An Algorithm for Influence Maximization and Target Set Selection for the Deterministic Linear Threshold Model

Anand Swaminathan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Madhav V. Marathe, Chair
Chris J. Kuhlman
V. S. Anil Vullikanti
Keith R. Bisset

May 16, 2014
Blacksburg, Virginia

Keywords: Influence maximization, complex contagion, linear threshold

An Algorithm for Influence Maximization and Target Set Selection for the Deterministic Linear Threshold Model

Anand Swaminathan

(ABSTRACT)

The problem of influence maximization has been studied extensively with applications that include viral marketing, recommendations, and feed ranking. The optimization problem, first formulated by Kempe, Kleinberg and Tardos, is known to be NP-hard. Thus, several heuristics have been proposed to solve this problem. This thesis studies the problem of influence maximization under the deterministic linear threshold model and presents a novel heuristic for finding influential nodes in a graph with the goal of maximizing contagion spread that emanates from these influential nodes. Inputs to our algorithm include edge weights and vertex thresholds. The threshold difference greedy algorithm presented in this thesis takes into account both the edge weights as well as vertex thresholds in computing influence of a node. The threshold difference greedy algorithm is evaluated on 14 real-world networks. Results demonstrate that the new algorithm performs consistently better than the seven other heuristics that we evaluated in terms of final spread size. The threshold difference greedy algorithm has tuneable parameters which can make the algorithm run faster. As a part of the approach, the algorithm also computes the infected nodes in the graph. This eliminates the need for running simulations to determine the spread size from the influential nodes. We also study the target set selection problem with our algorithm. In this problem, the final spread size is specified and a seed (or influential) set is computed that will generate the required spread size.

Acknowledgments

I would never have been able to finish my thesis without the guidance of my committee members, help from friends, and support from my family.

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Madhav V. Marathe for the continuous support of my Masters study and research.

I offer my sincerest gratitude to my mentor, Dr. Chris Kuhlman, who has supported me throughout my thesis with his patience and knowledge. I will cherish my experience working with him throughout the rest of my career, and I hope to continue to collaborate with him in the future.

I would like to thank the rest of my committee members, Dr. V. S. Anil Vullikanti and Dr. Keith R. Bisset, for giving valuable comments about my thesis.

I gratefully acknowledge the funding received for my research from DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, and NSF SDCI Grant OCI-1032677.

To Dr. Godmar Back, for allowing me to work with him on his project LibX.

To Daniel Hung, for providing me the opportunity to work as Graduate Assistant at the office of Institutional Research and Effectiveness. I thank him for his flexibility and guidance.

I am grateful for my mother who is my first inspiration, my father for his support, and my sister for her encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Collective behavior refers to the behavior that is diffused or dispersed over large distances. The web and the social media have allowed for such rapid distribution of information around the world. Research shows that people trust information obtained from their close social circle far more than information obtained from general advertisement channels. Thus a minor piece of information can pass from ear to ear in a network and become a viral phenomenon. This type of information dispersal has led to an increase in interest among researchers in modeling the spread of such diffusion among a given population. A social network can be modeled as a graph with nodes representing individuals and edges representing connections, or relationships, between individuals. Information, behavior (e.g., joining a protest, adopting a fad), and other entities that can be propagated through a social network are referred to as **contagions** (e.g., [13]). There are two models of diffusion that are widely studied in the area of social networks - **simple** and **complex**. In a **simple contagion model**, each individual can contract a contagion if only one of its neighbors possesses it. An example of a simple contagion would be the spread of a contagious disease like the flu in a community wherein each individual has a chance of getting infected if he/she comes in contact with another infected person. In a **complex contagion model**, multiple sources of exposure are needed before an individual adopts the change of behavior [13].

Online social networking websites like Twitter and Facebook have provided an effective medium for diffusing ideas and spreading influence. These social networking websites provide a platform for marketing products and businesses online. Due to budgetary constraints in marketing, the ideal strategy is to influence a set of users who will start using the product and who will in turn influence their friends to use the product and so on. Informally, the problem of *influence maximization* as defined in [35], is the problem of finding a small set of seed nodes (that initially possess a contagion) in a social network that maximizes the spread

of influence. Kempe et al. proved that this optimization problem is NP-hard, and presented a greedy approximation algorithm guaranteeing that influence spread is $(1 - \frac{1}{e} - \varepsilon)$ of optimal influence spread [35]. Just as important, their work has motivated the development of a huge body of literature on the topic of influence maximization. Influence maximization has applications in viral marketing, feed ranking, recommendations and several other areas.

A complex contagion requires multiple contacts for an individual to change his/her state and exhibits a different diffusion pattern compared to a simple contagion. Let us take a real-world example that exhibits complex contagion: a tense atmosphere prevailing in a city which can potentially lead to a public protest. It has been argued [13] and demonstrated empirically [31] that the possibility of an individual participating in a protest depends on the number of that person's neighbors who are already part of the protest. In particular, for this and other scenarios, we focus on the linear threshold model. Informally, the model is as follows: A social network is given. Each directed edge (v_1, v_2) from one person v_1 to another person v_2 has an edge weight w_{v_1,v_2} associated with it that quantifies v_1's influence on v_2. A person v_2 who has not contracted a contagion is influenced by all of its distance-1 neighbors that have contracted it. The weights of the edges formed with these contagious neighbors are added, and if this sum is at least equal to v_2's threshold for acquiring the contagion, then v_2 will do so. A formal definition is given in Chapter 2. Our goal in this work is to identify, for a social network whose contagion dynamics are those just described, a minimal set of seed nodes that initially possess the contagion so that the contagion will propagate to a large number of people. As will be described later, this problem of identifying seed nodes is formally hard. Thus, we present a new algorithm for computing a small seed node set.

We also make a distinction that carries through the entire thesis. The linear threshold model described in [35] assumes that thresholds are assigned randomly to vertices of a graph, and that threshold assignments are made stochastically as part of the model implementation. That is, just prior to the executing the linear threshold dynamics on a network, thresholds are assigned to vertices. We use, in contrast, a deterministic linear threshold model in which thresholds are assigned deterministically to vertices. This difference in threshold assignments has important subtle ramifications; see [44].

1.2 Contribution

1. **New influence maximization algorithm:** the major contribution of this thesis is a new influence maximization algorithm for the deterministic linear threshold model. The nodes' threshold values are deterministic and are provided as an input to our algorithm. The new algorithm called the Threshold Difference Greedy (TDG) algorithm takes into account both edge weights as well as node thresholds to compute the influence spread. The space and time complexities of the algorithm have also been analyzed.

2. **Experimental evaluation of influence maximization algorithms:** the proposed TDG algorithm in this paper has been evaluated against seven existing algorithms, namely, the High Degree Heuristic, the Random Heuristic, the influence maximization algorithm by Chen et al. for the independent cascade model [58], the influence maximization algorithm by Chen et al. for the linear threshold model [20], the Eigenvector Heuristic, the Degree Discount Heuristic, and the Betweenness Centrality Heuristic. Experiments were conducted on 14 real-world networks with average degree ranging from 5 to 50. The TDG algorithm performs better than other approaches for the great majority of conditions considered. All the evaluations are done through experiments using the *InterSim* simulator.

3. **Tuneable parameters for the TDG algorithm:** the TDG algorithm has tuneable parameters that trade off computational speed for solution accuracy. We show that a single set of parameters consistently gives better results in terms of outbreak size than other heuristics, while minimizing the algorithm execution time. For instance, for the Epinion graph with 75877 nodes and a very low threshold of 0.1 (onerous conditions), TDG completes in 199 seconds while other other state of the art approaches take thousands of seconds to complete.

4. **Algorithm for both influence maximization and target set selection:** the TDG algorithm computes infected nodes as part of the algorithm and thus, unlike other influence maximization heuristics, can be used to address both the influence maximization problem and the target set selection problem. The experimental results for addressing both of the problems are provided in this thesis.

5. **Parallel implementations of several influence maximization algorithms:** the influence maximization algorithm by Chen et al. for the independent cascade model [58], the LDAG algorithm for the linear threshold model [20], and the TDG algorithm have been implemented to execute in a parallel fashion. The parallelism is achieved by threading and not through distributed processing. The results of the improvement in the running time are available in Chapter 8.

6. **Verified software applications for integration into CINET:** all of the implemented heuristics that have been evaluated are tested and have inbuilt log structures, so that these algorithms can be extended or modified in the future. The package containing all these heuristics can be integrated into CINET [1].

Chapter 2

Dynamics Model

2.1 Synchronous Graph Dynamical Systems

We use the **graph dynamical system** (GDS) [48, 40, 46, 47] formalism, which is also referred to as a **discrete dynamical system** (e.g. [4, 3, 5]), to model contagion propagation on social networks. That is, the dynamics are discrete in time and discrete in node states. Let $\mathbb{B}$ denote the Boolean domain {0,1}. A GDS $\mathcal{S}$ over $\mathbb{B}$ is a triple $\mathcal{S} = (G, \mathcal{F}, R)$, where

(a) $G(V, E)$, a directed **graph** with node set V and edge set E where $n = |V|$ and $m = |E|$, represents the underlying social network on which a contagion propagates,

(b) $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is a collection of functions in the system, with f_v denoting the **local transition function** or **vertex function** associated with vertex or node v_i, $1 \leq i \leq n$, and

(c) R is the **update scheme** that specifies the execution sequence of the function f_v.

For an undirected graph G, an edge $\{a, b\}$ is equivalent to two directed edges: an edge from a to b, written (a, b), and an edge b to a, written (b, a). Throughout this manuscript, we use the convention that a directed edge (a, b) denotes that a influences b.

Each node in graph G has a state value from $\mathbb{B}$. Each function f_v specifies the local interaction between node v and its distance-1 (i.e., adjacent) neighbors in G. Vertex function f_v computes the next state for the vertex v. We use the convention that a node is not a neighbor of itself, but this convention is irrelevant for the dynamics that we study. In this paper, our main focus is the **deterministic linear threshold (LT) model**. The deterministic linear threshold model works the same way as the linear threshold model [35] explained in Section 2.2. The only difference is that, in this model, the threshold values on the nodes are deterministic and are prefixed.

The update scheme we use throughout this work is the **synchronous update scheme**, meaning that to compute the states of nodes at time t, all inputs to f_v $(1 \leq v \leq n)$ are quantities at time $(t-1)$. We provide an example below to make this concrete, but we note here that there are other update schemes [3]. The synchronous update scheme means that the GDS is a **synchronous dynamical system** (SyDS), and we use SyDS henceforth to emphasize the synchronous update approach; i.e. $\mathcal{S}$ is an SyDS.

A **configuration** $\mathcal{C}(t)$ of an SyDS at any time is an n-vector $(s_1, s_2, \ldots, s_n)$, where $s_i \in \mathbb{B}$ is the state of v. A single SyDS transition from one configuration to another can be expressed by the following pseudocode, where each of the two steps is executed in parallel, but the steps themselves are executed serially.

for each node v **do in parallel**
  (i) Compute the value of $f_v(t+1)$ from $\mathcal{C}(t-1)$. Let $s_i' = s_i(t)$ denote this value.
  (ii) Update the state of v to s_v'.
end for

If an SyDS has a transition from configuration $\mathcal{C}_1 = \mathcal{C}(t)$ to configuration $\mathcal{C}_2 = \mathcal{C}(t+1)$, we say that $\mathcal{C}_2$ is the **successor** of $\mathcal{C}_1$ and $\mathcal{C}_1$ is a **predecessor** of $\mathcal{C}_2$. For deterministic f_v, as is the case here, a successor is unique. A configuration $\mathcal{C}_1$ is called a **fixed point** if the successor of $\mathcal{C}_1$ is $\mathcal{C}_1$ itself. It is known (e.g., [38]) that progressive threshold systems always reach a fixed point. A configuration $\mathcal{C}$ which does not have a predecessor is called a **Garden of Eden** configuration. A **forward trajectory** is the sequence $(\mathcal{C}(t))_{t=0}^{t_f}$ of configurations from time $t = 0$ to the time t_f, at which time the deterministic system reaches and traverses all of its **limit cycles** (i.e., a limit cycle is a repeating sequence of configurations). For a deterministic discrete dynamical system on a finite graph (sometimes called a **finite dynamical system**), a limit cycle will always be reached. A fixed point defined earlier is a limit cycle of length 1.

2.2 Deterministic Linear Threshold Model [35], [44]

We note that the linear threshold model [35] and the deterministic linear threshold model [44] use the same parameters and state functions, and consequently, the description here applies to both the models.

In the linear threshold model, each directed edge (u, v) in the graph is assigned a fixed weight $w_{(u,v)}$, with $0 \leq w_{(u,v)} \leq 1$. Weights are not necessarily symmetric; i.e., in general $w_{(u,v)} \neq w_{(v,u)}$. Each vertex v is assigned a fixed threshold θ_v such that $0 \leq \theta_v \leq 1$. A vertex v transitions its state, s_v, $0 \to 1$, if the sum of weights of incoming edges from nodes that are in state 1 is greater than or equal to v's threshold; otherwise a node in state 0 remains in state 0. A node in state 1 remains in state 1. This is called a **progressive** threshold model [35]. Formally, the vertex function f_v is as follows:

1. If a vertex v is in state 1 at time $t-1$ (i.e., $s_v(t-1)=1$), then $s_v(t)=f_v(t)=1$ (i.e., the state does not change).

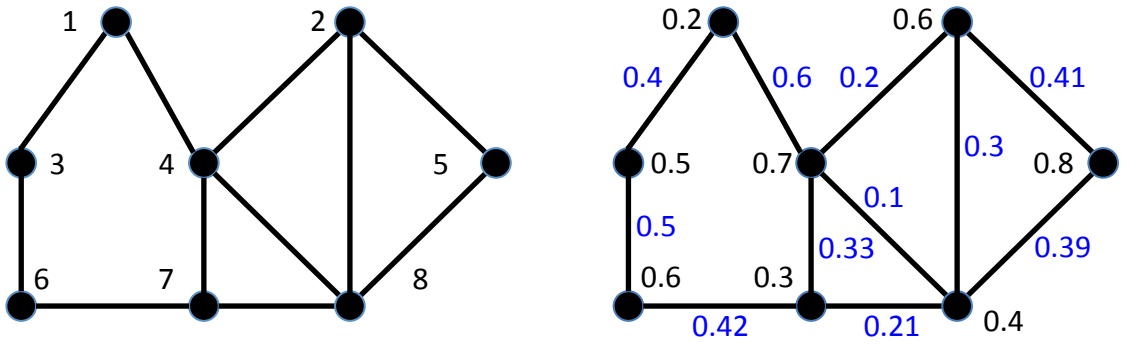
2. If $s_v(t-1)=0$, then $s_v(t)=1$ if the following condition is satisfied

$$\sum_{\substack{u\in N^{in}(v)\\ s_u(t-1)=1}} w_{(u,v)} \geq \theta_v \ . \tag{2.1}$$

3. Otherwise $s_v(t)=0$.

Here, $N^{in}(v)=\{u \mid (u,v)\in E\}$; i.e., $N^{in}(v)$ is the set of distance-1 neighbors of v (each element of $N^{in}(v)$ forms an edge *to* v). $N^{in}(v)$ is the set of *in-neighbors* of v, and these are the nodes that influence v to change state. Also, in this model, a node u that changes to state 1 at time t^* will influence its out-neighbors $N^{out}(u)$ at each time $t>t^*$ (This contrasts with the dynamics in some other models; e.g., independent cascade [35]). Also we adopt the convention that θ_v, $w_{(u,v)}$ have values between $0\leq\theta_v$, $w_{(u,v)}\leq 1$. Other values of thresholds and edge weights can be used, in general.

Example 1: Consider the $\mathcal{S}$ whose underlying graph is shown in Figure 2.1(a). Suppose for each node v, $1\leq v\leq 8$, the vertex function f_v is the linear threshold function with vertex thresholds and edge weights as given in Figure 2.1(b). Hence, this is a complex contagion [13], since for at least some vertices, multiple neighbors must contribute to a vertex's state change. For example, for node 2, $\theta_2=0.6$, but no weight on a single edge that is incident on node 2 will cause node 2 to transition state. A forward trajectory is given in Table 2.1. We detail a few of those state transitions here. In the initial configuration, the **seed set** I (i.e., the nodes initially in state 1) is $I=\{6\}$ and all other nodes are in state 0. During the first time step ($t=1$), the state s_3 of node 3 changes to 1 because $w_{(6,3)}=\theta_3=0.5$, and hence the criterion of Equation (2.2) is satisfied. Similarly for node 7, $w_{(6,7)}=0.42>\theta_7=0.3$, and so vertex 7 transitions to state $s_7=1$. No other vertex changes state. At time $t=2$, the state of node 1 changes to $s_1=1$ because $w_{(3,1)}=0.4>\theta_1=0.2$, and hence the criterion of Equation (2.2) is satisfied. No other vertex changes state at this time. From Table 2.1, we see that there are no vertex state transitions from $t=3$ to $t=4$, indicating that the system has reached a fixed point at $t=3$. The **spread size** is the number of nodes in state 1 at the end of contagion propagation, and the **spread fraction** is the corresponding fraction of nodes. Here, the spread fraction is 5/8=0.625. We call this sequence of configurations that completely specifies the system dynamics from one seed node set a **diffusion instance**. Hence, a forward trajectory is also a diffusion instance. In the simulations described later, we are computing diffusion instances.



(a) (b)

Figure 2.1: Network to illustrate contagion dynamics on a graph. (a) Graph with node numbers. (b) Linear threshold model parameters: thresholds (in black) and edge weights (in blue). Edge weights are assumed symmetric; i.e., $w_{(u,v)} = w_{(v,u)}$ in this example.

Table 2.1: Sequence of configurations and system transitions that result in a fixed point at time $t = 3$ when vertex 6 is seeded; i.e., $I = \{6\}$ and all other vertices are initially in state 0.

Time	**System Configuration** $\mathcal{C} = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$
0	(0, 0, 0, 0, 0 1, 0, 0)
1	(0, 0, 1, 0, 0 1, 1, 0)
2	(1, 0, 1, 0, 0 1, 1, 0)
3	(1, 0, 1, 1, 0 1, 1, 0)
4	(1, 0, 1, 1, 0 1, 1, 0)

2.3 Deterministic threshold model

In the deterministic threshold model, each directed edge (u, v) is assigned a fixed weight 1. Each vertex v is assigned a fixed threshold θ_v such that $\theta_v \geq 1$ and is an integer. A vertex v transitions its state, s_v, $0 \to 1$, if the sum of weights of incoming edges from nodes that are in state 1 is greater than or equal to v's threshold; otherwise a node in state 0 remains in state 0. A node in state 1 remains in state 1. Formally, the vertex function f_v is as follows:

1. If a vertex v is in state 1 at time $t-1$ (i.e., $s_v(t-1) = 1$), then $s_v(t) = f_v(t) = 1$ (i.e., the state does not change).

2. If $s_v(t-1) = 0$, then $s_v(t) = 1$ if the following condition is satisfied

$$\sum_{\substack{u \in N^{in}(v) \\ s_u(t-1)=1}} w_{(u,v)} \geq \theta_v \ . \tag{2.2}$$

3. Otherwise $s_v(t) = 0$.

2.4 Independent Cascade Model

The **independent cascade** model is described here since one of the algorithms used in the influence maximization studies assumes this dynamics model. The network has a fixed weight $w_{(u,v)}$ on each directed edge (u, v) (i.e., u influences v) where $0 \leq w_{(u,v)} \leq 1$. The vertex function for this progressive model is as follows :

1. If a vertex v is in state 1 at time $t-1$ (i.e., $s_v(t-1) = 1$), then $s_v(t) = f_v(t) = 1$ (i.e., the state does not change).

2. If a vertex v has state $s_v(t-1) = 0$, then for each $u \in N^{in}(v)$ that changed to state 1 at $t-1$, perform a Bernoulli trial. If for at least one edge (u, v) the random number $r \leq w_{(u,v)}$, then $s_v(t) = f_v(t) = 1$. (Note there are separate random numbers r for each u.)

3. Otherwise, $s_v(t) = 0$.

Note that once a vertex u transitions to state 1, it has one chance (at the next time) to independently influence each of its distance-1 neighbors in state 0 to transition to state 1.

Chapter 3

Formal Problem Statement

In this chapter, we provide formal problem statements for the problems that are addressed in this thesis.

We provide a formal statement for the problem of finding the most effective nodes to seed, in order to cause a large number of nodes to become affected. Two dynamics models were investigated in the initial work [35]: the IC model and the LT model (cf. Chapter 2).

We now address the influence maximization under the deterministic linear threshold model. The only change necessary in the problem statement for the linear threshold model is that vertex thresholds are assigned at random. The decision problem variant is as follows.

Influence Maximization Problem for Deterministic Linear Threshold Model (IM Decision Problem) [35]

Given: A social network $G(V, E)$ with vertex set V and edge set E where $n = |V|$ and $m = |E|$, a set $K = \{0, 1\}$ of vertex states, a set W of edge weights w_{ij} for the influence of vertex i on vertex j, a set T of thresholds θ_i for vertices $1 \leq i \leq n$, a sequence $(f_v)_{v=1}^{n}$ of vertex functions where f_v describes how vertex v changes state in the linear threshold model, an update scheme for sequencing the f_v, and two integers ρ and σ_G, such that both integers are $\leq n$.

Question: Does $\mathcal{S}$ have a seed set I with $|I| \leq \rho$, whose elements each have initial state 1, such that the spread size $\sigma_G(I)$ is at least σ_G?

The corresponding optimization problem variant follows.

Influence Maximization Problem for Deterministic Linear Threshold Model (IM Optimization Problem) [35]

Given: A social network $G(V, E)$ with vertex set V and edge set E where $n = |V|$ and $m = |E|$, a set $K = \{0, 1\}$ of vertex states, a set W of edge weights w_{ij} for the influence of vertex i on vertex j, a set T of thresholds θ_i for vertices $1 \leq i \leq n$, a sequence $(f_v)_{v=1}^{n}$ of

vertex functions where f_v describes how vertex v changes state in the linear threshold model, an update scheme for sequencing the f_v, and an integer $k < n$.

Find: The set I of seed nodes, of size $k = |I|$, whose initial state is 1, such that the spread size $\sigma_G(I)$ is maximum.

As mentioned in Chapter 1, our focus is the deterministic linear threshold model [44], which differs in some aspects from the linear threshold model [35]. In the deterministic model, the thresholds are assigned to vertices deterministically, while in the linear threshold model, they are assigned uniformly at random, taking on values between 0 and 1. These have a significant affect on complexity results. In [35], it was shown that the influence maximization optimization problem is NP-hard. They also showed that the optimization problem is polynomial time approximable to within a factor $e/(e-1)+\epsilon$ for any $\epsilon > 0$ under selected diffusion models. For the deterministic linear threshold model, there is no $n^{1-\epsilon}$ factor polynomial time approximation unless P=NP [44]. Furthermore, the complexity of determining the influence from a given seed set (influential set) is different for the two models. Chen et al [20], showed the problem to be #P-hard for the linear threshold model. However, for the deterministic linear threshold model, the solution is efficiently computable [44]. Note that while these differences exist, the dynamics in Section 2.2 are the same in both the deterministic linear threshold model and the linear threshold model.

The problem of finding the smallest set I of seed nodes (called a **target set [14] or conversion set** [25]) to ensure that all nodes become affected under a deterministic threshold model was first presented as a decision problem in [24, 25] and concurrently in [14, 15]. Their dynamics model is the threshold model in section 2.3; i.e., f_v for $v \in V$ is a threshold function. We make a small modification in the optimization variant to allow specification of any (final) spread size, following [15].

Target Set Selection Decision Problem [25, 14]

Given: A social network $G(V, E)$ with vertex set V and edge set E where $n = |V|$ and $m = |E|$, a set $K = \{0, 1\}$ of vertex states, a set W of edge weights w_{ij} for the influence of vertex i on vertex j, each vertex i, $1 \leq i \leq n$ has threshold θ_i, a sequence $(f_v)_{v=1}^n$ of vertex functions where f_v for a vertex v describes how vertex v changes state in the linear threshold model, an update scheme for sequencing the f_v, and an integer $\rho \leq n$.

Question: Does the $\mathcal{S}$ have a target set (i.e., a seed set whose elements are initially in state 1) I where $|I| \leq \rho$, such that all nodes become affected?

The optimization problem variant, of finding the smallest set of seed nodes to ensure that at least σ_G nodes become affected, is formalized below. This is a variant considered in [14, 15].

Target Set Selection Optimization Problem [25, 14]

Given: A social network $G(V, E)$ with vertex set V and edge set E where $n = |V|$ and $m = |E|$, a set $K = \{0, 1\}$ of vertex states, a set W of edge weights w_{ij} for the influence of vertex i on vertex j, each vertex i, $1 \leq i \leq n$ has threshold θ_i, a sequence $(f_v)_{v=1}^n$ of vertex

functions where f_v for a vertex v describes how vertex v changes state in the linear threshold model, an update scheme for sequencing the f_v, and an integer $\sigma_G \leq n$.

Find: The set I of seed nodes, with the least cardinality, whose elements are nodes with initial state of 1 with all other vertices initially in state 0, such that the spread size $\sigma_G(I)$ (i.e., the final number of nodes in state 1) is at least σ_G.

When the required number of nodes to be affected σ_G is less than n, this problem is called the Target Set Selection (TSS) Problem. When $\sigma_G = n$, this problem is called the Perfect Target Set Selection (PTSS) problem in [14, 15]. Other variants are presented in [2]. The decision problem is NP-Complete [25] for the case where all the edge weights are 1.

Chapter 4

Related Work

4.1 Influence Maximization

The problem of selecting individuals to market to, who might then influence others to also purchase a commodity, was first posed in [23]. One of the contentions of the authors' was that a person's value in marketing included a component related to her position in her social network and the influence she exerted on others. They used a probabilistic cost-benefit model to identify individuals with the greatest network value. They also attempted to predict future purchasing activities and applied their method to a movie dataset. In [53], they extended their model for improved computational speed and investigated knowledge sharing networks. They laid the groundwork for the study of influence maximization by positing how to select a subset of agents (nodes) to query, in order to learn the node's (trusted) neighbors, that leads to greatest profits. This has led to a large body of work on influence maximization, to which we now turn.

Arguably the seminal work in influence maximization is [35], which addressed a discrete formulation of the optimization problem posed in [23]. It is useful to detail some of this work, since it laid the groundwork for the great majority of influence maximization work that followed. The influence maximization problem they addressed is formalized in Chapter 3.

The dynamics models studied in [35] for investigating the influence maximization question are *progressive*. In almost all works, the update scheme is taken as synchronous and the vertex state set $K = \{0, 1\}$, although there are exceptions (e.g., for competing contagion models). Two dynamics models are studied in [35]: independent cascade (IC) and linear threshold (LT). These were presented in Chapter 2

The work [35] made a number of contributions that are regularly used today in the analysis of network dynamics: (*i*) it popularized the IC and LT models described above; (*ii*) it showed that the influence maximization problem for both models is NP-hard; (*iii*) it popularized

the use of submodular functions, which have been used in a host of influence studies (e.g., [36, 37, 41, 17, 20, 11, 39, 50]); and (*iv*) it showed that both IC and LT models were submodular.

This last contribution has particular ramifications. It means that the greedy hill-climbing algorithm introduced in [49] could be used for obtaining approximate solutions to the influence maximization problem. Further, this hill-climbing algorithm has a provable performance bound that is within $(1 - 1/e - \epsilon)$ (=63%) of optimum for arbitrarily small $\epsilon > 0$ (that is, it is a $(1 - 1/e - \epsilon)$-approximation) [49]. Thus, the spread size resulting from seeding the nodes in the computed influential node set I is guaranteed to be within 63% of the optimal spread size for the IC and LT models.

Submodular functions capture the idea that the incremental increase to a function f, with the addition of a node u to an argument set S, will be at least as great as the incremental increase if u is added to a set T, where $S \subseteq T$. Formally, f is submodular if $f(S \cup \{u\}) - f(S) \geq f(T \cup \{u\}) - f(T)$ for $S \subseteq T$. A function f is *monotone* if for all $S \subseteq T$, $f(S) \leq f(T)$. In the formulations of [35], f and the spread size $\sigma(I)$ from a seed set I, are both submodular and monotone [35].

Not all functions are submodular. For example, the threshold model of section 2.3, used in numerous empirical and theoretical studies (e.g., [15, 25, 31, 51, 55, 34, 38]), is not submodular. To see this, consider the graph $G(V, E)$ that is a clique on $n \geq 3$ nodes. To each node assign the θ-threshold function, where $1 < \theta < n$. Let $\sigma_G(Q)$ be the function that computes the number of affected nodes, given a set Q of nodes that are in state 1. Let $S = T = \emptyset$; i.e., S and T are initially empty. Then $\sigma(S) = \sigma(T) = 0$ for threshold θ. Select any one node $u \in V$, and $S \cup \{u\} = \{u\}$. The resulting number of affected nodes, is $\sigma(S \cup \{u\}) = 1$, since all nodes have threshold-2 (the 1 comes from the fact that $s_u = 1$). Now, add to T the set whose elements are u and any other $\theta - 1$ vertices $\{y_1, y_2, \ldots, y_{\theta-1}\}$ in V. Then $\sigma(T \cup \{u, y_1, \ldots, y_{\theta-1}\}) = n$ because any θ nodes in the clique G that are affected will cause all other nodes to be affected. Now, examine the submodularity definition. We have $S \subseteq T$. But $f(S \cup \{u\}) - f(S) = 1 - 0 = 1 \not\geq f(T \cup \{u\}) - f(T) = n - 0 = n$. Thus, the threshold function is not submodular.

The basic greedy hill climbing algorithm used in [35], a variant of which was proposed in [23], is given in Algorithm 1. Note that this is a particular implementation of the greedy strategy where simulation is employed to compute spread sizes $\sigma(I)$ so that the influential node sets I can be determined; cf. line 1. This algorithm provides a $(1 - 1/e)$ approximation gaurantee for both the IC and LT models [35]. The general algorithm is provided in Algorithm 2. Note that line 1 in Algorithm 2 is the generalization of line 1 in Algorithm 1. The goal of many of the influence maximization schemes described below is to find methods to compute v in line 1 of Algorithm 2 in less time, using less memory, and/or providing a better solution.

Parts of this lineage of [35] is illustrated in Figure 4.1, which provides selected methods from the literature and our view of their dependencies. Greedy hill climbing is the work of [35]; other works are described below. The methods are arranged top-down and left to right, from

Algorithm 1: Greedy Hill Climbing of KKT [35].

input : Graph $G(V, E)$, dynamics model M, number of diffusion instances n_j, maximum time per diffusion instance t_{max}, number k of influencing nodes to find.
output: Set A of maximum influencers, of cardinality k.

Read in inputs: $G(V, E)$, n_i, t_{max}, k.
// Loop over diffusion instances.
for *($j = 1$; $j \leq n_j$; $++j$)* **do**
    Set A_j, the influence set for diffusion instance j, to empty.
    for *($i = 1$ to k)* **do**
1   Using simulation over t_{max} time steps with the dynamics model M, where there is one seed node $v_i \in V$ per simulation instance, let v_i be the node (approximately) maximizing the marginal gain $\sigma(A_j \cup \{v\}) - \sigma(A_j)$.
        Set $A_j = A_j \cup \{v\}$.
Assign to A the maximum k nodes that occur most frequently in the n_j sets A_j.
Return A.

Algorithm 2: General Greedy Hill Climbing

input : Graph $G(V, E)$, dynamics model M, number k of influencing nodes to find.
output: Set A of maximum influencers, of cardinality k.

Read in inputs: $G(V, E)$, k.
Set A to empty set.
for *($i = 1$ to k)* **do**
1   Let $v \in V \backslash A$ be the node (approximately) maximizing the marginal gain $\sigma(A \cup \{v\}) - \sigma(A)$.
    Set $A = A \cup \{v\}$.
Return A.

earliest works to most recent works. For example, LDAG and SIMPATH are related in that both characterize local graph structure in the vicinity of a node v to estimate the influence on v and to estimate v's influence on nearby nodes. The LDAG method uses DAGs (directed acyclic graphs) while SIMPATH uses all paths between v and all of its nearby nodes.

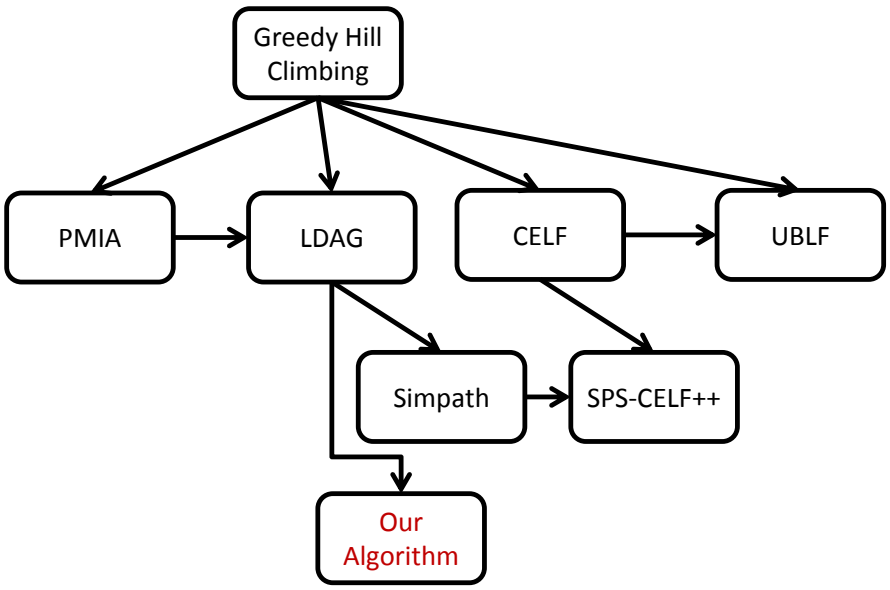


Figure 4.1: Relationships among selected influence maximization algorithms from the literature, and our algorithm. For example, the PMIA and SIMPATH influence maximization methods are related in that both use the local graph structure for a node v to compute the influence (contagion spread) from nodes in the vicinity of v. See Table 4.1 for meanings of arrows. Our method builds on the LDAG method.

We now turn to other works on influence maximization. Optimal sensor placement to detect cascades is the subject of [41]. However, the problem can also be cast in a manner—using *reductions* in penalty functions— so that it becomes the same as an influence maximization problem. The reduction function can be written as an absolute reduction, or as a penalty reduction per unit cost. By using both formulations, and at each time, choosing the node v that provides the largest *gain* in penalty reduction considering both methods, a submodular scheme is produced, generating a $(1/2)(1-1/e)$ approximation guarantee. By noticing that most networks are sparse (i.e., average degree $\ll O(n)$) an inverted index approach and a

Table 4.1: Meanings of arrows in Figure 4.1.

From	To	Meaning
CELF	SIMPATH	CELF algorithm used in SIMPATH.
CELF	SPSCELF++	CELF algorithm used in SPS-CELF++.
Greedy Hill Climbing	PMIA, LDAG, SIMPATH, CELF	These algorithms use the submodularity ideas from the Greedy Hill Climbing approach.[35].
PMIA	LDAG	The idea that local structures around a vertex approximately defines the influence on, and influence exerted by, a node v; this is an idea applied to the IC model, and then used in the LT model.
LDAG	SIMPATH	The idea that local structures around a vertex approximately defines the influence on, and influence exerted by, a node v; this is an presented in an LT model that is used in another LT model.
SIMPATH	SPS-CELF++	SPS-CELF++ uses SIMPATH-SPREAD to estimate spread size
CELF	UBLF	UBLF reduces the number of Monte-Carlo simulations of CELF using a upper bound function.

priority queue can be employed. Because of the submodularity property, nodes are ordered in non-increasing order of the reduction penalty. Then, after a node s is selected as an influential node, one starts at the top of the ordered list of nodes and continues to evaluate nodes (to update their gain in penalty reduction) until a node s''s recomputed gain in penalty reduction does not change much, and hence its rank does not change. By submodularity, no node with lesser *previous* gain can produce a larger gain than s', after s is removed. Hence, the entire set of nodes does not have to be re-evaluated, and significant savings is realized in not recomputing the gain in penalty reduction for each node; this is the *lazy* property in the cost-effective lazy forward (CELF) selection process. The sparsity condition also helps to localize changes in gain in penalty reduction for nodes, further reducing the number of nodes whose gain must be recomputed. They note that in one case, this lazy approach results in a 700× improvement in speed, compared to the non-lazy scheme. The algorithm is tested on small networks, from hundreds to a couple tens of thousands of nodes, and very small average degrees, or on a larger network with, again, a very small average degree.

Several algorithms for influence maximization are considered in [19]. A NewGreedyIC approach uses the same idea in [35] for reasoning about the IC model: pre-compute active edges (edges over which contagion spreads) in graph G. Here, graphs are undirected and edge weights are symmetric. One can compute both $\sigma(S)$ and $\sigma(S \cup \{u\})$ efficiently by depth or breadth first search. They find that correlation biases are not significant on real networks. A MixedGreedyIC method uses the NewGreedyIC method for the first iteration (i.e., the first

influential node), and then uses CELF [41] for all further iterations. The NewGreedyWC method uses a slightly different approach to address undirected graphs with unsymmetric edge weights (i.e., $w_{(u,v)} \neq w_{(v,u)}$). The approach essentially focuses on strongly connected components and edges between them. MixedGreedyWC uses the NewGreedyWC method to compute the first influential node, and CELF to compute all subsequent influential nodes. SingleDiscount selects as the next most influential node the node with the greatest discounted degree. A node's discounted degree is its original degree, minus the number of its distance-1 neighbors that have already been selected as influential nodes. The DegreeDiscountIC method discounts a node v's degree based on the number of its distance-1 neighbors that are already in the influential set, but the discount process is based on an analysis of a star subgraph centered at v; this discount is greater than the discount in the SingleDiscount model. The models are applied to two realistic networks. When execution time is critical, the degree discount methods are more attractive, but when quality of solution is important, the two mixed strategies (involving CELF) are preferred.

Influence maximization under the IC model is studied in [17]. It is shown that the problem of computing the spread size from a specified seed set I in the IC model is NP-hard. The authors' in [17] devise a maximum influence arborescence (MIA) model to compute influential nodes, and show that the function for computing $\sigma(I)$ using the MIA model is submodular and monotone. Therefore, the $1 - 1/e$-approximation bound holds for the MIA, based on [49]. MIA uses the idea that a node v interacts mostly with other nodes that are within a short geodesic distance of it. This assumption paves the way for a more efficient method for computing influential nodes. To compute node v's influence, and the influence exerted on v by other nodes, efficient, they use arborescences rooted at v. An in-arborescence (respectively, out-arborescence) is a tree in a directed graph such that all edges are directed towards (respectively, away from) the root. An in-arborescence quantifies other nodes' influence on v; an out-arborescence quantifies v's influence on its neighbors. The depth of these arborescence structures is controlled by a tunable parameter η such that the probability of a path from v to another node u (which is the product of the weights of the edges forming the path) is at least η. A large value of η leads to arborescences of lesser depth and hence more restricted consideration of local influence. A small value for η produces larger arborescences, but at greater cost (execution time). Activation probabilities are computed from the arborescences that characterize influence. These activation probabilities are used to compute influential nodes. Each additional influential node can require updates to other nodes' activation probabilities, and they provide efficient linear update schemes with provable properties. Other refinements include altering arborescences of nodes that have not yet been selected as influential, so that these arborescences do not include paths containing already-selected influential nodes. This modification, called prefix excluding MIA (PMIA) is shown to be *sequence submodular*. These latter modifications are detailed in [18, 58]. They run tests on four networks and compare their model to CELF [41], degree discount [19], a shortest path heuristic [37], and PageRank [52]. Interestingly, all of the networks have small average degrees, in the range 4 to 13. One wonders how these methods would perform on networks with higher average degree where arborescences would presumably be larger.

Influence maximization under the LT model is studied in [20, 21]. It is shown that the problem of computing the spread size from a specified seed set I in the LT model is NP-hard. The authors' in [21] describe a heuristic to compute influential nodes for the LT model. It is called Linear Threshold Directed Acyclic Graph (LDAG) heuristic. Conceptually, they use the same approach as in [17] for the IC model: evaluate local influence of (and on) a node v by assessing directed cyclic graphs (DAGs) rooted at v. They assume that v's influence is only propagated on the DAG rooted at v. Again, the depth of a DAG is controlled by a parameter η where $0 < \eta \leq 1$ controls the minimum weight of a path to v, in a DAG rooted at v (the path weight is the product of the weights of the edges that form it). They show that the influence maximization problem in their LDAG dynamics model, where nodes only propagate their influence on these respective DAGs, is NP-hard. Now, $\sigma(I)$ in the LDAG dynamics model is both submodular and monotone, and hence the seed set I in the LDAG model has a $1 - 1/e$ performance guarantee on the spread size, just as the original models evaluated in [35]. However, one can compute the spread size on DAGs efficiently. At issue is how to construct these DAGs. They specify the following properties for a DAG rooted at v, DAG(v), on a graph $G(V, E, W)$, where W is the set of edge weights: (i) DAG(v) should be an induced subgraph of G; (ii) $Inf_D(u, v) \geq \eta$ (meaning that the total path weight $Inf_D(u, v)$; i.e., the influence probability from u to v for all paths from u to v, should be at least η); and (iii) $\sum_{y \in X} Inf_D(u, v)$ is maximum among all DAGs rooted at v, where X is the node set of a DAG. They show that it is NP-hard to find such DAGs, and propose a greedy algorithm that constructs DAG(v) by successively adding a node u from X (i.e., to nodes currently in DAG(v)) that has an edge to at least one node of DAG(v) in G, such that the new path weight to v is maximum among all candidate u. This scheme is efficient. Note that this LDAG algorithm does not have a performance guarantee. They use methods to update influence, in the form of activation probabilities, as new influential nodes are found. There are two points of interest. First, their approach does not make use of the thresholds assigned to nodes. Their companion method for the IC model does (since there is an implicit threshold of 1 for each node). They show that their LDAG method performs just as well or better than other methods in identifying seed nodes to spread a contagion, and that it scales well. Hence, it is natural to wonder if including thresholds would increase the model's effectiveness. Second, all of the networks again have small average degrees, in the range 4 to 13; they are the same networks used in [17]. So, one wonders again how the methods would do with greater-degree networks where arborescences would presumably be larger.

An upper bound on the spread size, which can be used to prune unnecessary spread estimations (Monte-Carlo calls) in the CELF algorithm is proposed in [59]. In the initialization step, CELF needs to estimate the spread size using Monte-Carlo simulations for each node in a graph, resulting in n Monte-Carlo calls. Consequently, the method is very slow for large graphs. Based on an upper bound, the authors propose a new greedy algorithm, Upper Bound-based Lazy Forward (UBLF), which outperforms the original CELF algorithm. The nodes are ranked based on their upper bound scores and algorithm uses the upper bound scores to limit the Monte-Carlo simulation. The authors then go on to explain the relation between upper bound estimated (which is used as node's influence) and the actual influence

of the node. Researchers in [10] have proposed a near optimal time algorithm for the IC model. The approach to seed selection is through a polling process where nodes are picked at random after a hyper graph construction. The main motive of the approach is not to compute influence for all seeds in the graph. Instead, their algorithm uses a sampling method to generate a sparse hyper-graph of the network which allows them to estimate the influence of a node. The hyper-graph encodes the influence estimates. The algorithm them proceeds by selecting node in a greedy fashion to provide a hyper-graph of maximum degree. The algorithm also provides provisions for early termination and runs in sublinear time.

Influence maximization strategies have also been studied for time varying graphs and under other constraints. The problem of influence maximization in dynamic networks is studied in [60]. They proposed a new algorithm for probing influence diffusion called Maximum Gap Probing (MGP), which maximizes the change of solution obtained by probing. The intuition is to probe the nodes which are expected to bring the biggest change to the approximated solution on the observed network. The idea of incorporating time factor in influence maximization algorithm is described in [43]. The time constrained influence maximization problem is defined based on Latency Aware Independent Cascade influence propagation model. The authors of [43] propose a greedy algorithm, a simulation, and two algorithms based on Influence Spread path to address time constrained influence maximization in social networks. The greedy algorithm proceeds by selecting nodes with the maximum marginal influence and adds them to the influential set. The simulation-based algorithm and the Influence Spreading Path algorithm take in the time factor into account and return a set of activated nodes given a initial seed set. Time critical influence maximization problem, where one wants to achieve maximum influence within a deadline is presented in [16]. To address the deadline, the Independent Cascade model is extended to incorporate a time delay. The MIA-M algorithm proposed in this paper uses an augmented length path to take the deadline constraint into account. Two algorithms have been proposed to this problem: one based on a dynamic programming procedure, and a second one converts the problem to a problem under the IC model and applies fast heuristics. The algorithm effectively computes Maximum Influence in-arborescences for all nodes with a propagation probability limit and an augmented length. The algorithm then greedily selects nodes and updates marginal gain using the nodes' in-arborescence.

