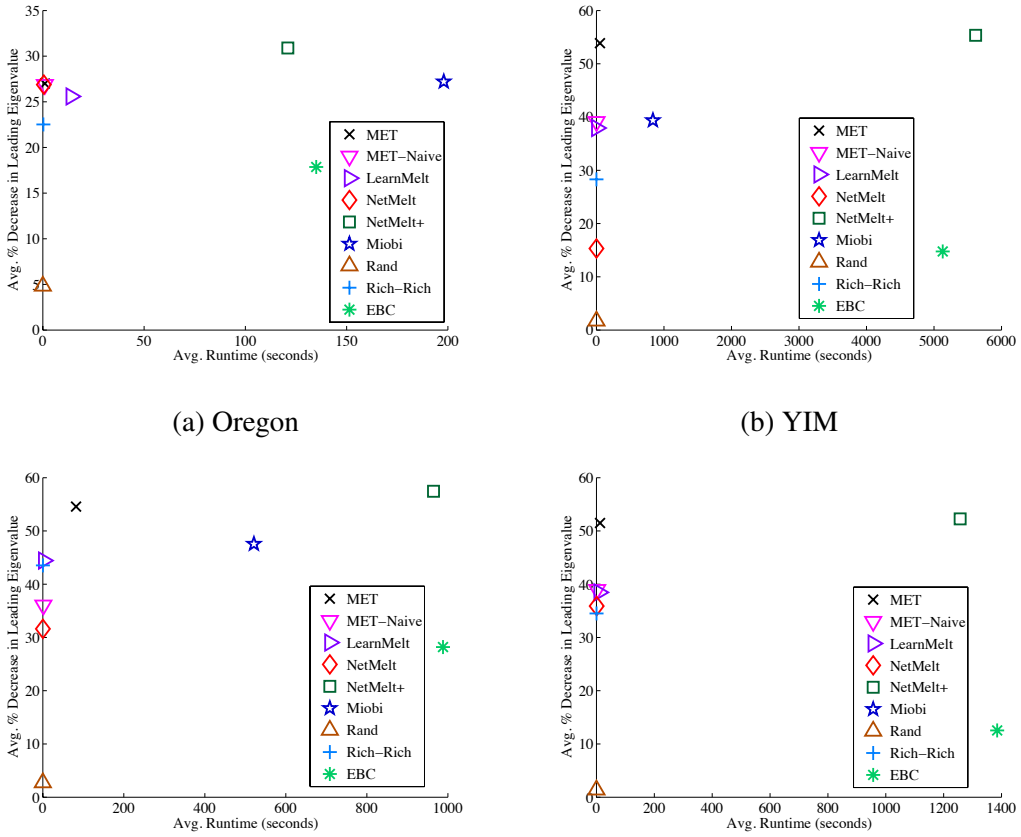
(a) Oregon

(b) YIM

(c) FB

(d) TT

Figure 6.1: (Best viewed in color.) Comparing *efficacy* (as measured by the percentage decrease in the leading eigenvalue λ) vs. budget k (i.e., the number of edges to delete). The larger the drop, the better the performance. *NetMelt* performs poorly in all social graphs, especially YIM and FB. *NetMelt+* has the best performance but it is very slow (see Figure 6.2). Our proposed method *MET-Naive* can beat *NetMelt*. *MET* performs well across both technological and social graphs. In social graph, *MET* is close to *NetMelt+*. When the exact topology of the graph is not available, *LearnMelt* always achieves the highest *efficacy*. (Recall that *Rich-Rich* also does not use the full topology of graph.)



(a) Oregon

(b) YIM

(c) FB

(d) TT

Figure 6.2: (Best viewed in color.) Trade-off between *efficacy* and *efficiency* when removing 1000 edges from our graphs. Higher percentage drop of the leading eigenvalue and lower runtime is better (i.e, the upper left corner of the plots). We depict *efficacy* as the average of percentage drop in the leading eigenvalue and the average runtime for all graphs of the same type. *MET* and *NetMelt+* achieve similar percentage drop of the leading eigenvalue, but *NetMelt+* is much slower (e.g., *NetMelt+* is 100 times slower than *MET* in YIM graph.). When we cannot access the full topology of graph, we can use *LearnMelt*, which is a learning approach and is scalable.

LearnMelt estimates the eigen-drop of all edges and picks the edges with highest estimated eigen-drop. Thus, *LearnMelt* is recommended when we can only access some structural features of the graph.

6.3.3 Performance Analysis of MET

6.3.3.1 Re-computation in MET

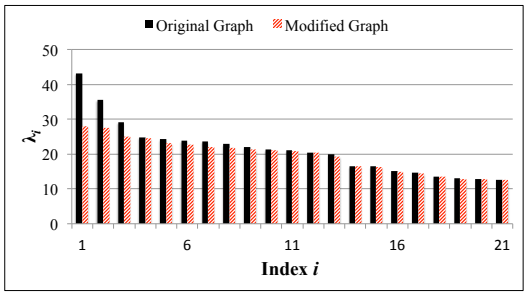
We also measure how many times *MET* re-computes the eigen-scores and the average number of eigenvalues *MET* tracks. Table 6.2 lists the average values of different datasets. We observe that *MET* needs to track a few eigenvalues with a small number of re-computations, which makes *MET* much faster than *NetMelt+*. In the FB graph, we need to track more eigenvalues and the re-computation is more frequent than in the YIM and TT graphs. The reason for this is that the gap between top eigenvalues in the FB graph are much closer than the gap in the YIM and TT graphs.