The first attempt to investigate the influence maximization problem in online social networks with both friend and foe relationship is described in [42]. This paper also studies the influence diffusion and the influence maximization in signed networks. The relationships are modeled using positive and negative edges in the graph. The paper extends the voter model to signed digraphs and provides a detailed mathematical analysis for the model. They show that the steady state dynamics depends on the graph structure: balanced graphs, anti-balanced graphs, and strictly balanced graphs. The influence maximization problem has been studied under the voter model for signed digraphs in two forms. One is instant influence, which counts the total number of influenced nodes at a time step $t > 0$ and average influence, which takes the average number of influenced nodes within the first t time steps. The

algorithm is evaluated extensively on both real-world and synthetic networks and the results demonstrate that the algorithm performs better a host of heuristics. The Least Cost Rumor Blocking ($LCRB$) problem where rumors originate from a community in the network and a notion of protectors being used to limit the bad influence of these rumors is studied in [26]. The problem is defined as identifying a minimal subset of individuals as initial protectors to minimize the number of people infected at the end of the diffusion process. The network structure of the graph is taken into consideration and minimum protectors are employed to protect bridge ends. The greedy algorithm proposed in the paper has been evaluated on different parameters such as network density, community size and rumor originators. The experimental results show that the greedy algorithm and the $SCBG$ (Set Cover based greedy algorithm) algorithm outperform the two heuristics: $MaxDegree$ and $Proximity$.

Researchers in [8] put forward important questions to answer before selecting influencers. Some of the questions are: where exactly does the influence of an influencer lie? How is it distributed? On what type of actions (or products) is an influencer influential? What are the demographics of its followers? The influencer validation improves the confidence and ensures marketer's satisfaction. The paper finds the solution to the questions raised and presents a greedy algorithm. They formulate the problem of providing explanations (called PROXI) as a discrete optimization problem of feature selection; i.e, to find upto k selections, each containing one or more features, while maximizing the coverage. The objective function and the intuitive greedy heuristic are evaluated on two real-world datasets Twitter and Flixster. [**?**] addresses the fact that individuals attribute different costs for becoming early adopters in the process of influence maximization. The paper discusses a constant-factor approximation mechanism for the Coverage, Linear Threshold, Independent Cascade, the Voter and general sub modular influence models. The mechanism is evaluated by experiments of real data sets - network data with a million nodes and 72 million edges, together with a cost-distribution by running a simulated campaign on Amazon's Mechanical Turk platform. When compared against the greedy benchmark, in this model, as the budget increases, the gap between the two mechanisms grow. [45] show that in a deterministic linear threshold model, there is no $n^{(1-e)}$ polynomial factor polynomial time approximation for the problem unless $P = NP$.

4.2 Minimum Sized Conversion Sets

The target set or the conversion set problem, presented in Chapter 3, was first posed in [14, 15, 24, 25]. The complexity result (the decision problem is NP-Complete) is shown to hold for the case where any single node has threshold $\theta \geq 3$ [25]; in [15], the result is also shown to hold when $\theta = 2$. Solutions (i.e., the sizes of conversion sets) for a number of stylized networks, including paths, cycles, bipartite graphs, and toroidal grids are given in [25]. Chen [15] gives the first approximability results for the majority thresholds model, showing that the majority thresholds model have the same approximation ratio as the general case; i.e.,

the problem cannot be approximated within a ratio of $O(2[\log n]^{1-\epsilon})$, for any fixed constant $\epsilon > 0$, unless NP$\subseteq$DTIME(n polylog (n)). For the special case of trees, [25] provides a polynomial time algorithm to compute optimal conversion sets when all node thresholds are the same. An algorithm for the non-uniform threshold case is provided in [14]. Both of these works assume that a node's threshold is not larger than its degree.

Beyond trees, [6, 7] provide an algorithm for computing conversion sets for graphs with bounded tree width. The tree width of an undirected graph is a number associated with the graph. It is the size of the largest vertex set in a tree decomposition of a graph or the size of the largest clique in a chordal completion of a graph. For a graph with n nodes and upper bound tree width w, the algorithm in [7] runs in time $n^{O(w)}$, and can be adapted to directed graphs, and graphs with weighted edges and nodes. However, there is no implementation of this algorithm, and correspondingly, no experimental evaluation of its performance. Also, they argue that tree width characterizes the time complexity of any target set selection algorithm, since it is unlikely that the time complexity could be less than $n^{o(\sqrt{w})}$. Finally, in [7] the authors provide the first results for a **non-progressive** (non-monotone) dynamics model. For each node v and its assigned threshold θ_v, v may transition from $0 \rightarrow 1$ and $1 \rightarrow 0$. It transitions up to 1 as in the progressive case: when the number of its neighbors in state 1 is at least θ_v. It transitions down to 0 when the number of its neighbors in state 1 is less than θ_v. They prove that the non-monotone target set selection problem is #P-hard.

A genetic algorithm is used [57] to compute solutions to the target set selection problem that converge in probability to the optimal solution. However, the graphs on which they run evaluations are quite small: at most 1500 nodes, with the maximum average degree of any graph being 15 (most are on the order of 3 to 5). Thus, whether the algorithm can scale to even moderate-sized graphs (e.g., roughly 20000 to 100000 node graphs) is an open issue.

A straight-forward, intuitively appealing algorithm for computing target sets for deterministic threshold dynamics is presented in [54]. The approach is to compute a set of seed nodes that will result in all nodes being affected. It uses a pruning process to remove nodes from a graph whose thresholds can be satisfied, based on their degrees. The process is fast, enabling evaluation of graphs with several million nodes. While there are no explicit steps to compute a small target set, the authors find in practice that target (seed) sets are routinely about 0.5% to 5% of the nodes in many graphs. A primary reason why such small target sets are produced is because thresholds for low-degree nodes are specifically reduced to enable these nodes to transition (i.e., their thresholds are reduced to be no more than their in-degrees). In real (e.g., mined) scale-free social networks, 35% to 50% of nodes are typically degree-1. If these nodes have thresholds > 1, then they cannot be affected unless they are seeded. An alternative approach that could be implemented is to identify nodes v that could not transition (because $\theta_v > d_v^{in}$), and simply remove them from the graph, in a pruning process that is similar to that for computing k-cores. Then one could focus on the resulting subgraph, and compute the target set on the subgraph of nodes that can possibly transition to state 1.

Chapter 5

Graphs

The set of graphs used to evaluate the influence maximization heuristics is shown in Table 6.1. All of the graphs in Table 6.1 are unweighted except *FB-02* and *FHS*. For the unweighted networks, the edge weights are assigned in the following fashion. For a vertex v, the weight on each incoming edge to v is set equal to $1/D_{in}(v)$ where $D_{in}(v)$ is the degree of v.

The graph classes SF and ED in Table 6.1 refer to scale free and exponential decay networks respectively. An ED (exponential decay) network has the following form (where exp is the exponential function):

$$p(d) = c_3 \exp(-c_4 * d)$$

where c_3 and c_4 are constants, d is a specified degree, and $p(d)$ is the probability that a node has that degree. An SF (scale-free) network has the following form:

$$p(d) = c_1 d^{-c_2}$$

where c_1 and c_2 are constants (both > 0), d is the specified degree, and $p(d)$ is the probability that a node has that degree.

Table 5.1: List of networks studied in this project. Networks were cleaned to eliminate redundant edges and self loops. Data here are for the giant component in each network. Two of the networks are weighted.

Networks	**Number Nodes,** n	**Number Edges,** m	**Average Degree,** d_{ave}	**Average Clustering Coefficient,** CC_{ave}	**Graph Class**
Epinions [33]	75877	508836	10.7	0.138	*SF*
Slashdot [33]	77360	905468	12.1	0.0555	*SF*
Wikipedia (Wiki) [33]	7066	103663	28.3	0.141	*SF*
Twitter/Tweet [22]	22405	59925	5.35	0.000850	*SF*
Enron (email) [33]	33696	180811	10.7	0.509	*SF*
Ca-Astroph [33]	17903	196972	22.0	0.633	*ED*
Ca-Condmat [33]	21363	91286	8.55	0.642	*ED*
Ca-Grqc [33]	4158	13422	6.46	0.557	*ED*
Ca-Hepth [33]	8638	24806	5.74	0.482	*ED*
Cit-Hepph [33]	34401	420783	24.5	0.286	*ED*
Ca-Hepph [33]	11204	117619	21.0	0.622	Ind.
FHS (weighted) [27]	10430	37103	7.11	0.530	*ED*
MONT-VA [12]	77,528	1,967,714	50.8	0.395	Neither
FB-01 [56]	63,392	816,886	25.8	0.222	*ED*
FB-02 (weighted) [56]	43,953	182,384	8.30	0.111	*ED*

Chapter 6

Pre-Existing Influence Maximization Algorithms

This chapter describes the existing influence maximization heuristics from the literature that have been implemented and evaluated in this study. These are listed in Table 6.1. We address the execution times for the various influence maximization algorithms in this chapter. The effectiveness of the algorithms to spread contagion is evaluated along with the results from the new algorithm in Chapter 7.

Table 6.1: List of Influence Maximization Algorithms Implemented from the Literature

Algorithm	**References**
Influence Maximization for the Independent Cascade model	[58]
Influence Maximization for the Linear Threshold model	[20]
Eigenvector Heuristic	[9]
High Degree Heuristic	[29]
Degree Discount Heuristic	[19]
Betweenness Centrality Heuristic	[28]
Random Heuristic	

6.1 Influence Maximization for the Independent Cascade model [58]

Chen et al. present a scalable algorithm that studies the influence maximization problem for large-scale social networks for the independent cascade model in [58]. The algorithm works by computing local arborescence structures for each node in the graph to approximate the influence propagation. The *PMIA* model presented in [58] assigns an influence value to each

node in the graph. The seed selection process picks the node with the maximum influence and updates the influence of other nodes due to the seed selected. For example, suppose vertices u and v can both infect vertex w. If v is selected as an influential node, then $u's$ influence on w is updated. This form of seed selection and update process continue until the desired number of influential seeds have been selected.
The algorithm was implemented in Java and used Java's Executor Service to achieve concurrency. We improved the running time of the algorithm by using multiple threads in parts of the algorithm which could be executed in parallel. The results of the performance improvements are presented in Chapter 8. The algorithm is divided into two major steps: (i) setup or the preprocessing step; (ii) seed selection and influence update step.

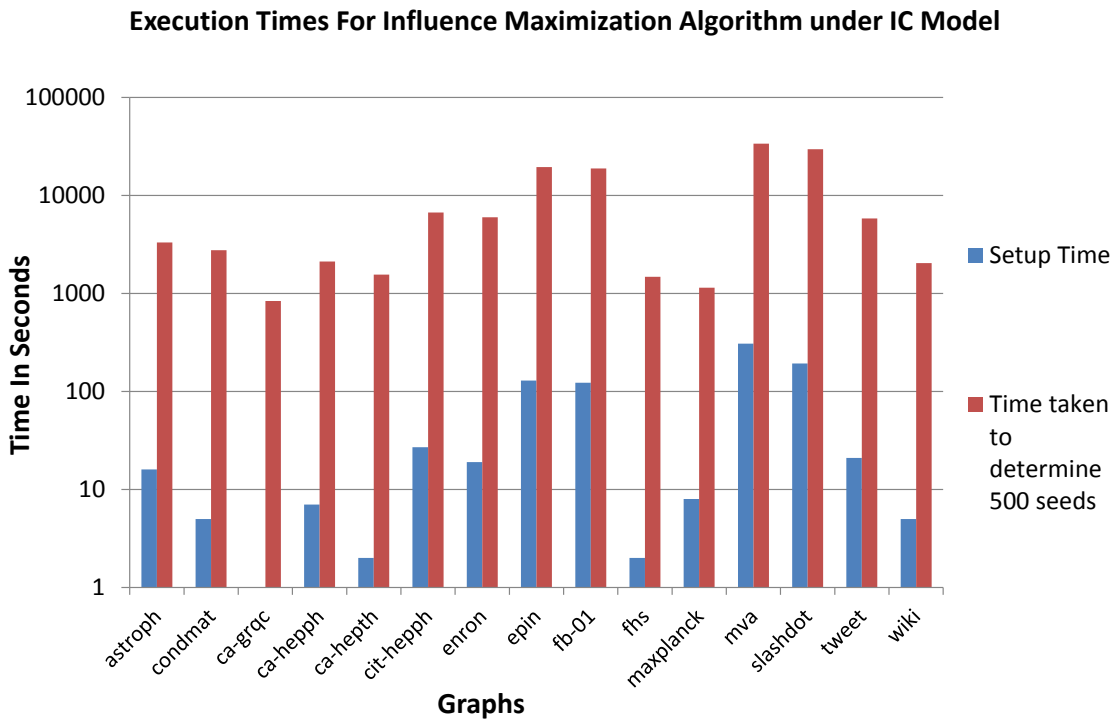


Figure 6.1: Execution Times for Chen's Influence Maximization Algorithm for the Independent Cascade Model.

Figure 6.1 shows the execution times for the two steps for various graphs. The blue line indicates the time taken for the preprocessing step and red line indicates the time taken for the seed selection and the update step for selecting 500 influential nodes. We use influential nodes, seeds and seed nodes synonymously. For the above experiments, 16 threads were used to execute the algorithm and the propagation probability limit, a parameter for the

algorithm was set as 0.0001 for all graphs except *FHS*. The propagation probability limit for *FHS* graph was set as 0.05. Most of the time taken involved in computing arborescence structures and recreating these structures for a few nodes after each seed selection step.

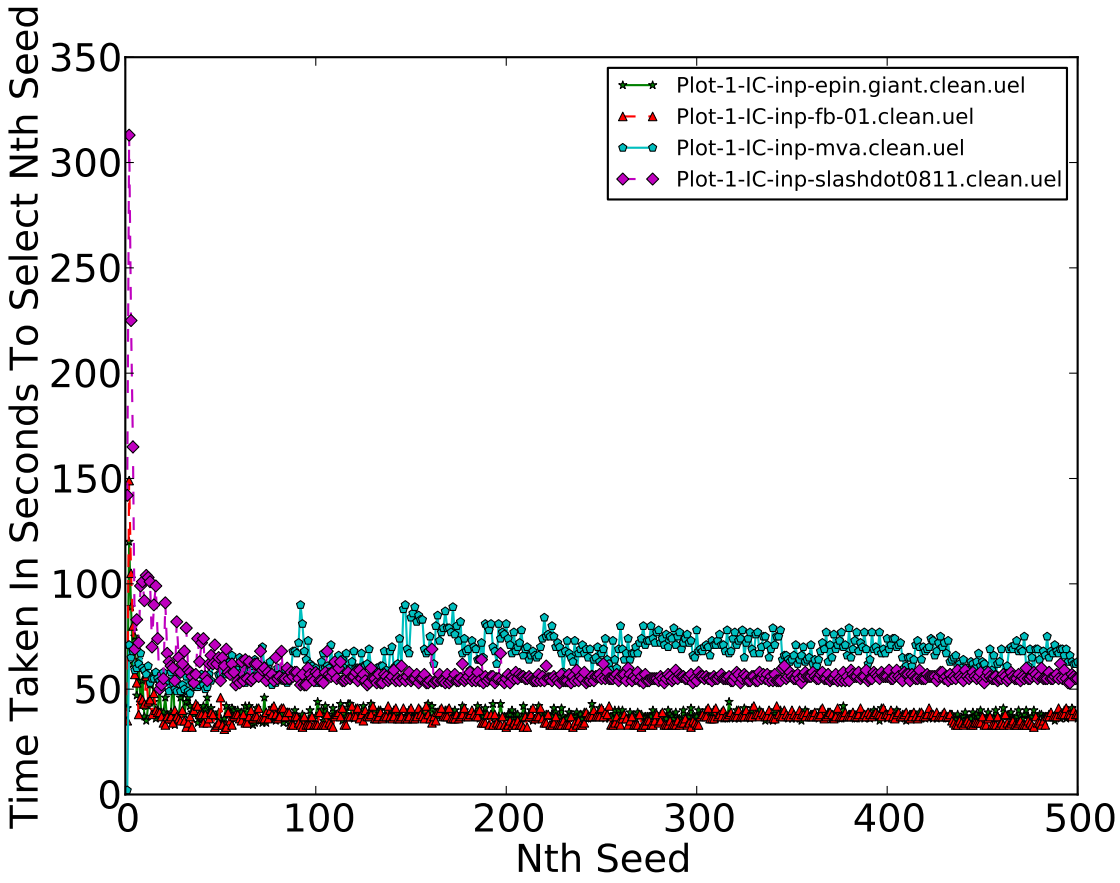


Figure 6.2: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Independent Cascade Model.

Figures 6.2, 6.3 and 6.4 show the time taken for each seed to be selected. The networks in Figures 6.2, 6.3 and 6.4 have been grouped arbitrarily. For most of the graphs, the time taken for selecting the initial set of seeds is relatively greater than those for later-chosen seeds. This is because initially the arborescence strutures are large, and a greater number of nodes require updating.

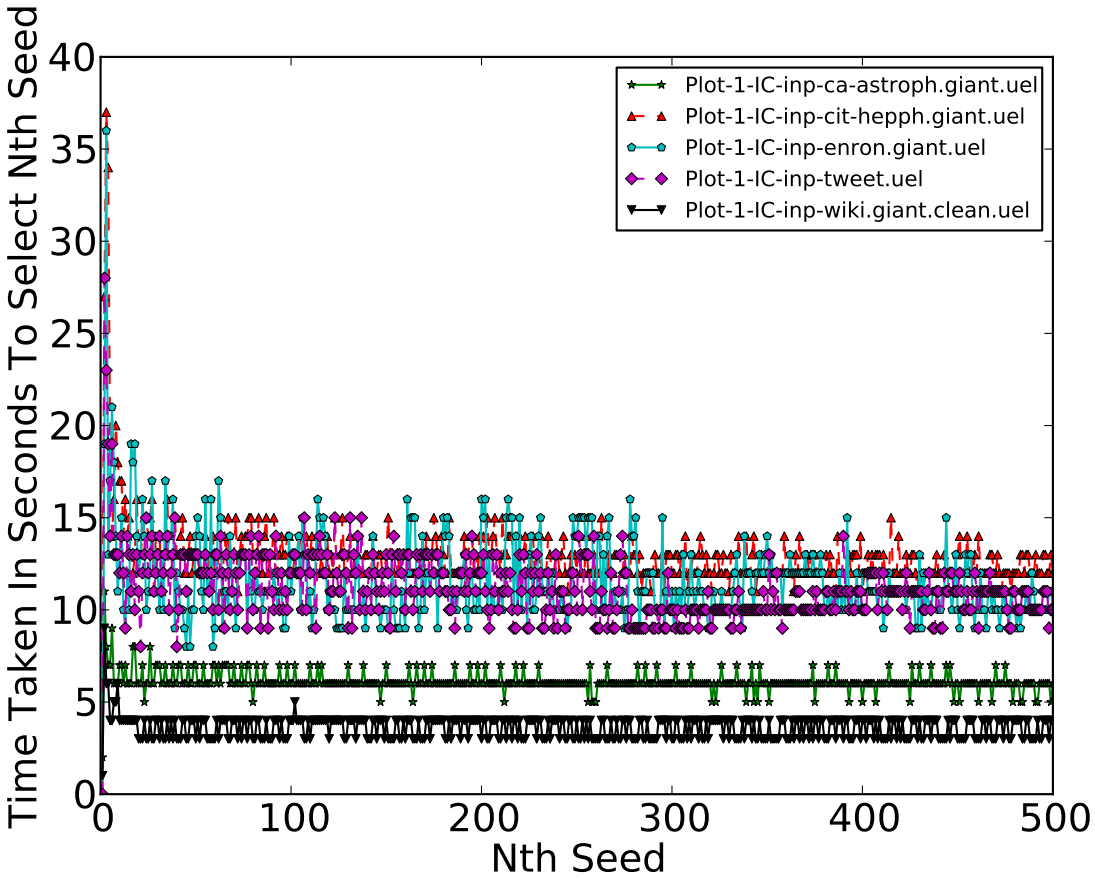


Figure 6.3: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Independent Cascade Model.

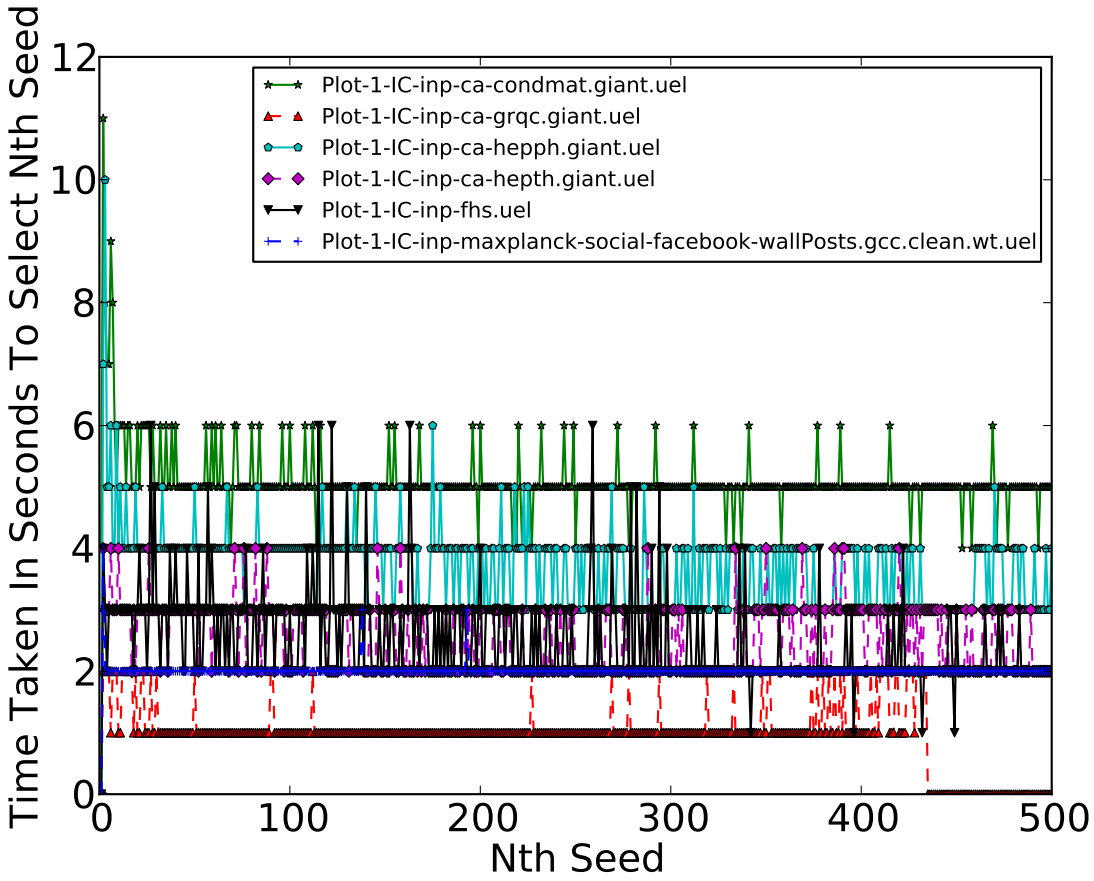


Figure 6.4: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Independent Cascade Model.

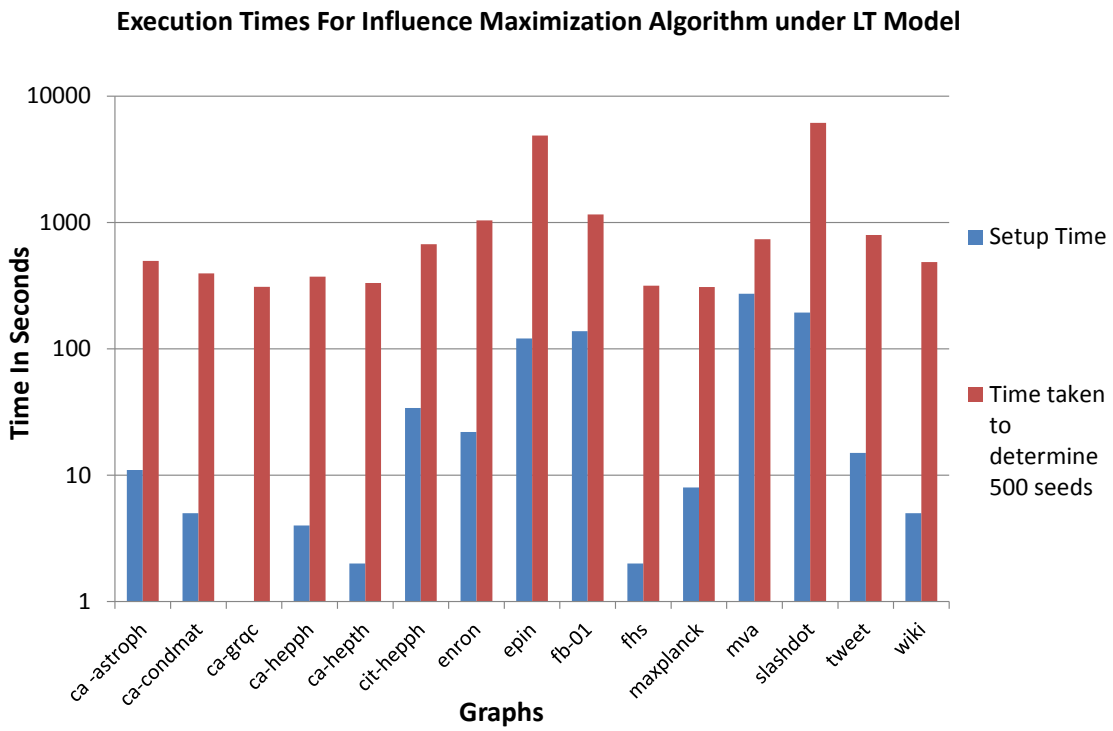


Figure 6.5: Execution Times for Chen's Influence Maximization Algorithm for the Linear Threshold Model.

6.2 Influence Maximization for the Linear Threshold model [20]

Chen et al. presented an algorithm for the influence maximization problem for the linear threshold model which involves computating $DAG's$ in time linear to the size of the graph. Unlike the algorithm for the independent cascade model in [58], the arborescence for each node v is computed only once during the preprocessing step. Only activation probabilities and influence values are updated during the seed selection step. Thus, less processing is performed.

The algorithm was implemented in Java and used Java's Executor Service to achieve concurrency similar to the parallel implementation in Section 6.1. The algorithm is divided into two major steps: (i) setup or the preprocessing step; (ii) seed selection and the influence update step.

Figure 6.5 shows the execution times for various graphs. The blue line indicates the time taken for the preprocessing step and the red line indicates the time taken for the seed selection and updating step for selecting 500 seed nodes. Similar to the experiments in Section 6.1, 16 threads were used to execute the algorithm. The propagation probability limit, a parameter for the algorithm, was set to 0.0001 for all graphs except *FHS*. The propagation probability limit for *FHS* graph was set as 0.05. The time taken to select seeds in this algorithm is less than the algorithm in Section 6.1. This is because the *LDAG* structures are not created or updated during the seed selection process.

Figures 6.6, 6.7 and 6.8 show the time taken for selecting each seed. The time taken for each seed selection in this algorithm is less than that taken by the algorithm in Section 6.1 as shown in these corresponding sets of figures. Thus the algorithm for the linear threshold model by Chen et al. runs much faster than the algorithm for the independent cascade model discussed in Section 6.1.

6.3 Other Heuristics

Along with the more intricate influence maximization algorithms described in Sections 6.1 and 6.2, several other heuristics from the literature have been implemented as shown in table 6.1.

The Betweenness centrality is a measure of a node's centrality in a network. It is equal to the number of shortest paths between all pairs of vertices that pass through a given node [28]. The Eigenvector heuristic assigns relative scores to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than similar connections to low-scoring nodes [9].The High Degree or the Degree Centrality computes the out-degree for each node in the graph where the out-degree

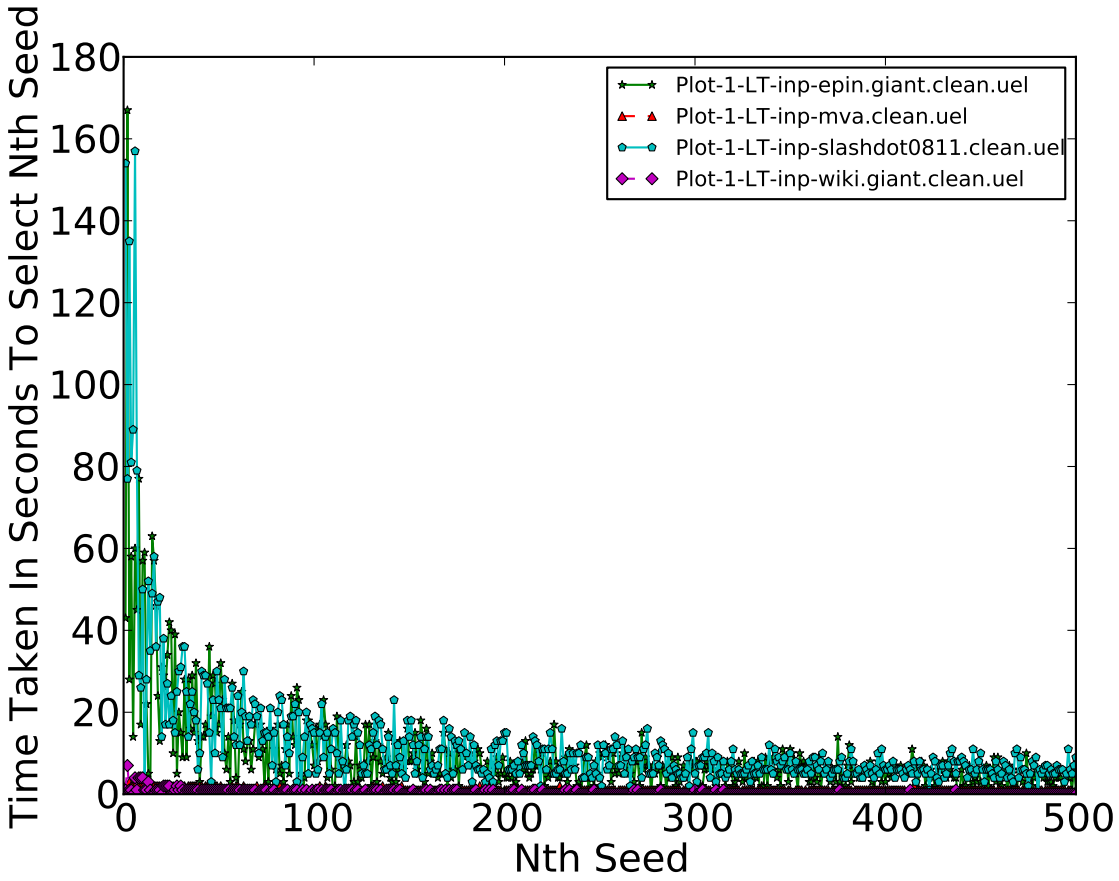


Figure 6.6: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Linear Threshold Model.

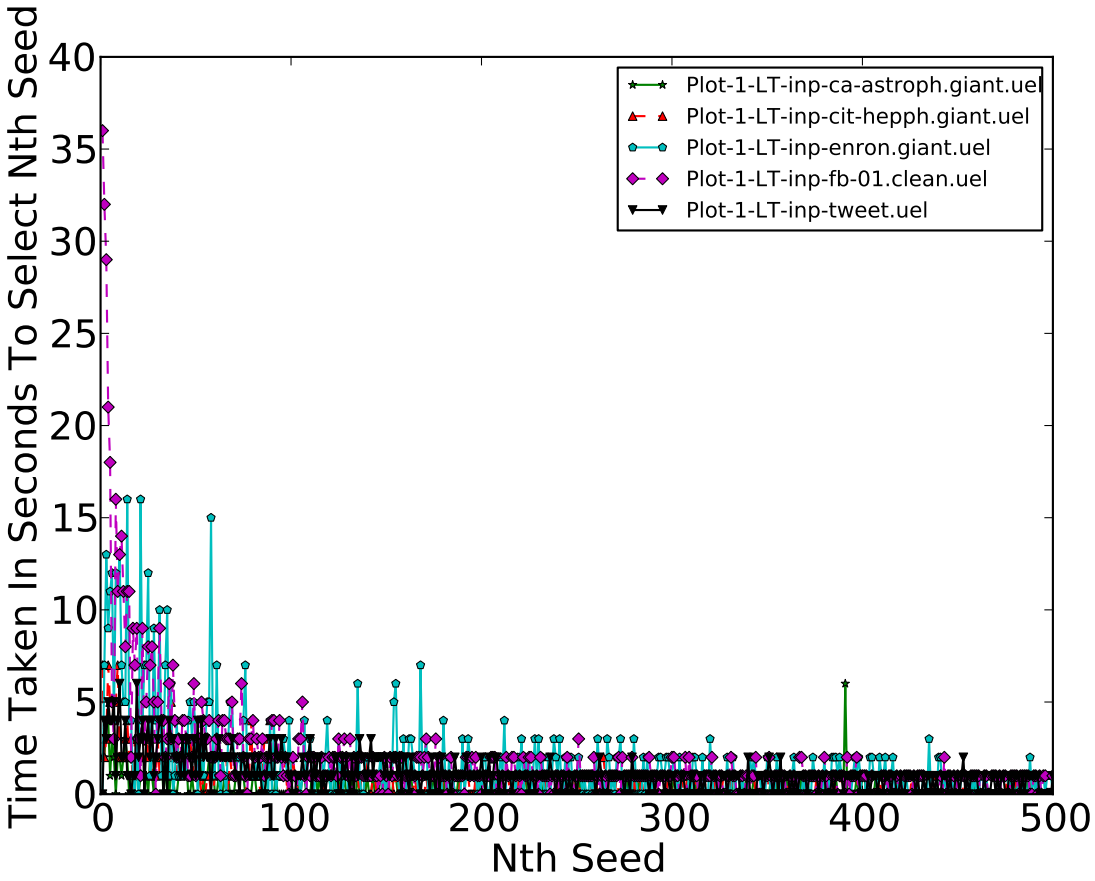


Figure 6.7: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Linear Threshold Model.

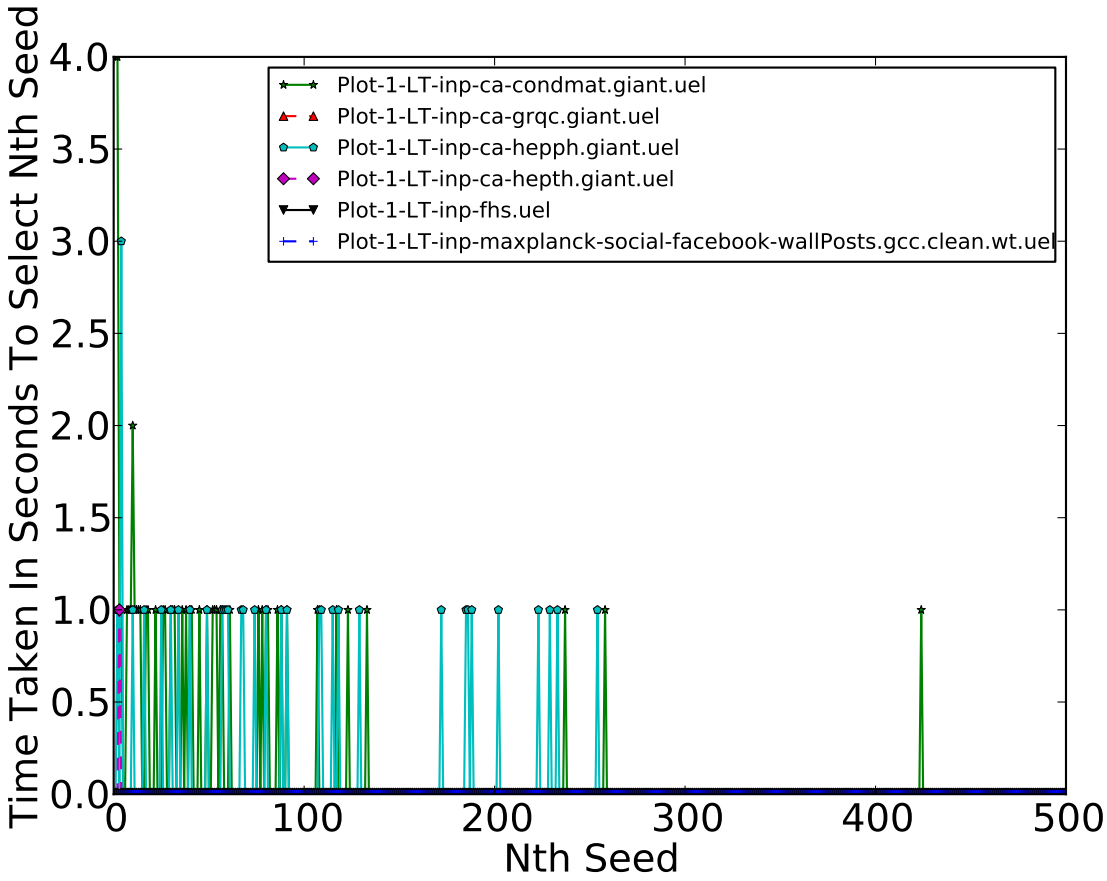


Figure 6.8: Time Taken to Select Each Individual Seed Based on Chen's Influence Maximization Algorithm for the Linear Threshold Model.

of a node is the number of outgoing edges from a given node [29]. In the Degree Discount heuristic, if node u is selected as a seed, then node u is removed from the graph and the degree of each of its neighbors is decremented by one [19]. The Random heuristic picks a node from the graph randomly.

These heuristics have been implemented in *Python* using *NetworkX* library [32]. *NetworkX* is a *Python* package for the creation, manipulation, and evaluation of the structure, dynamics, and functions of graph networks. All the heuristics have been implemented in a serial fashion.

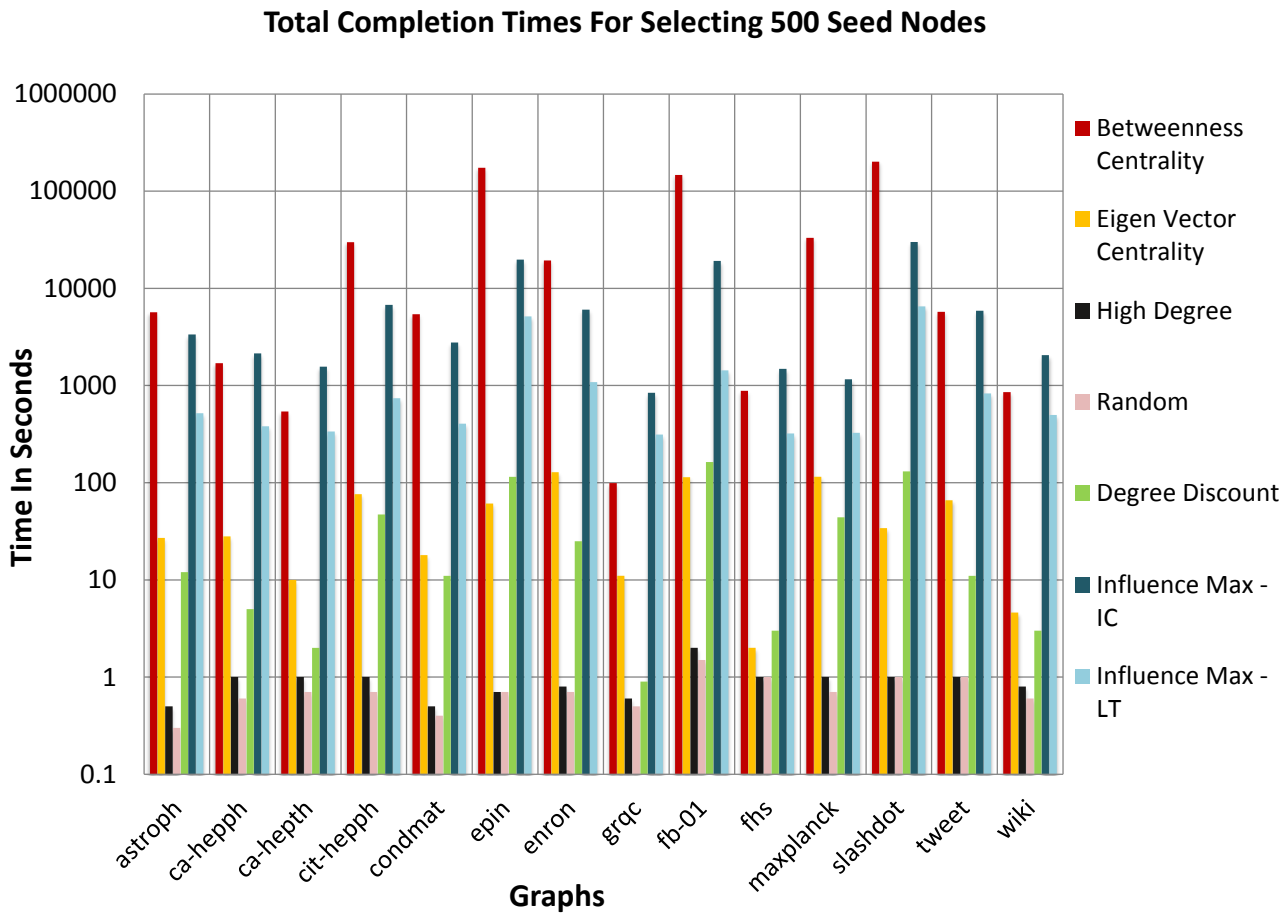


Figure 6.9: Comparison of Completion Times of Various Algorithms to Select the 500 Most Influential Seeds for Various Graphs.

Figure 6.9 shows the completion times of all the algorithms for various graphs. The Betweenness Centrality takes a very long time to complete compared to other algorithms. The Random and the High Degree heuristic take less than 1 second to complete in most cases. The influence maximization algorithms by Chen et al. of Section 6.1 and 6.2 take more time to complete than Eigenvector heuristic, High Degree, Random and Degree Discount

heuristic but is faster than the Betweenness Centrality for most of the graphs examined. The multi-threaded implementation of the algorithms in Sections 6.1 and 6.2 helps drive down the execution times.

Chapter 7

New Influence Maximization Algorithm

This chapter covers our proposed heuristic for the deterministic linear threshold dynamic model.

7.1 Threshold Difference Greedy Algorithm (TDG)

The Threshold Difference Greedy (TDG) algorithm works as follows. Informally, a node v's influence is captured recursively by determining whether it causes neighboring nodes to transition, and if not, by the residual that remains. We assume that nodes' threshold values are known. As an example, take a node v with an out-going edge to node u. The weight of edge (v, u) (i.e., edge from v to u) is w_{vu}, and the threshold of u is θ_u. The **residual** r_{vu} edge weight for edge (v, u) is

$$r_{vu} = w_{vu} - \theta_u \tag{7.1}$$

If $r_{vu} \geq 0$, then v influences u to transition from state 0 to 1. If $r_{vu} < 0$ then the edge weight is insufficient to cause u to transition state, and the value of r indicates how far away u is from transitioning. The influence of node v, $IncInfl(v)$, is the amount of influence exerted by v on all other nodes in the graph; i.e., it quantifies how much v influences all other nodes to transition from state 0 to state 1. The incremental influence of a node v, denoted $IncInfl(v)$, is the amount of influence assigned to v based on its effect on u. In the above example, if v by itself influences u to change state; i.e., if $r_{vu} \geq 0$, then $IncInfl(v)$ is incremented by a value 1. If instead v contributes to the transition of node u, but cannot cause u to transition on its own, then v gets partial influence in causing u to transition. If v contributes to u's transition, but u does not change state based on v's influence, then the

incremental influence $IncInfl(v)$ is incremented by value $(w_{vu} \div \theta_u)$. The intuition is that the closer w_{vu} comes to θ_u, the higher is the magnitude of $(w_{vu} \div \theta_u)$. The magnitude of $IncInfl(v)$ increases as w_{vu} increases. We illustrate some details of our algorithm through the following examples :

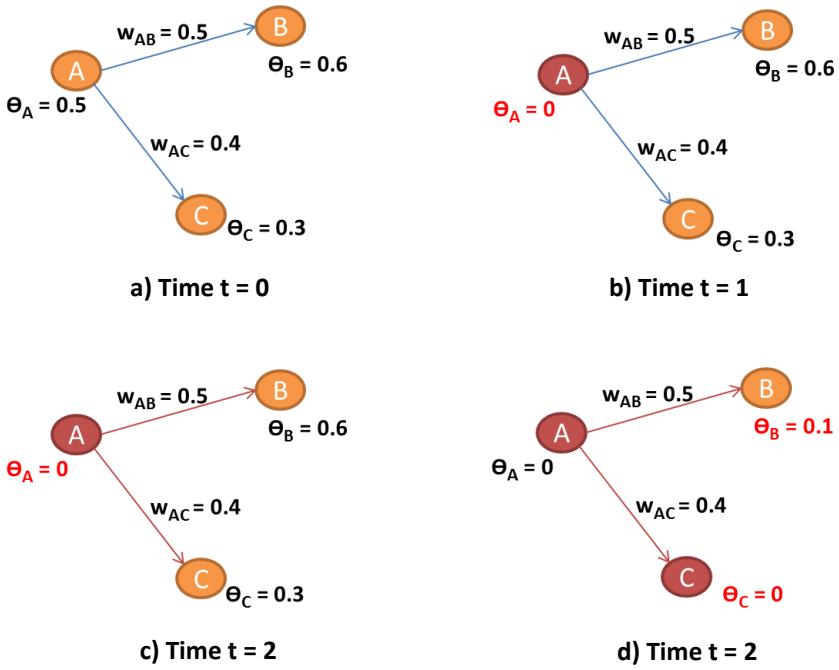


Figure 7.1: Figure showing diffusion spreading through nodes at various time stamps. θ indicates the threshold on each node and w indicates the edge weight.

Let us consider the graph in Figure 7.1. Node A has two edges pointing to nodes B and C. The thresholds of the three nodes A, B, and C at time t=0 are 0.5, 0.6, and 0.3 respectively. Initially the $IncInfl$ of all nodes is set to zero. We compute $IncInfl$ for these three nodes. Since nodes B and C do not have any outgoing edges from them, their $IncInfl$ is zero; i.e, $IncInfl(B) = 0$ and $IncInfl(C) = 0$. Node A has two outgoing edges to two other nodes. For node B, the weight of the edge w_{AB} is less the θ_B, and so A partially influences B. $IncInfl(A) = IncInfl(A) + ((w_{AB} \div \theta_B)) = 0 + (0.5 \div 0.6) = 0.833$. This also indicates that A contributes to meeting 83.3% of $B's$ threshold. For node C, the weight of the edge w_{AC} is more the θ_C, and A enables C to change state. $IncInfl(A) = IncInfl(A) + 1 = 0.833 + 1 = 1.833$.

Let us consider another example in Figure 7.2. Nodes A and C both have edges incident on B.

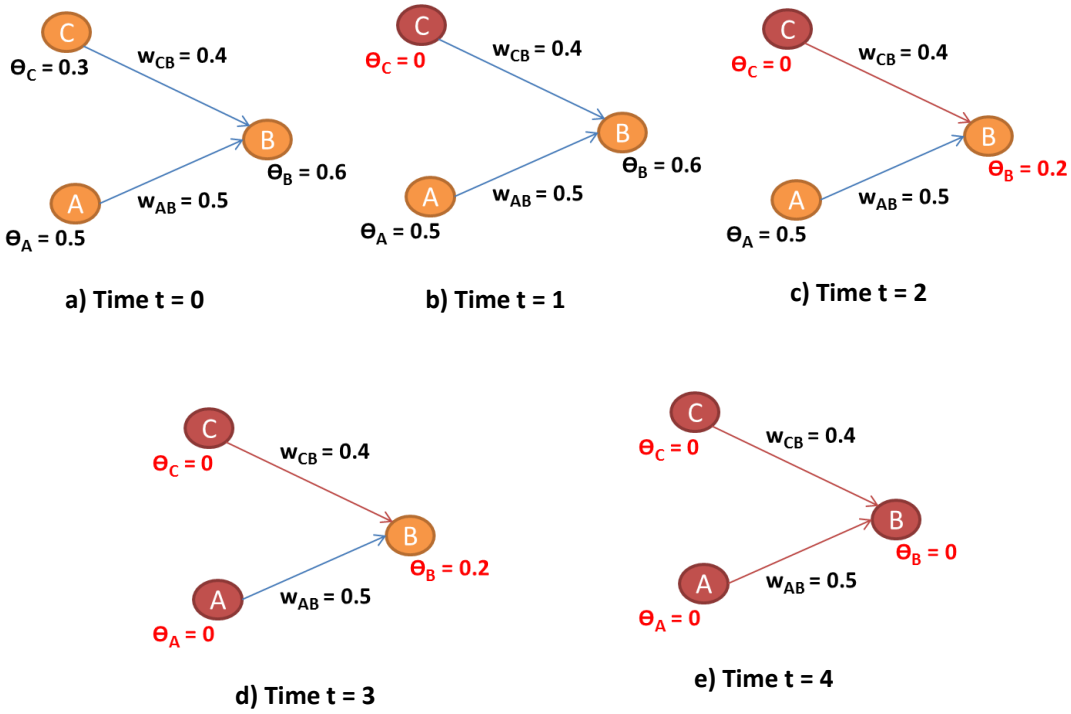


Figure 7.2: Figure showing diffusion spreading through nodes at various time stamps. θ indicates the threshold on each node and w indicates the edge weight.

Due to other nodes in the graph(not shown in the figure), the $IncInfl(A)$ and $IncInfl(C)$ are computed as 5 and 7 respectively and $IncInfl$ for edges to B are yet to be computed for both nodes. For edge w_{AB}, since the weight of the edge is less the θ_B, A partially influences B. Then, $IncInfl(A) = IncInfl(A) + ((w_{AB} \div \theta_B)) = 5 + (0.5 \div 0.6) = 5.833$. For edge w_{CB}, since weight of the edge is less the θ_B, C partially influences B. Then, $IncInfl(C) = IncInfl(C) + ((w_{AB} \div \theta_B)) = 7 + (0.4 \div 0.6) = 7.66$. The value of $IncInfl(B)$ is zero.

Continuing with Figure 7.2, the node with maximum influence C gets picked as a seed at time $t = 1$. Node C influences B and changes $B's$ threshold to 0.2. Node B's threshold is decremented by the influence provided to it by C; i.e, $\theta_B = \theta_B - w_{CB}$. Since $B's$ threshold has changed, $IncInfl(A)$ needs to be updated. Now since $w_{AB} > \theta'_B$ for the new threshold of B, A can completely influence B and changes B's state. So, $IncInfl(A) = IncInfl(A) + 1 = 6$. The *ComputeInfluence* function invoked for Node A will evaluate $IncInfl$ of A to 6. Once A is selected as a seed, then A can change the state of B as shown in the figure. Suppose a node v solely influences u such that u transitions state, then u's influence on its distance-1 neighbors is computed, and these influences are also attributed to v (not just u). This transference of influence is continued recursively until no such nodes can transition, or until a user-specified maximum depth d_1 is reached. Note that d_1 may equal ∞. The distance d_1 characterizes the extent of the graph that is used to compute the influence of v. As d_1 increases, the $IncInfl(v)$ is more accurate, but this comes at an increased cost. The computation takes place level by level starting from the initial node. This scenario has been explained in Figure 7.3. Node A has edges to nodes B, C, and D. Node A can completely influence nodes B and C but can only partially influence D as $w_{AD} < \theta_D$. But since A can change the state of B and C, B and C can in turn cooperate with A and influence D (each contributes a weight of 0.1) and the final outcome is that D's threshold is met and Node D changes state. Thus, if initially $IncInf(A)$ is α, then the scenario just described results in $IncInf(A)$ becoming $\alpha + 3$. The $IncInfl$ for D's change of state is credited to A.

The node v that is selected and added to the influential set is the node that has the highest $IncInfl(v)$ value among nodes which have not been selected as seeds, and which have not been infected. Once v is selected, the diffusion process is performed to a user-defined depth d_2 from v to determine the number of nodes that are affected owing to v's addition to influential set S. Note that d_2 may nor may not be equal to d_1, and can be set to ∞. As d_2 increases, the influence due to v is more accurately captured but at a greater cost.

Finally, we specify a distance d_3 such that the minimum distance between a new candidate seed v and any node in the infected set I is at least d_3. This is motivated by the complex contagion dynamics that we seek to characterize. A summary of all algorithm parameters is given in Table 7.1.

The Algorithm 3 presents a pseudo code of the TDG algorithm. It has subroutines to Algorithm 4 and Algorithm 5. Lines 5 to 7 in Algorithm 3 indicate a call to subroutine ComputeInfluence in Algorithm 4 invoked for every node in the graph. Note that the Com-

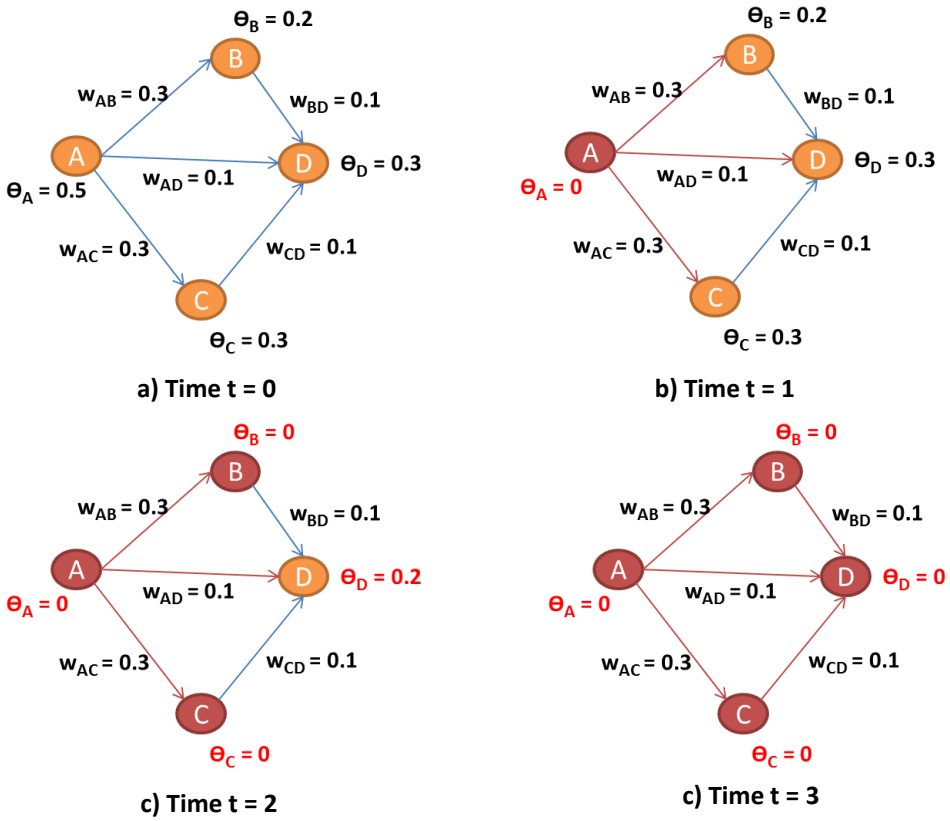


Figure 7.3: Figure showing diffusion spreading through nodes at various time stamps. θ indicates the threshold on each node and w indicates the edge weight.

Table 7.1: Parameters for the TDG Algorithm.

Parameter	**Description**
d_1	Depth to which influence of each node is computed v.
d_2	Depth to which diffusion is run to compute the number of affected nodes and update thresholds.
d_3	Minimum distance between a new node v to put into to the influential set, and any node already in the set I.

puteInfluence method for each node in the graph can be executed concurrently. Once the $IncInfl$ for all the nodes have been computed, the execution proceeds to Line 8 to 13 in Algorithm 3. In lines 9 to 11, the most influential node or the node with the highest value of $IncInfl$ is picked and is added to the set of seeds and the set of infected nodes. Line 13 involves a call to subroutine UpdateForNewSeed in Algorithm 5.

The Algorithm 4 contains the ComputeInfluence subroutine and shows the computation of $IncInfl$ for a node in the graph. The $IncInfl(v)$ is represented as Inf_v in the algorithm. The subroutine ComputeInfluence proceeds execution from node v using a level order traversal or the BFS traversal on $v's$ out-neighbors. The subroutine uses Queue Q for this purpose. In Line 7 of Algorithm 4, a node is fetched from the queue. Lines 8 to 17 indicate the influence computation due the neighbors of the node fetched from the queue. If the edge weight is greater than the threshold, then lines 10 to 13 are executed, the $IncInfl$ is incremented by 1, and the neighbor is added to the queue. If the edge weight is lesser than the threshold, then lines 15 to 17 are executed. Note that in line 11 there is a condition to see if the level is less than d_1. This condition must be true for a node to be inserted into the queue. The execution stops when the queue becomes empty.

The Algorithm 5 contains the UpdateForNewSeed subroutine. This function takes care of updating thresholds, incremental influence $IncInfl$, and the set of infected nodes. The subroutine executes similar to the ComputeInfluence subroutine. It proceeds execution by performing a BFS traversal on out-neighbors of seed node s. Lines 6 to 15 in the function perform the steps for updating the thresholds and the set of infected nodes. As shown in lines 6 to 11, if the edge weight of the neighbor is greater than the neighbor's threshold value, then the neighbor is added to the queue and the set of infected neighbors. Its threshold is updated to zero. If the edge weight of the neighbor is less than the neighbor's threshold value then its threshold is only updated (shown in line 13). For each of the neighbor w, a call to the function UpdateIncomingNeighborInfluence is made to update the incremental influence of all the nodes that can affect w (shown in line 15). Note that in line 10 there is a condition to see if the level is less than d_2. This condition must be true for the node to be inserted into the queue. The execution stops when the queue becomes empty.

The Algorithm 6 contains the function UpdateIncomingNeighborInfluence. This function uses a queue and performs a BFS traversal similar to Algorithms 4 and 5. But this subroutine traverses the incoming neighbors of a given node. For each of the neighbor traversed, the function calls the ComputeInfluence subroutine in Algorithm 4. If the edge weight is greater than the in-neighbor's threshold. then the entry is added to the queue. Note that in Line 8 there is a condition to see if the level is less than d_1. This condition must be true for the node to be inserted into the queue. The execution stops when the queue becomes empty.

The execution of the algorithm is explained with an example in Figures 7.4 and 7.5. Figure 7.4 shows the execution of the preprocessing step. Here the values for d_1 and d_2 are set as ∞. The $IncInfl$ for all the nodes are computed, of which the computation of $IncInfl$ for the three nodes A, D, and C are shown in the figure. After this step, node A which has the

Algorithm 3: InfluenceMaximization

input : Graph $G(V, E)$; Number of seed nodes required n_{max}; Vector θ indicating thresholds for each node with $|\mathrm{v}_\theta| = n$; A set of edge weights W, with $p_{(u,v)} \in W$ being the weight on the edge from u to v; Depth to which incremental influence is evaluated d_1; Depth through which infected nodes are computed d_2; Distance between the seeds selected d_3

output: Set S containing the most influential nodes of size n_{max}.

```
Read in all inputs
Set S = ∅, R = ∅
Set Inf_v = 0
// Compute initial influence for each node.
for (v ∈ V) do
    // See Algorithm 4.
    ComputeInfluence(v,R,v_θ,W,d_1,G,Inf)
for (i = 1 to n_max) do
    Select s = argmax_{v∈V\R}{Inf_v}
    S = S ∪ {s}
    R = R ∪ {s}
    // See Algorithm 5.
    UpdateForNewSeed(s,R,v_θ,W,d_1,d_2,G, Inf)
return S
```

Algorithm 4: ComputeInfluence

input : Node v; Infected node set R; Threshold vector θ; set of edge weights W with $p_{(u,v)} \in W$ being the weight on the edge from u to v; Depth to which incremental influence is evaluated d_1; Graph G; Influence vector Inf.

output: Updated influence vector Inf.

```
1  Instantiate: empty queue Q, empty pathlist L, level l.
2  /* Inf_v is the influence exerted by node v on its neighborhood
3  up through, and including, level d_1. */
4  Set Inf_v=0; P = P.append(v); Q.enqueue(v).
5  Store θ values into θ_temp
6  while (Q.notEmpty() ) do
7      u=Q.dequeue.
8      for (w ∈ G.OutNeighbors(u) \(R ∪ L) ) do
9          if ( p_(u,w) ≥ θ_w ) then
10             Inf_v = Inf_v + 1
11             if (l ≤ d_1) then
12                 Q.enqueue(w)
13             L = L ∪ {w}
14         else
15             Inf_v = Inf_v + (p_(u,w) ÷ θ_w)
16             // Temporarily update θ_w
17             θ_w = θ_w - p_(u,w)
18     Check and update l
19 Restore back θ values from θ_temp.
```

Algorithm 5: UpdateForNewSeed

input : Seed node s; Infected nodes set R; Threshold vector θ; set of edge weights W with $p_{(u,v)} \in W$ being the weight on the edge from u to v; Depth to which incremental influence is evaluated d_1; Depth to which diffusion is simulated d_2; Graph G; Influence vector Inf.

output: Updated threshold vector θ and Infected Nodes set R

```
1  Instantiate: empty queue Q, empty pathlist L, level l.
2  Set l = 1; P = P.append(s); Q.enqueue(s).
3  while (Q.notEmpty() ) do
4      u=Q.dequeue.
5      for (w ∈OutNeighbors(u) \(R ∪ L) ) do
6          if ( p_(u,w) ≥ θ_w ) then
7              R = R ∪ {w}
8              L = L ∪ {w}
9              θ_w = 0
10             if (l ≤ d_2) then
11                 Q.enqueue(w)
12         else
13             θ_w = θ_w − p_(u,w)
14         // See Algorithm 6.
15         UpdateIncomingNeighborInfluence(w, R, v_θ, W, d_1, G, Inf)
16     Check and update l
```

Algorithm 6: UpdateIncomingNeighborInfluence

input : Node v; infected node set R; threshold vector θ; set of edge weights W with $p_{(u,v)} \in W$ being the weight on the edge from u to v; depth to which incremental influence is evaluated d_1; graph G; influence vector Inf.
output: Updated threshold vector θ.

```
1  Instantiate: empty queue Q, empty pathlist L, level l.
2  Set l = 1; P = P.append(v); Q.enqueue(v).
3  while (Q.notEmpty() ) do
4      u=Q.dequeue.
5      for (w ∈InNeighbors(u) \(R ∪ L) ) do
6          ComputeInfluence(w,R,w_θ,W,d_1,G,Inf)
7          if ( p_(u,w) ≥ θ_w ) then
8              if (l ≤ d_1) then
9                  Q.enqueue(w)
10     Check and update l
```

highest $IncInfl$ value is selected as the seed node. Once node A gets selected as the seed, the update step is executed. The update step updates the thresholds and $IncInfl$ of other nodes in the graph. The nodes with updated threshold values is shown is Figure 7.5. It is to be noted that the $IncInfl$ of nodes C and D have changed after the update step of seed node A. Now Node C has the highest $IncInfl$ and is selected as the next seed node. All the nodes in the graph get infected after node C is selected as the seed. Thus the algorithm finishes after this step.

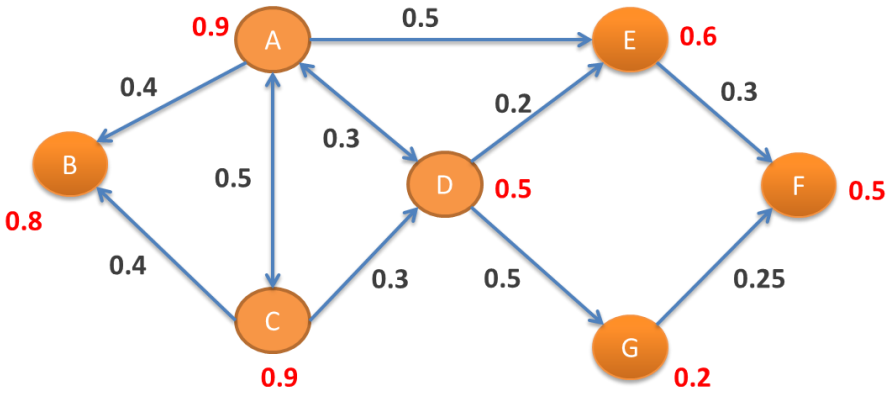
Figure 7.6 presents an example where the TDG algorithm does not match the optimum result. In this example, Node D has the highest value of $IncInfl$ and thus gets selected as the seed node. In order to infect all the nodes in the graph, the TDG algorithm selects nodes D, A, F as seeds. The optimum result selects nodes F and C as seed nodes. Nodes F and C together can infect all the nodes in the graph.

The algorithm presented in this Section 7.1 has the following advantages :

1. Multiple steps in the preprocessing stage can be executed concurrently. The $ComputeInfluence$ function can be invoked for each individual node in parallel.

2. The algorithm gives more information than other existing influence maximization algorithms. It produces (a) the influential nodes in the graph, and (b) the spread size (i.e., the number of affected nodes due to seed set).

3. It can be tailored for different depths to be consistent with theory proposed in [30].

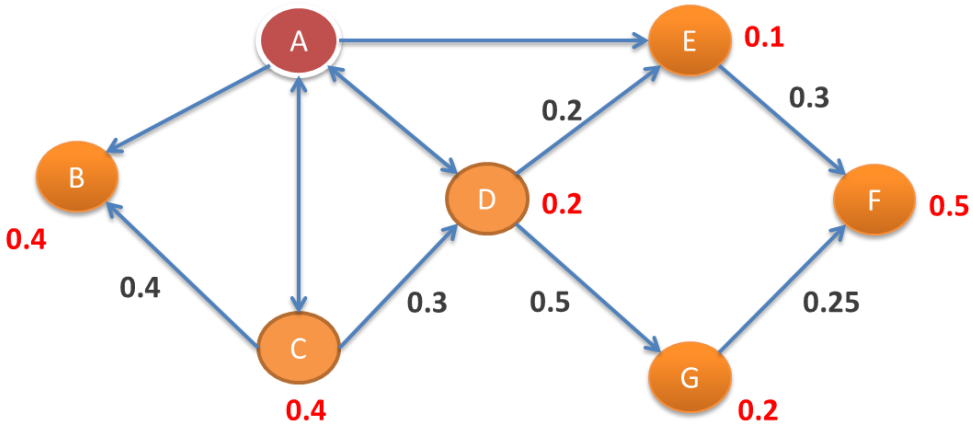


IncInfl(A) = (0.5/0.6) + (0.5/0.9) + (0/4/0.8) + (0.3/0.5) = 2.48

IncInfl(C) = (0.5/0.9) + (0.4/0.8) + (0.3/0.5) = 1.65

IncInfl(D) = (0.3/0.9) + (0.2/0.6) + 1 + (0.25/0.5) = 2.16

Figure 7.4: Figure showing the computation of $IncInfl$ in a graph. The values in red indicate threshold of nodes and values in black indicate edge weights.



IncInfl(C) = 1 + 1 + 1 + 1 + 1 = 5

IncInfl(D) = 1 + 1 + 1 = 3

Figure 7.5: Figure showing the $IncInfl$ calculation for the graph from Figure 7.4 with threshold values updated.

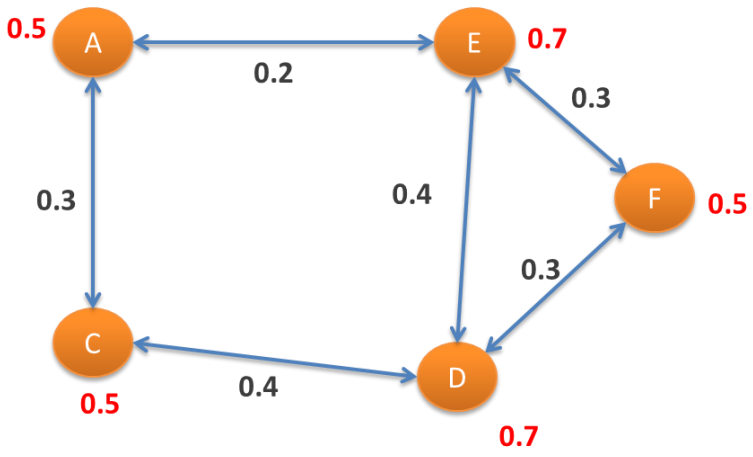


Figure 7.6: Figure showing an example of a graph where the TDG algorithm does not provide optimum results.

Thus as a part of computing the influential node set, one obtains the extent of contagion diffusion that is achievable, by setting d_2 to ∞. As a result, a separate simulation to compute the total number of affected nodes for a particular influential set is not required. On the downside, the running time of the algorithm is greater when the number of affected nodes is high. Execution time comparisons are shown in Section 7.3.2. As the diffusion process is deterministic, the algorithm can be used to identify the number of influential nodes required to achieve a particular spread size (target set selection problem). A modification to the algorithm takes in an input: the desired spread size and outputs the minimum set of influential nodes in the graph needed to attain the desired spread. The results for this Target Set Selection problem have been presented in Section 7.3.3 for various graphs and different spread size values.

7.2 Time and Space Complexity

Let n denote the number of nodes in the graph and m denote the number of edges in the graph.

Proposition : The time complexity of the TDG algorithm in $O((n*M_1)+(M_1*M_2))$ where $M_1, M_2 \leq m$. The value of M_1 depends on D_{avg}, θ and d_1, and the value of M_2 depends on D_{avg}, θ and d_2. Here D_{avg} is the average degree of graph, θ represents the threshold values assigned to nodes, d_1 and d_2 are parameters for the TDG algorithm.

Proof : The three subroutines ComputeInfluence, UpdateForNewSeed, and UpdateIncomingNeighborInfluence in Algorithms 4, 5 and 6 respectively perform a BFS traversal upto a level(or depth). For the subroutines ComputeInfluence, UpdateIncomingNeighborInfluence the level limit is d_1. The level limit for the subroutine UpdateForNewSeed is d_2.

The preprocessing step is performed for every node v in the graph (as indicated in lines 5 to 7 in Algorithm 3). The subroutine ComputeInfluence is invoked for every node in the graph. For each node, a breadth first search is performed and nodes are added into the queue until level d_1. Each node added to the queue contributes to $IncInfl$ and this proceeds until the queue becomes empty. The number of lookups performed by the ComputeInfluence function for each node is $M_1 \leq m$ and this value depends on D_{avg}, θ and d_1. Thus in the worst case, m look ups may be performed. Since the ComputeInfluence function is invoked for every node in the graph, the running time of the preprocessing step is $O(n * M_1)$. In the worst case, the value of M_1 is equal to m.

Once the preprocessing step completes, the most influential node is picked as seed in $O(n)$ and the update process begins. The update process performs two operations:

1. Update the threshold of nodes for the seed selected. This execution proceeds upto level

(depth) d_2.

2. For each of the node u whose threshold gets updated in the previous step, update the $IncInfl$ for all nodes that can affect u's threshold. This execution can proceed upto level d_1.

For a given graph with edge weights, the number of nodes that will have their thresholds modified for each seed selected depends on D_{avg}, θ and d_2. The number of update operations that can be performed is $M_2 \leq m$. For each of the update operation, the compute influence subroutine is invoked. Thus the running time for the seed selection and update process is $O(M_1 * M_2)$ where value of $M_1, M_2 \leq m$.

Thus the time complexity of the algorithm in $O((n * M_1) + (M_1 * M_2))$ where $M_1, M_2 \leq m$.

Space complexity : The algorithm does not compute any arborescence structures unlike the LDAG algorithm. It uses queues for intermediate computation. The only values stored are $IncInfl$ for each node. So the **space complexity** of the algorithm is $O(n + m)$ where n is the number of nodes in the graph and m is the number of edges in the graph.

7.3 Results

This section presents the comparison results of our algorithm and the algorithms in Chapter 6 for the graphs of Chapter 5. For the experiments, the vertex thresholds are assigned to the nodes in the graph in four different ways: uniform threshold of 0.5, uniform threshold of 0.8, a threshold between 0.3 and 0.7 assigned uniformly at random, a threshold between 0.1 and 0.9 assigned uniformly at random.

In the plots, the TDG algorithm has been labeled as `LTMNew`. The name also indicates the parameter used for that particular curve in the plot. For example `LTMNew_5_20_3_1` indicates that the new algorithm with parameter $d_1 = 5$, $d_2 = 20$, $d_3 = 3$. The last entry 1 indicates a flag for parameter d_3. If value of the flag is 1, then d_3 is considered ∞. The value '@' in the legend indicates ∞. We perform experiments with eight different combinations of parameter values for d_1, d_2, d_3.

7.3.1 Outbreak Results

A subset of the results for various graphs is presented in this section. The complete set of plots is available in Appendix A. The plots not only compare results of the existing algorithms with the TDG algorithm but also show how different parameter values, d_1, d_2, and d_3, for the TDG algorithm impact the result. For all the plots, the red lines indicate the TDG algorithm. The black line with an inverted triangle as the marker denotes the

algorithm described in Section 6.2 based on [20], and this algorithm is considered as one of the best algorithms for the linear threshold model.

As shown in Figures 7.7, 7.8, 7.9, and 7.10, the TDG algorithm performs better than all the existing algorithms for a uniform threshold of 0.5. For the graphs Astroph, Enron, and Facebook the algorithm with parameter $d_1 = 2$ and $d_2 = 2$ performs as good as parameter values $d_1 = \infty$ and $d_2 = \infty$ when the number of seeds are low. But as the number of seeds increase, the outbreak size for parameter values $d_1 = 2$ and $d_2 = 2$ becomes slightly less and this outbreak size is still better than the other existing algorithms. The LT algorithm in [20] is the second best in most cases. For Astroph and Enron, as the number of seeds become more than 200, the Degree discount and the Eigenvector heuristic starts performing very well. The intuition we derive is that the seeds picked by these heuristics reside in different location affecting a smaller set of nodes around them but as the seed count increase above a limit, they together start a cascade and cause large number of nodes to be affected. This effect is most dominant for the epinion graph. For the Eigenvector Centrality and High degree heuristic, the number of affected nodes for 400 seeds is 8301 but for 500 seeds it becomes 54979.

As shown in Figures 7.11, 7.12, 7.13, and 7.14, the TDG algorithm performs better than all the existing algorithms for a uniform threshold of 0.8. This supports our intuition that the existing algorithms might select seeds which may be highly influential, but a high threshold value to the neighbors of these selected seeds can reduce the influence propagation. For all the graphs with uniform threshold 0.8, the TDG algorithm with parameter $d_1 = 2$ and $d_2 = 2$ performs as good as $d_1 = \infty$ and $d_2 = \infty$.

Figures 7.15, 7.16, 7.17, and 7.18 show results for graphs with nodes having a random threshold value between 0.3 and 0.7. Figure 7.15, for the Ca-hepph graph, shows the Degree discount heuristic performing well for 100 and 200 seed nodes, and the Betweenness Centrality performing better than the LDAG algorithm [20] for 300 and 400 nodes. Figure 7.16, for the Cit-hepth graph, shows the TDG algorithm producing large number of affected nodes compared to other algorithms for 500 seed nodes. Figure 7.17, for the Epinion graph, shows High degree, Degree discount, Eigenvector heuristics producing huge spread sizes for 400 nodes which are better than other algorithms. But for 500 seed nodes, all the algorithms except the IC algorithm in [58] and the Random heuristic reach the 60000 node mark. Figure 7.18 shows results for the FHS graph with real edge weights. As shown in the figure, the spread size shoots up to around 4000 nodes for 5 seed nodes and grows less rapidly as the number of seed nodes increase. This is because, the FHS graph has few nodes with very high edge weights and others having extremely low edge weights.

Figures 7.19, 7.20, 7.21, and 7.22 show results for graphs with nodes having a random threshold value between 0.1 and 0.9. Figure 7.20 shows the results for maxplanck social facebook network, which is a graph with real edge weights. The edge weights in this graph are extremely low compared to the threshold values. As a result, the spread sizes are also very small. The contagion spread is very little, and therefore results of the algorithm are

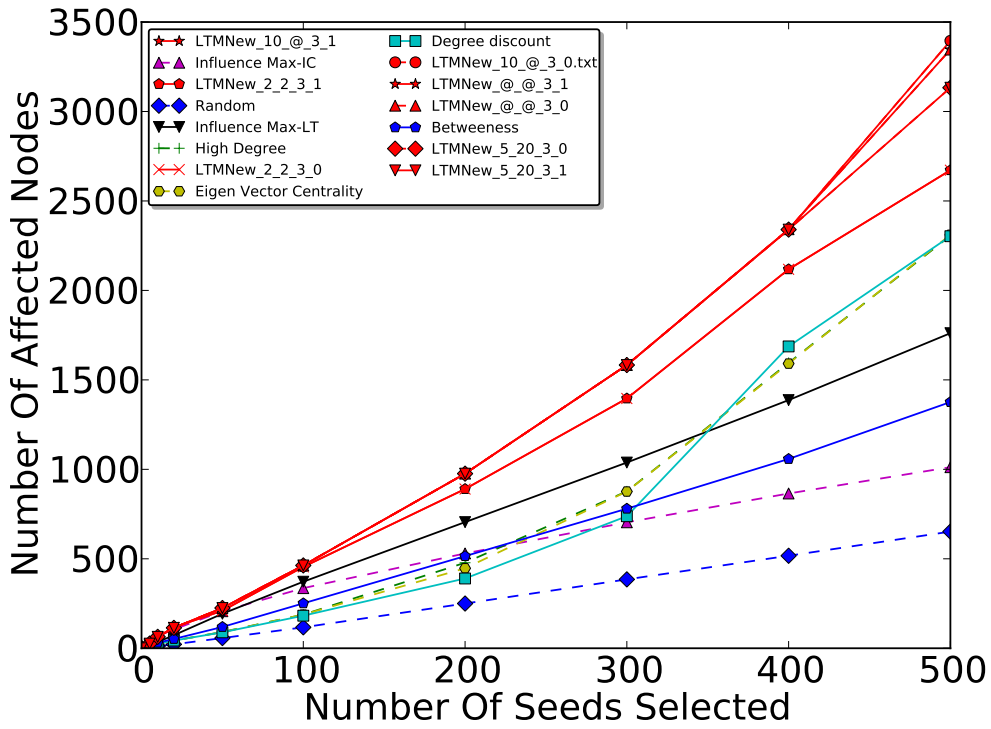


Figure 7.7: Final Outbreak Results for Astroph Graph with Nodes Having a Uniform Threshold of 0.5.

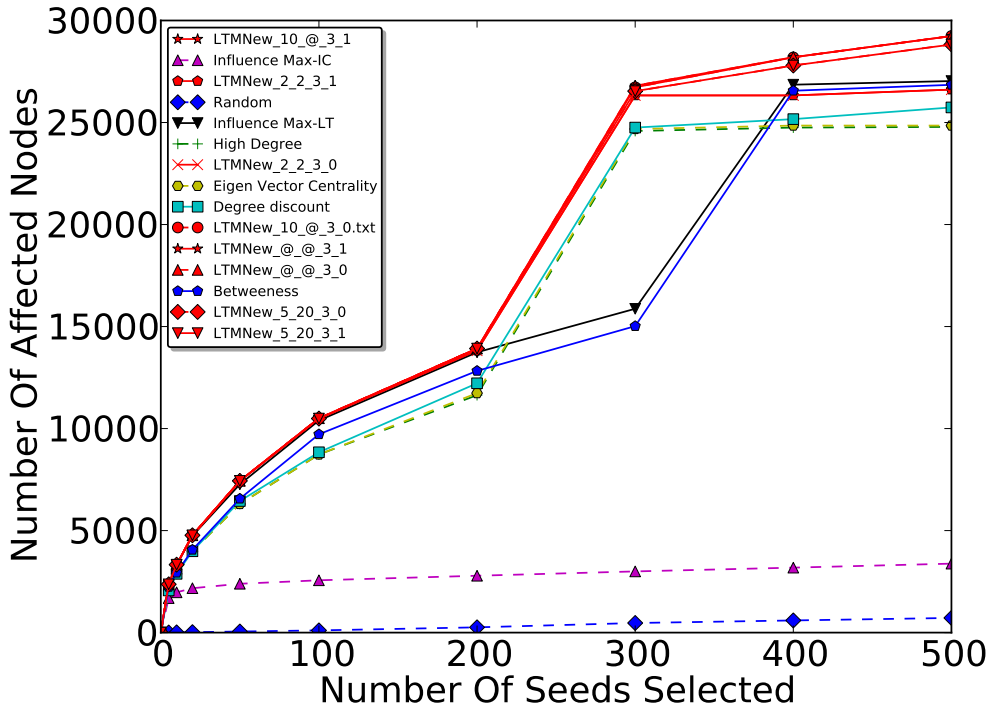


Figure 7.8: Final Outbreak Results for Enron Graph with Nodes Having a Uniform Threshold of 0.5.

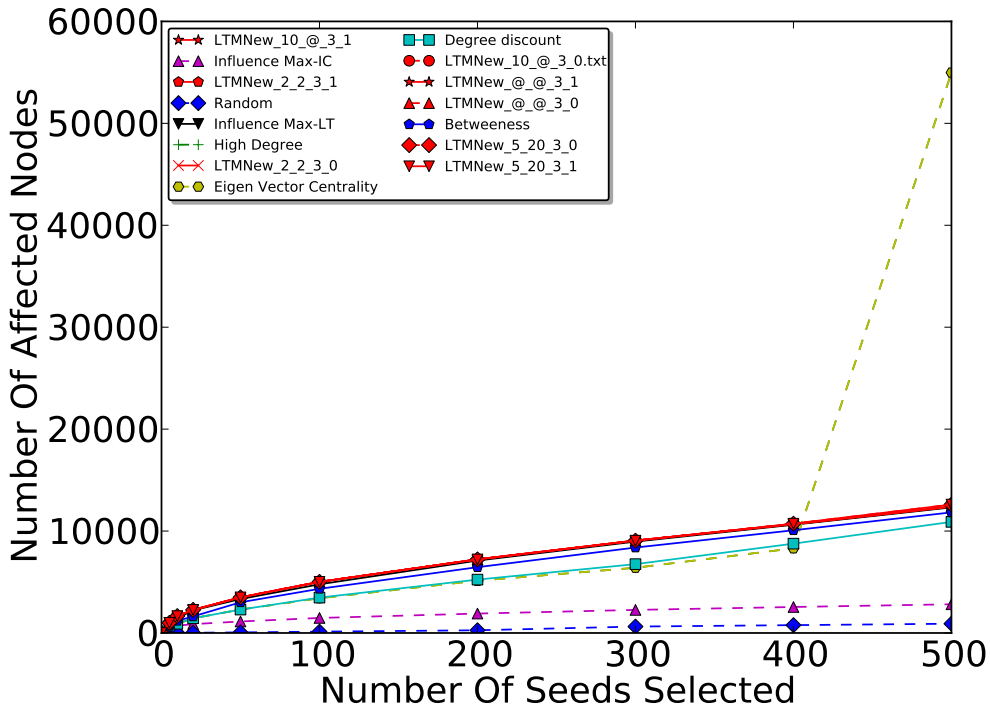


Figure 7.9: Final Outbreak Results for Epinion Graph with Nodes Having a Uniform Threshold of 0.5.

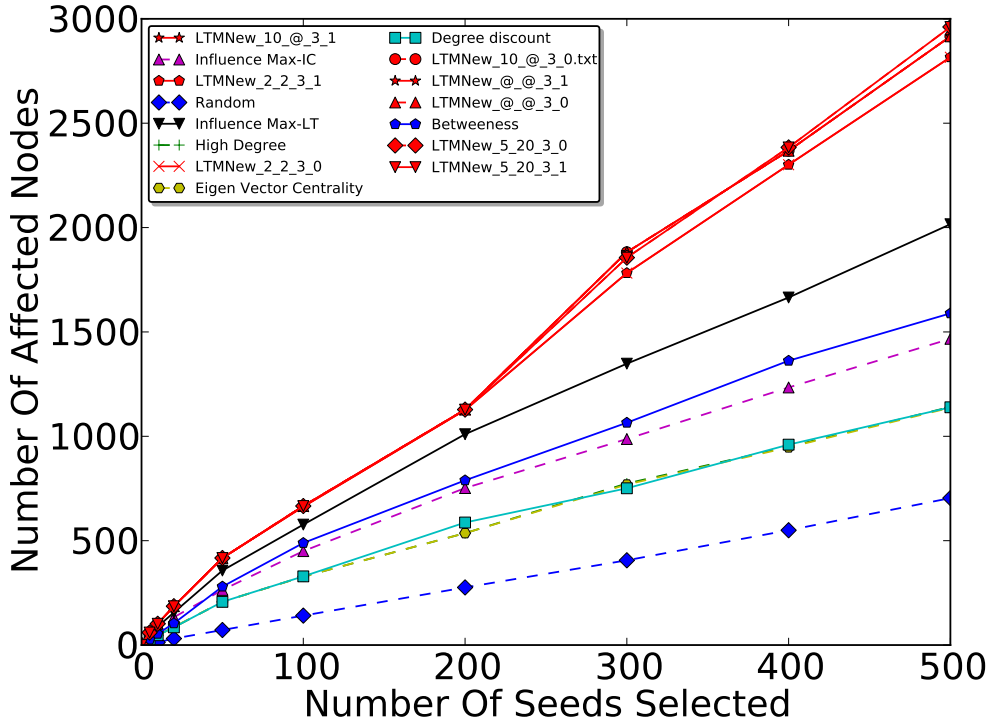


Figure 7.10: Final Outbreak Results for Facebook Graph with Nodes Having a Uniform Threshold of 0.5.

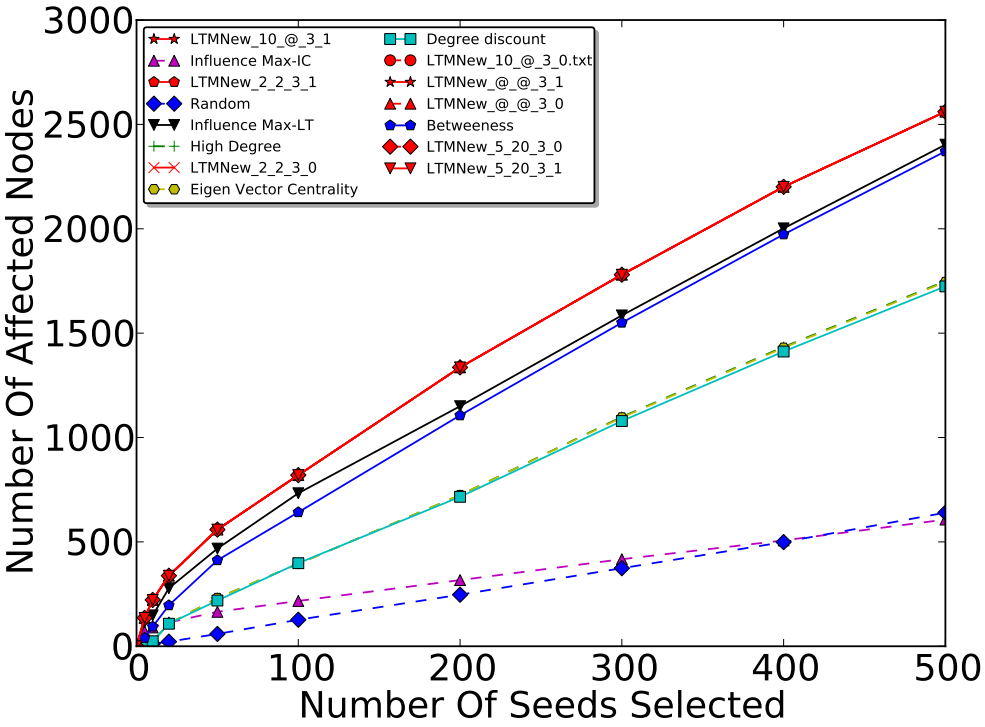


Figure 7.11: Final Outbreak Results for Wikipedia Graph with Nodes Having a Uniform Threshold of 0.8.

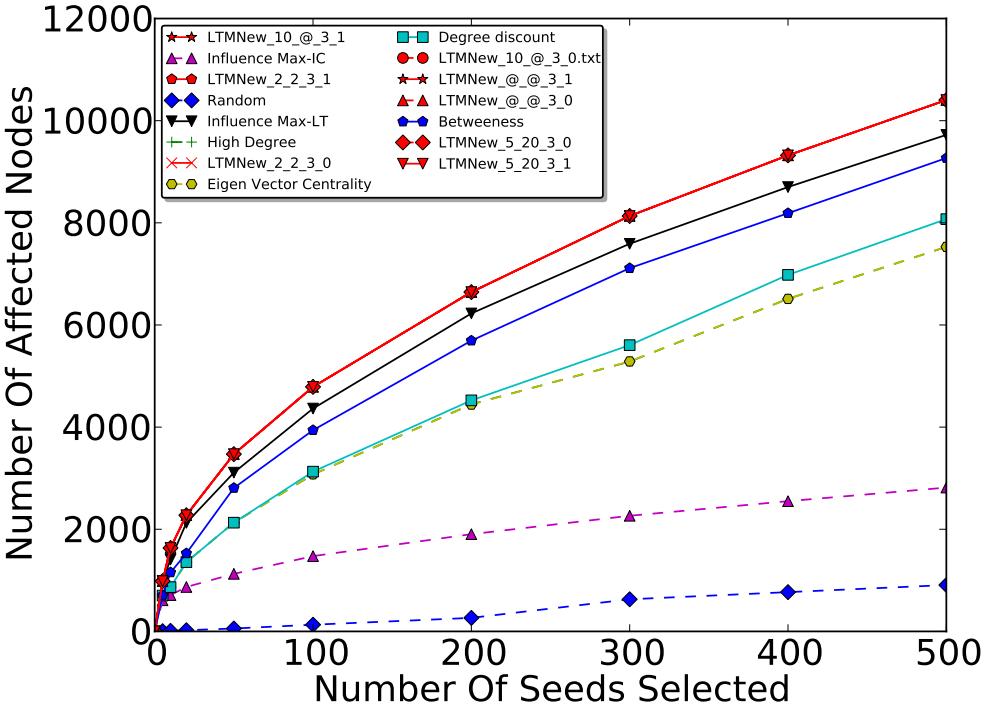


Figure 7.12: Final Outbreak Results for Epinion Graph with Nodes Having a Uniform Threshold of 0.8.