Dataset	**Avg. # eigenvalues being tracked**	**Avg. # of re-computations**
Oregon	2.08	1.25
YIM	5.17	10.3
FB	6.83	24.6
TT	3.03	9.4

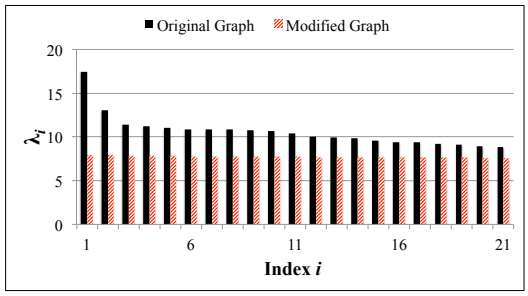
Table 6.2: Average number of eigevanlues being tracked in *MET* and average number of times *MET* re-computes eigen-scores when removing $k = 1000$ edges from each graph. *MET* tracks small numbers of eigenvalues and the re-computations are not often, which makes *MET* a fast algorithm.

6.3.3.2 The Motivation for Introducing the Slack Variable $epsilon$ in *MET*

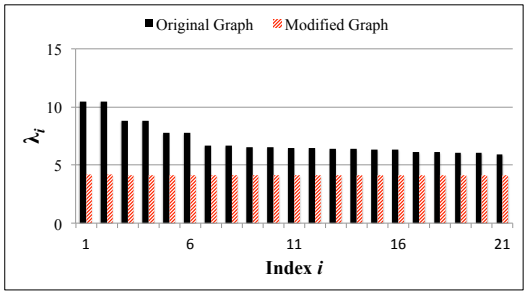
Figure 6.3 plots the top eigenvalues of the original graph, and after *MET* removes 1000 edges. The gaps between λ_1 and λ_{21} are as small as 0.37 in Yahoo! IM, 0.36 in TT graph, and 0.05 in FB graph. When this situation happens, the re-computation condition is triggered too



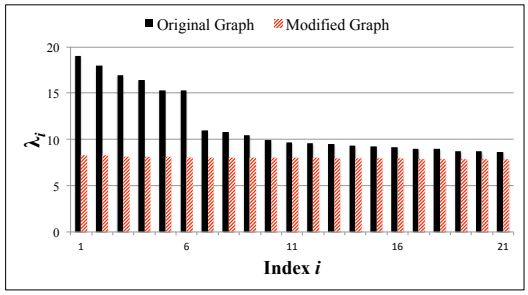
(a) λ's of Oregon-1



(b) λ's of YIM-1



(c) λ's of FB-1



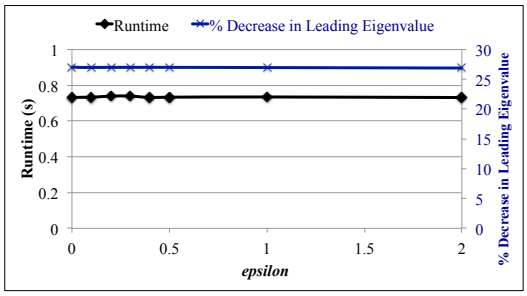
(c) λ's of TT-1

Figure 6.3: Top eigenvalues of the "Original graph" and "Modified Graph" (i.e., the graph after removing 1000 edges selected by *MET*). The top eigenvalues in social networks are "very close" after removing 1000 edges using *MET*. After removing the edges, the gaps between λ_1 and λ_{21} in *YIM-1*, *FB-1*, and *TT-1* are 0.37, 0.05, and 0.36 respectively.

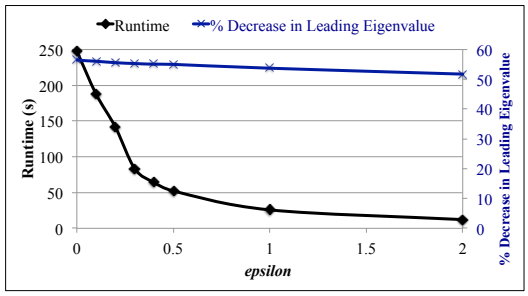
often (i.e., $max(estimated\lambda_i) < \lambda_T$ after few deletions). One possible solution is to track even more eigenvalues, which also increases *runtime*. We allow trade-off between *efficacy* and *runtime* by introducing a slack variable $epsilon$. Figure 6.4 shows the effect of increasing $epsilon$. In practical applications, one may allow a small drop in the *efficacy* to improve the *runtime*. In order to achieve the highest *efficacy*, we simply set $epsilon$ to 0. In this case, *MET* can preserve more than 99% of *efficacy* compared with *NetMelt+* in all social graphs.

6.3.4 Performance Analysis of LearnMelt

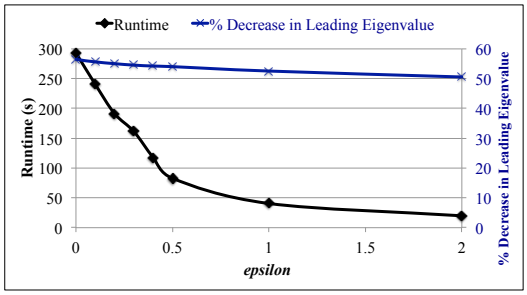
We also investigated how much the performance of *LearnMelt* drops when the size of the training set is reduced by a half (i.e., going from 2K edges in the training set to 1K edges). For this experiment, we defined *Efficacy Drop* as follows. Let $\Delta\lambda_1$ be the average eigen-drop when



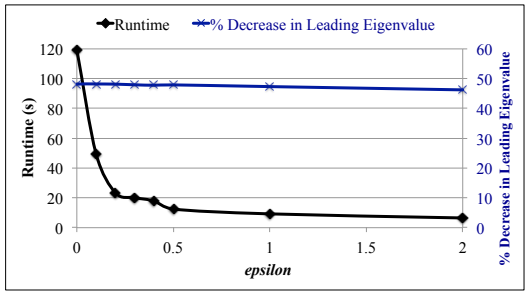
(a) Oregon



(b) YIM



(c) FB



(c) TT

Figure 6.4: The effects of selecting $epsilon$ values on *efficacy* and *runtime* in *MET*. The *runtimes* and *percentage drops in the leading eigenvalue* are the averages over graphs of the same type. Smaller $epsilon$ values create higher eigen-drops but also higher *runtimes*. Note that when $epsilon$ equals to 0, the runtimes are still lower than *NetMelt+* (i.e., *MET* is 23 times faster in Yahoo! IM graph, 4 times faster in FB graphs, and 11 times faster in TT graphs while preserving 99.9%, 99.2%, and 99.9% *efficacy* respectively). The performance of *MET* on Oregon graphs is not affected by the value of $epsilon$ due to the large gap between the largest eigenvalues (i.e., $epsilon$ does not trigger re-computation).

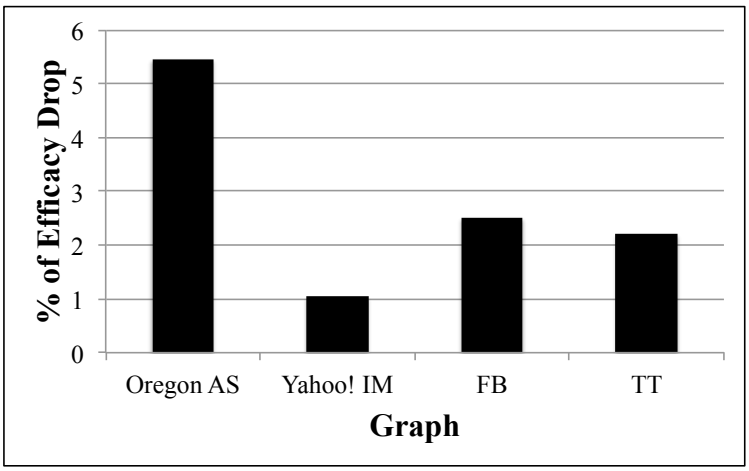


Figure 6.5: The percentage *efficacy* drop of *LearnMelt* when reducing the size of training set by a half. We define *efficacy* drop as the relative change in performance. The decrease in performance is no more than 6% when we reduce the training set from 2K to 1K edges. For these experiments, we used a budget of k=1000.

using the default setting and $\Delta\lambda_2$ be the average eigen-drop when using half of the training set. Then, *Efficacy Drop* equals $\frac{\Delta\lambda_1 - \Delta\lambda_2}{\Delta\lambda_1}$. Figure 6.5 shows that the *Efficacy Drop* is less than 6% across all sets of graphs. Thus, reducing the size of the training set significantly (e.g., by a half) does *not* significantly affect the *efficacy* (as measured by the decrease in the leading eigenvalue).

6.3.4.1 *Efficacy* vs. Network Similarity

In *LearnMelt*, we build a model on a training graph and use it on never-seen-before graphs. Here, we evaluate the relationship between the similarity of the training and test networks vs. the *efficacy* of the learnt model in *LearnMelt*. We use NetSimile [1] to measure similarity between graphs. NetSimile uses structural features of nodes to build a 'signature' vector for a graph. Then, it uses Canberra Distance [6] to measure how different two graphs are. A Canberra Distance of zero indicates perfect similarity.

Figure 6.6 reports the *efficacy* of *LearnMelt* (again in terms of the percentage decrease in

[6] $Canberra(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} \frac{|p_i - q_i|}{|p_i| + |q_i|}$, where n is the number of dimensions in $\mathbf{p}, \mathbf{q}$ [22].

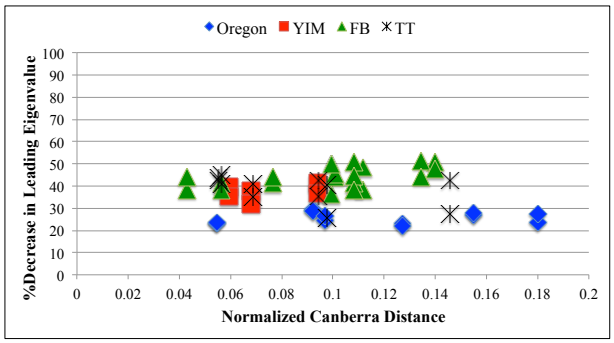
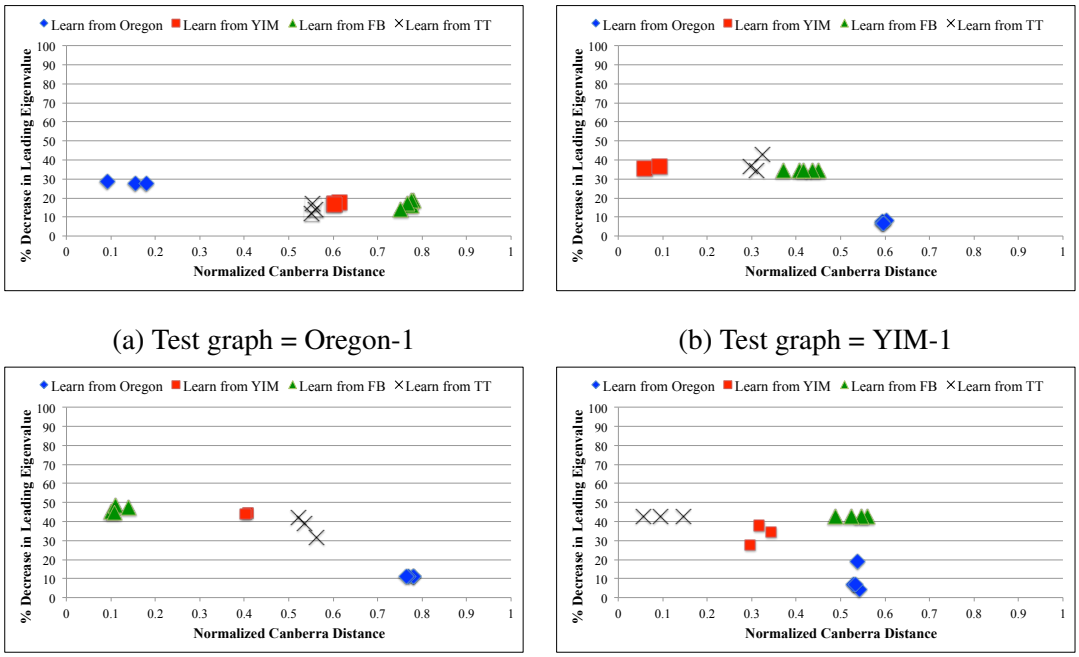