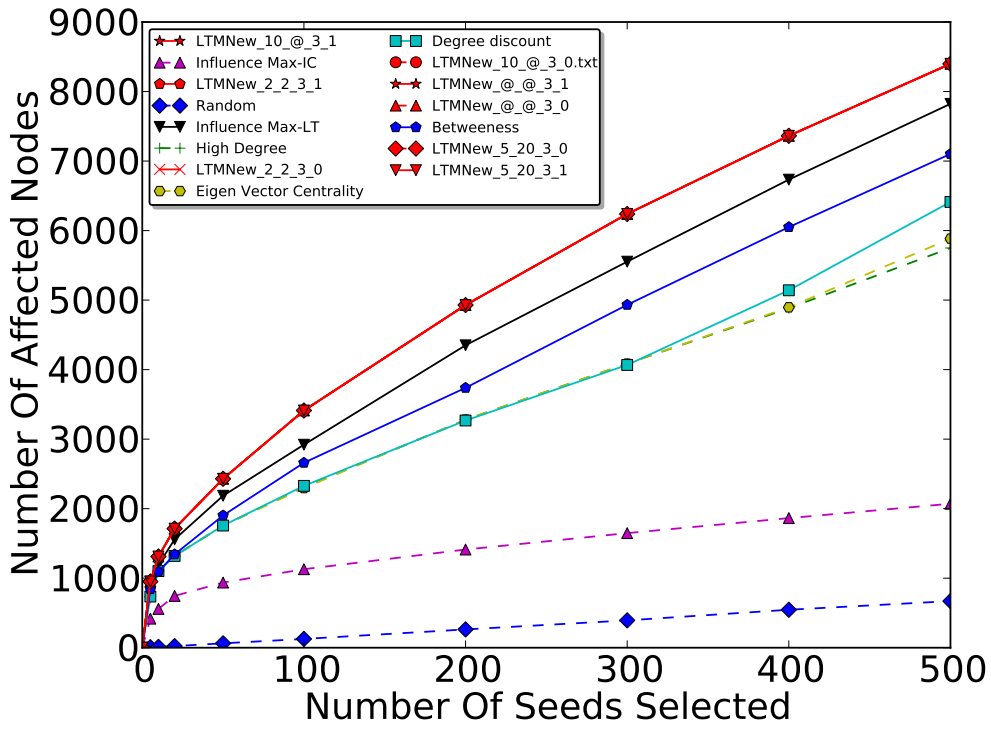


Figure 7.13: Final Outbreak Results for Slashdot Graph with Nodes Having a Uniform Threshold of 0.8.

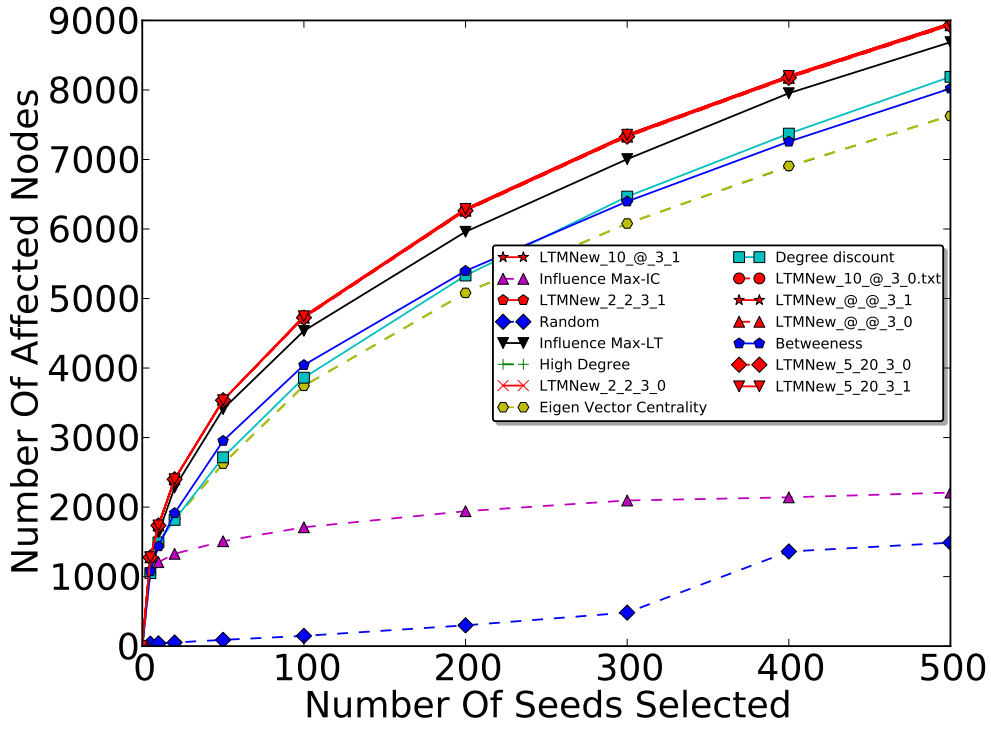


Figure 7.14: Final Outbreak Results for Twitter graph with Nodes Having a Uniform Threshold of 0.8.

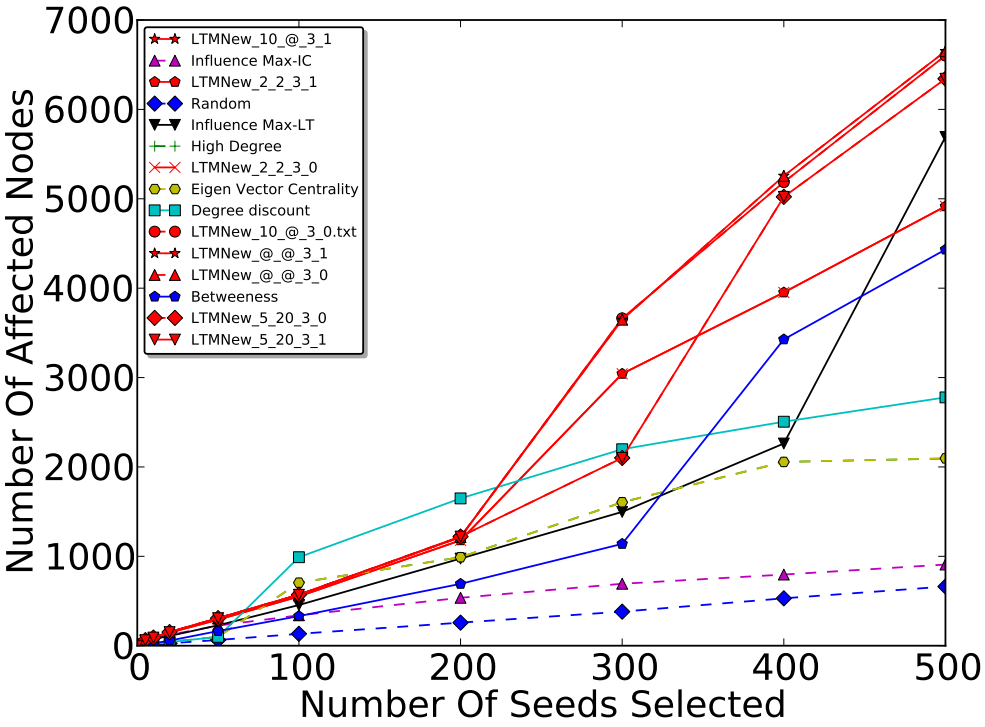


Figure 7.15: Final Outbreak Results for Ca-hepph Graph with Nodes Having a Random Threshold between 0.3 and 0.7.

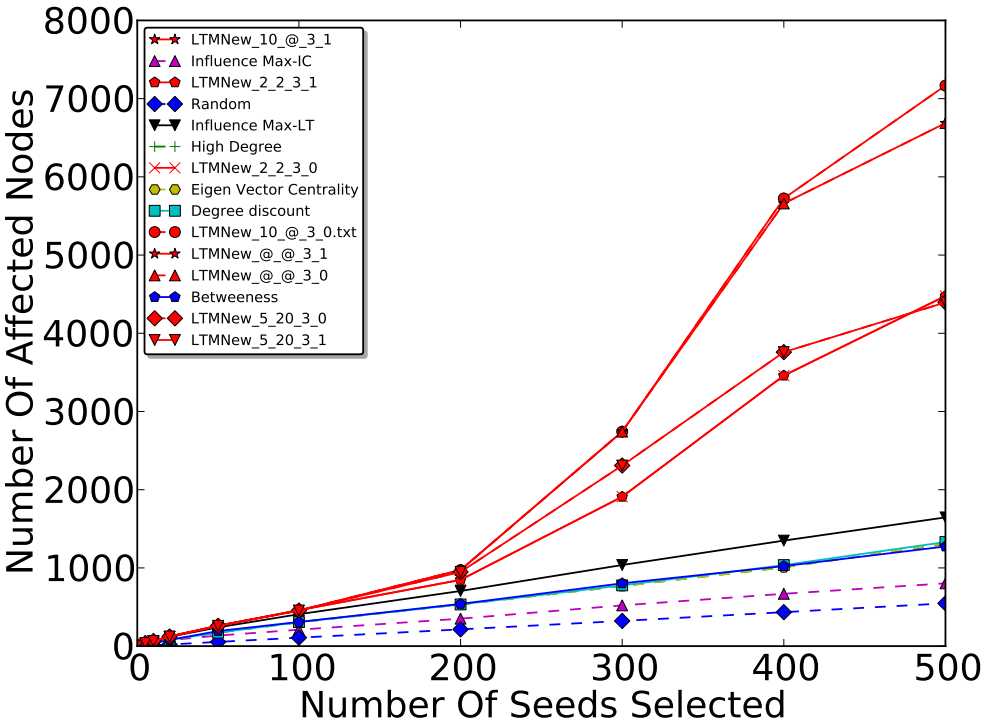


Figure 7.16: Final Outbreak Results for Cit-hepph Graph with Nodes Having a Random Threshold between 0.3 and 0.7.

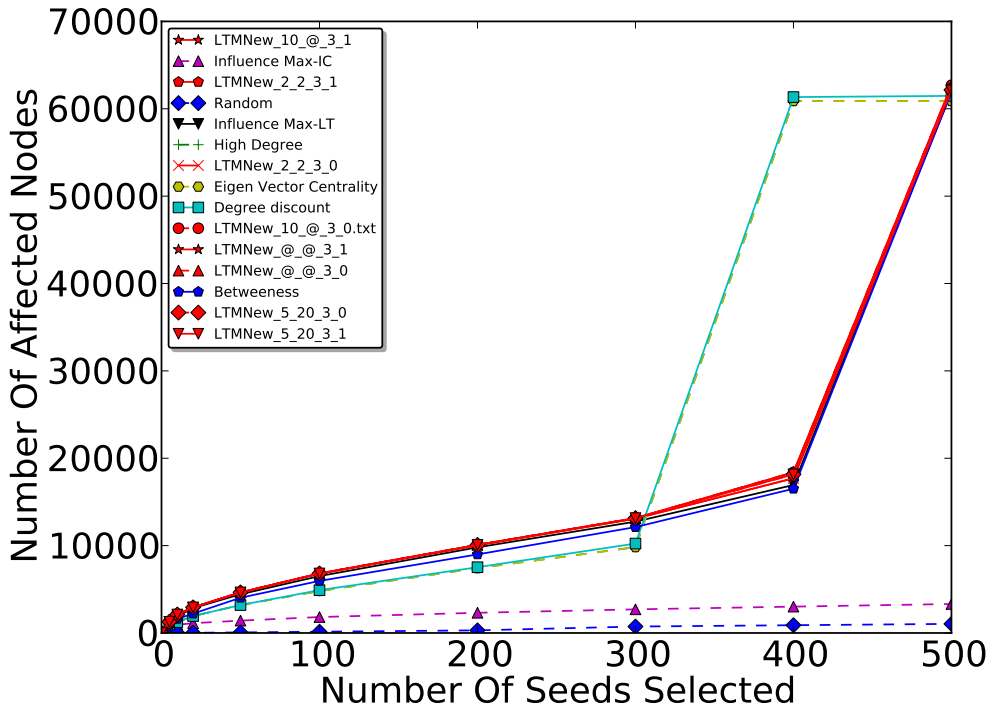


Figure 7.17: Final Outbreak Results for Epinion Graph with Nodes Having a Random Threshold between 0.3 and 0.7.

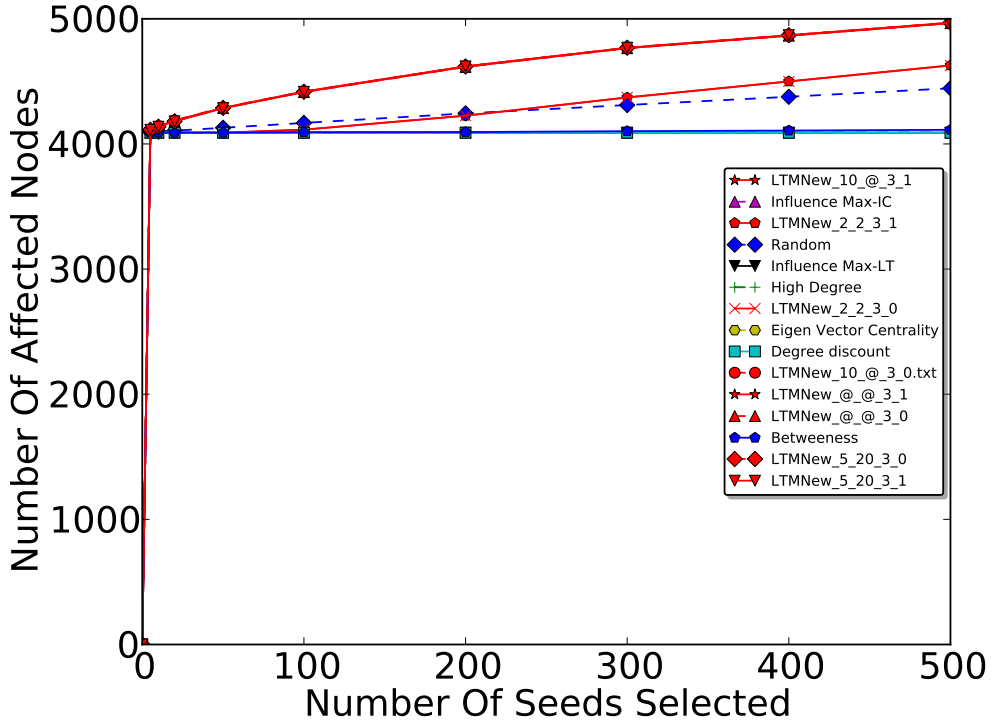


Figure 7.18: Final Outbreak Results for Fhs graph with Nodes Having a Random Threshold between 0.3 and 0.7.

not sensitive to the parameter values. In Figure 7.21, for the Slashdot graph, the Degree discount, the Eigenvector Centrality and the High Degree heuristic performs extremely well for 200 seed nodes. Also for 200 seeds the TDG algorithm with parameter values $d_1 = 5$ and $d_2 = 20$ performs better than other parameter values considered.

7.3.2 Execution Times

This section compares the execution times of the TDG algorithm with the LDAG algorithm for the linear threshold model [20]. The experiments were executed using 16 threads in a single compute node with 16 core processors. We compare both the setup/preprocessing time and the time to select 500 seed nodes. For both the algorithms, the preprocessing step was executed concurrently using 16 threads. But the seed selection and the update process was executed serially using a single thread for the TDG algorithm and concurrently using 16 threads for the LDAG algorithm in [20]. As shown in Figure 7.23, the TDG algorithm runs faster than the LDAG for most of the graphs for uniform threshold of 0.5. The yellow-orange bars in Figure 7.23 represent execution times of the LDAG algorithm [20]. The TDG algorithm with parameters $d_1 = 2$ and $d_2 = 2$, as indicated by the pink-red bar, runs faster than the LDAG algorithm for all the graphs. When the parameter for the TDG algorithm is set as $d_1 = \infty$ and $d_2 = \infty$, the algorithm runs slower than its counter part for FHS and Twitter graph as indicated by the blue bar. This is because, the number of nodes that changed their states due to the influential set was relatively high. The update process for the influential seed selected went to greater depths to update the threshold values. Thus the time taken to determine 500 seeds is high for these two graphs. After a closer examination into the execution times of these two graphs, we found that for Twitter, the 194^{th} made 3030 nodes to change state and the update step for this seed took 90 seconds to complete. Also the 228^{th} seed selected made 5488 nodes to change state and the update step for this node took 787 seconds to complete. For certain graphs like grqc, the algorithm finishes in less than 1 second and we do not see any bars in the Figure 7.23 for these graphs.

Figure 7.24 shows the results of execution times for TDG algorithm and the LDAG algorithm [20] for graphs with a random threshold value between 0.1 and 0.9 assigned to each node. Similar to the results for a uniform threshold of 0.5, and the TDG algorithm with parameters $d_1 = 2$ and $d_2 = 2$ faster than the LDAG algorithm for all the graphs, as indicated by green bar in the figure 7.24. When the parameter for the TDG algorithm is set as $d_1 = \infty$ and $d_2 = \infty$, the algorithm runs slower than its counter part for FHS and Slashdot graph as indicated by the blue bar in Figure 7.23. For the TDG algorithm, the time taken for selecting the most influential node is higher if the update process affects large number of nodes over a greater depth.

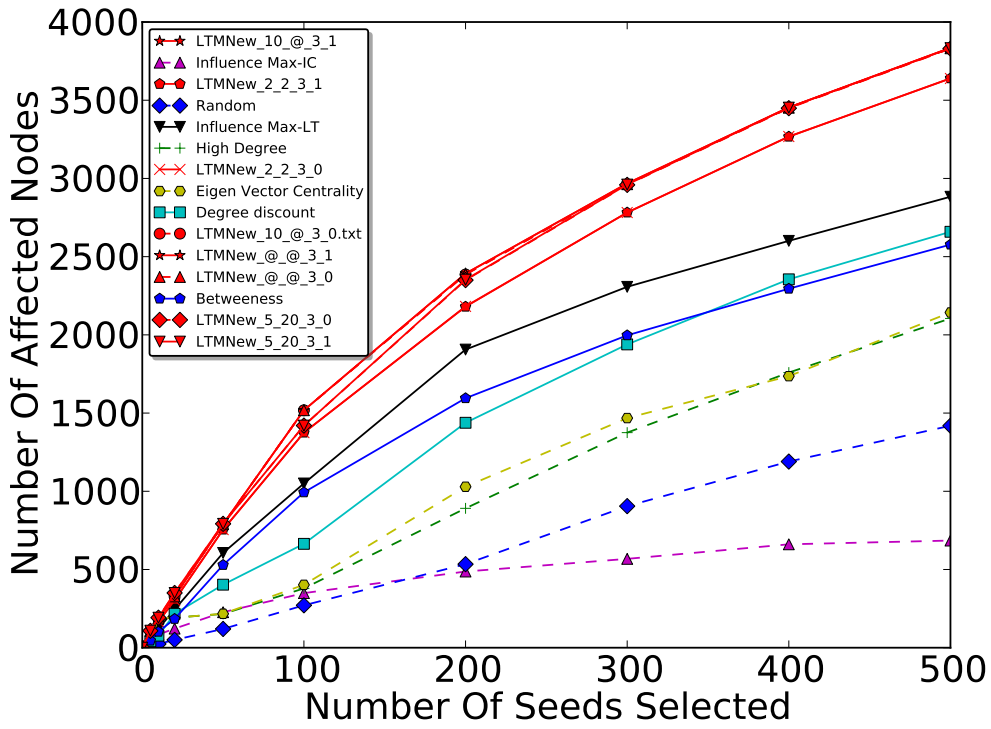


Figure 7.19: Final Outbreak Results for Grqc Graph with Nodes Having a Random Threshold between 0.1 and 0.9.

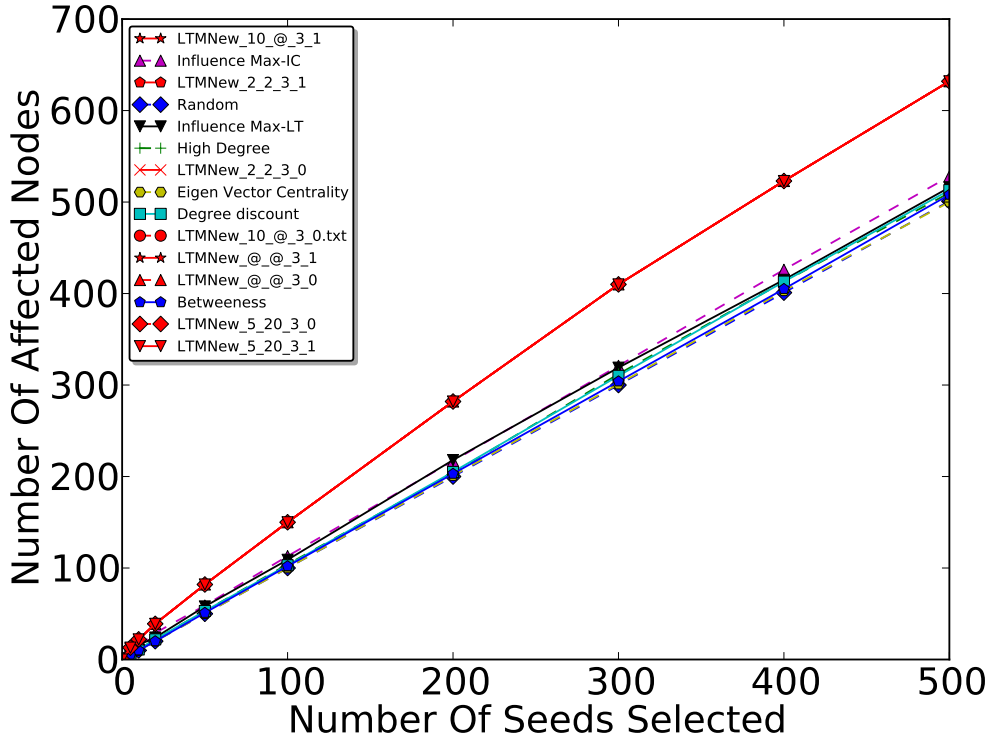


Figure 7.20: Final Outbreak Results for Maxplanck Social Facebook Graph with Random Threshold between 0.1 and 0.9.

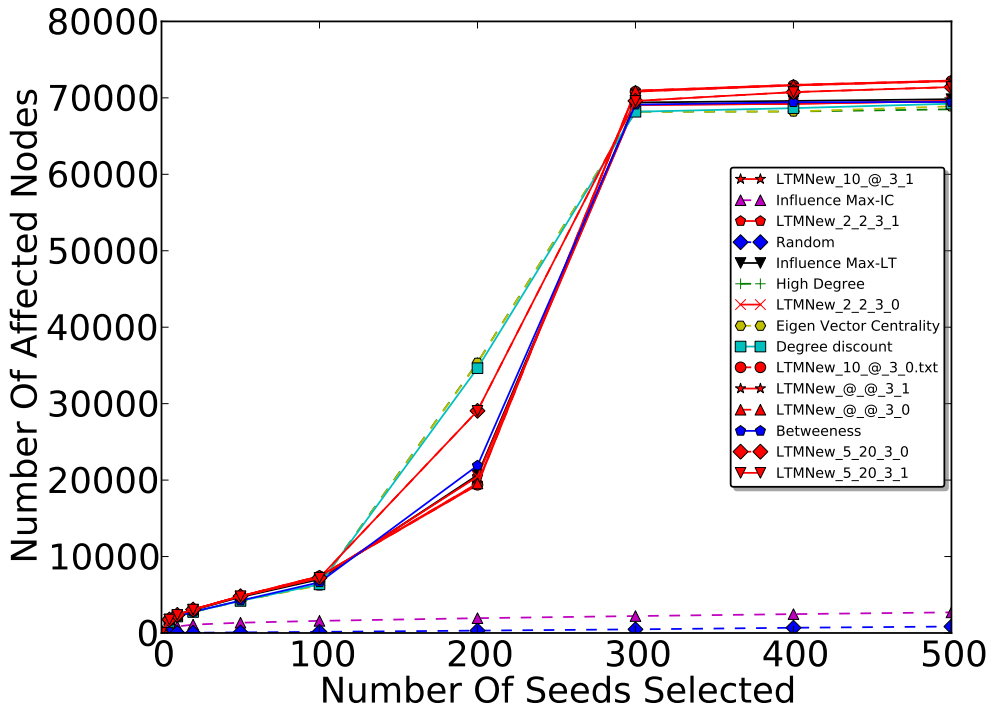


Figure 7.21: Final Outbreak Results for Slashdot Graph with Nodes Having a Random Threshold between 0.1 and 0.9.

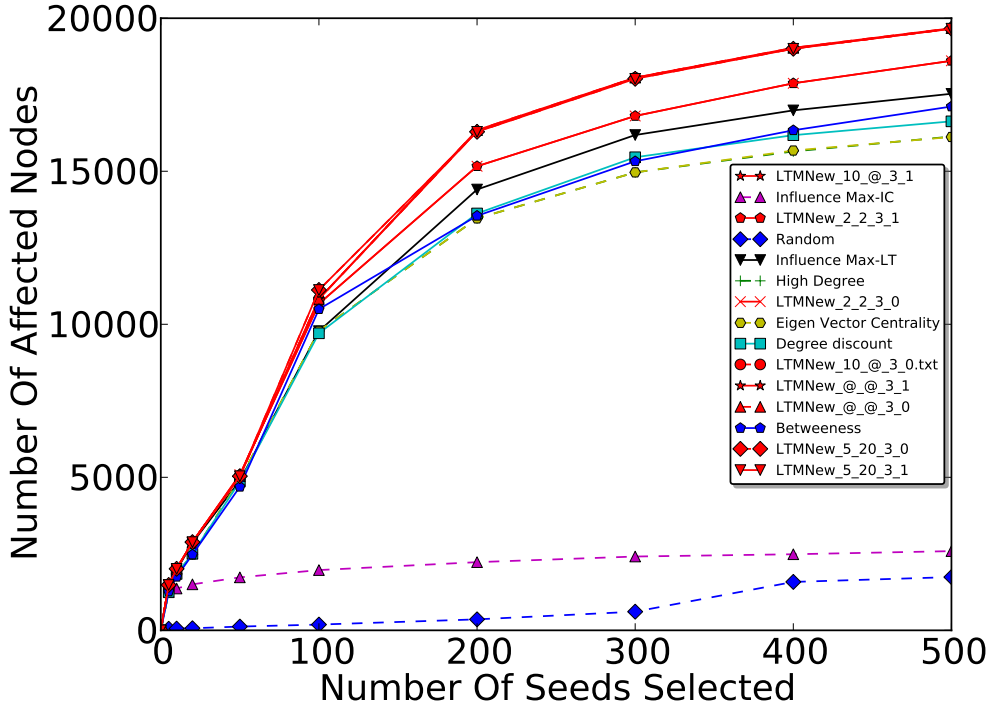


Figure 7.22: Final Outbreak Results for Twitter graph with Nodes Having a Random Threshold between 0.1 and 0.9.

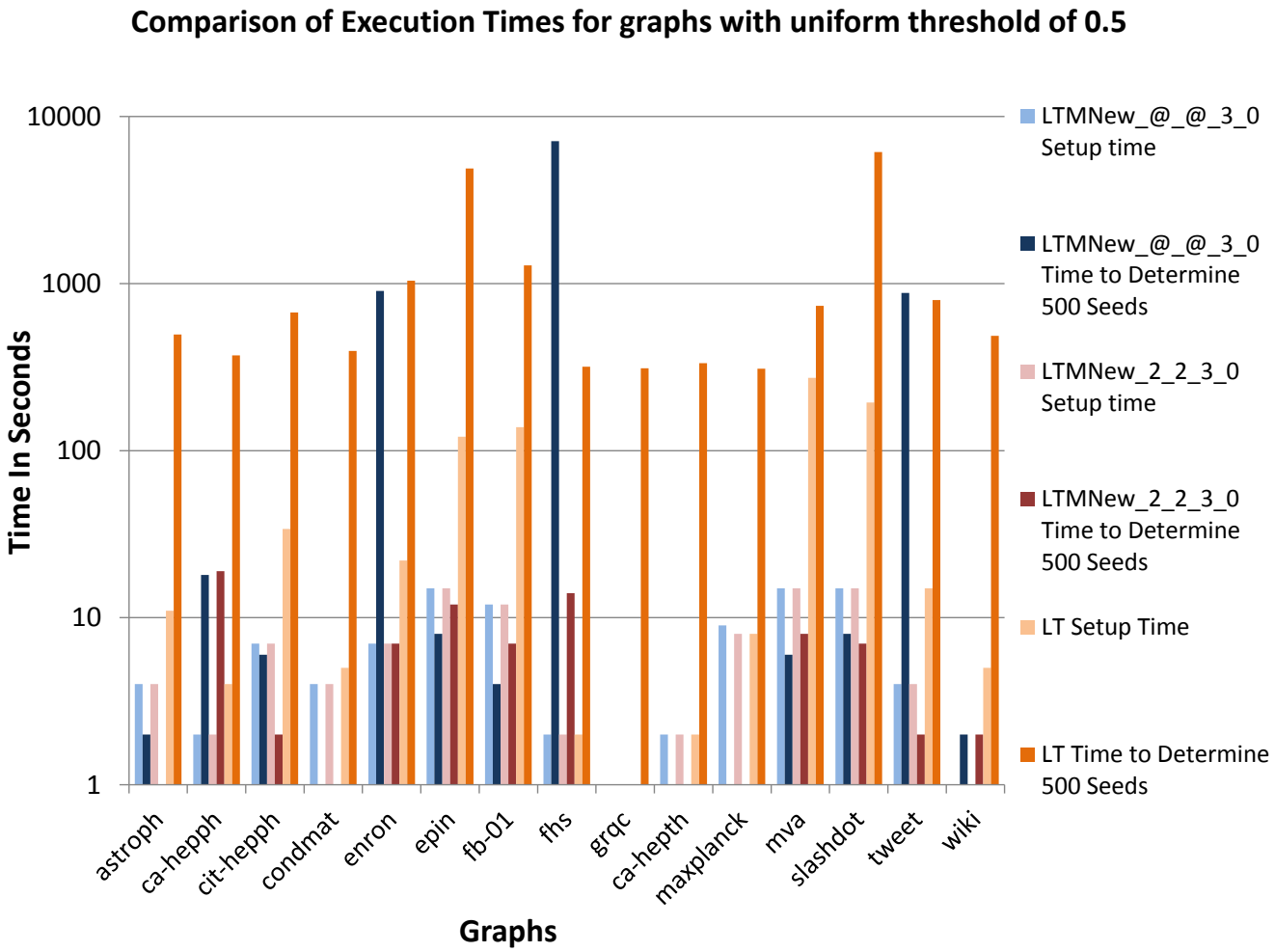


Figure 7.23: Comparison of Execution Times between the LDAG Algorithm and the TDG algorithm for Graphs with Nodes Having a Uniform Threshold of 0.5.

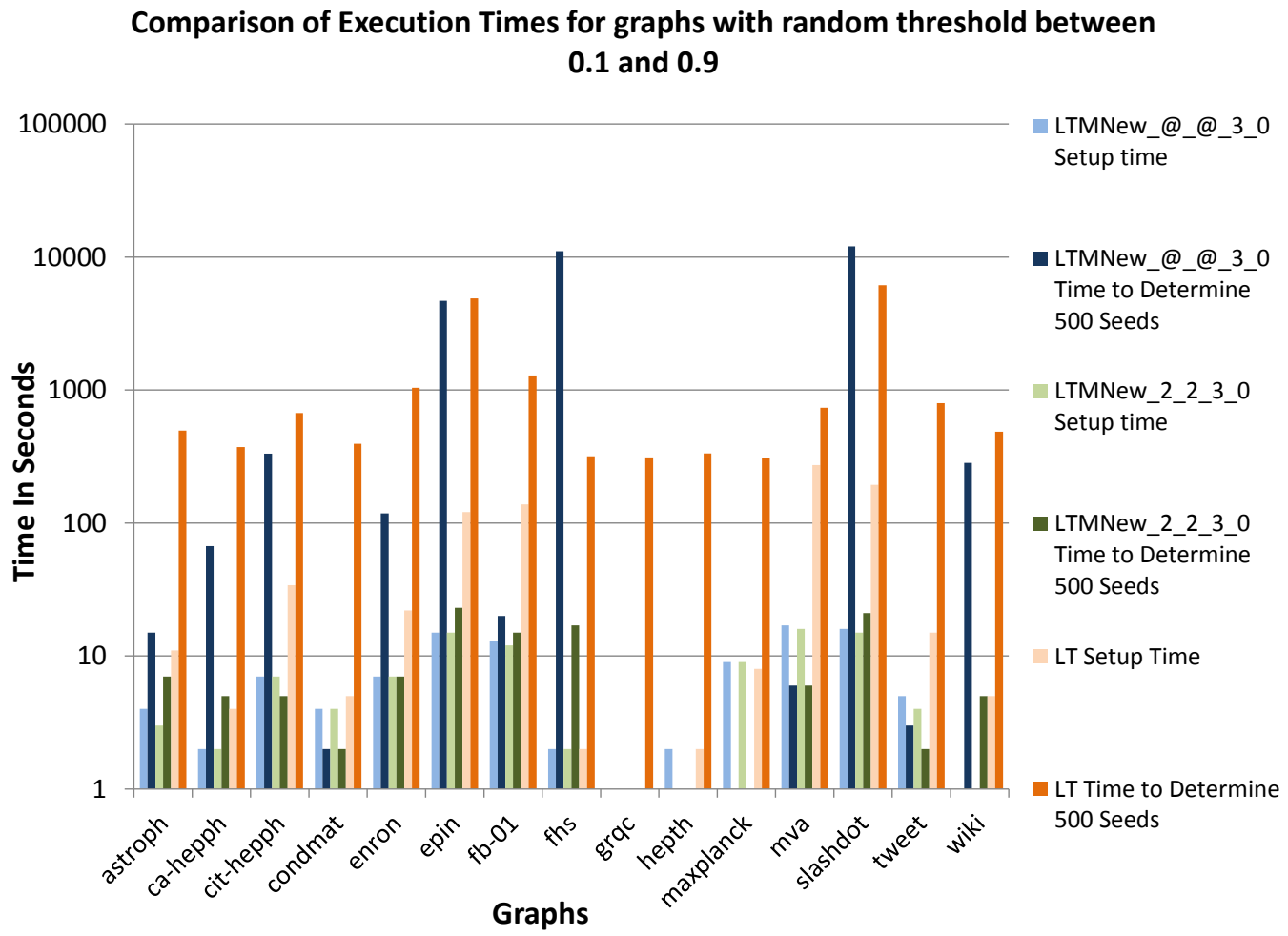


Figure 7.24: Comparison of Execution Times between the LDAG Algorithm and the TDG Algorithm for Graphs with Nodes Having a Random Threshold assignment between 0.1 and 0.9.

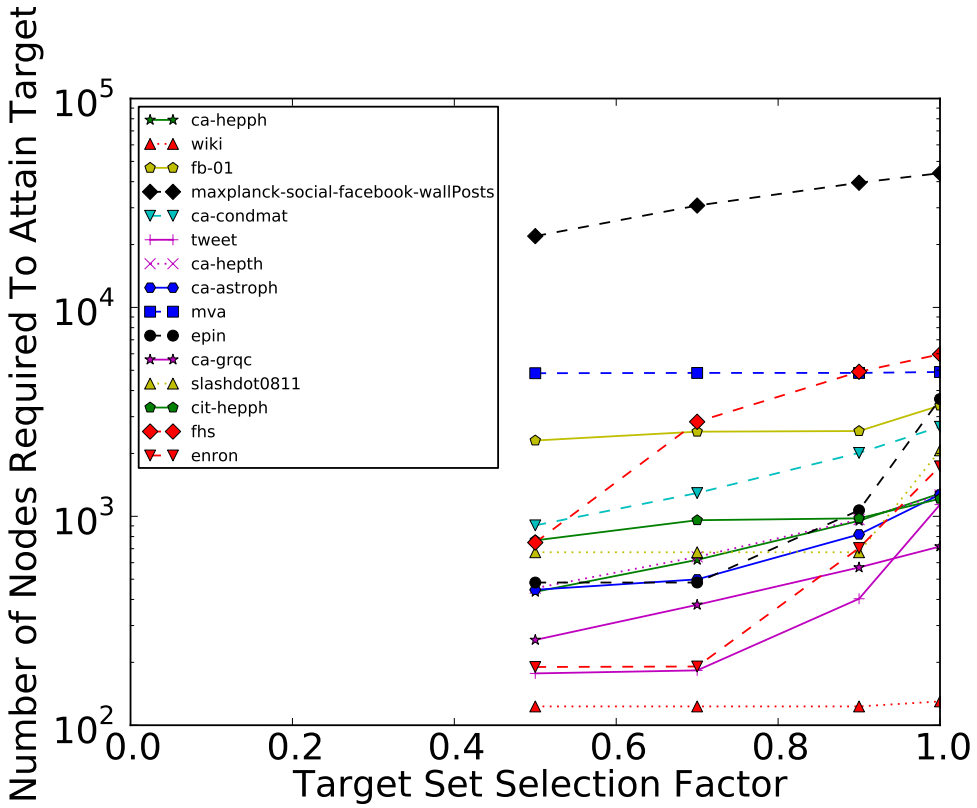


Figure 7.25: Target Set Selection Results for Various Graphs, Indicating the Number of Seed Nodes Required to Affect the Specified Fraction of Nodes in the Graph.

7.3.3 Target Set Selection(TSS)

As a part of studying the influence maximization problem, we also looked into the problem of Target Set Selection. The TDG algorithm with parameter d_2 set as ∞, determines the number of affected nodes after the seed selection step. This process can be used to study the Target Set Selection(TSS) problem where the goal is to find the minimum set of seeds that can produce an outbreak of desired size. Note that the LDAG algorithm cannot be used for TSS problem. For studying the TSS problem, the TDG algorithm was executed with parameter values $d_1 = 10$ and $d_2 = \infty$, and no limitation on d_3 by setting the flag to 1. The TSS factor is the prescribed fraction of nodes in the graph that needs to be affected. The algorithm was run for TSS factor of 0.5, 0.7, 0.9 and 1 for all graphs. The algorithm like in the previous sections, was executed using 16 threads on a single computing node with 16 core processors.

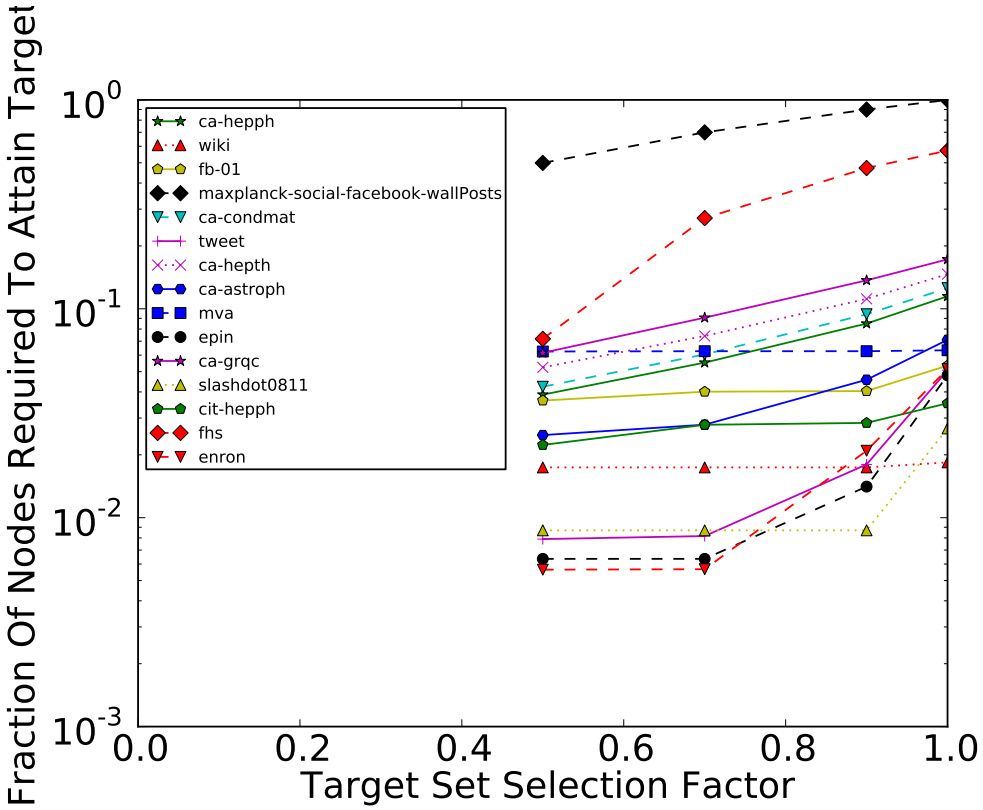


Figure 7.26: Target Set Selection Results for Various Graphs, Indicating the Minimum Fraction of Nodes Required to Affect the Specified Fraction of Nodes in the Graph.

Figures 7.25, 7.26, and 7.27 show the target set selection results for various graphs with nodes assigned a random threshold value between 0.1 and 0.9. The straight line in the Figure 7.25 for the Mva graph between 0.5 and 1, indicates that comparatively lesser number of seeds were required to affect all the nodes in the graph as compared to 50% of the nodes. But

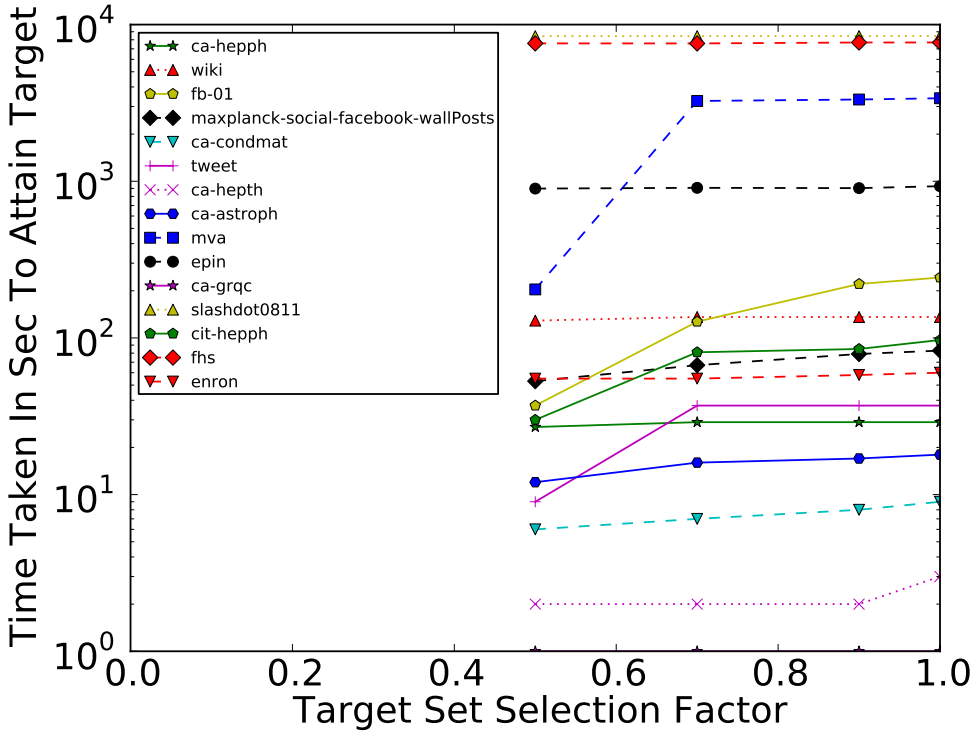


Figure 7.27: Target Set Selection Results for Various Graphs, Indicating the Time Taken by the TDG algorithm to Affect the Specified Fraction of Nodes in the Graph.

for the FHS and the Enron graph, higher number of seeds were required to reach target set selection factor of 1 from 0.5.

In the Figure 7.26, it can be seen that to completely affect all the nodes in a graph, i.e, to attain a target set selection factor of 1, almost all nodes were needed to be selected as seeds in the case of the maxplanck social facebook graph(43913 seeds out of 43953 nodes). For the wiki graph, only 1.84% of the nodes were required to affect all the nodes in the graph.

Figure 7.27 shows the amount of time taken by the TDG algorithm to compute seeds to attain a given TSS factor. The algorithm computes seeds to affect all the nodes for the Ca-grqc in close to 1 second while for the Slashdot graph, the algorithm takes the maximum time of 8451 seconds to compute seeds to affect all the nodes in the graph. Also for the Mva graph, one can see a steep rise in time taken from 0.5 to 0.7. It was found that 16 seeds beyond those required for the TSS factor of 0.5, were required to reach TSS of 0.7 and 0.9 and these 16 seeds affected 38009 nodes. The update process to compute affected nodes took close to 3000 seconds for these 16 seeds.