Figure 6.6: (Best viewed in color.) *efficacy* of *LearnMelt* vs. the normalized Canberra distance of train-test graph pairs in the same domain (k=1,000). The learnt model in *LearnMelt* is able to maintain their *efficacy* even when the distance between the train-test graph pairs increases from 0.05 to 0.2. In our graphs, the Canberra distances of train-test graphs in the same set (e.g., FB) are less than 0.2.

the leading eigenvalue) vs. the normalized Canberra Distance of the train-test graph pairs across our four sets of graphs. We used a budget of k=1000 for these experiments. We observe that across our four sets of graphs, the learnt model in *LearnMelt* is able to maintain its efficacy even as the normalized Canberra Distance increases between the train-test graph pairs. That is, *LearnMelt* is indeed generalizing across graphs in the same group. We observe that this generalization is better maintained in social graphs than in technological graphs.

Figure 6.7 reports the *efficacy* of *LearnMelt*'s inference on Oregon-1, YIM-1, FB-1, and TT-1 graphs when it is trained across different graph sets. In Figure 6.7(b), (c), and (d), we observe that *LearnMelt* maintains its *efficacy* when it is trained on Facebook, Twitter and tested on YIM-1 (and vice versa); but *LearnMelt*'s performance drops when it is trained on the Oregon AS graph and tested on a social graph such as YIM-1, FB-1 or TT-1. Similarly, Figure 6.7(a) shows that *efficacy* of *LearnMelt* on Oregon-1 decreases when it is trained on Yahoo! IM, Twitter or Facebook graphs. This is not surprising. The structures of technological graphs are very different from social graphs, which make transfer-learning approaches such as *LearnMelt* difficult across unrelated domains.



(a) Test graph = Oregon-1

(b) Test graph = YIM-1

(c) Test graph = FB-1

(d) Test graph = TT-1

Figure 6.7: (Best viewed in color.) *Efficacy* of *LearnMelt* vs. normalized Canberra distance across different train-test graph pairs (k=1,000). (a) shows the efficacy of *LearnMelt* on Oregon-1 when it was trained on Oregon AS (blue diamonds), on Yahoo! IM (red squares), on Facebook data (green triangles), and on Twitter data (black crosses). (b), (c) and (d) depict the same for YIM-1, FB-1, and TT-1, respectively. *LearnMelt* is able to successfully transfer knowledge between social graphs (Yahoo! IM, Facebook, and Twitter), but has difficulty doing so between Oregon AS and Yahoo! IM (or Oregon AS and Facebook). This is not surprising since the structures of technological and social graphs are very different.

6.3.4.2 Feature Importance in LearnMelt

Here, we quantify feature importance in *LearnMelt*. The most popular method to evaluate the feature importance in random forests is measuring the mean square error of prediction [15]. When building the regression trees in random forests, the *out-of-bag (OOB)* samples are used to compute the prediction error. For each feature f, f is randomly permuted in the OOB samples and the prediction error is computed. The change in prediction error is used to measure the importance of feature f.

Figure 6.8 depicts feature importance in *LearnMelt*. The higher the value, the more important the feature. Generally, the degree-based features are more important, while the clustering-coefficient based features are less important in *LearnMelt*. Intuitively, this observation makes sense. Degree of a node is a better indication of the number of paths it is involved in than its clustering-coefficient. Deleting an edge that has high degrees in both of its endpoints can reduce the number of paths in a graph significantly. The experiments in the previous section confirm that the $Rich$-$Rich$ method can achieve high eigen-drop. Clustering-coefficient based features are less important since clustering coefficient hardly affects the number of paths in graph. Figure 6.9 illustrates an example of the effect of clustering coefficient. Let's consider two cases: (i) The edges *u2-u3* and *u3-u4* do not exist, (ii) There are edges *u2-u3* and *u3-u4* in the graph, which cause higher clustering coefficient for node u. Low or high clustering-coefficient of node *u* hardly affects the connectivity of the graph when deciding to delete the edge *u-v*.