7.4 Threshold Difference Greedy Method with Alpha

This section presents a different way of computing $IncInfl$ for the TDG algorithm compared to the one presented in Section 7.1. A parameter α is used in computing influence when a node completely affects other nodes. The algorithm is similar to that in Section 7.1 except for the way $IncInfl$ is calculated. Consider an example, take a node v with an out-going edge to node u. The weight of edge (v, u) (i.e., edge from v to u) is w_{vu}, and the threshold of u is θ_u. The **residual** r_{vu} edge weight as shown in Section 7.1 for edge (v, u) is $w_{vu} - \theta_u$. If $r_{vu} \geq 0$, then v influences u to transition state $0 \rightarrow 1$. If $r_{vu} < 0$ then the edge weight is insufficient to cause u to transition state, and the value of r indicates how far away u is from transitioning. In this method if $r_{vu} \geq 0$, then $Infl(v)$ is incremented by a value α. If instead v contributes to the transition of node u, but cannot cause u to change state on its own, then v gets partial influence in causing u to transition. If v contributes to u's transition, but u does not change state based on v's influence, then the incremental influence $IncInfl(v)$ is increased by value $1 + (w_{vu} - \theta_u)$. The intuition is that the closer w_{vu} comes to θ_u, then $(w_{vu} - \theta_u)$ has lesser negative magnitude.

Figures 7.28, 7.29, and 7.30 show results where the red line indicates the outbreak size achieved through the Threshold Difference greedy algorithm with alpha. For some graphs, this method did not perform well and the optimal value of α varies for each graph. These shortcomings are not present is the Threshold Difference greedy algorithm of Section 7.1 and the algorithm performs better without the value of α.

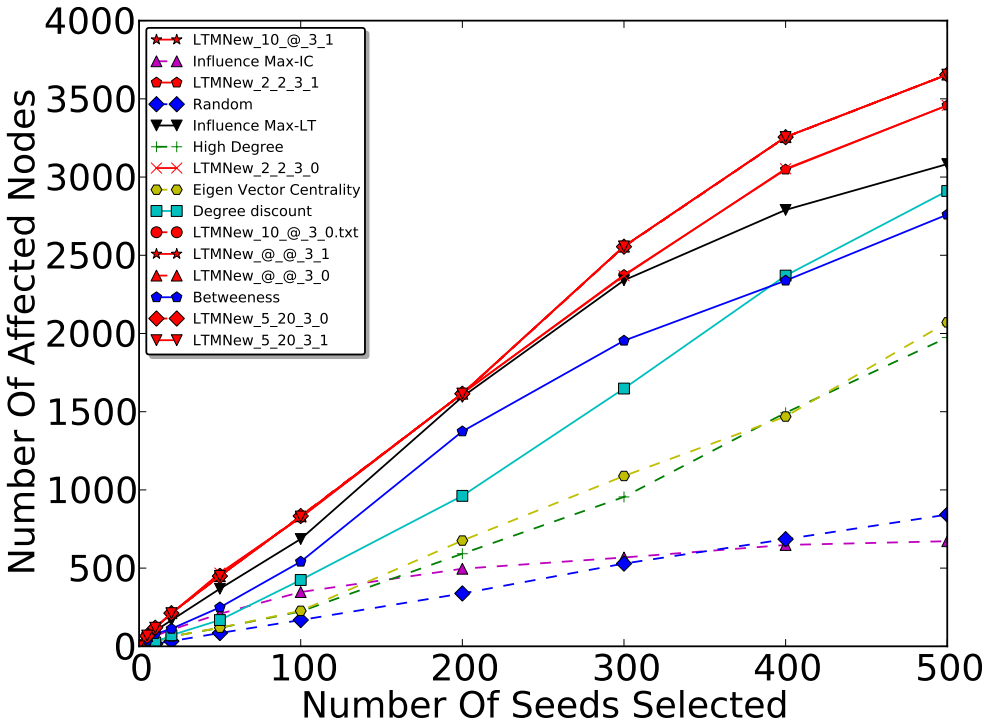


Figure 7.28: Final Outbreak Results for Grqc Graph with Nodes Having a Uniform Threshold of 0.5.

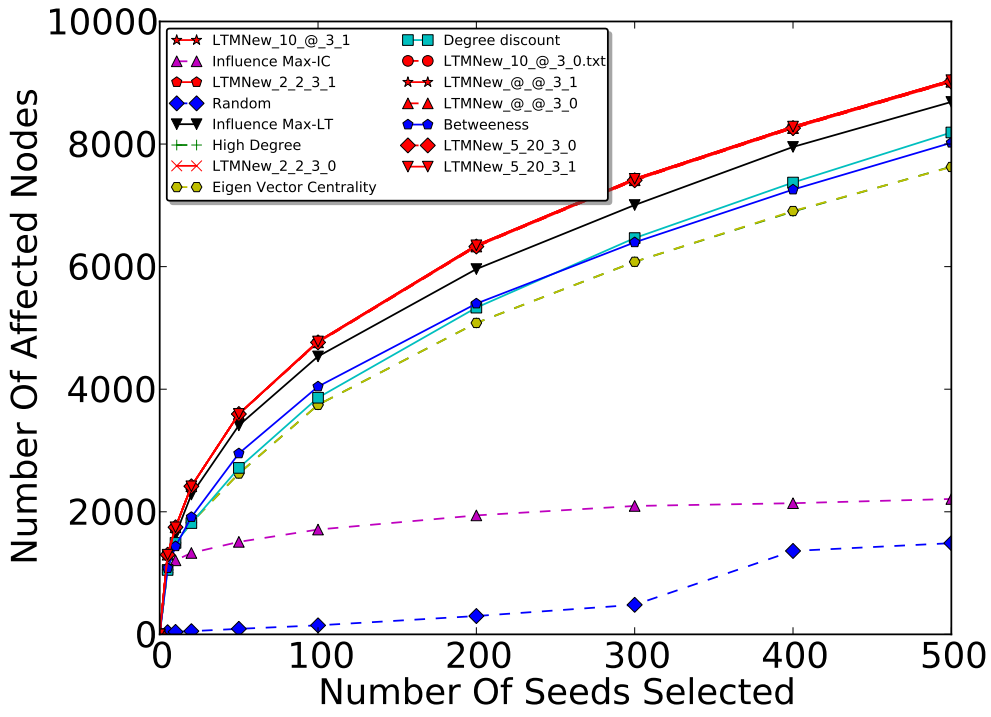


Figure 7.29: Final Outbreak Results for Twitter Graph with Nodes Having a Uniform Threshold of 0.8.

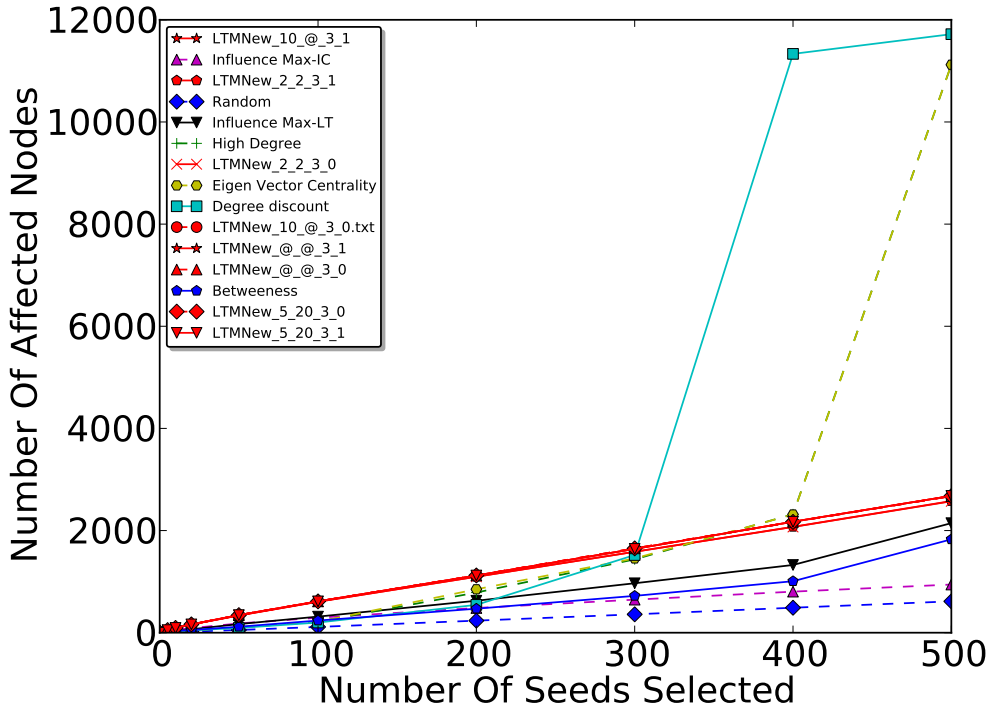


Figure 7.30: Final Outbreak Results for Astroph Graph with Nodes Having a Random Threshold between 0.3 and 0.7.

Chapter 8

Performance Improvements and Evaluation

Influence maximization algorithms typically take a long time to run. Depending on the size and the structure of the graphs, the execution time can range from seconds to several days. As a part of the implementation of influence maximization algorithms proposed by Chen et al in Sections 6.1 and 6.2, and the threshold difference greedy algorithm (TDG), we addressed the bottlenecks in these algorithms and improved the performance of these algorithms. This chapter presents the results of the performance improvement achieved and the changes made to the algorithm in order to achieve it. These influence maximization algorithms have been paralellized to run concurrently on a single compute node with multiple cores. The implementation does not support distributed processing across several nodes.

The algorithms are implemented using Java's ExecutorService library with cached thread pools. Our intention is to use multiple threads to execute steps in the algorithm that can be parallelized, and reduce access to common (shared) data structures. In the case of the influence maximization algorithm for the IC model [58] and the LDAG algorithm [20], both the steps in the algorithm are executed in parallel. The two steps are : a) setup or the preprocessing step, b) seed selection and the update step. Due to concurrent execution, the value of $Incremental\ Influence$ for a particular node can be modified by multiple threads. Such update operations need to be synchronized. It was found that the priority queue used by the authors in [58] and [20] had to be modified by multiple threads at the same time. When one thread makes an update to the priority queue for a particular node in the graph, all the other threads have to wait before gaining access to the queue. Thus, the priority queue caused lots of overhead for concurrent execution of these algorithms. The priority queue implementation was removed to improve performance. In the case of the TDG algorithm, only the preprocessing step runs in a parallel fashion.

The experiments to measure the performance improvement achieved were run on a single cluster node with 16 core processors. Figures 8.1 and 8.2 show the execution times for the

influence maximization algorithm under IC model and the LDAG algorithm respectively. The propagation probability limit has been set to 0.0001 for these experiments. As shown in the figure 8.1, for the ca-astroph graph when the worker threads were increased from 1 to 8, the execution time for preprocessing/setup step reduced from 210 seconds to 16 seconds, and the time to compute 500 seeds reduced from 4701 seconds to 3271 seconds. For the Cit-hepph graph, when the worker threads were increased from 1 to 15, the execution time for the preprocessing/setup step reduced from 491 seconds to 26 seconds, and the time to determine 500 seeds reduced from 12178 seconds to 7488 seconds.

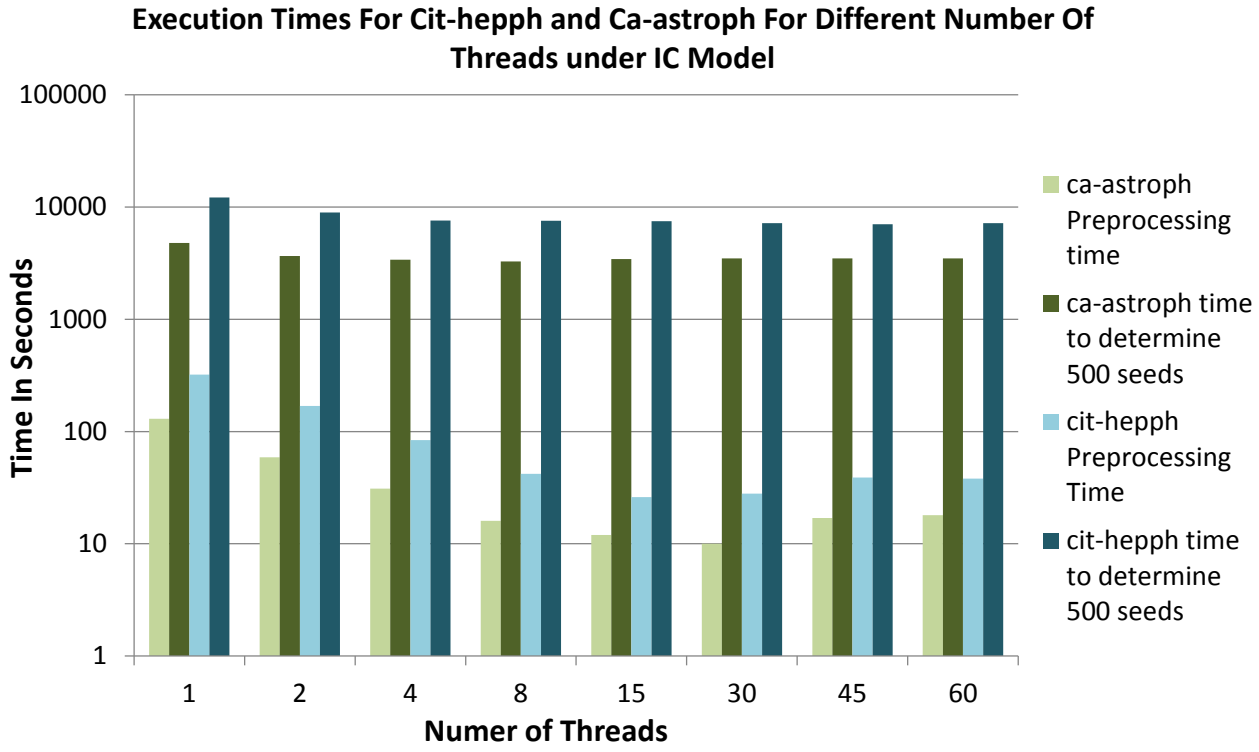


Figure 8.1: Performance Improvement Achieved Due to Multithreading for Chen's Influence Maximization algorithm for Independent cascade model.

As shown in the figure 8.2, for the ca-astroph graph when the worker threads were increased from 1 to 30, the execution time for preprocessing/setup step for the LDAG algorithm reduced from 119 seconds to 11 seconds, and the time to compute 500 seeds reduced from 498 seconds to 487 seconds. For the Cit-hepph graph, when the worker threads were increased

from 1 to 30, the execution time for the preprocessing/setup step of the LDAG algorithm reduced from 309 seconds to 26 seconds, and the time to determine 500 seeds reduced from 673 seconds to 671 seconds.

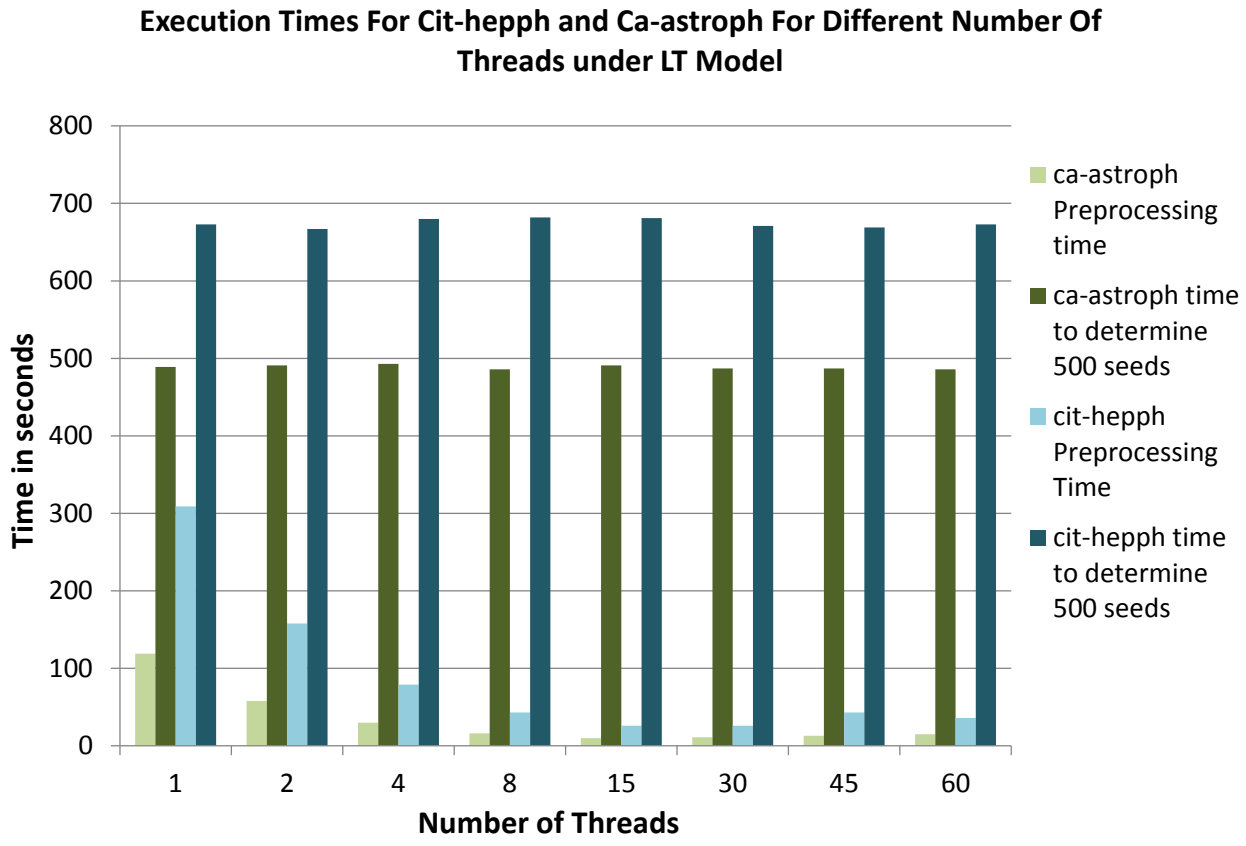


Figure 8.2: Performance Improvement Achieved Due to Multithreading for Chen's Influence Maximization algorithm for Linear Threshold model.

Figure 8.3 shows the execution times taken by the preprocessing step in TDG algorithm for different number of threads. As shown in the figure, for the Epinion graph when the worker threads were increased from 1 to 45, the execution time for the preprocessing/setup step reduced from 143 seconds to 18 seconds. For the Mva graph, when the worker threads were increased from 1 to 45, the execution time for the preprocessing/setup step reduced from 98 seconds to 14 seconds.

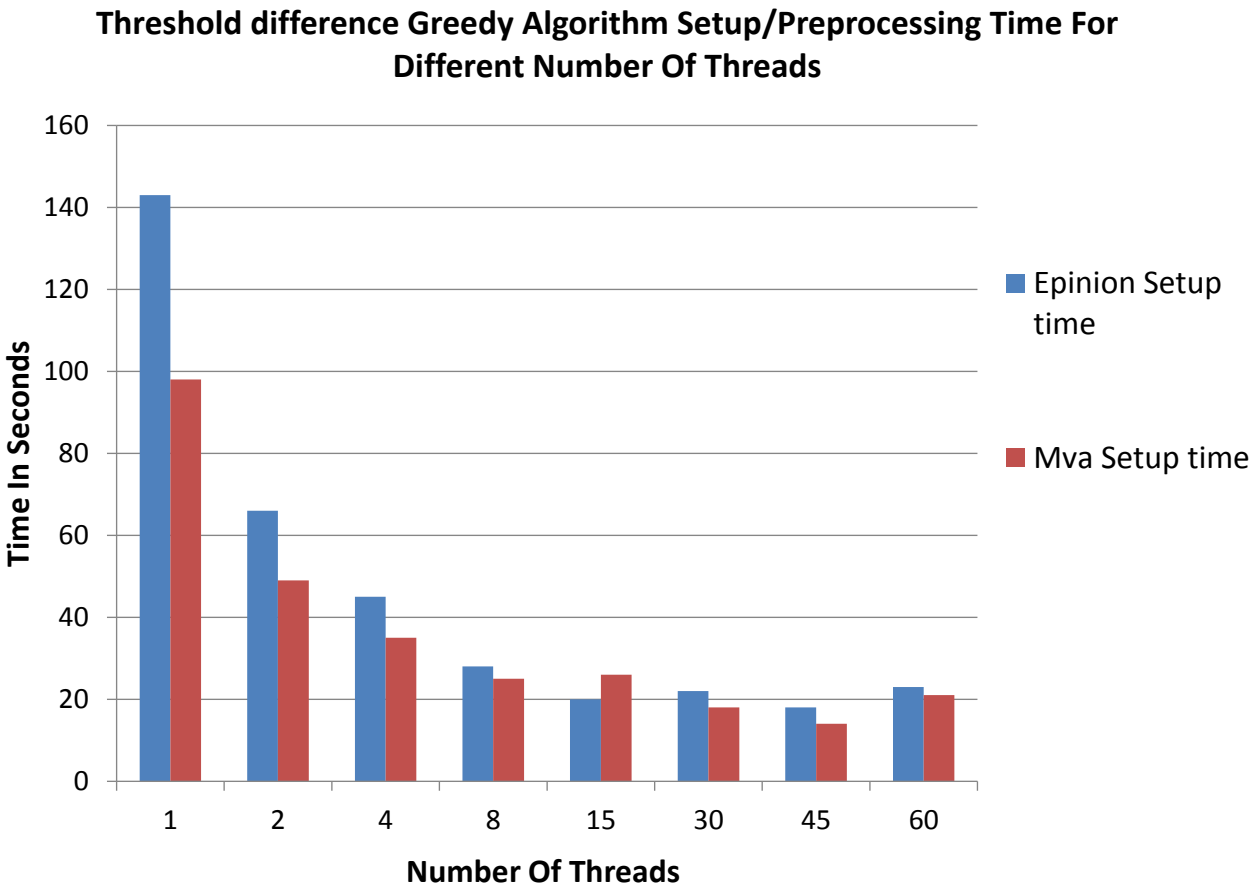


Figure 8.3: Time Taken by the Setup/Preprocessing to Complete under the TDG Algorithm for Different Number of Threads.

Chapter 9

Conclusion

The research in the field of influence maximization is growing rapidly. We have looked into a broad range of algorithms from the literature, and have provided a brief summary for each of the algorithms examined. In this thesis, I have presented the threshold difference greedy (TDG) algorithm for the deterministic linear threshold model which addresses both the influence maximization problem as well as the target selection problem. With extensive experiments on 14 real-world networks of varying size and density, I have shown that the novel approach using vertex thresholds is better than the seven other algorithms taken from the literature. Since the execution time is a crucial factor for any influence maximization heuristic, the tuneable parameters are essential in controlling the execution times. The performance improvement attained through threading was important in reducing the execution times for several graphs. Through this thesis, I have shown that the threshold of a node is an important input for a graph and data mining techniques to compute threshold values for real world networks can provide more accurate results for the influence maximization problem.

Bibliography

[1] Sherif Elmeligy Abdelhamid, Richard Alo, S. M. Arifuzzaman, Pete Beckman, Md Hasanuzzaman Bhuiyan, Keith Bisset, Edward A. Fox, Geoffrey C. Fox, Kevin Hall, S.M.Shamimul Hasan, Anurodh Joshi, Maleq Khan, Chris J. Kuhlman, Spencer Lee, Jonathan P. Leidig, Hemanth Makkapati, Madhav V. Marathe, Henning S. Mortveit, Judy Qiu, S.S. Ravi, Zalia Shams, Ongard Sirisaengtaksin, Rajesh Subbiah, Samarth Swarup, Nick Trebon, Anil Vullikanti, and Zhao Zhao. CINET: A CyberInfrastructure for Network Science. In *8th IEEE International Conference on eScience*, 2012.

[2] Eyal Ackerman, Oren Ben-Zwi, and Guy Wolfovitz. Combinatorial model and bounds for target set selection. *Theoretical Computer Science*, 411:4017–4022, 2010.

[3] Christopher L. Barrett, Harry B. Hunt III, Madhav V. Marathe, S. S. Ravi, Daniel J. Rosenkrantz, and Richard E. Stearns. Complexity of reachability problems for finite discrete sequential dynamical systems. *Journal of Computer and System Sciences*, 72:1317–1345, 2006.

[4] Christopher L. Barrett, Harry B. Hunt III, Madhav V. Marathe, S. S. Ravi, Daniel J. Rosenkrantz, and Richard Edwin Stearns. Reachability problems for sequential dynamical systems with threshold functions. *Theoretical Computer Science*, 295(1-3):41–64, 2003.

[5] Christopher L. Barrett, Harry B. Hunt III, Madhav V. Marathe, S. S. Ravi, Daniel J. Rosenkrantz, Richard Edwin Stearns, and Mayur Thakur. Predecessor existence problems for finite discrete dynamical systems. *Theoretical Computer Science*, 386:3–37, 2007.

[6] Oren Ben-Zwi, Danny Hermelin, Daniel Lokshtanov, and Ilan Newman. An exact almost optimal algorithm for target set selection in social networks. In *ACM Conference on Electronic Commerce (EC 2009)*, pages 355–362, 2009.

[7] Oren Ben-Zwi, Danny Hermelin, Daniel Lokshtanov, and Ilan Newman. Treewidth governs the complexity of target set selection. *Discrete Optimization*, 8:87–96, 2011.

[8] Glenn Bevilacqua, Shealen Clare, Amit Goyal, and Laks V. S. Lakshmanan. Validating Network Value of Influencers by means of Explanations. In *2013 IEEE 13th International Conference on Data Mining (ICDM 2013)*, 2013.

[9] Phillip Bonacich. Some unique properties of eigenvector centrality. *Social Networks*, 29(4):555–564, 2007.

[10] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, 2014.

[11] Ceren Budak, Divyakant Agrawal, and Amr El Abbadi. Limiting the Spread of Misinformation in Social Networks. In *Proceedings of the 20th International Conference of World Wide Web Conference (WWW 2011)*, 2011.

[12] Barrett C, Beckman R, Khan M, Kumar VS Anil, Marathe M, Stretz P, Dutta T, and Lewis B. Generation and analysis of large synthetic social contact networks. In *Winter Simulation Conference*, pages 1003–1014, 2009.

[13] D. Centola and M. Macy. Complex Contagions and the Weakness of Long Ties. *American Journal of Sociology*, 113(3):702–734, 2007.

[14] Ning Chen. On the approximability of influence in social networks. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2008)*, pages 1029–1037, 2008.

[15] Ning Chen. On the approximability of influence in social networks. *SIAM J. Discrete Mathematics*, 23:1400–1415, 2009.

[16] Wei Chen, Wei Lu, and Ning Zhang. Time-Critical Influence Maximization in Social Networks with Time-Delayed Diffusion Process. In *(CoRR 2012)*, 2012.

[17] Wei Chen, Chi Wang, and Yajun Wang. Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2010)*, pages 1029–1038, 2010.

[18] Wei Chen, Chi Wang, and Yajun Wang. Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks. Technical report, Microsoft Research Technical Report MSR-TR-2010-2, 2010.

[19] Wei Chen, Yajun Wang, and Siyu Yang. Efficient Influence Maximization in Social Networks. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2009)*, 2009.

[20] Wei Chen, Yifei Yuan, and Li Zhang. Scalable Influence Maximization in Social Networks under the Linear Threshold Model. In *2010 IEEE 10th International Conference on Data Mining (ICDM 2010)*, 2010.

[21] Wei Chen, Yifei Yuan, and Li Zhang. Scalable Influence Maximization in Social Networks Under the Linear Threshold Model. Technical report, Microsoft Research Technical Report MSR-TR-2010-133, 2010.

[22] M. Conover, J. Ratkiewicz, M. Francisco, B Goncalves, A. Flammini, and F. Menczer. Political polarization on twitter. In *Proc. of the Fifth International AAAI Conference onWeblogs and Social Media (AAAI 2011)*, 2011.

[23] P. Domingos and M. Richardson. Mining the Network Value of Customers. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2001)*, pages 57–61, 2001.

[24] Paul A. Dreyer. *Applications and Variations of Domination in Graphs*. Ph.D. Thesis, Rutgers University, 2000.

[25] Paul A. Dreyer and Fred S. Roberts. Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion. *Discrete Applied Mathematics*, 157:1615–1627, 2009.

[26] Lidan Fan, Zaixin Lu, Weili Wu, Bhavani Thuraisingham, Huan Ma, and Yuanjun Bi. Least cost rumor blocking in social networks. In *2013 IEEE 33rd International Conference on Distributed Computing (ICDCS 2013)*, pages 540–549, 2013.

[27] James H. Fowler and Nicholas A. Christakis. Dynamic spread of happiness in a large social network: longitudinal analysis over 20 years in the Framingham Heart Study. *BMJ*, 337, 2008.

[28] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[29] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979.

[30] Sharad Goel, Duncan J Watts, and Daniel G Goldstein. The structure of online diffusion networks. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, pages 623–638. ACM, 2012.

[31] Sandra Gonzalez-Bailon, Javier Borge-Holthoefer, Alejandro Rivero, and Yamir Moreno. The Dynamics of Protest Recruitment Through an Online Network. *Nature Scientific Reports*, pages 1–7, 2011. DOI: 10.1038/srep00197.

[32] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[33] J. Leskovec website, 2011. `http://cs.stanford.edu/people/jure/`.

[34] Fariba Karimi and Petter Holme. Threshold model of cascades in empirical temporal networks. *Physica A: Statistical Mechanics and its Applications*, 392:3476–3483, 2013.

[35] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the Spread of Influence Through a Social Network. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2003)*, pages 137–146, 2003.

[36] David Kempe, Jon Kleinberg, and Eva Tardos. Influential Nodes in a Diffusion Model for Social Networks. In *Proc. Intl. Conf. on Automata, Languages and Programming (ICALP 2005)*, pages 1127–1138, 2005.

[37] Masahiro Kimura and Kazumi Saito. Tractable Models for Information Diffusion in Social Networks. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (PKDD 2006)*, pages 259–271, 2006.

[38] Chris J. Kuhlman, V. S. Anil Kumar, Madhav V. Marathe, S. S. Ravi, and Daniel J. Rosenkrantz. Inhibiting diffusion of complex contagions in social networks: Theoretical and experimental results. *J. Data Mining and Knowledge Discovery*, 2014. To appear.

[39] Chris J. Kuhlman, V. S. Anil Kumar, and S. S. Ravi. Controlling opinion propagation in online networkss. *Journal of Computer Networks*, 57:2121–2132, 2013.

[40] V. S. Anil Kumar, Matthew Macauley, and Henning S. Mortveit. Limit set reachability in asynchronous graph dynamical systems. In *Reachability Problems (RP) 2009*, volume 5797 of *Lecture Notes in Computer Science*, pages 217–232, Berlin/Heidelberg, 2009. Springer.

[41] Jure Lekovec, Andreas Krause, Carlos Guestrin, and Christos Faloutsos Jeanne VanBriesen Natalie Glance. Cost-effective Outbreak Detection in Networks. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2007)*, 2007.

[42] Yanhua Li, Wei Chen, Yajun Wang, and Zhi-Li Zhang. Influence Diffusion Dynamics and Influence Maximization in Social Networks with Friend and Foe Relationships. In *6th ACM International Conference on Web Search and Data Mining (WSDM 2013)*, pages 657–666, 2013.

[43] Bo Liu, Gao Cong, Dong Xu, and Yifeng Zeng. Time Constrained Influence Maximization in Social Networks. In *2012 IEEE 12th International Conference on Data Mining (ICDM 2012)*, 2012.

[44] Zaixin Lu, Wei Zhang, Weili Wu, Joonmo Kim, and Bin Fu. The complexity of influence maximization problem in the deterministic linear threshold model. *Journal of Combinatorial Optimization*, 24(3):374–378, 2012.

[45] Zaixin Lu, Wei Zhang, Weili Wu, Joonmo Kim, and Bin Fu. The complexity of influence maximization problem in the deterministic linear threshold model. *Journal of Combinatorial Optimization*, DOI 10.1007/s10878-011-9393-3, 2012.

[46] Matthew Macauley and Henning S. Mortveit. Cycle equivalence of graph dynamical systems. *Nonlinearity*, 22(2):421–436, 2009. math.DS/0709.0291.

[47] Matthew Macauley and Henning S. Mortveit. Update sequence stability in graph dynamical systems. *Discrete and Continuous Dynamical Systems S.*, 4(6):1533–1541, 2011. Preprint: math.DS/0909.1723.

[48] Henning S. Mortveit and Christian M. Reidys. *An Introduction to Sequential Dynamical Systems*. Universitext. Springer Verlag, 2007.

[49] G. L. Nemhauser, L. A. Wosley, and M. L. Fisher. An Analysis of Approximations for Maximizing Submodular Functions. *Mathematical Programming Study*, 14:265–294, 1978.

[50] Nam P. Nguyen, Guanhua Yan, My T. Thai, and Stephan Eidenbenz. Containment of Viral Spread in Online Social Networks. In *Proceedings of the ACM Web Science Conference (WebSci 2012)*, June 2012.

[51] Mark G. Orr and Clare Rosenfeld Evans. Understanding long-term diffusion dynamics in the prevalence of adolescent sexual initiation: A first investigation using agent-based modeling. *Research in Human Development*, 8:48–66, 2011.

[52] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[53] M. Richardson and P. Domingos. Mining Knowledge-Sharing Sites for Viral Marketing. In *Proc. ACM Intl. Conf. on Data Mining and Knowledge Discovery (KDD 2002)*, pages 61–70, 2002.

[54] Paulo Shakarian and Damon Paulo. Large Social Networks can be Targeted for Viral Marketing with Small Seed Sets. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, 2012.

[55] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon Kleinberg. Structural Diversity in Social Contagion. *Proceedings of the Naitonal Academy of Sciences (PNAS 2012)*, 109(9):5962–5966, 2012.

[56] B. Viswanath, A. Mislove, M. Cha, and K. Gummadi. On the evolution of user interaction in Facebook. In *Proc. WOSN*, pages 7–12, 2009.

[57] Cheng Wang, Lili Deng, Gengui Zhou, and Meixian Jiang. A global optimization algorithm for target set selection problems. *Information Sciences*, 267:101–118, 2014.

[58] Chi Wang, Wei Chen, and Yajun Wang. Scalable influence maximization for independent cascade model in large-scale social networks. *Journal of Data Mining and Knowledge Discovery*, DOI 10.1007/s10618-012-0262-1, 2012.

[59] Chuan Zhou, Peng Zhang, Jing Guo, Xingquan Zhu, and Li Guo. UBLF: An Upper Bound Based Approach to Discover Influential Nodes in Social Networks. In *2013 IEEE 13th International Conference on Data Mining (ICDM 2013)*, 2013.

[60] Honglei Zhuang, Yihan Sun, Jie Tang, Jialin Zhang, and Xiaoming Sun. Influence Maximization in Dynamic Social Networks. In *2013 IEEE 13th International Conference on Data Mining (ICDM 2013)*, 2013.

Appendix A

Plots

A.0.1 Execution Times for existing Influence maximization algorithms

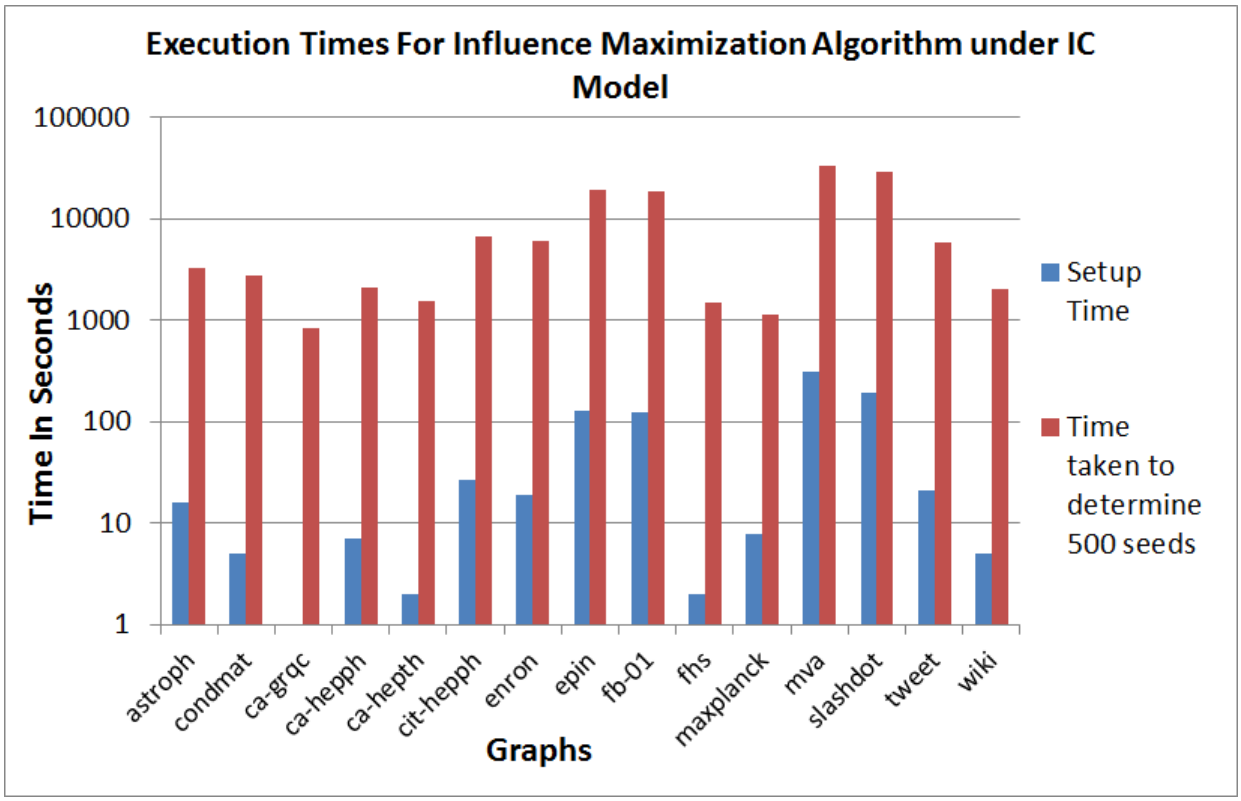


Figure A.1: Execution times for Chen's Influence Maximization algorithm for Independent cascade model

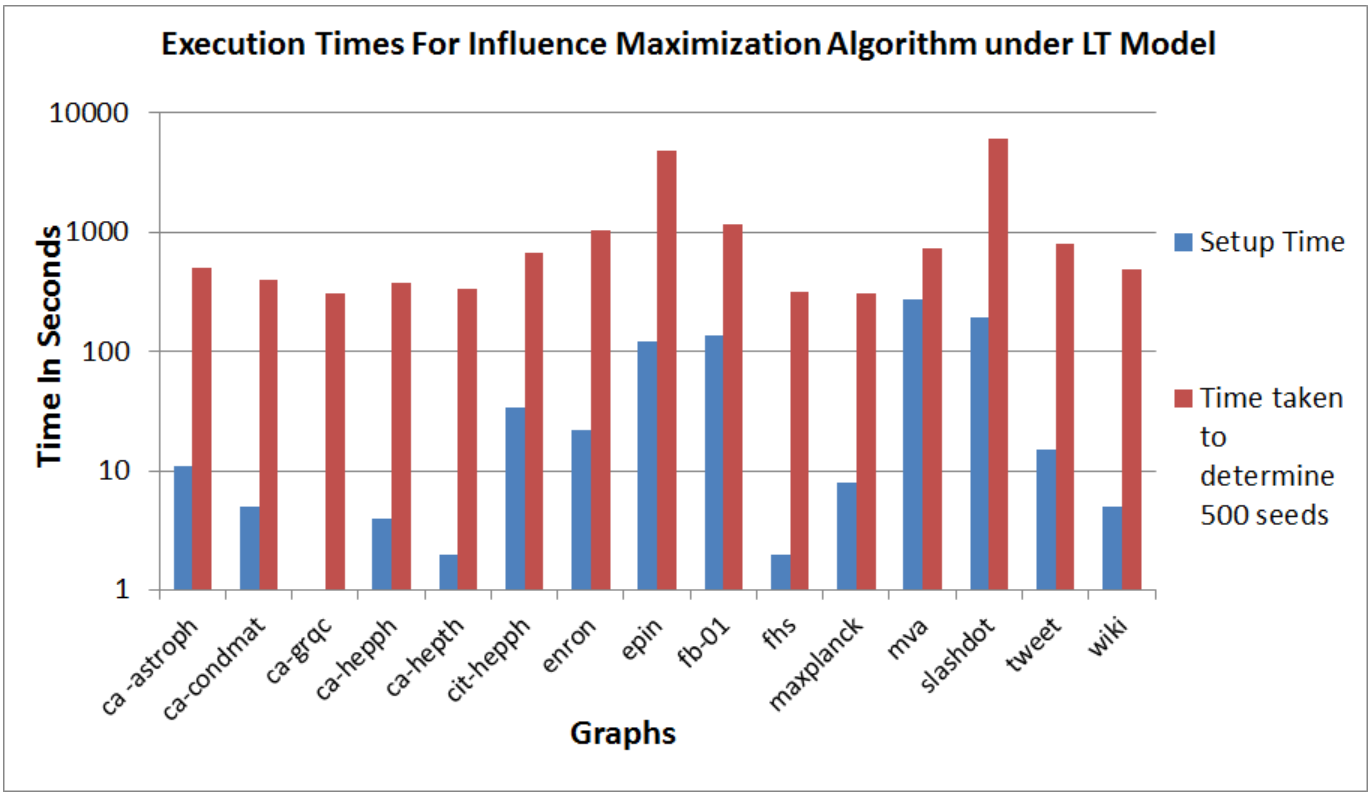


Figure A.2: Execution times for Chen's Influence Maximization algorithm for Linear Threshold model

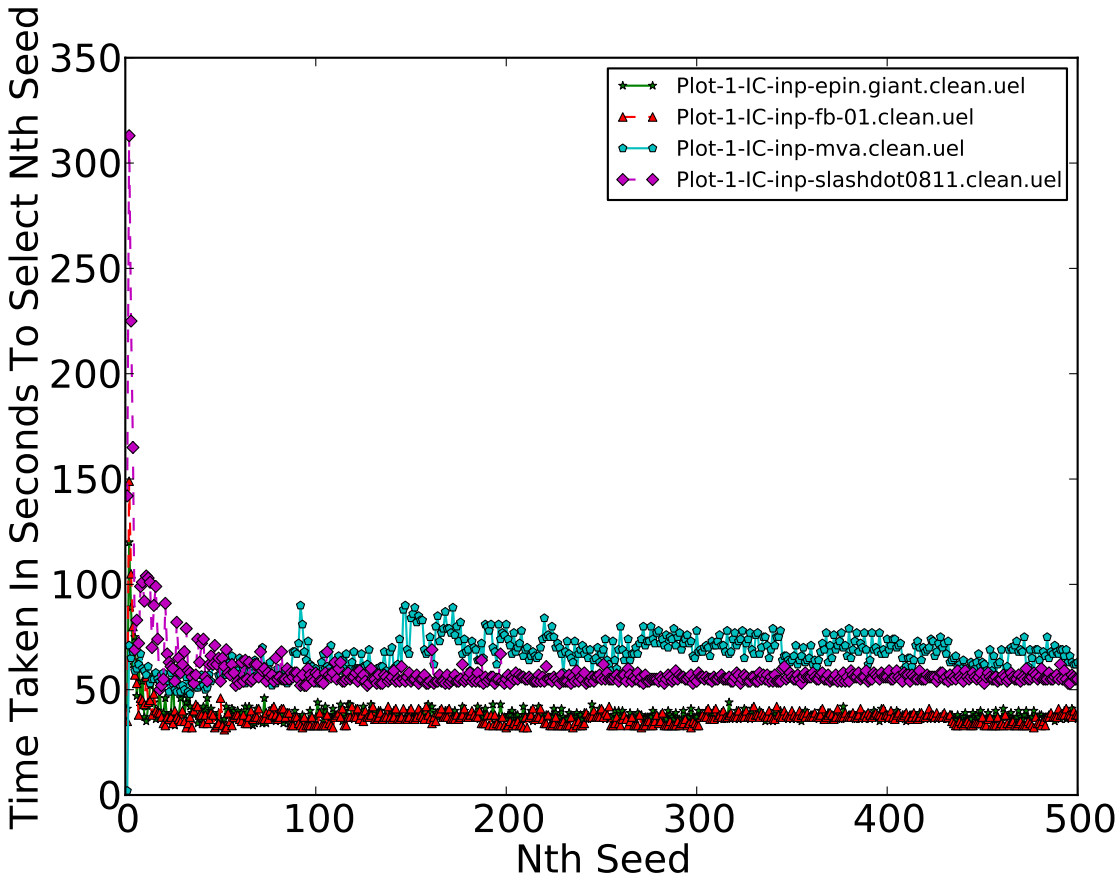


Figure A.3: Time taken to select each individual seed for Epinion, Facebook, Mva and Slashdot graphs based on Chen's Influence Maximization Algorithm for Independent cascade model