6.3.5 Simulating Virus Propagation

As we mentioned before, the leading eigenvalue λ of the adjacency matrix affects the dissemination of a virus. Here, we simulate the popular virus model called SIS (short for Susceptible-Infected-Susceptible) to evaluate the efficacy of minimizing the virus by different methods. In the SIS model, the node can be in one of two states: susceptible or infected. A susceptible node will be infected at some infection rate β if it connects with infected neighbors. In the mean time, an infected node also tries to recover by itself at a rate δ. In our experiment, we removed $k = 1000$ edges from the original graph and evaluated how many nodes were infected. We ran

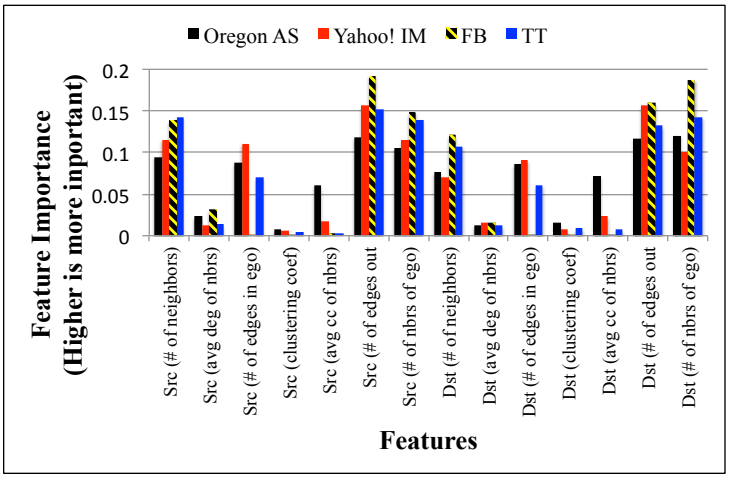


Figure 6.8: Feature importance in *LearnMelt*. Features with higher values are more important in predicting the eigen-drop. In order to evaluate the feature importance in random forests, we measure the mean square error of prediction [15]. When building the regression trees in random forests, the *out-of-bag (OOB)* samples are used to compute the prediction error. For each feature f, f is randomly permuted in the OOB samples and the prediction error is computed. The change in prediction error is used to measure the importance of feature f. The values are normalized such that the total importance values of all features is 1. In *LearnMelt*, the degree-based features are more important than the clustering-coefficient based features.

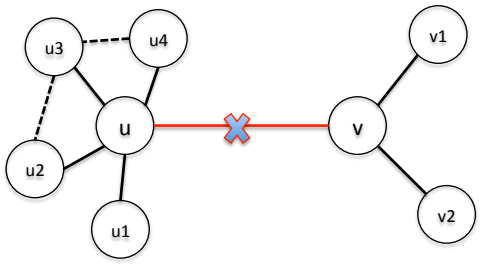


Figure 6.9: Clustering-coefficient is not an important feature in *LearnMelt* when deleting the connection between *u,v*. Edges *u2-u3* and *u3-u4* hardly affect the connectivity of the graph when deciding to delete the edge *u-v*

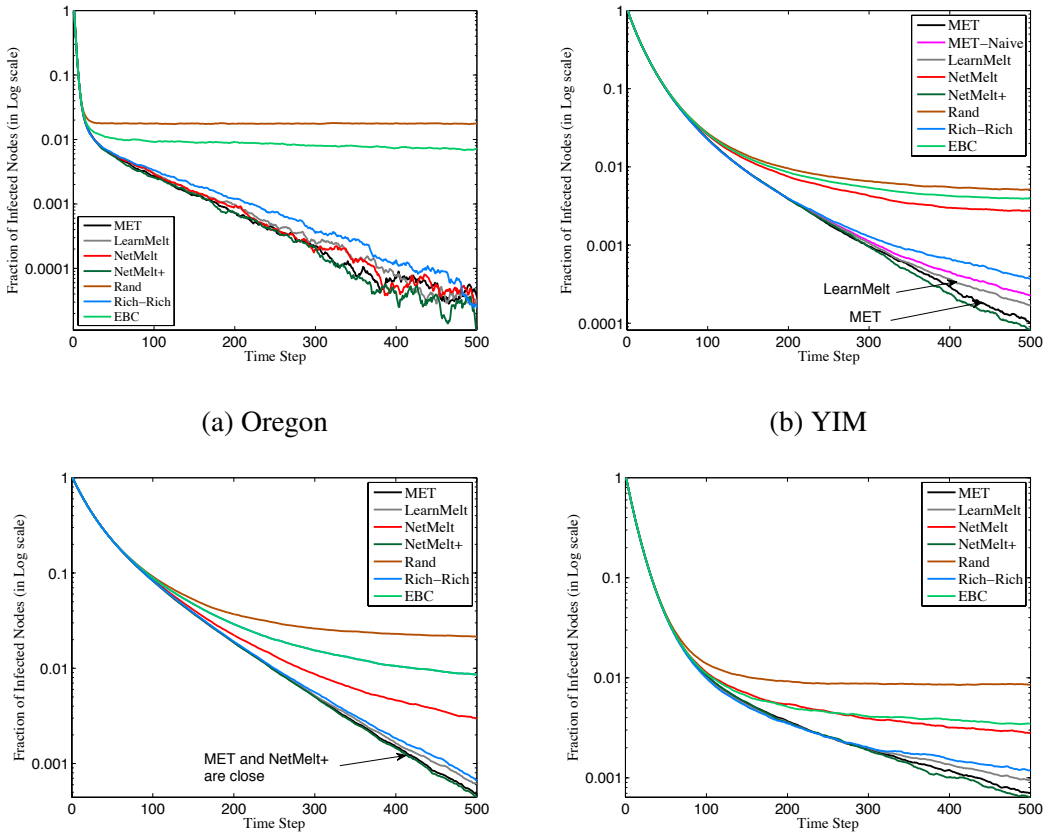
experiments for 100 times and report the average. In Figure 6.10, the x-axis is the time step and the y-axis is the fraction of nodes infected at particular time. A lower fraction of infected nodes is better. We observe that our method *MET* always achieves similar performance to *NetMelt+*. Among methods that only use structural features, *LearnMelt* is the best method. Again, *Rich-Poor* and *Poor-Poor* perform poorly and are omitted for brevity.