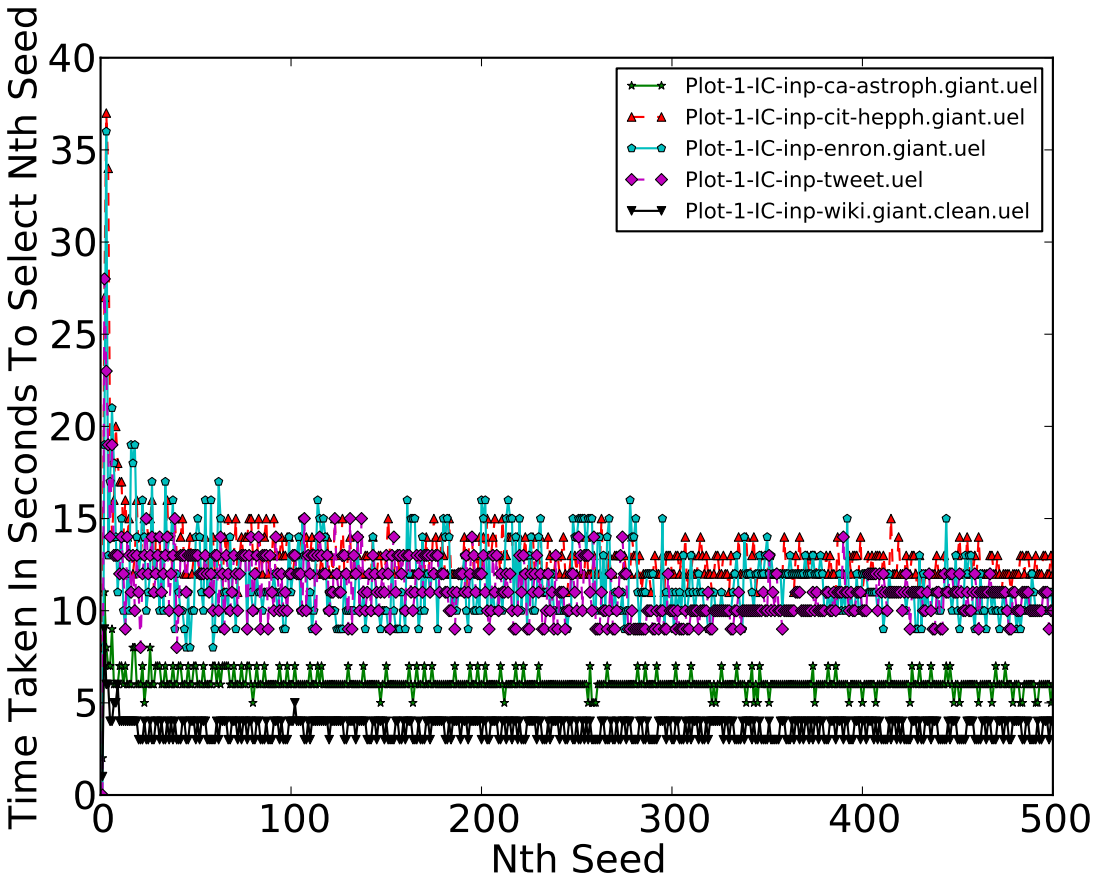


Figure A.4: Time taken to select each individual seed for Ca-Astroph, Cit-Hepph, Enron, Twitter and Wiki graphs based on Chen's Influence Maximization Algorithm for Independent cascade model for

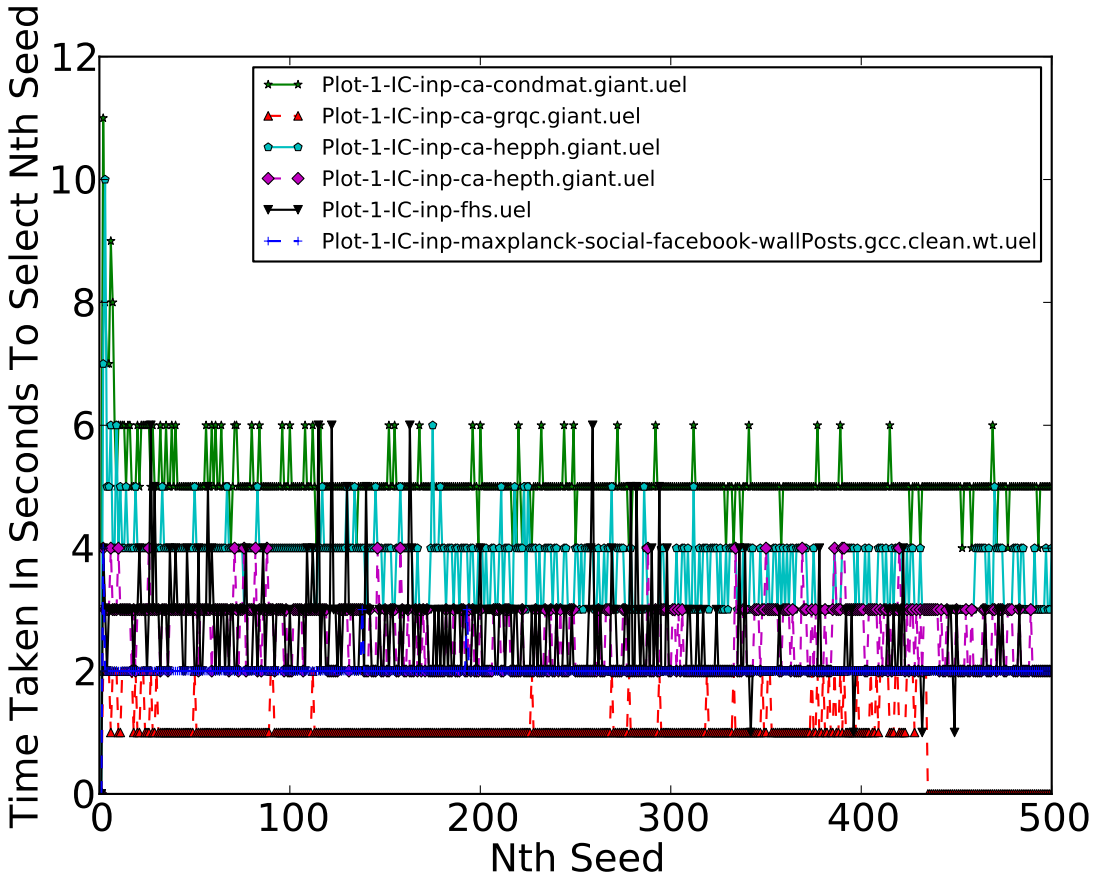


Figure A.5: Time taken to select each individual seed for Condmat, Grqc, Ca-Hepph, Ca-Hepth, Fhs and Maxplanck-social-facebook graphs based on Chen's Influence Maximization Algorithm for Independent cascade model

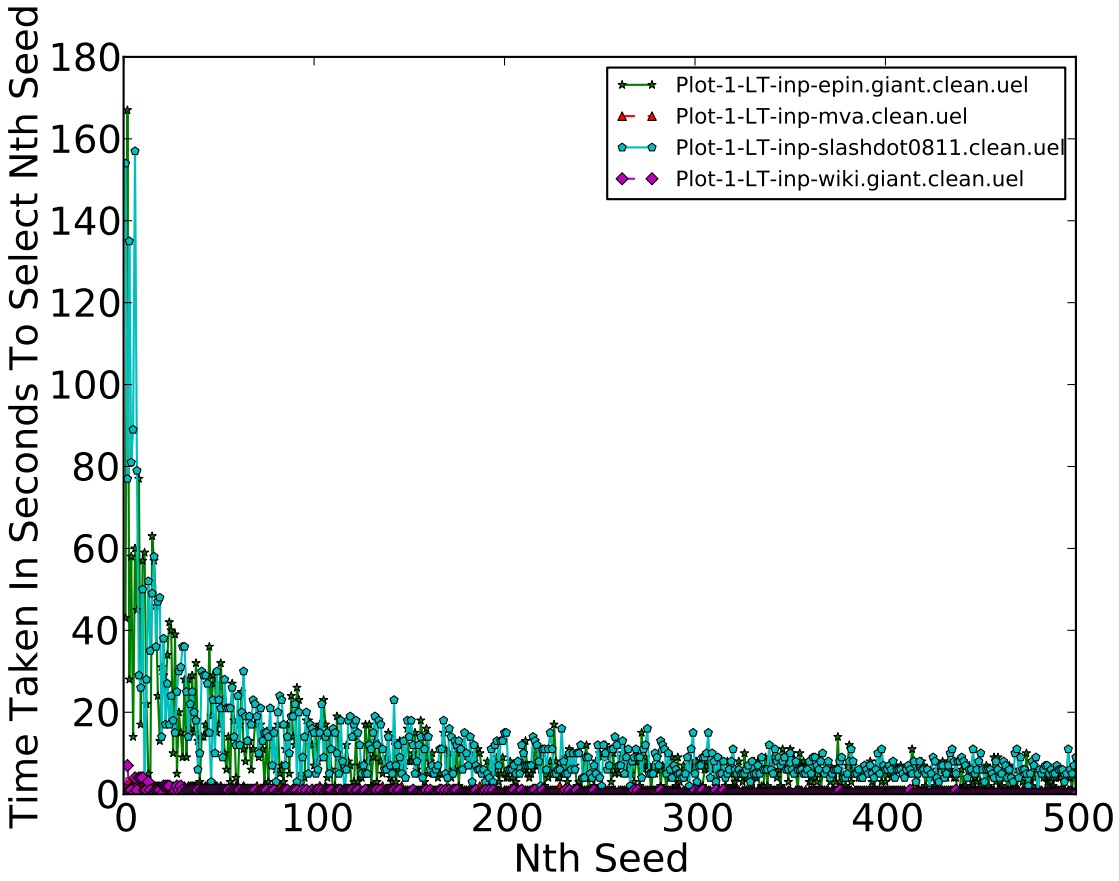


Figure A.6: Time taken to select each individual seed for Epinion, Mva, Slashdot and Wiki graphs based on LDAG algorithm for Linear Threshold model

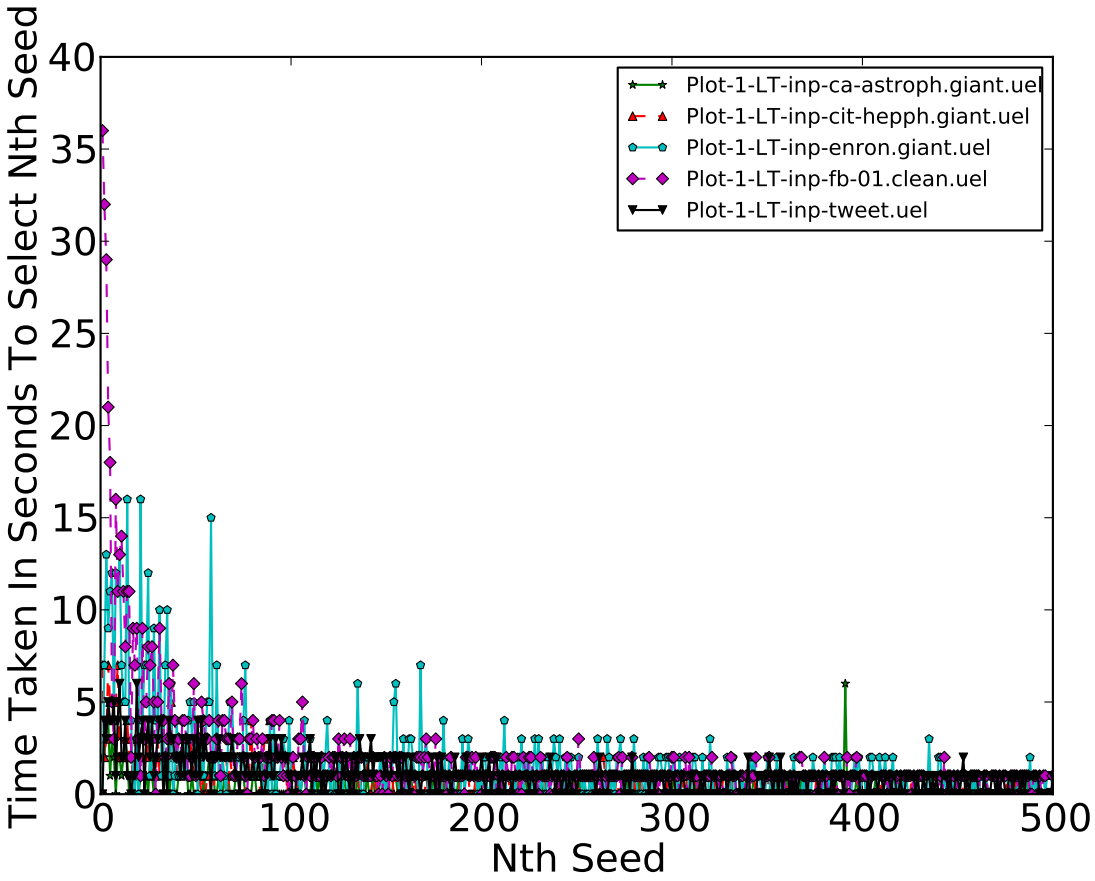


Figure A.7: Time taken to select each individual seed for Ca-Astroph, Cit-Hepph, Enron, Twitter and Facebook graphs based on LDAG algorithm for Linear Threshold model

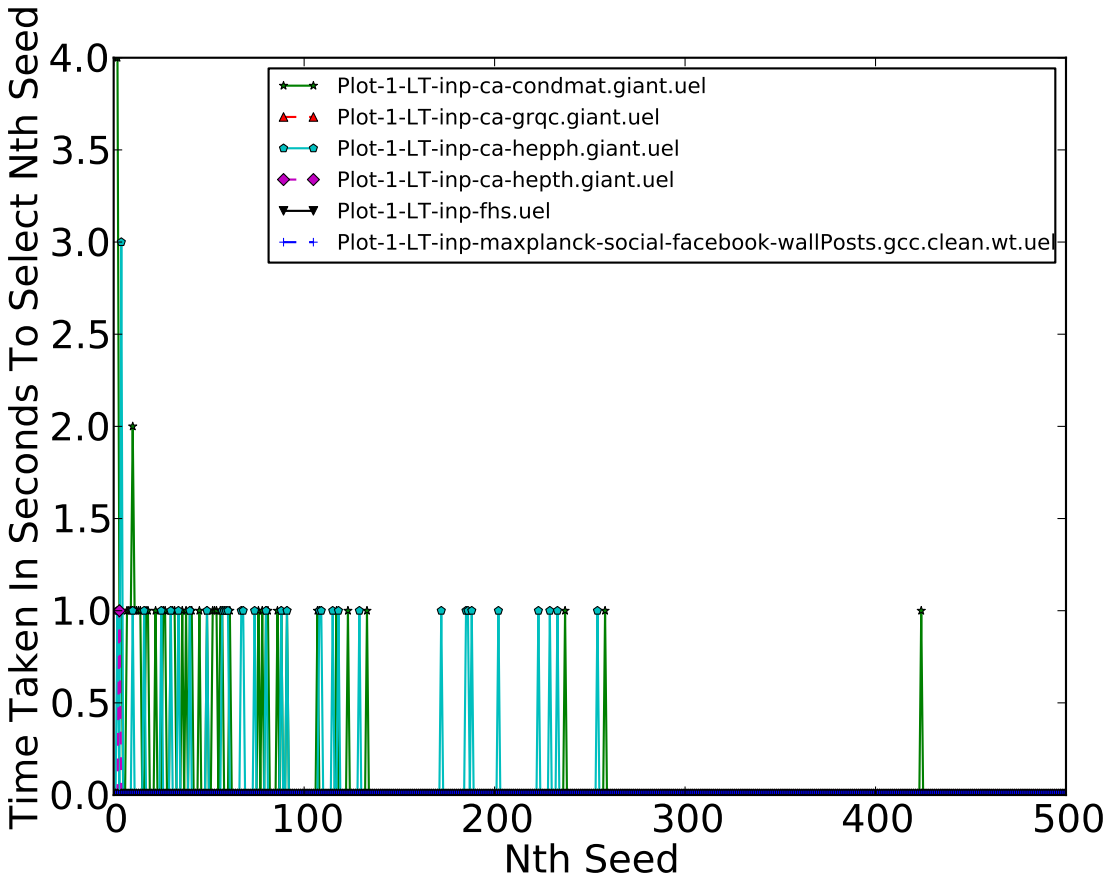


Figure A.8: Time taken to select each individual seed for Condmat, Grqc, Ca-Hepph, Ca-Hepth, Fhs and Maxplanck-social-facebook graphs based on LDAG algorithm for Linear Threshold model

A.0.2 Total completion time comparison between different influence maximization algorithms and existing heuristics

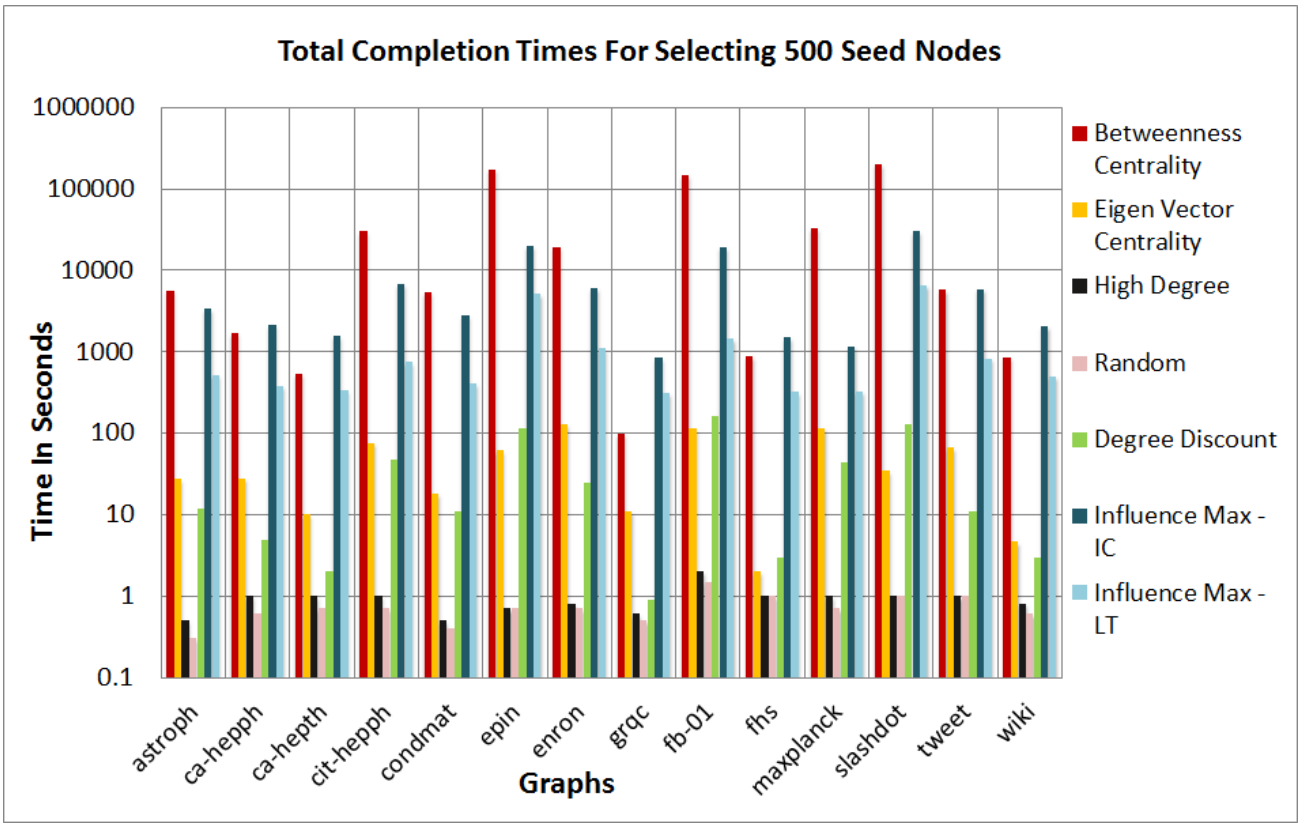


Figure A.9: Comparison of completion times of various algorithms to select 500 most influential seeds

A.0.3 Performance improvement achieved due to concurrent/parallel processing

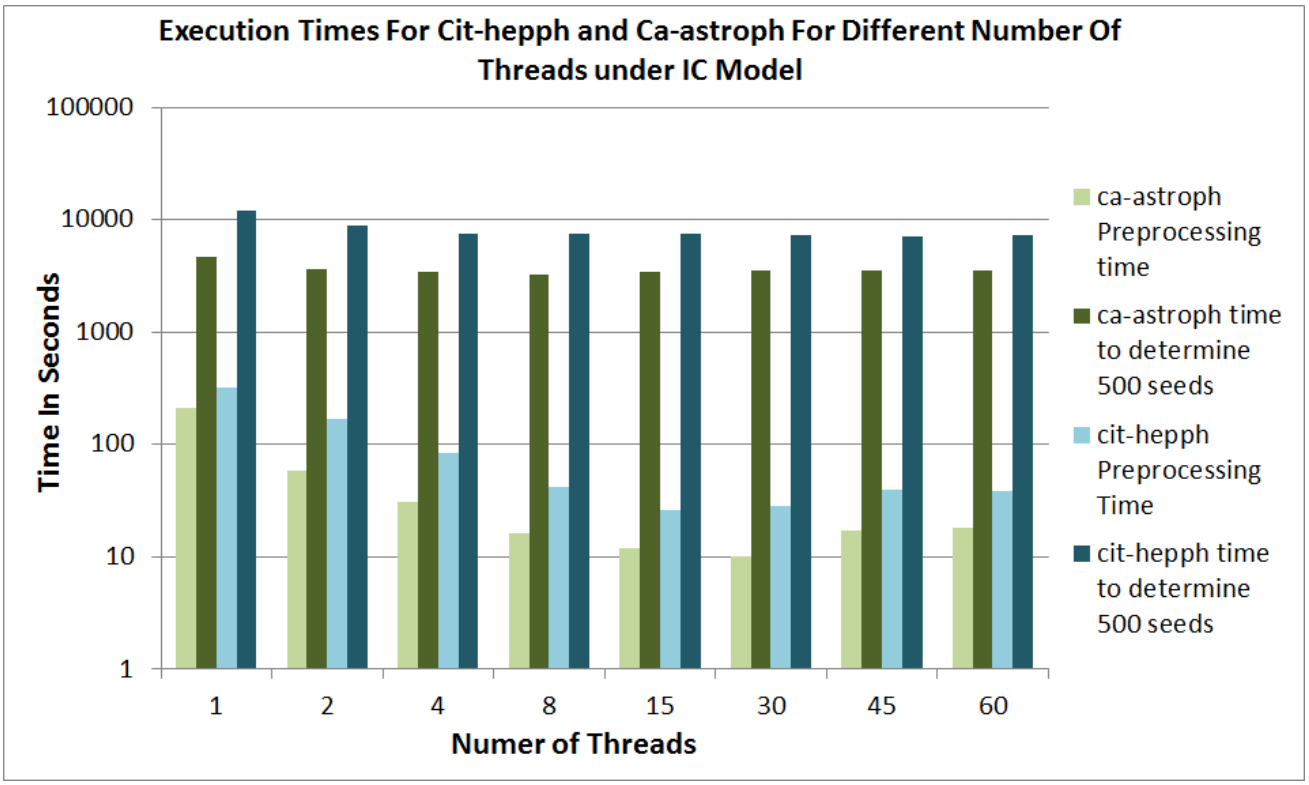


Figure A.10: Performance improvement achieved due to multithreading for Chen's Influence Maximization algorithm for Independent cascade model

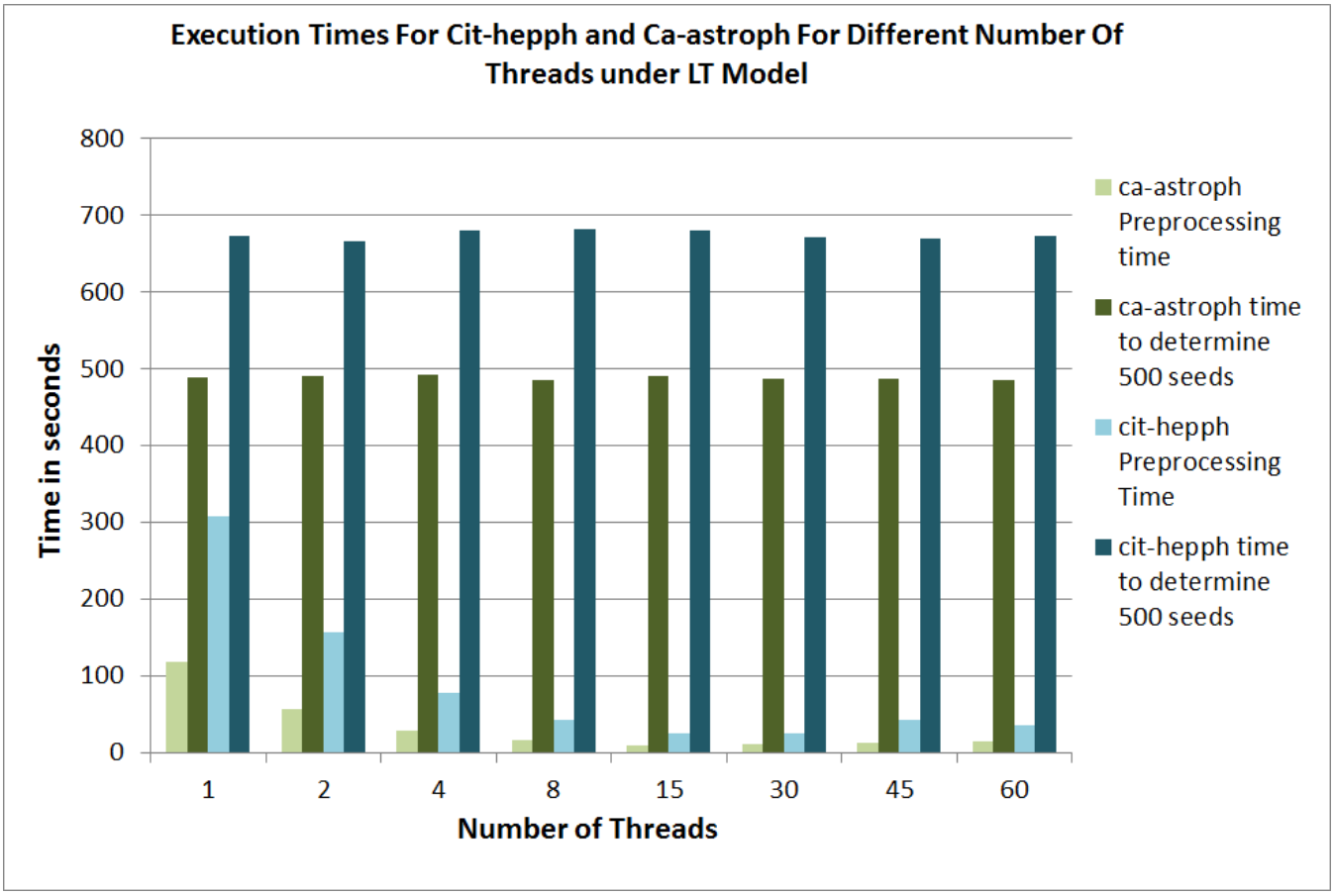


Figure A.11: Performance improvement achieved due to multithreading for Chen's Influence Maximization algorithm for Linear Threshold model.

A.0.4 New algorithm execution time comparison

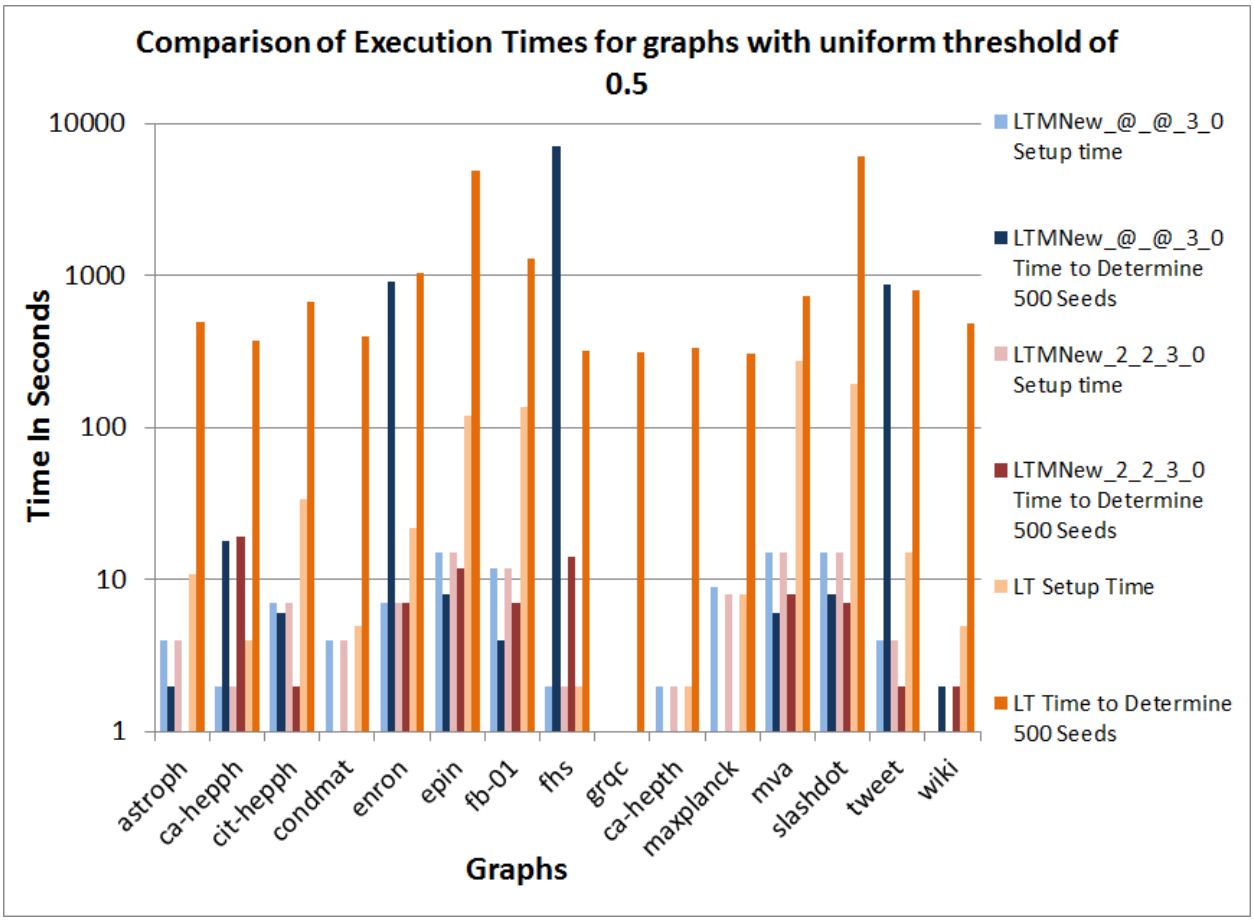


Figure A.12: Comparison of execution times between LDAG algorithm and TDG algorithm for graphs with uniform threshold of 0.5

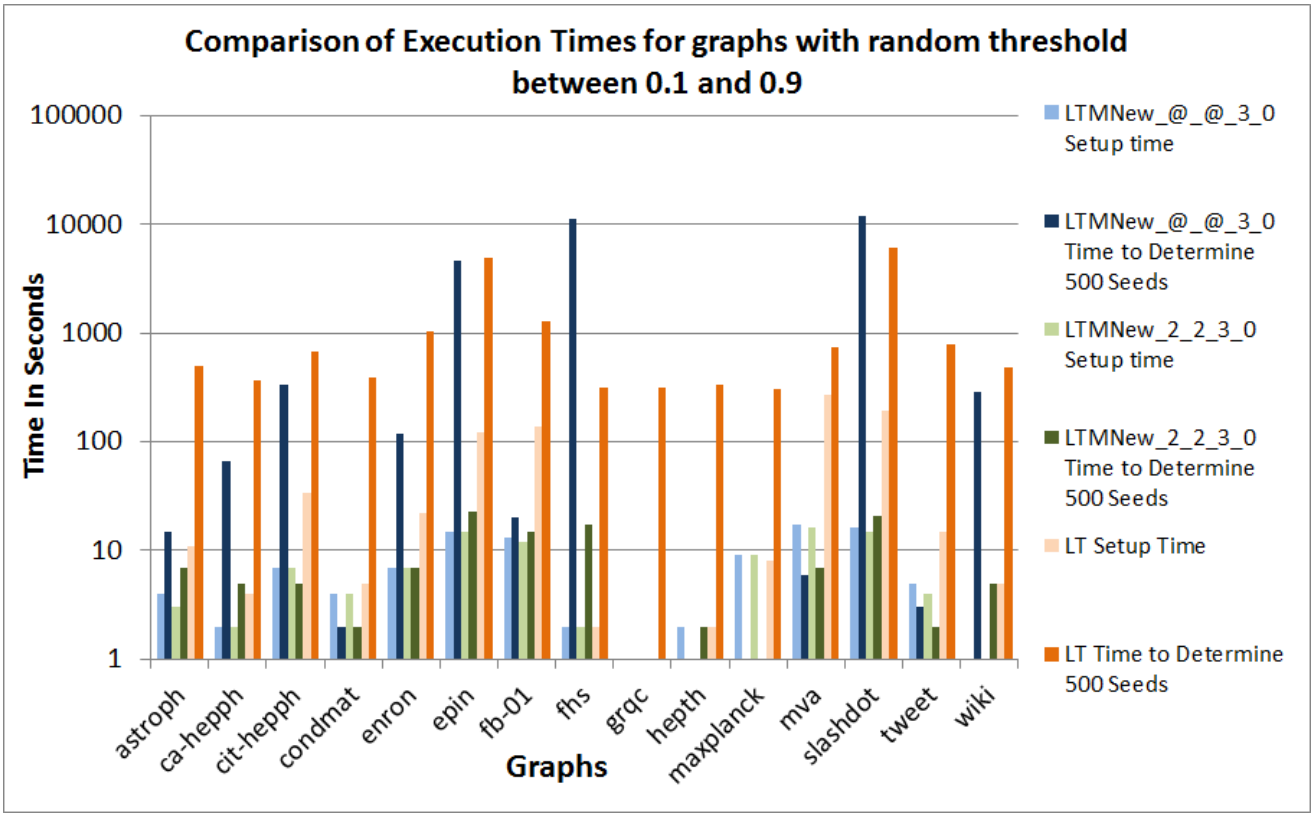


Figure A.13: Comparison of execution times between LDAG algorithm and TDG algorithm for graphs with random threshold between 0.1 and 0.9

A.0.5 Target Set Selection

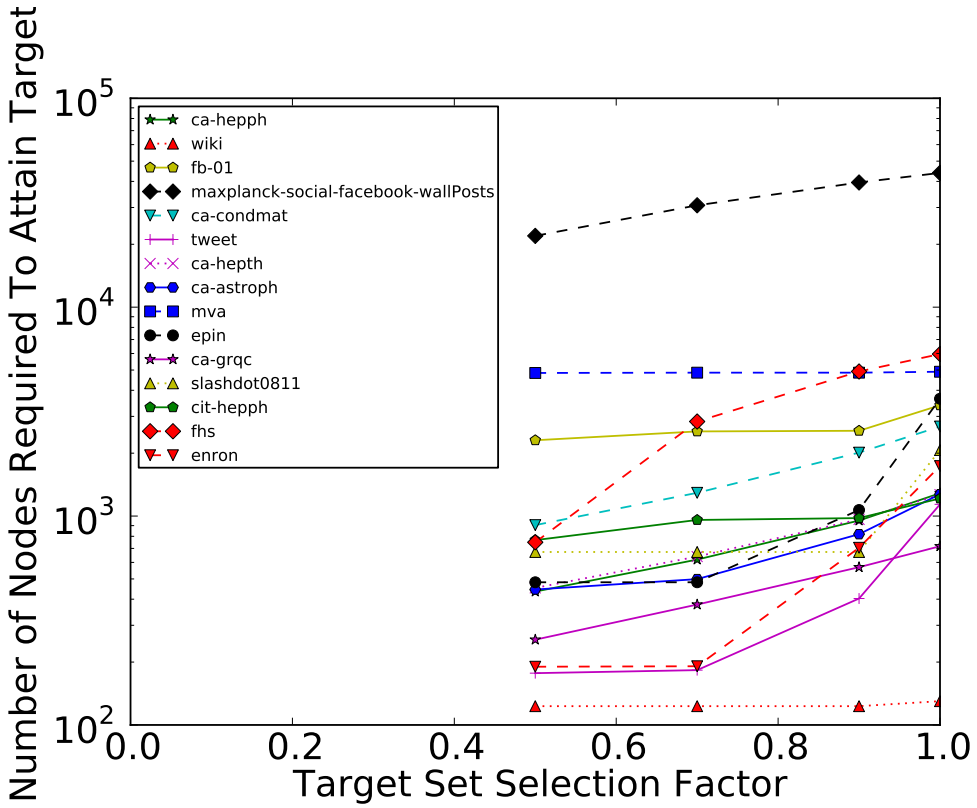


Figure A.14: Target set selection results for various graphs indicating number of seed nodes required to affect fraction of nodes in a graph

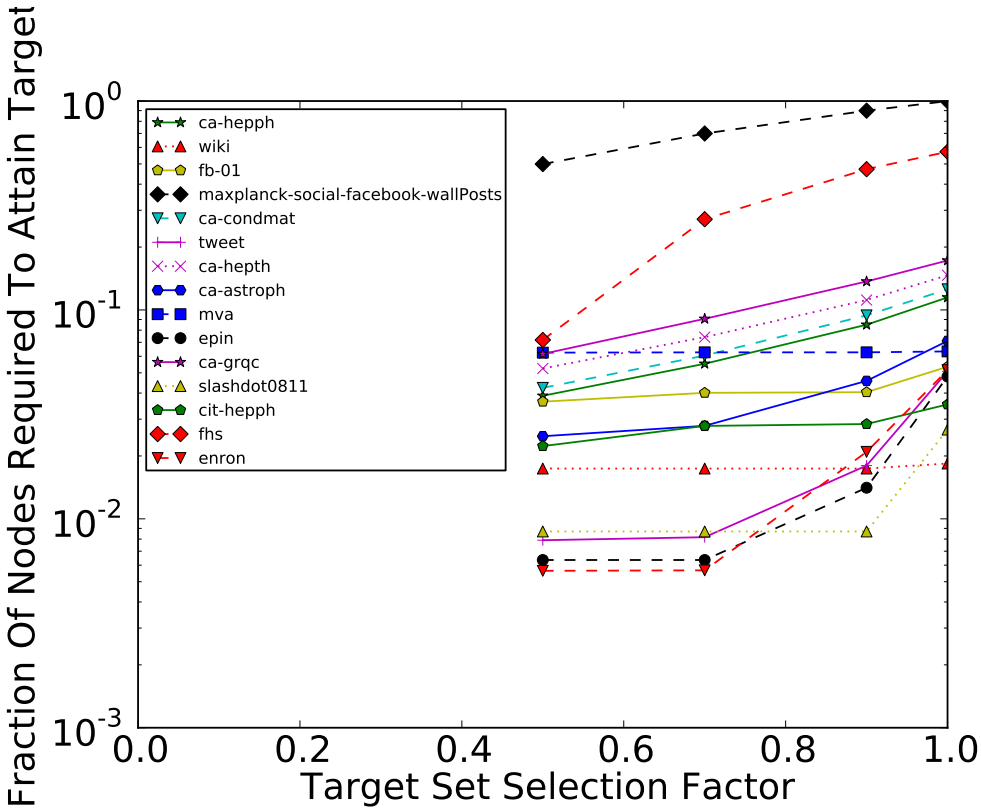


Figure A.15: Target set selection results for various graphs indicating minimum fraction of nodes required to affect a given fraction of nodes in a graph

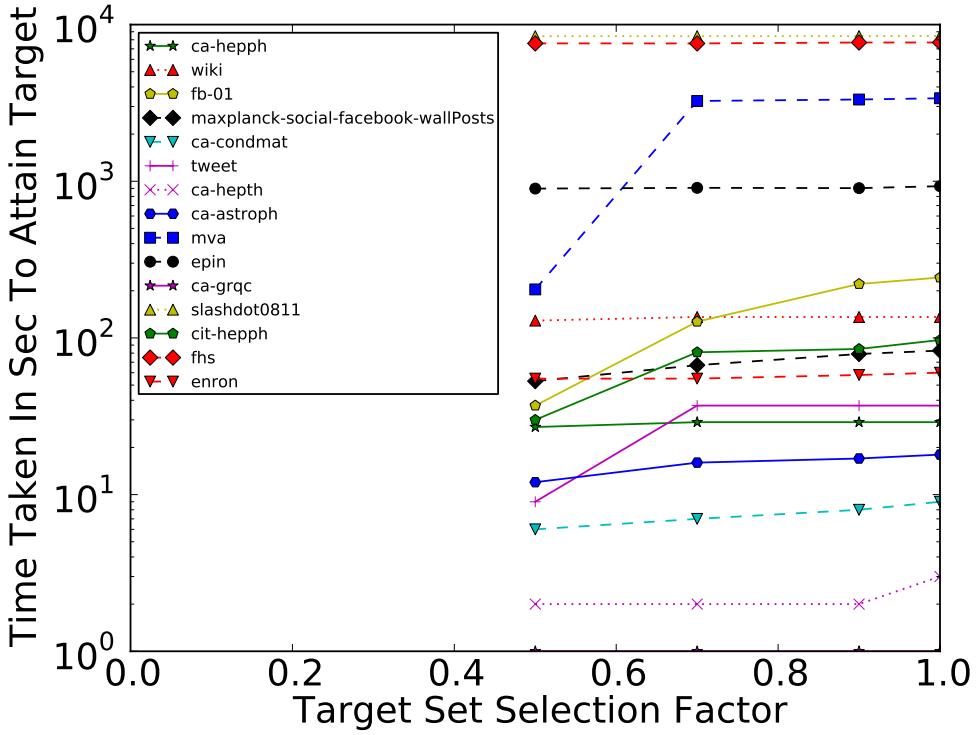


Figure A.16: Target selection results for various graphs indicating time required to affect fraction of nodes in a graph

A.0.6 Final outbreak Results

Uniform threshold of 0.5

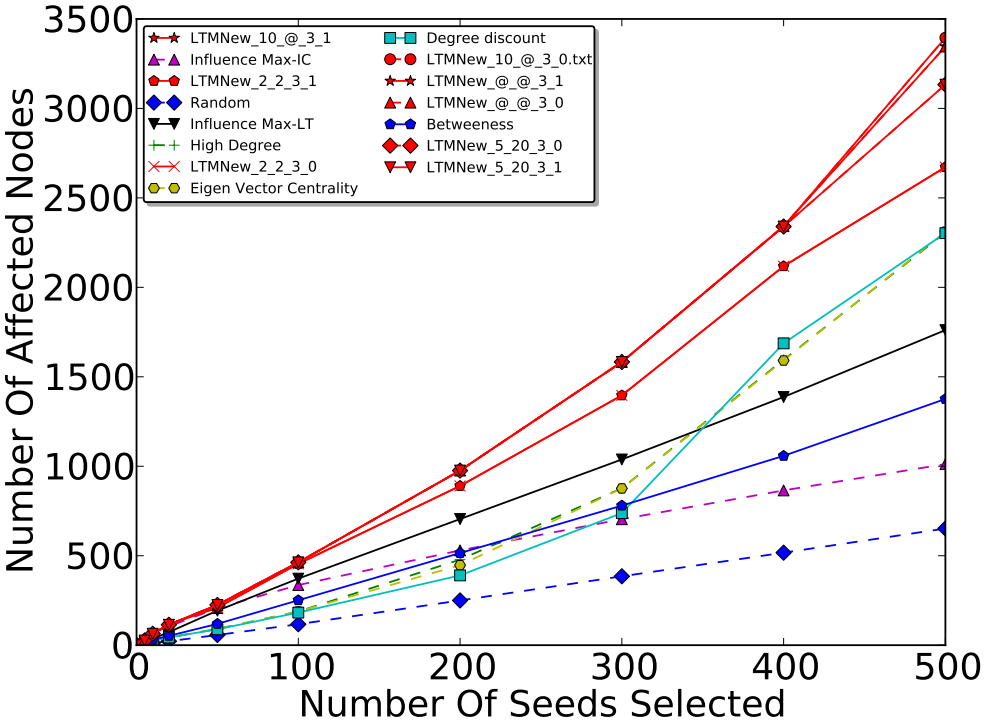


Figure A.17: Final outbreak Results for Astroph graph with uniform threshold of 0.5