6.4 Discussion

One possible strategy for improving $NetMelt^+$ is to optimize the eigen-scores computation steps. As mentioned before, we use the power iteration method to compute the eigenvector for λ. The power iteration method uses an initial random eigenvector u_0 and performs the iteration $u_{i+1} = \frac{Au_i}{||Au_i||}$. In $NetMelt^+$, we store the current eigenvector for the next iteration of deletions. When re-computing the eigen-scores, we use the stored eigenvector as the initial vector for the power iteration calculation. This method can improve runtime in the Oregon graphs by 13.4%, but it does not improve the runtimes on Yahoo! IM, Facebook, and Twitter graphs. The reason for this is because the gap between the leading and the second highest eigenvalues is small. Thus, it takes a significant number of iterations to converge. Also, social graphs are often more volatile than technological graphs, where one deletion can change the eigenvectors significantly. Therefore, optimizing the power iteration will not solve the aforementioned *efficiency* problems of $NetMelt^+$ when dealing with social graphs.

The runtime of *MET* is $100+$ times faster than *NetMelt+* in the YIM graphs, 30 times faster in the TT graphs, but only $10+$ times faster in the FB graphs. The reason for this is that the top T eigenvalues of the FB graph become close together after removing k edges (as in Figure 6.3, the difference between λ_1 and λ_{21} in FB graph can be as small as 0.05). When the top-T eigenvalues are so close as in the FB graph, *MET* has to track more eigenvalues or re-computes the eigen-scores more often, which leads to the higher runtimes.

We see that finding the best k edges to delete is more difficult in social graphs, especially when many of the top eigenvalues are close to each other. In this case, we need to track more eigenvalues which makes the computation more expensive. In real applications, we may allow a small loss in *efficacy* by introducing a slack variable $epsilon$, which makes *MET* faster.



(a) Oregon (b) YIM

(c) FB (c) TT

Figure 6.10: (Best viewed in color.) Comparing the capability of different methods to minimize the number of infected nodes in the SIS model. Lower on the y-axis is better. We removed $k = 1000$ edges from the graph using different algorithms, ran 100 simulations and took the average number of infected nodes. The normalized virus strength $\simeq 1.5$. The virus strength is $\frac{\beta}{\delta}$, where β, δ are the infection and death rates of the virus, respectively. Our method *MET* achieves similar performance to *NetMelt+* in minimizing the number of infected node. *MET* always beats other methods in social graphs, who often have small eigen-gaps. *LearnMelt* beats *Rich-Rich*, which is another applicable method when the exact topology is not available.

Chapter 7

Conclusions and Future Work

Our work studied different approaches to minimize dissemination on graphs. We observed that the small gap between the top eigenvalues is prevalent in social graphs, which makes the problem of minimizing the leading eigenvalue a challenge. We proposed a novel approach, *MET*, to tracking multiple eigenvalues and their associated eigen-scores. Our proposed method can find edges whose deletions result in the largest drop in the leading eigenvalue, while being scalable for large graphs. Our method performs well across different graph types, from technological to social networks.

Our work also considered the aforementioned problem when the exact topology of the graph is not available. In some domains, such as finance, the exact connectivity of the graph might be too sensitive to release and we might have access to structural features only. In this case, we proposed a learning approach called *LearnMelt* that learns which edges to delete based on available graph. Our experiment showed that the degree-based features are very important when deciding which edges to delete.

We evaluated our algorithms extensively on multiple large real graphs and showed the *efficacy* and *efficiency* of our methods. In our comparative study, we found that: (i) our *MET* achieves the best combination of *efficacy* and *efficiency*, and (ii) our *LearnMelt* achieves highest *efficacy* when the exact topology of the graph is not available.

In future work, we plan to improve our estimation of the eigen-drop, which can help in improving the runtime of our algorithms. We hope to develop a distributed version of MET to reduce the computational cost of our algorithms MET. At the moment, our algorithms and existing algorithms use the full topology of the graph, which is hard to distribute computation. We will also consider the problem of when we have partial observation of the graph. For example, we can pick up some nodes in graph and extract the egonet around these nodes. In

this case, we can only use the partial information of the graph to decide which edge to delete. Developing a distributed algorithm that can work with partially observable graph is challenging but promising.

References

[1] M. Berlingerio, D. Koutra, T. Eliassi-Rad, and C. Faloutsos. Network similarity via multiple social theories. In *ASONAM*, pages 1439–1440, 2013.

[2] F. Bonchi. Influence propagation in social networks: A data mining perspective. In *IEEE Intelligent Informatics Bulletin*, page 2, 2011.

[3] L. Breiman. Random forests. *Mach. Learn.*, pages 5–32, 2001.

[4] A. Brouwer and W. H. Haemers. *Spectral of Graph*. Springer, 2009.

[5] B. Cao, N. N. Liu, and Q. Yang. Transfer learning for collective link prediction in multiple heterogenous domains. In *ICML*, pages 159–166, 2010.

[6] D. Chakrabarti, Y. Wang, C. Wang, J. Leskovec, and C. Faloutsos. Epidemic thresholds in real networks. *ACM Trans. Inf. Syst. Secur.*, pages 1:1–1:26, 2008.

[7] H. Chan, L. Akoglu, and H. Tong. Make it or break it: Manipulating robustness in large networks. In *SDM*, pages 325–333, 2014.

[8] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[9] S. Datta, A. Majumder, and N. Shrivastava. Viral marketing for multiple products. In *ICDM*, pages 118–127, 2010.

[10] N. Du, L. Song, M. Gomez-rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, pages 3147–3155, 2013.

[11] A. Ganesh, L. Massoulie, and D. Towsley. The effect of network topology on the spread of epidemics. In *IEEE INFOCOM*, pages 1455–1466, 2005.

[12] A. Gionis, E. Terzi, and P. Tsaparas. Opinion maximization in social network. In *SDM*, pages 387–395, 2013.