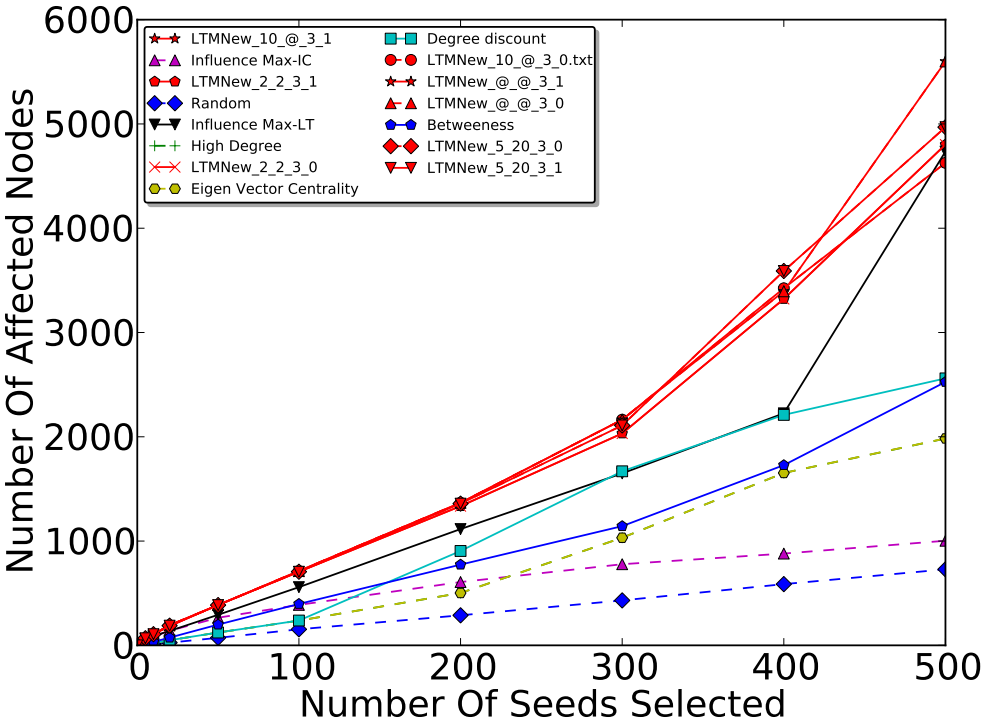


Figure A.18: Final outbreak Results for Ca-hepph graph with uniform threshold of 0.5

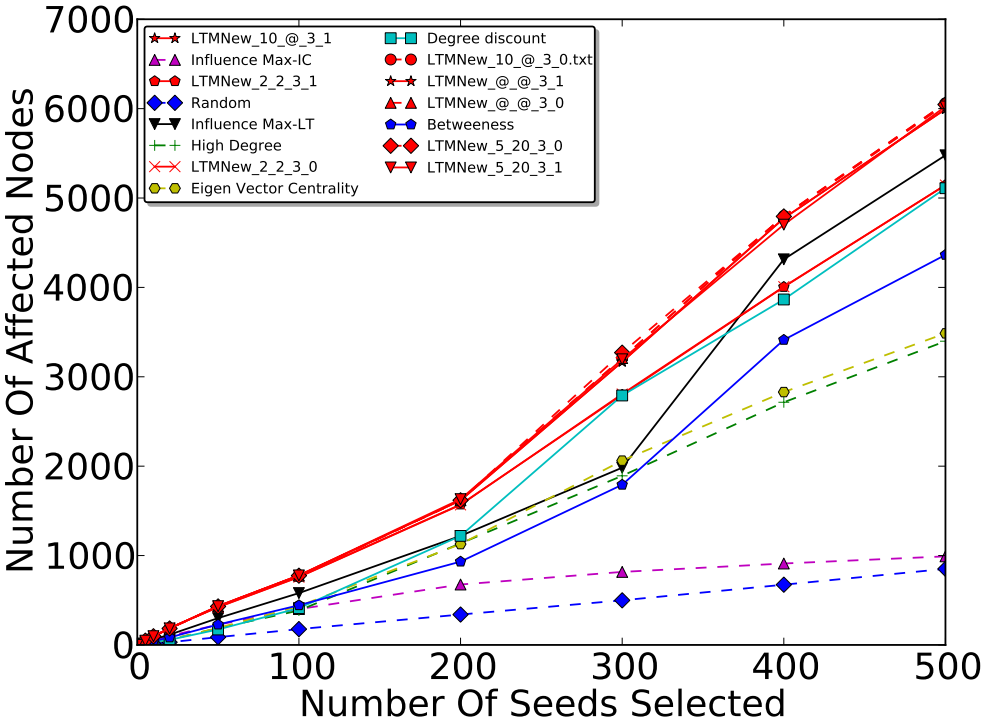


Figure A.19: Final outbreak Results for Ca-hepth graph with uniform threshold of 0.5

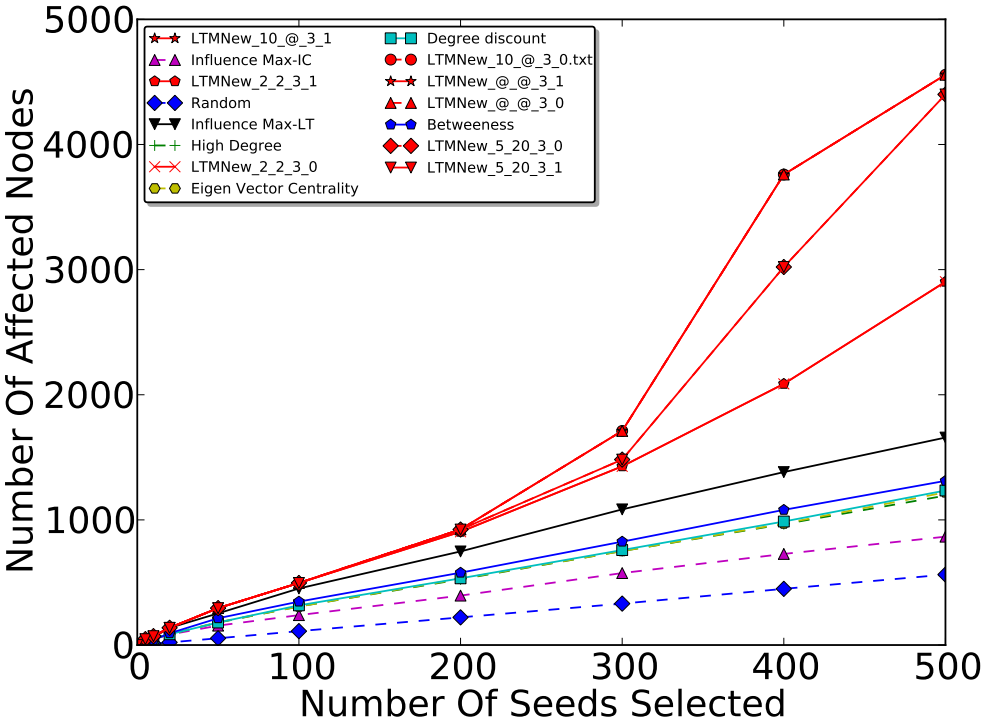


Figure A.20: Final outbreak Results for Cit-hepph graph with uniform threshold of 0.5

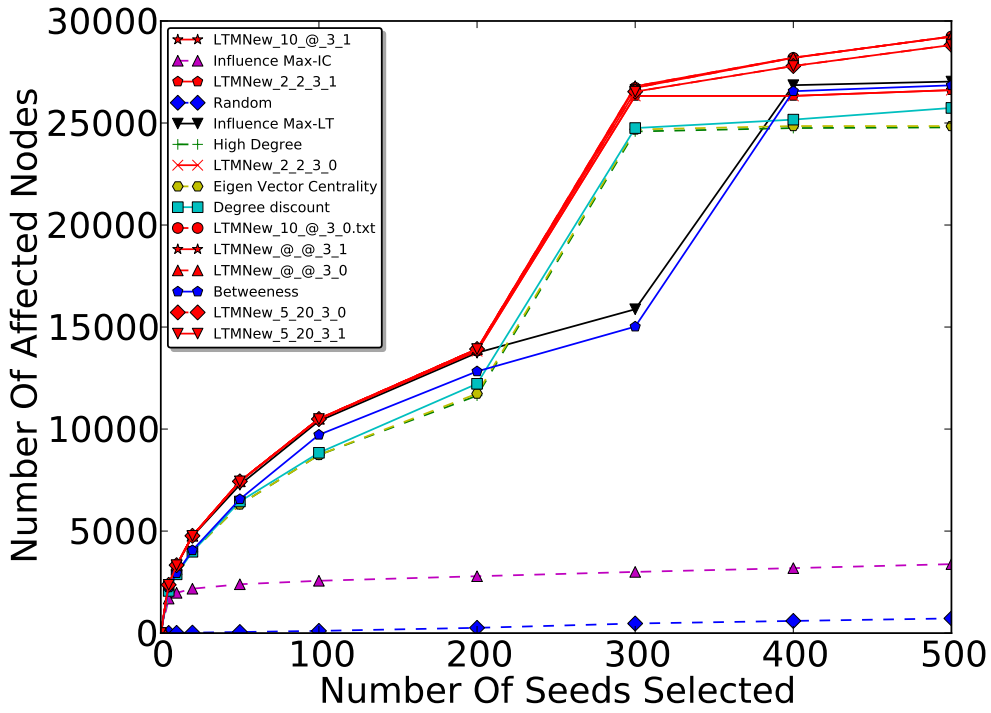


Figure A.21: Final outbreak Results for Enron graph with uniform threshold of 0.5

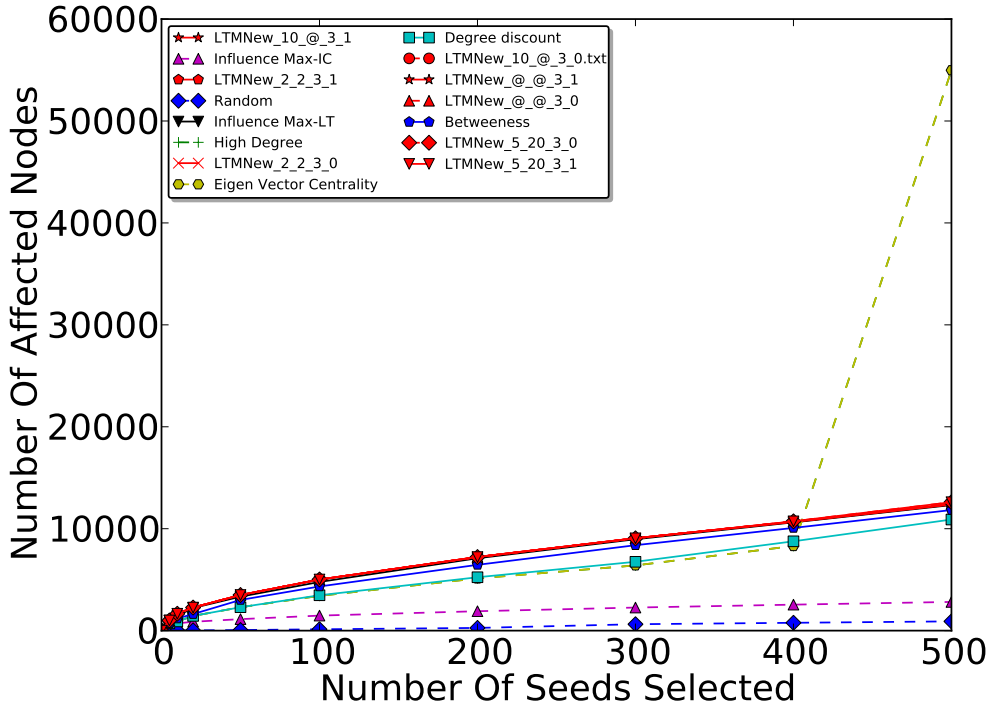


Figure A.22: Final outbreak Results for Epinion graph with uniform threshold of 0.5

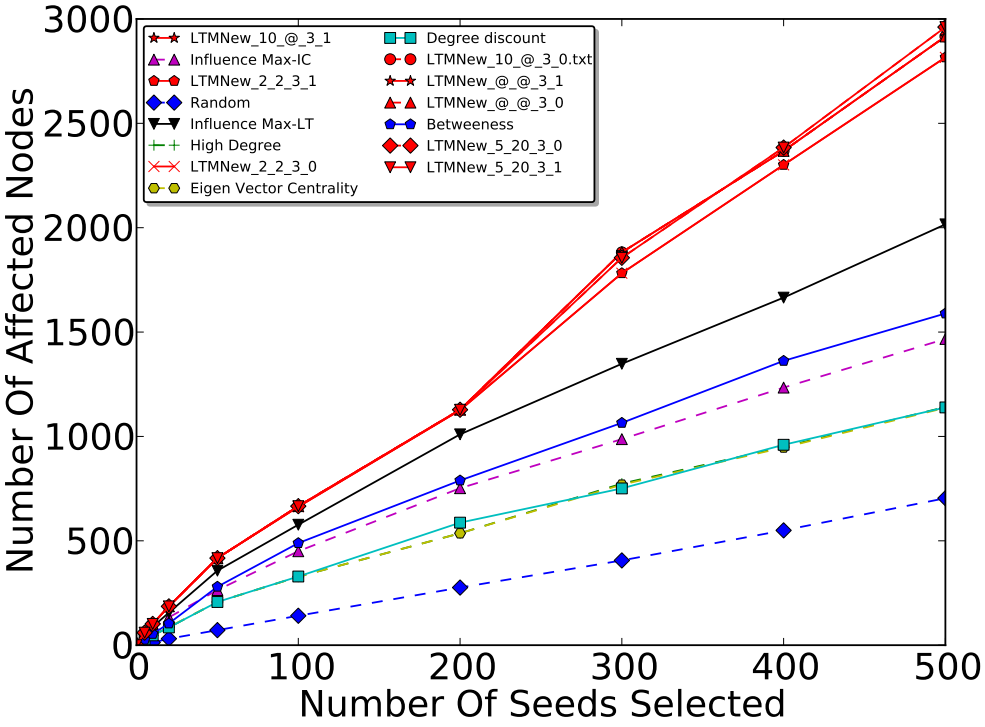


Figure A.23: Final outbreak Results for Facebook graph with uniform threshold of 0.5

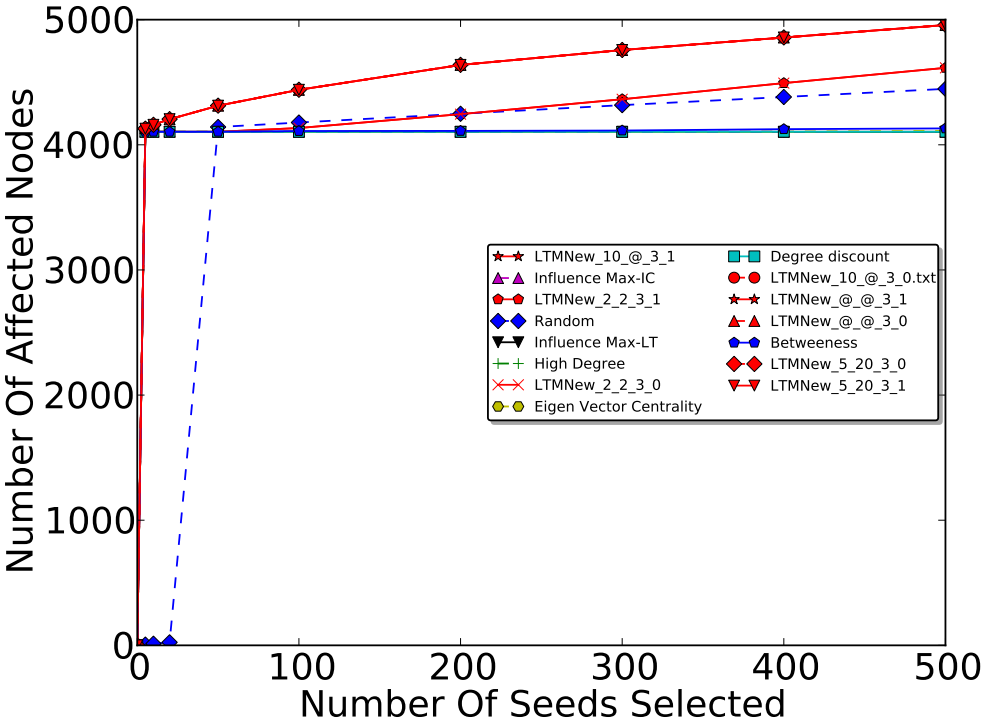


Figure A.24: Final outbreak Results for Fhs graph with uniform threshold of 0.5

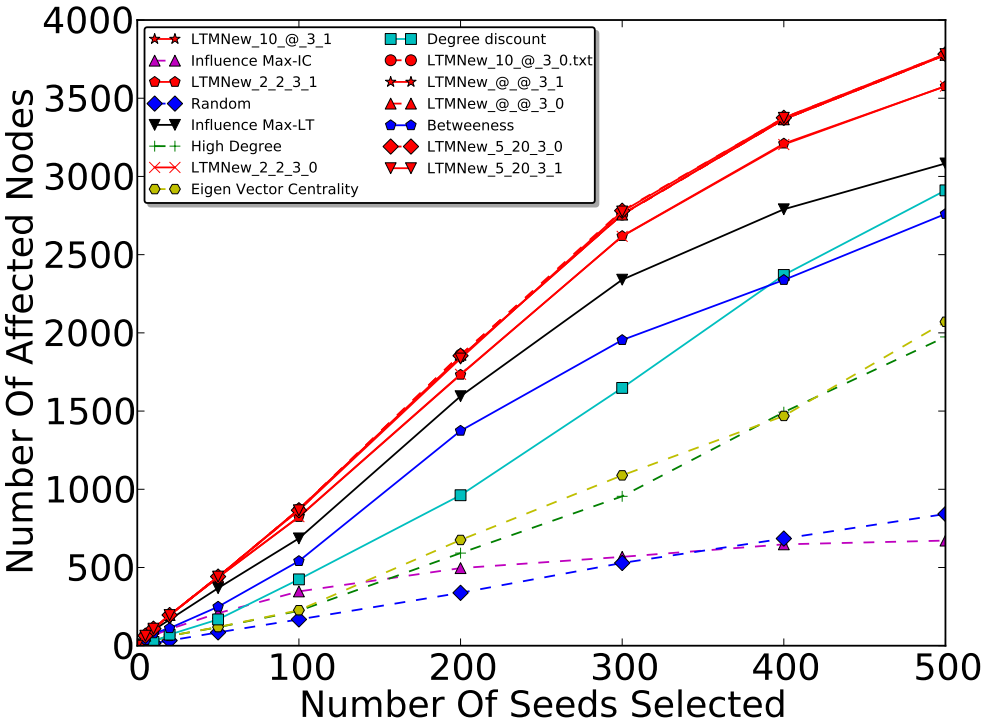


Figure A.25: Final outbreak Results for grqc graph with uniform threshold of 0.5

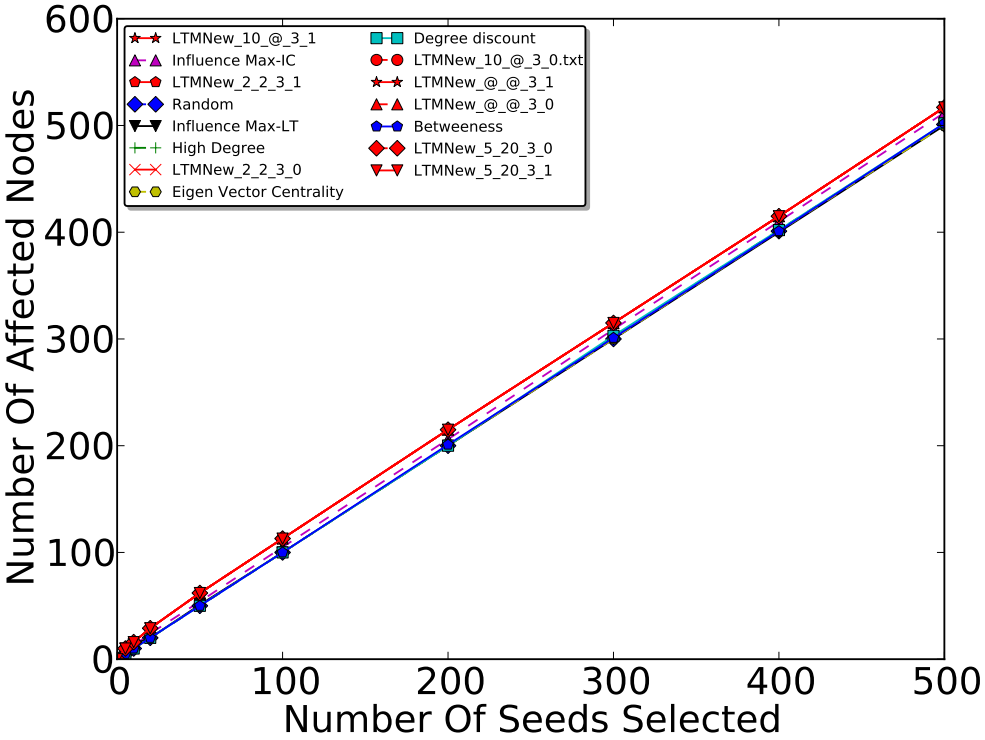


Figure A.26: Final outbreak Results for Maxplank social facebook graph with uniform threshold of 0.5

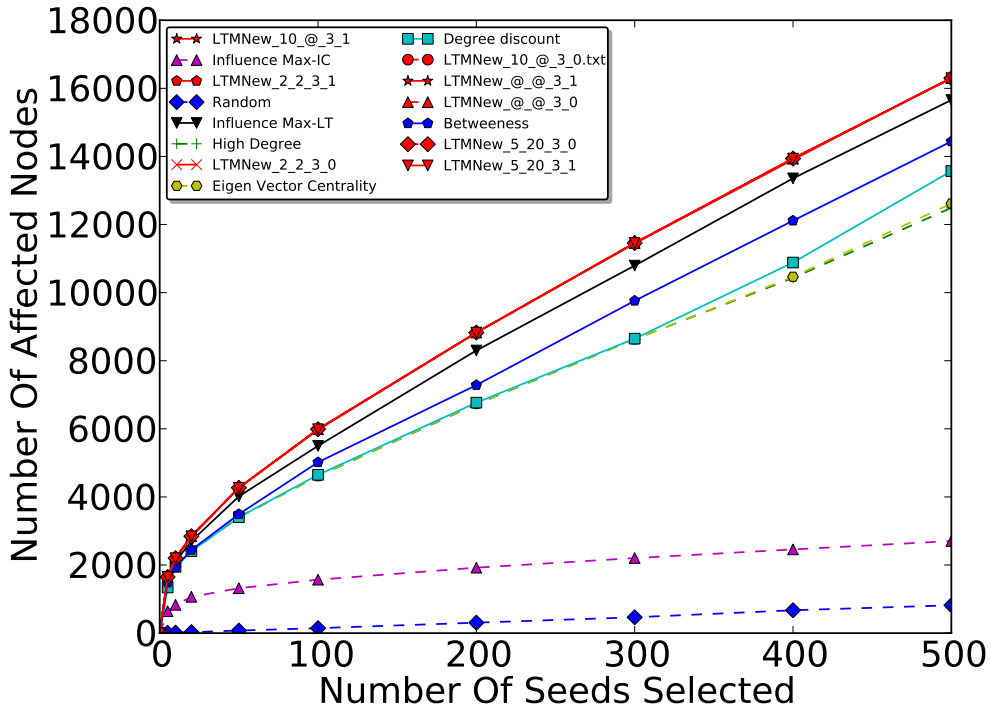


Figure A.27: Final outbreak Results for Slashdot graph with uniform threshold of 0.5

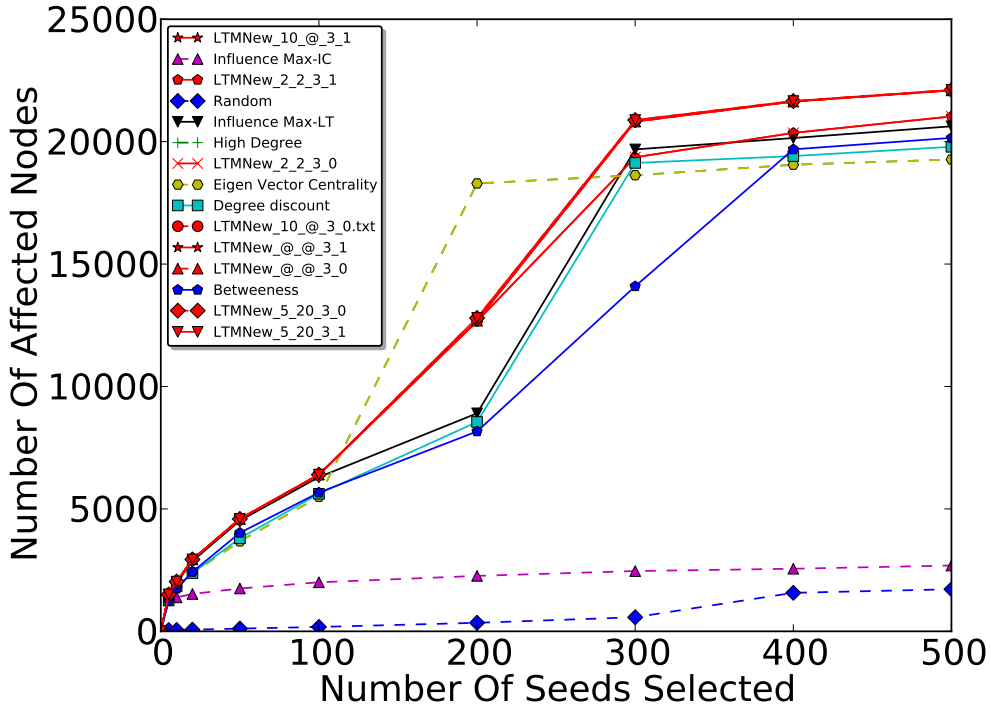


Figure A.28: Final outbreak Results for Twitter graph with uniform threshold of 0.5

Uniform threshold of 0.8

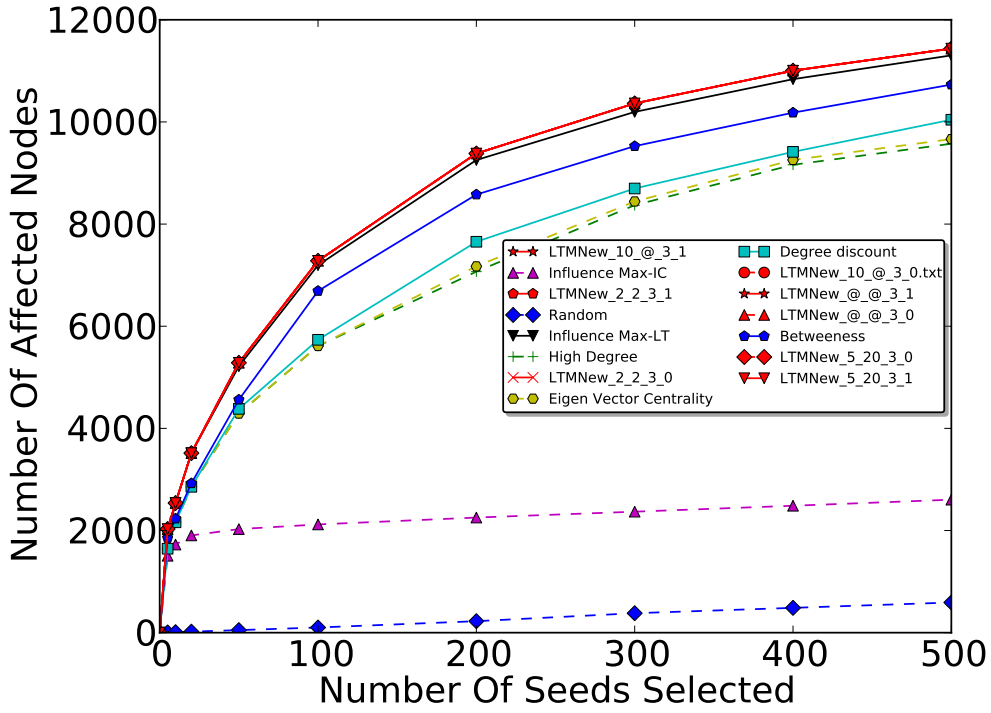


Figure A.29: Final outbreak Results for Enron graph with uniform threshold of 0.8

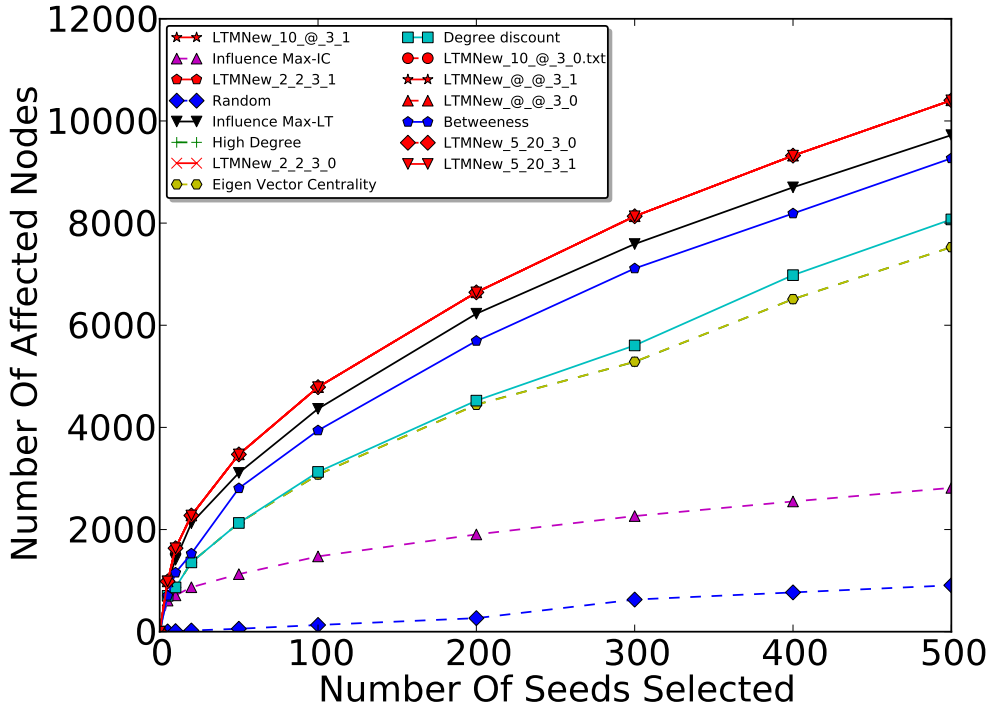


Figure A.30: Final outbreak Results for Epin graph with uniform threshold of 0.8

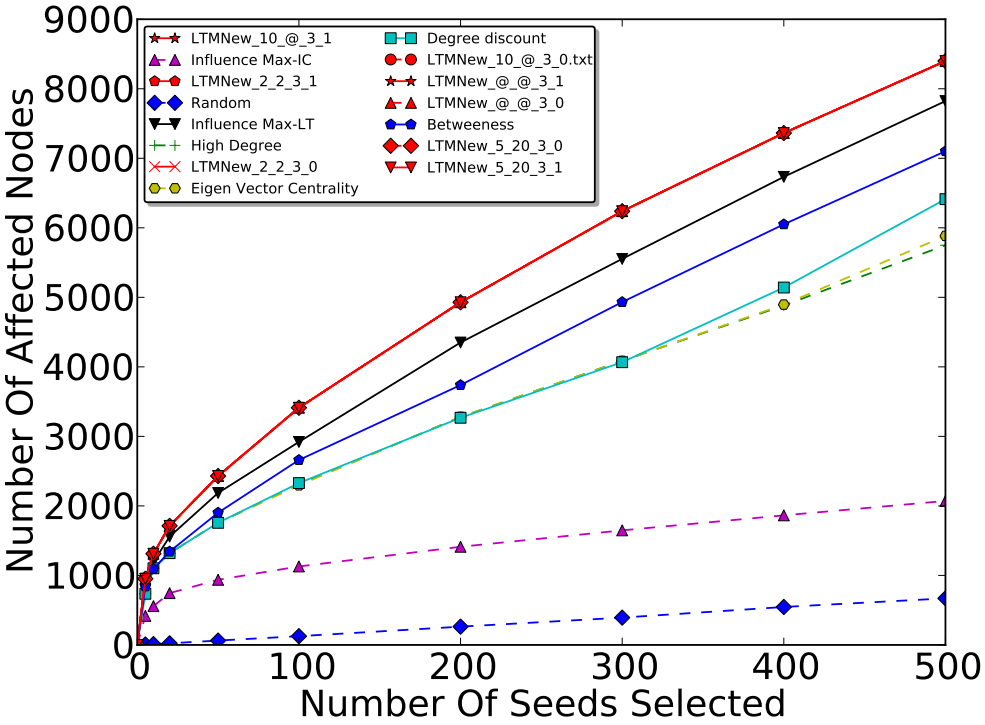


Figure A.31: Final outbreak Results for Slashdot graph with uniform threshold of 0.8

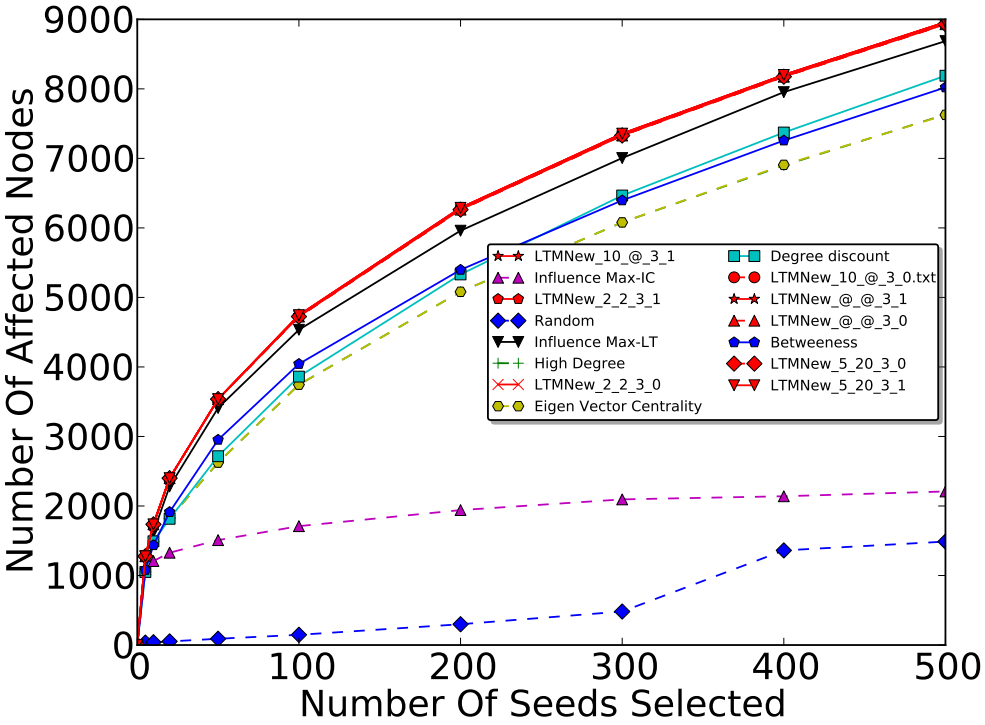


Figure A.32: Final outbreak Results for Twitter graph with uniform threshold of 0.8

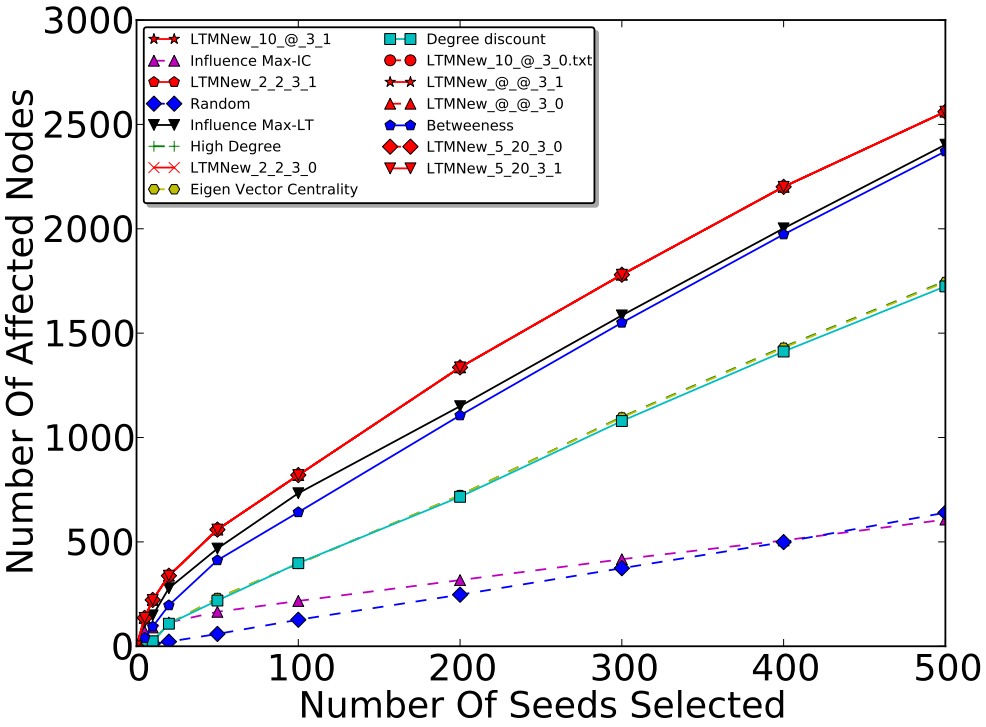


Figure A.33: Final outbreak Results for Wikipedia graph with uniform threshold of 0.8

Random threshold between 0.3 and 0.7

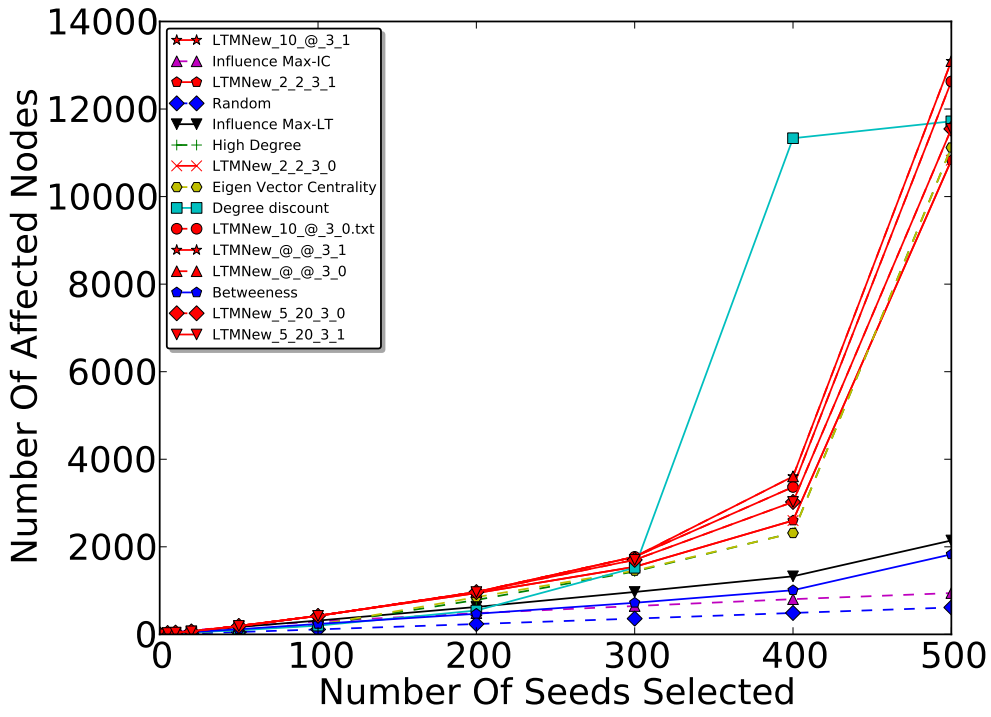


Figure A.34: Final outbreak Results for Astroph graph with random threshold between 0.3 and 0.7

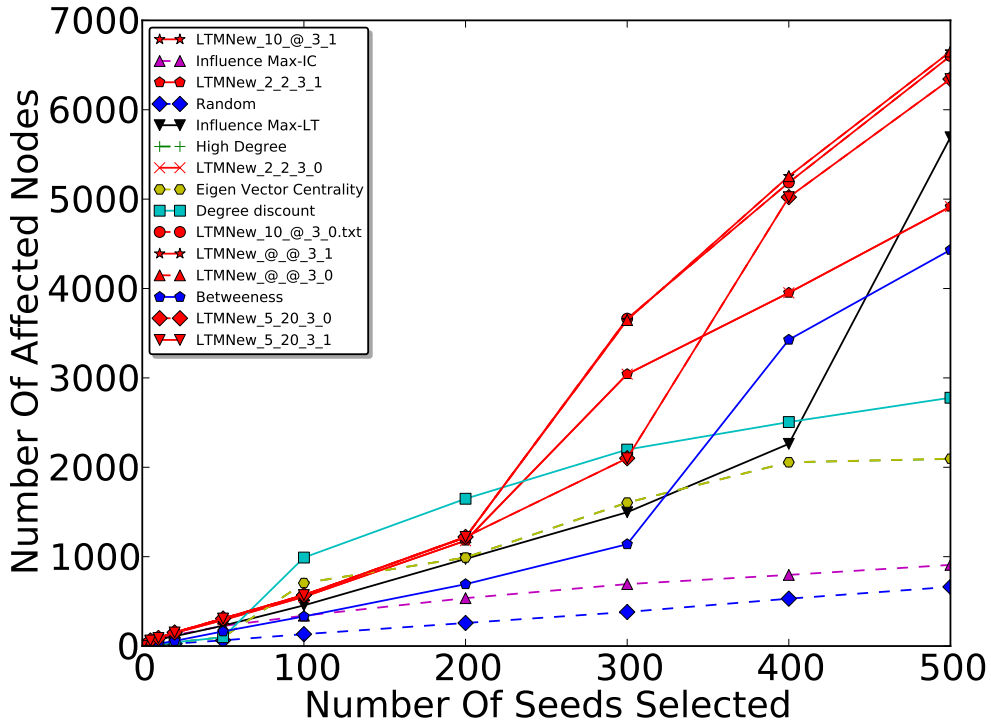


Figure A.35: Final outbreak Results for Ca-hepph graph with random threshold between 0.3 and 0.7

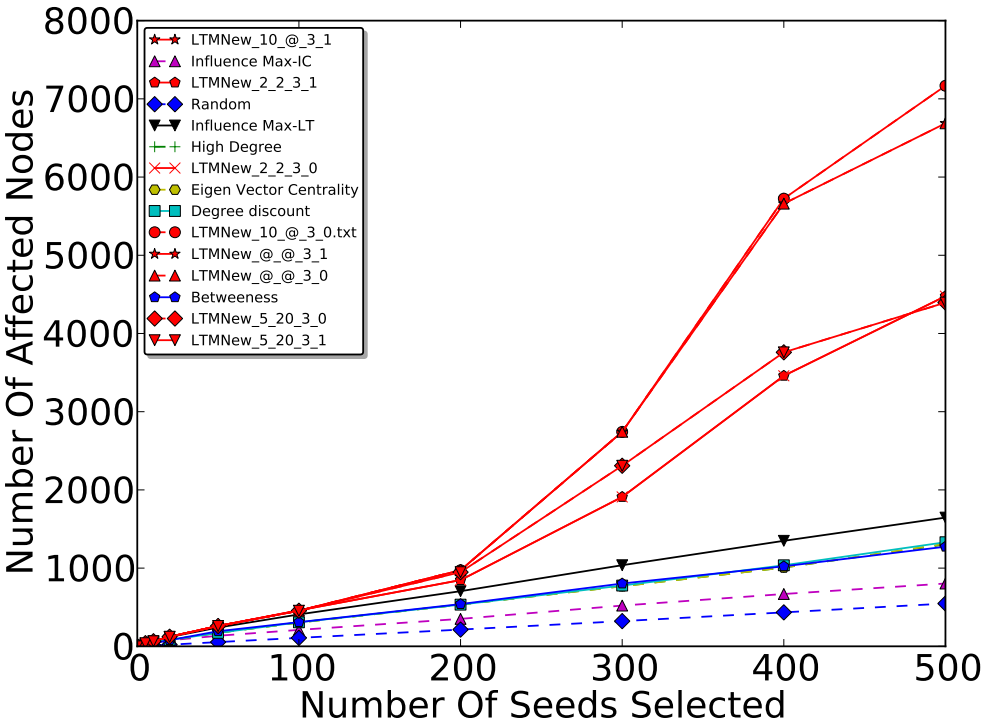


Figure A.36: Final outbreak Results for Cit-hepph graph with random threshold between 0.3 and 0.7