[13] A. Goyal, F. Bonchi, L. V. S. Lakshmanan, and S. Venkatasubramanian. On minimizing budget and time in influence propagation over social networks. *Social Netw. Analys. Mining*, 3:179–192, 2013.

[14] A. Goyal, W. Lu, and L. V. S. Lakshmanan. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *ICDM*, pages 211–220, 2011.

[15] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2009.

[16] K. Henderson, T. Eliassi-Rad, C. Faloutsos, L. Akoglu, L. Li, K. Maruhashi, B. A. Prakash, and H. Tong. Metric forensics: A multi-level approach for mining volatile graphs. In *KDD*, pages 163–172, 2010.

[17] K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. Rolx: Structural role extraction and mining in large graphs. In *KDD*, pages 1231–1239, 2012.

[18] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[19] D. Kempe, J. Kleinberg, and E. Tardos. Influential nodes in a diffusion model for social networks. In *ICALP*, pages 1127–1138, 2005.

[20] E. B. Khalil, B. Dilkina, and L. Song. Scalable diffusion-aware optimization of network topology. In *KDD*, pages 1226–1235, 2014.

[21] C. J. Kuhlman, G. Tuli, S. Swarup, M. V. Marathe, and S. S. Ravi. Blocking simple and complex contagion by edge removal. In *ICDM*, pages 399–408, 2013.

[22] G. N. Lance and W. T. Williams. Mixed-data classificatory programs i - agglomerative systems. *Australian Computer*, 1:15–20, 1967.

[23] C. Lanczos. An iterative method for the solution of the eigenvalue problem of linear differential and integral. *J. Res. Nat. Bur. Stand.*, pages 255–282, 1950.

[24] T. Lappas, E. Terzi, D. Gunopulos, and H. Mannila. Finding effectors in social networks. In *KDD*, pages 1059–1068, 2010.

[25] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, pages 1345–1359, 2010.

[26] B. A. Prakash, D. Chakrabarti, M. Faloutsos, N. Valler, and C. Faloutsos. Threshold conditions for arbitrary cascade models on arbitrary networks. In *ICDM*, pages 537–546, 2011.

[27] B. A. Prakash, J. Vreeken, and C. Faloutsos. Spotting culprits in epidemics: How many and which ones? In *ICDM*, pages 11–20, 2012.

[28] V. M. Preciado and A. Jadbabaie. Moment-based spectral analysis of large-scale networks using local structural information. *IEEE/ACM Trans. Netw.*, 21:373–382, 2013.

[29] M. Purohit, B. A. Prakash, C. Kang, Y. Zhang, and V. Subrahmanian. Fast influence-based coarsening for large networks. In *KDD*, pages 1296–1305, 2014.

[30] S. Soundarajan, T. Eliassi-Rad, and B. Gallagher. A guide to selecting a network similarity method. In *SDM*, pages 1037–1045, 2014.

[31] G. W. Stewart and J. guang Sun. *Matrix Perturbation Theory*. Academic Press, 1990.

[32] H. Tong, B. A. Prakash, T. Eliassi-Rad, M. Faloutsos, and C. Faloutsos. Gelling, and melting, large graphs by edge manipulation. In *CIKM*, pages 245–254, 2012.

[33] H. Tong, B. A. Prakash, C. Tsourakakis, T. Eliassi-Rad, C. Faloutsos, and D. H. Chau. On the vulnerability of large graphs. In *ICDM*, pages 1091–1096, 2010.

[34] N. C. Valler, B. A. Prakash, H. Tong, M. Faloutsos, and C. Faloutsos. Epidemic spread in mobile ad hoc networks: Determining the tipping point. In *Networking*, pages 266–280, 2011.

[35] P. Van Mieghem, J. Omic, and R. Kooij. Virus spread in networks. *IEEE/ACM Trans. Netw.*, pages 1–14, 2009.

[36] Z. Wang, Y. Song, and C. Zhang. Knowledge transfer on hybrid graph. In *IJCAI*, pages 1291–1296, 2009.

[37] Z. Wu and V. M. Preciado. Laplacian spectral properties of a graph from random structural samples. In *SDM*, pages 343–351, 2014.

[38] R. Zafarani and H. Liu. Social computing data repository at ASU (http://socialcomputing.asu.edu), 2009.

Appendix A

Appendix

A.1 Relationship Between Eigenvalues and Eigenvectors

Lemma A.1.1. *Given a graph G represented by a symmetric matrix $\mathbf{A}$ and an edge e in G, let λ_i be the i^{th} eigenvalue of $\mathbf{A}$ and $\bar{\lambda_i}$ be the corresponding eigenvalue after removing the edge e. We have:*

$$\Delta\lambda_i = \lambda_i - \bar{\lambda_i} \approx eigenscores(e, i) \tag{A.1}$$

Proof. Let $\Delta\mathbf{A}$ be the perturbation matrix of $\mathbf{A}$ when removing edge e (i.e., all entries are zeros except the corresponding entries of edge e. Let $\Delta\mathbf{u}_i$ be the perturbation of the i^{th} eigenvector. By definition of eigenvalue and eigenvector, we have:

$$\mathbf{A}\mathbf{u}_i = \lambda_i\mathbf{u}_i \tag{A.2}$$

$$(\mathbf{A} - \Delta\mathbf{A})(\mathbf{u}_i - \Delta\mathbf{u}_i) = (\lambda_i - \Delta\lambda_i)(\mathbf{u}_i - \Delta\mathbf{u}_i) \tag{A.3}$$

Ignoring the high-order perturbations such as $\Delta\mathbf{A}\Delta\mathbf{u}_i, \Delta\lambda_i\Delta\mathbf{u}_i$, and using Eq. A.2, we have:

$$\mathbf{A}\Delta\mathbf{u_i} + \Delta\mathbf{A}\mathbf{u}_i \approx \lambda_i\Delta\mathbf{u}_i + \Delta\lambda_i\mathbf{u}_i \tag{A.4}$$

Multiplying both sides of Eq. A.4 by $\mathbf{u}_i'$ (i.e., the transpose of vector $\mathbf{u}_i$), leads to:

$$\mathbf{u}_i'\mathbf{A}\Delta\mathbf{u_i} + \mathbf{u}_i'\Delta\mathbf{A}\mathbf{u}_i \approx \mathbf{u}_i'\lambda_i\Delta\mathbf{u}_i + \mathbf{u}_i'\Delta\lambda_i\mathbf{u}_i \tag{A.5}$$

Reordering the scalar values, we get:

$$\mathbf{u}_i'\mathbf{A}\Delta\mathbf{u_i} + \mathbf{u}_i'\Delta\mathbf{A}\mathbf{u}_i \approx \lambda_i\mathbf{u}_i'\Delta\mathbf{u}_i + \Delta\lambda_i\mathbf{u}_i'\mathbf{u}_i \tag{A.6}$$

Using the unitary characteristic of $\mathbf{u}_i$ (i.e,. $\mathbf{u}_i'\mathbf{u}_i = 1$) and the symmetric property of $\mathbf{A}$ (i.e., $\mathbf{A}\mathbf{u}_i = \lambda_i\mathbf{u}_i \Rightarrow (\mathbf{A}\mathbf{u}_i)' = (\lambda_i\mathbf{u}_i)' \Rightarrow \mathbf{u}_i'\mathbf{A} = \lambda_i\mathbf{u}_i'$), we have:

$$\Delta\lambda_i = \mathbf{u}_i'\Delta\mathbf{A}\mathbf{u}_i \approx eigenscores(e,i) \tag{A.7}$$

□

A.2 Relationship Between the Leading Eigenvalue and the Maximum Degree of a Graph

Lemma A.2.1. *Let d_{max} be the maximum degree of graph G, and λ_1 be the largest eigenvalue of the adjacency matrix. Then:*

$$\sqrt{d_{max}} \leq \lambda_1 \tag{A.8}$$

Proof. If $\mathbf{A}'$ is a sub-matrix of $\mathbf{A}$, according to Proposition 3.1.1 in [4]), we have:

$$\lambda_1(\mathbf{A}) \geq \lambda_1(\mathbf{A}') \tag{A.9}$$

From the definition of λ, we can write:

$$\lambda_1 = \max_{\mathbf{x}} \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \tag{A.10}$$

We see that $\sqrt{d_{max}}$ is an eigenvalue of a star graph with the highest degree of d_{max}. This can be shown by choosing the values of vector $\mathbf{x}$ in Eq. A.10 as $\sqrt{d_{max}}$ for the node with highest degree and 1 for other nodes.

From the facts A.9 and A.10, we have $\sqrt{d_{max}} \leq \lambda_1$.

□

A.3 An Approach for Finding the Upper-bound of the Optimal Solution when Removing k Edges

It has been shown that finding the optimal solution for minimizing the largest eigenvalue is NP-hard [32]. Here, we show the upper-bound on the largest degree of a graph after removing k edges. This helps us in upper-bounding the drop of the leading eigenvalue in the optimal solution.

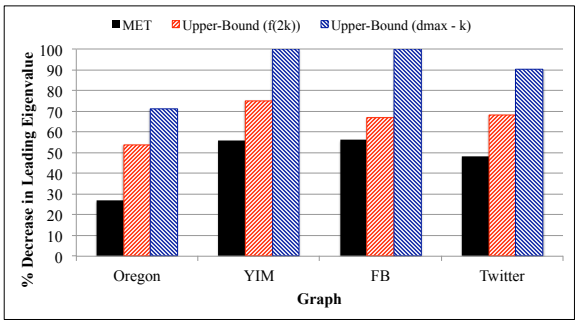


Figure A.1: Upper-bound on the percentage decrease in λ_1. Budget $k = 1000$. The optimal solution must be in between the black and red bars. Equation A.11 provides a tighter bound than Equation A.12.

Suppose we have a sequence of numbers: $d_1 \geq d_2 \geq ... \geq d_n$ (this is not a degree sequence-just a sequence of numbers). Consider the trivial problem of minimizing the maximum element in this sequence by reducing one element at a time. Obviously the best way to do this is by reducing d_1 until it is equal to d_2, then reduce d_1 and d_2 alternately until they are equal to d_3, etc. Let $f(x)$ be the maximum value of this specific sequence after one performs x such reductions.

In the case of a graph, if we are allowed to remove k edges in a given graph, the maximum degree of the modified graph can not be smaller than $f(2k)$. We can find the upper-bound of the drop of the leading eigenvalue as in Eq. A.11. Notice that $0 < \sqrt{f(2k)} \leq \bar{\lambda_1} \leq \lambda_1$.

$$\%drop = \frac{100 * (\lambda_1 - \bar{\lambda_1})}{\lambda_1} \leq \frac{100 * (\lambda_1 - \sqrt{f(2k)})}{\lambda_1} \tag{A.11}$$

There is a simpler approach for finding the upper bound of the optimal solution. Let the highest degree of original graph be d_{max}, then the highest degree after removing k edges will be at least $d_{max} - k$. Thus,

$$\%drop = \frac{100 * (\lambda_1 - \bar{\lambda_1})}{\lambda_1} \leq \frac{100 * (\lambda_1 - (d_{max} - k))}{\lambda_1} \tag{A.12}$$

Figure A.1 depicts the upper-bound of the optimal solution when using $(d_{max} - k)$ to bound the highest degree of modified graph. We see that $f(2k)$ provides a tighter bound.