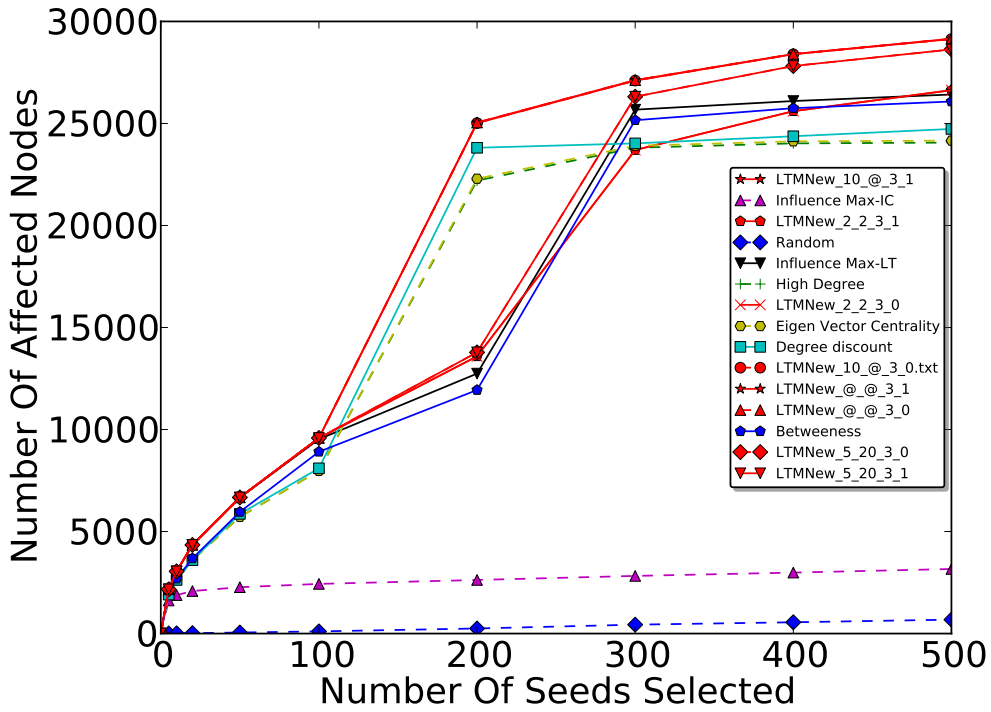


Figure A.37: Final outbreak Results for Enron graph with random threshold between 0.3 and 0.7

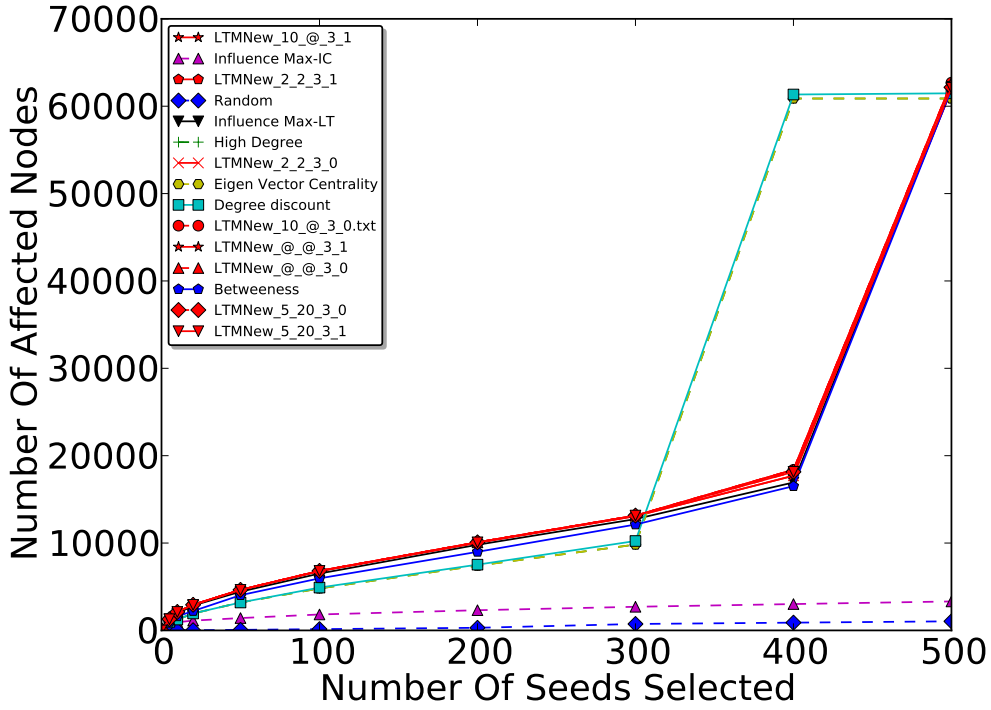


Figure A.38: Final outbreak Results for Epinion graph with random threshold between 0.3 and 0.7

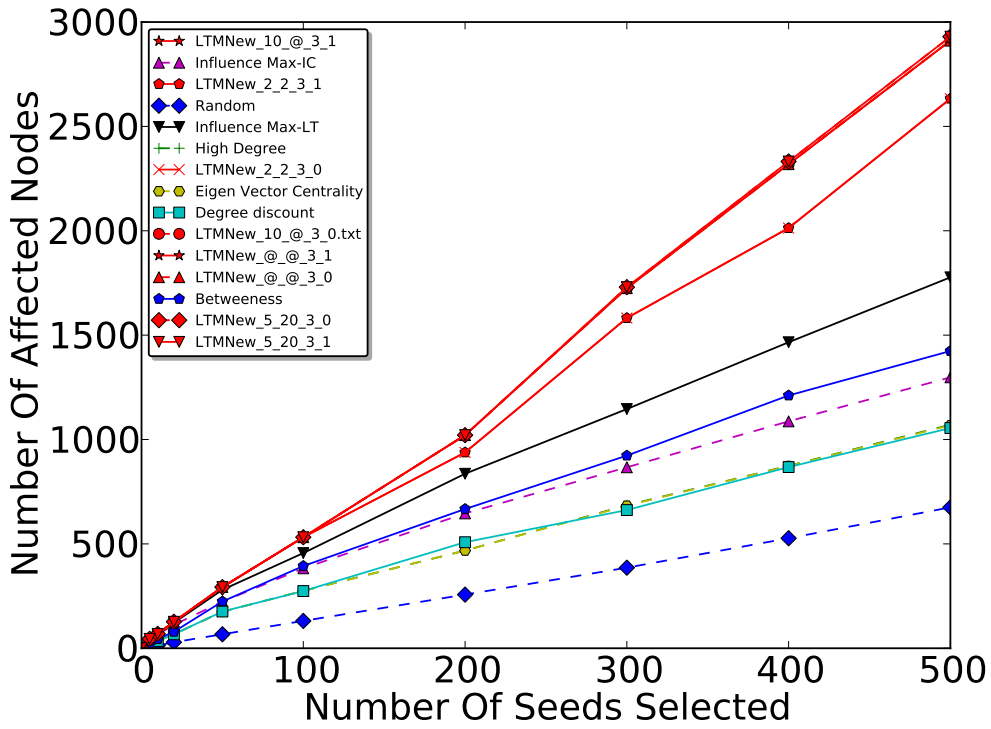


Figure A.39: Final outbreak Results for Facebook graph with random threshold between 0.3 and 0.7

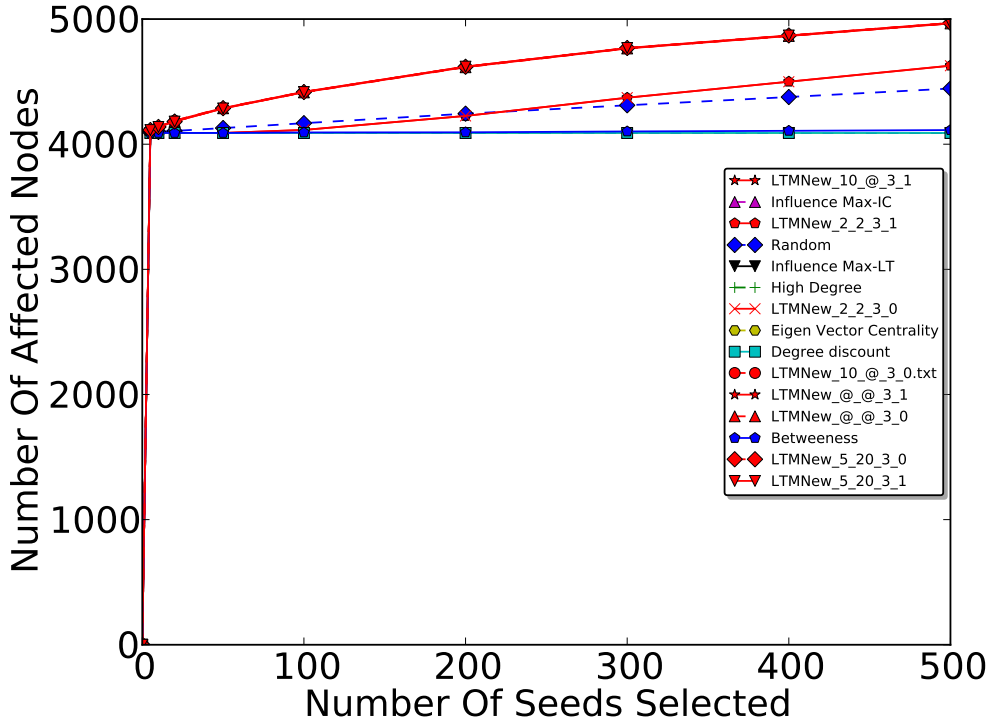


Figure A.40: Final outbreak Results for Fhs graph with random threshold between 0.3 and 0.7

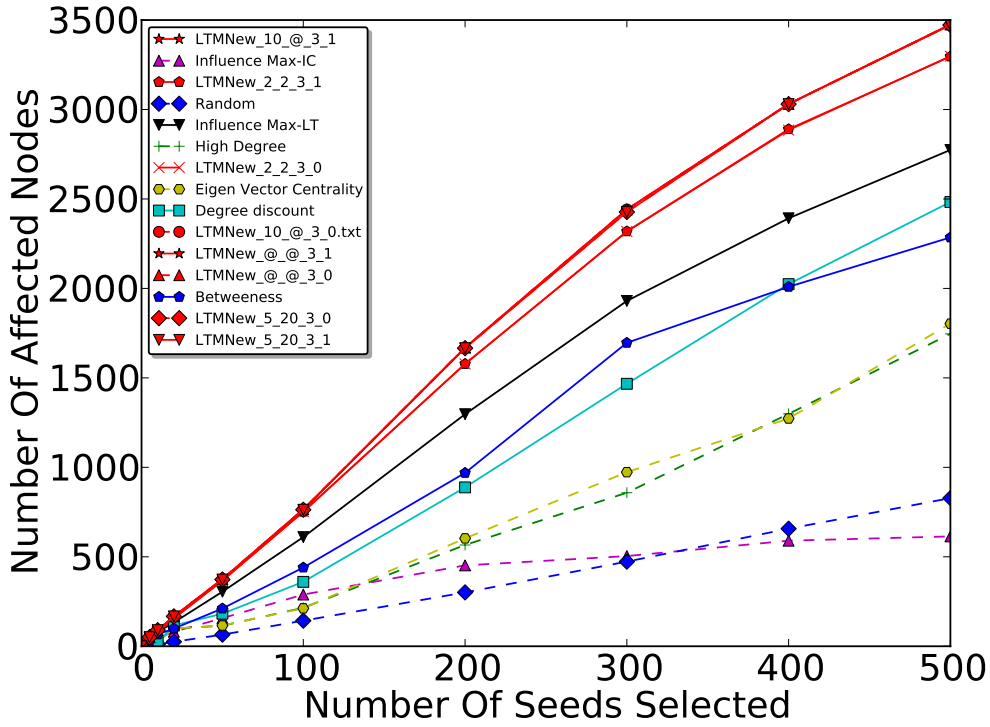


Figure A.41: Final outbreak Results for grqc graph with random threshold between 0.3 and 0.7

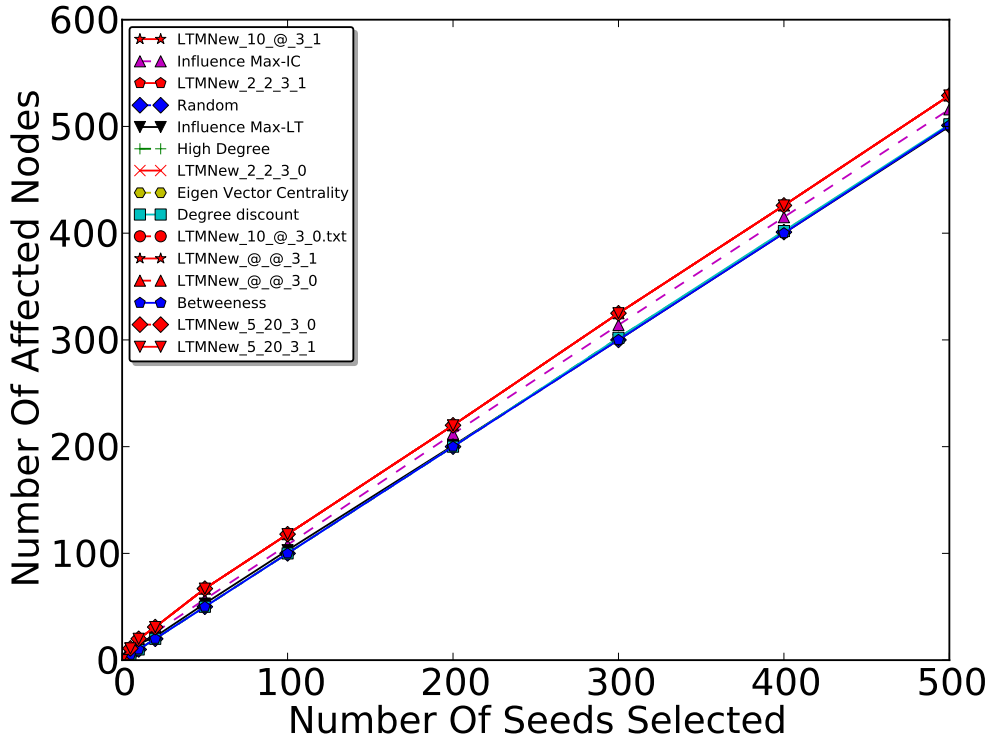


Figure A.42: Final outbreak Results for Maxplank social facebook graph with random threshold between 0.3 and 0.7

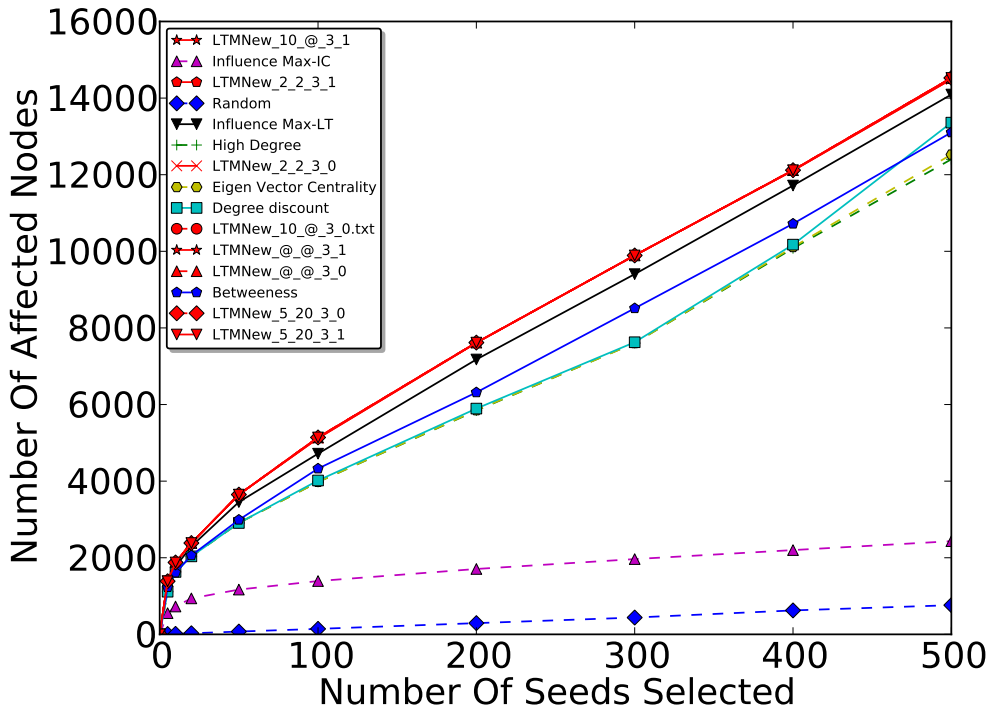


Figure A.43: Final outbreak Results for Slashdot graph with random threshold between 0.3 and 0.7

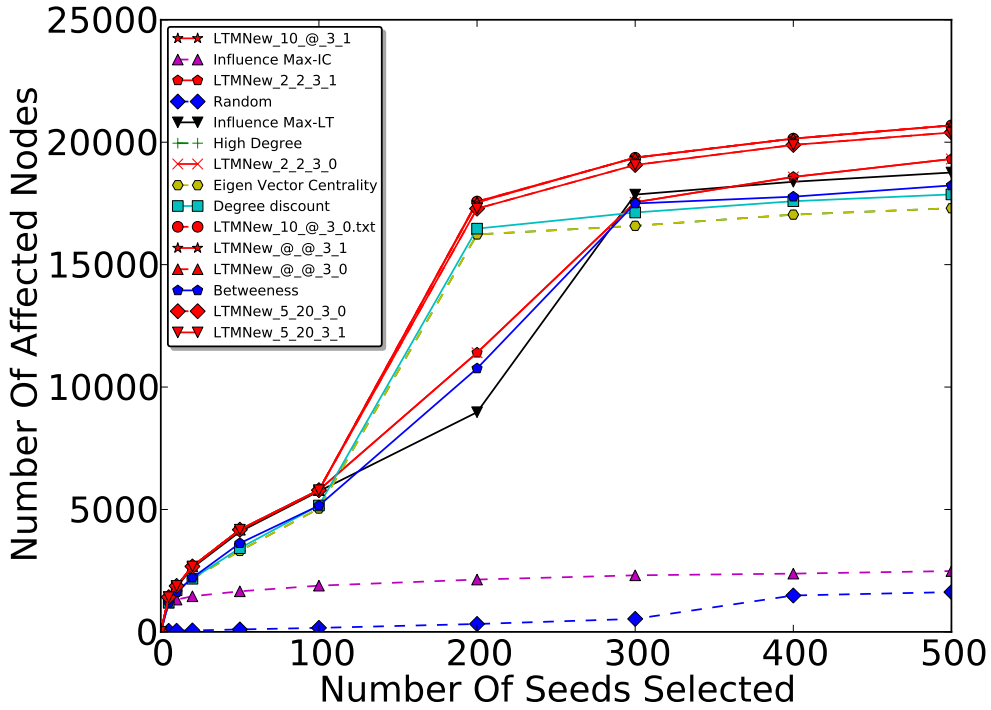


Figure A.44: Final outbreak Results for Twitter graph with random threshold between 0.3 and 0.7

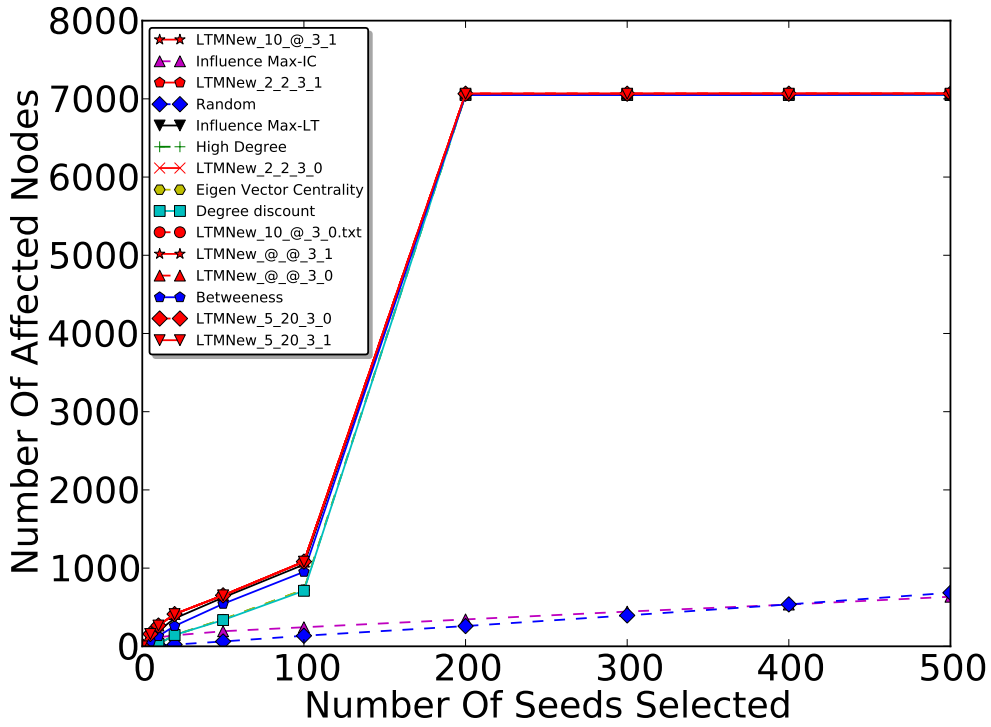


Figure A.45: Final outbreak Results for Wiki graph with random threshold between 0.3 and 0.7

Random threshold between 0.1 and 0.9

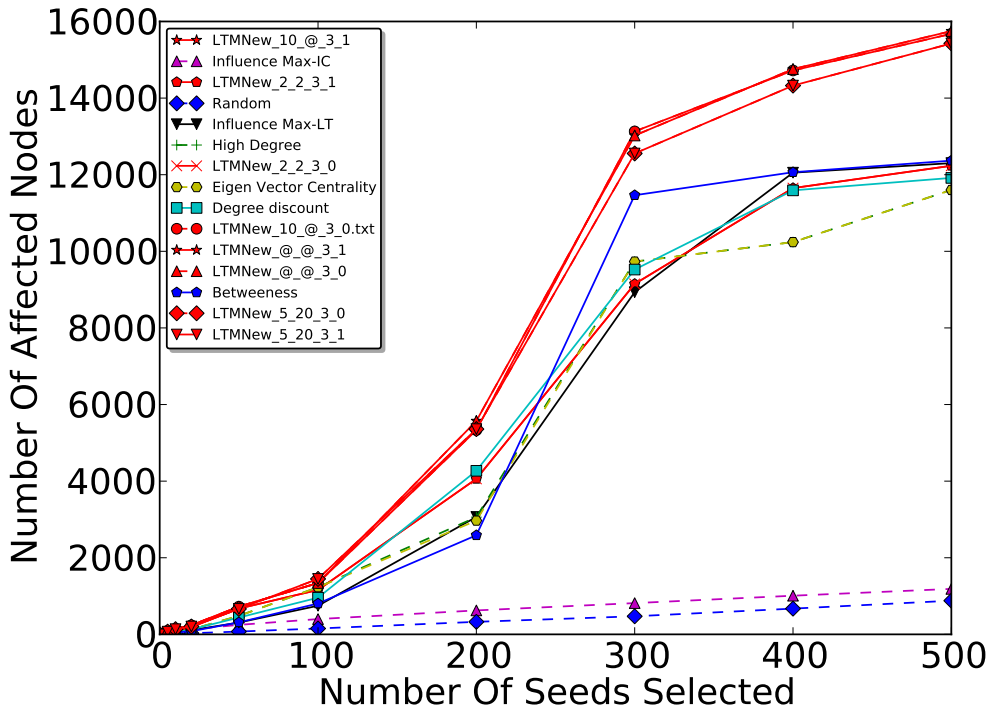


Figure A.46: Final outbreak Results for Astroph graph with random threshold between 0.1 and 0.9

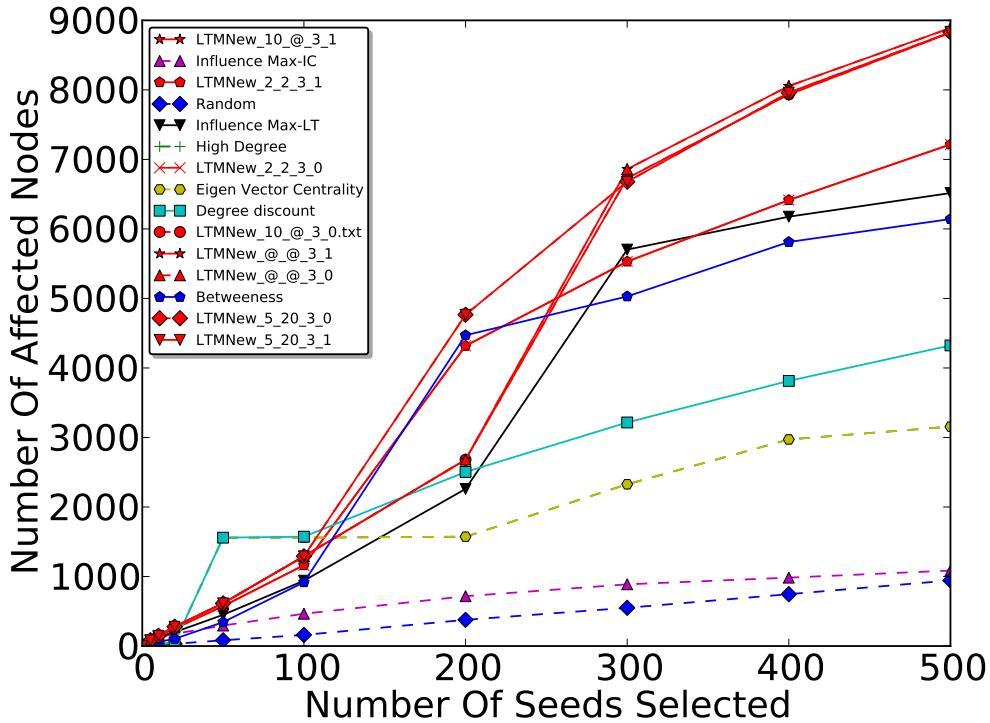


Figure A.47: Final outbreak Results for Ca-hepph graph with random threshold between 0.1 and 0.9

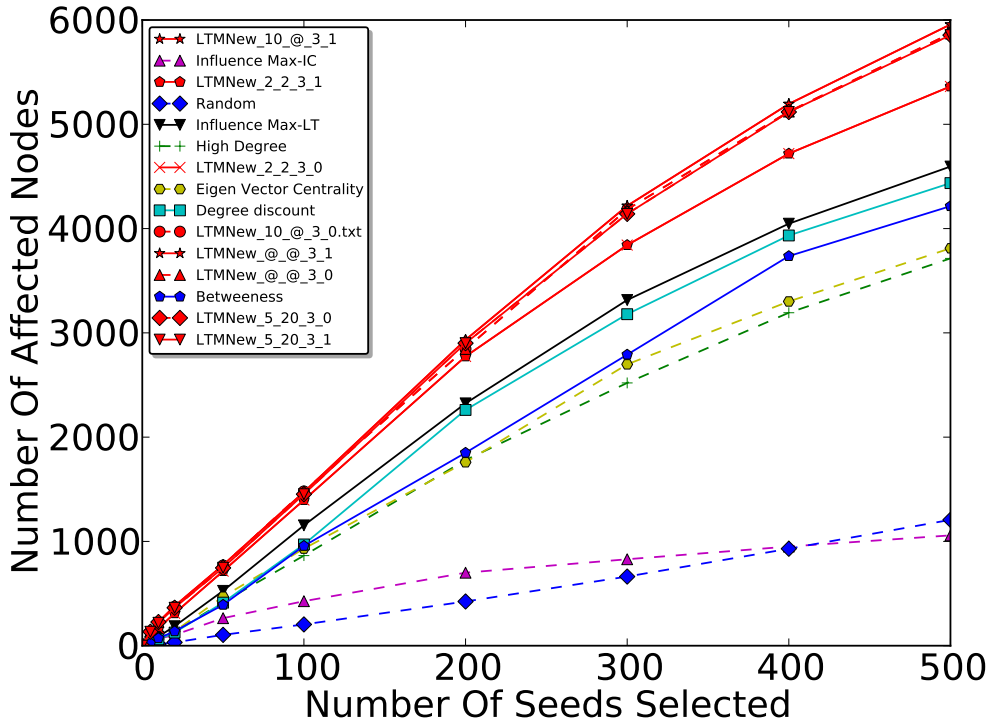


Figure A.48: Final outbreak Results for Ca-hepth graph with random threshold between 0.1 and 0.9

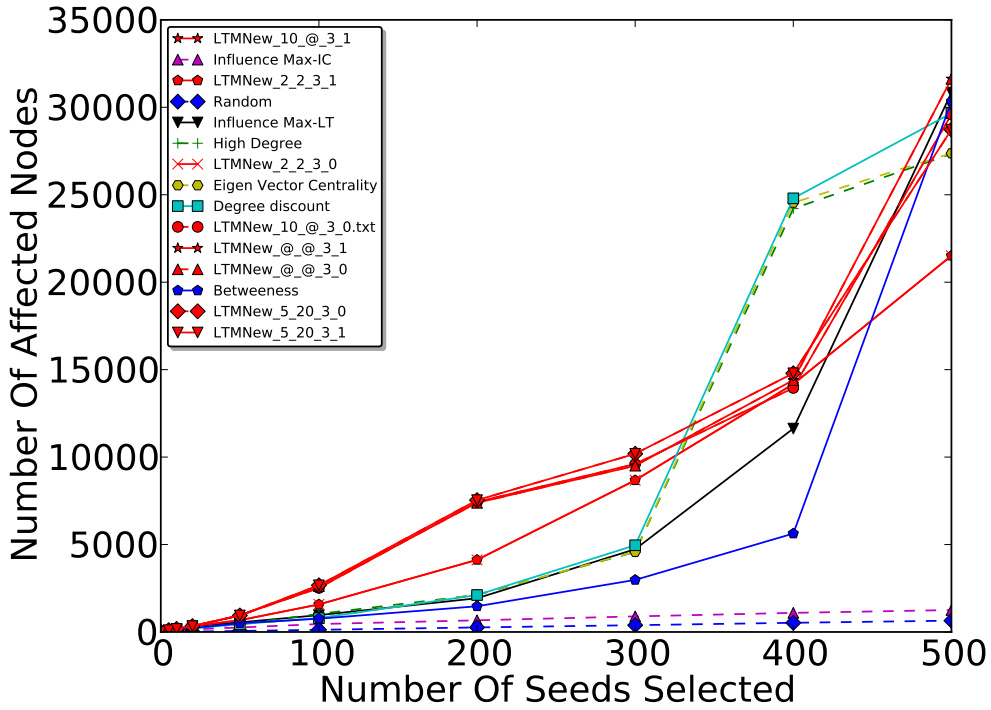


Figure A.49: Final outbreak Results for Cit-hepph graph with random threshold between 0.1 and 0.9

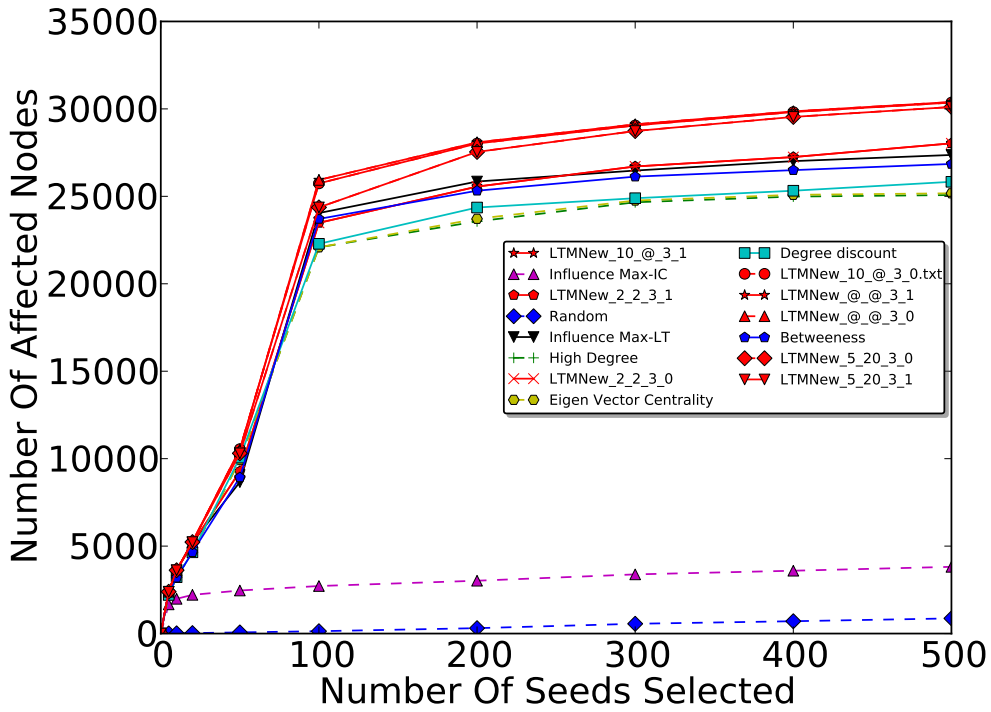


Figure A.50: Final outbreak Results for Enron graph with random threshold between 0.1 and 0.9

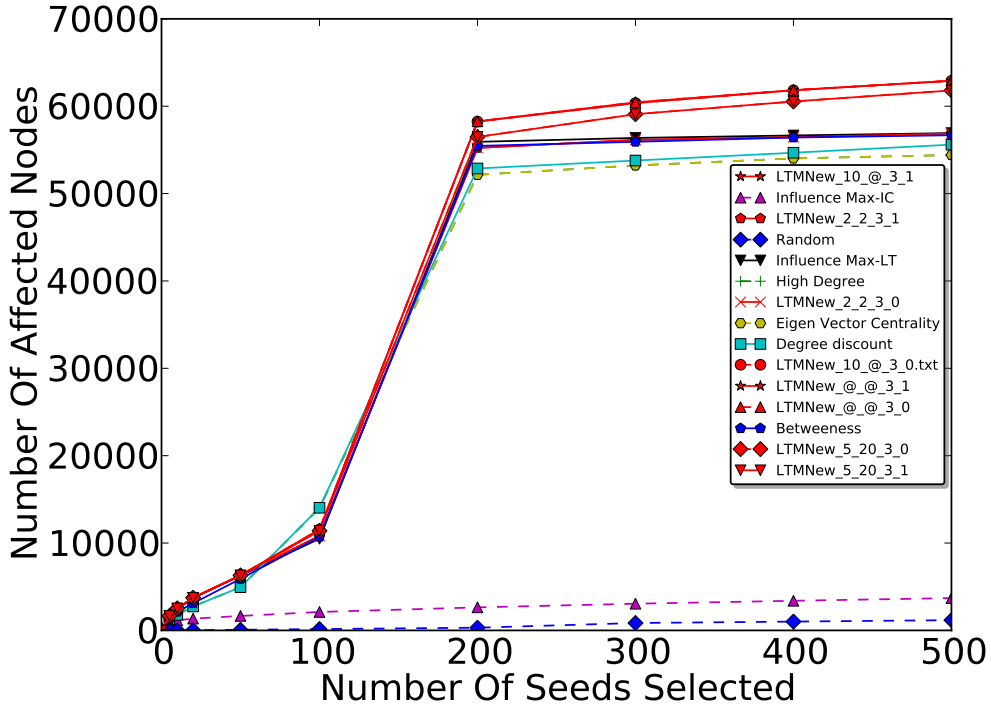


Figure A.51: Final outbreak Results for Epinion graph with random threshold between 0.1 and 0.9

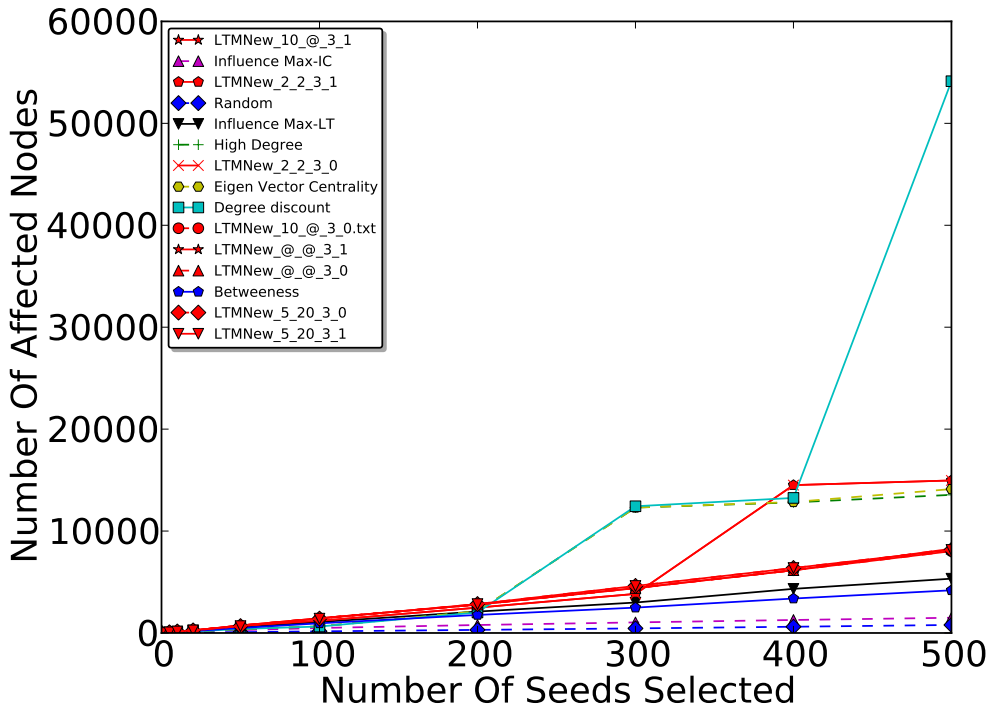


Figure A.52: Final outbreak Results for Facebook graph with random threshold between 0.1 and 0.9

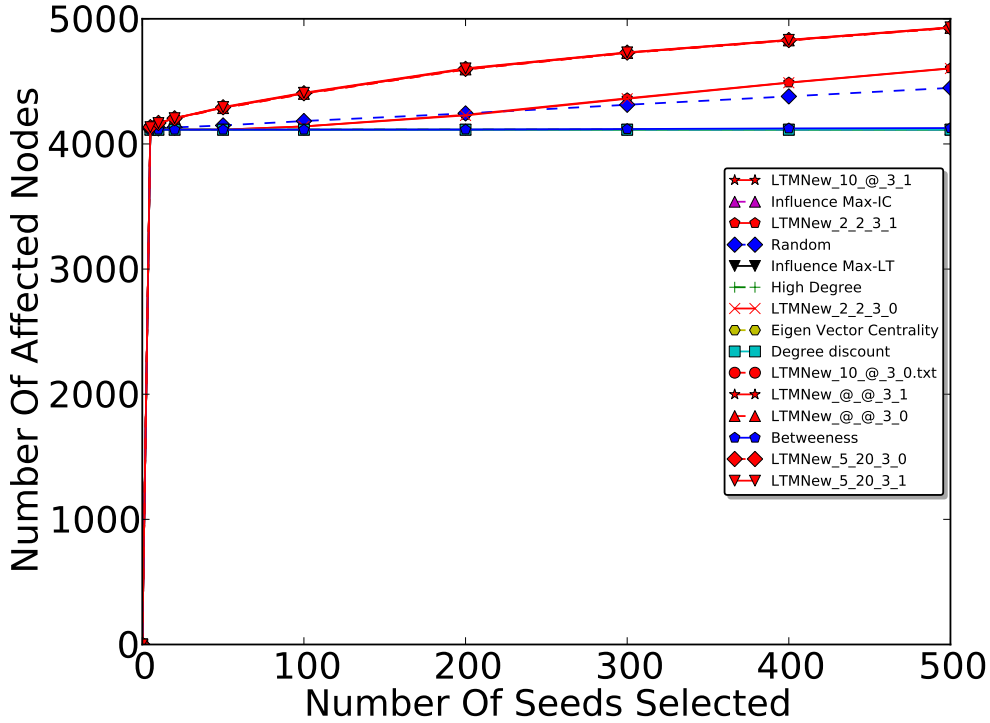


Figure A.53: Final outbreak Results for Fhs graph with random threshold between 0.1 and 0.9

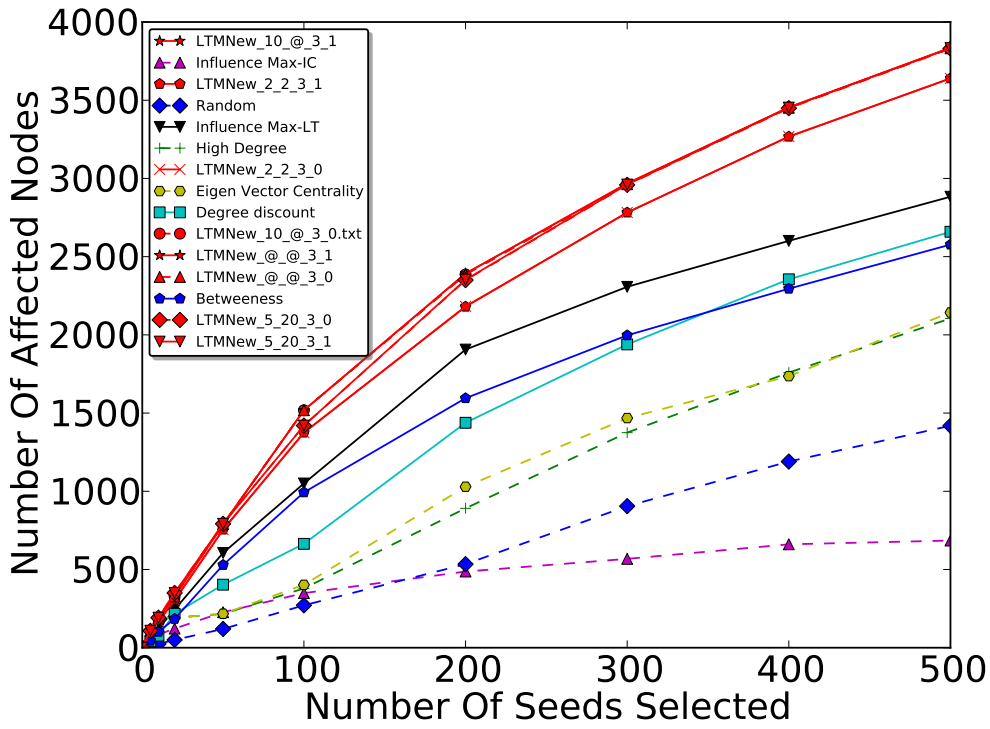


Figure A.54: Final outbreak Results for grqc graph with random threshold between 0.1 and 0.9

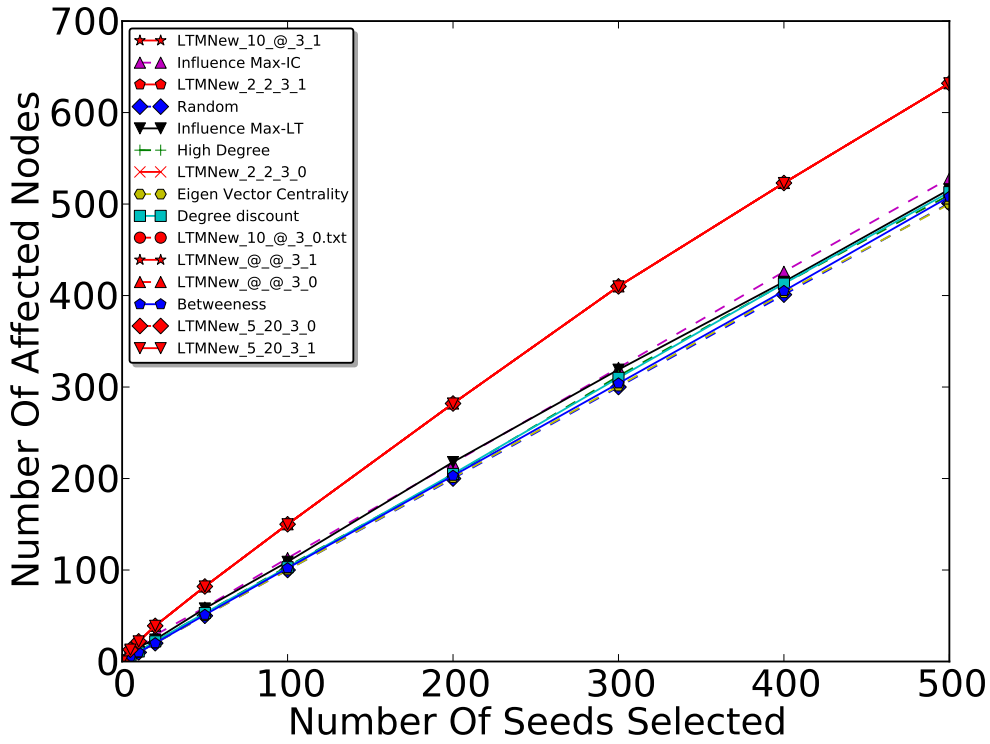


Figure A.55: Final outbreak Results for Maxplank social facebook graph with random threshold between 0.1 and 0.9

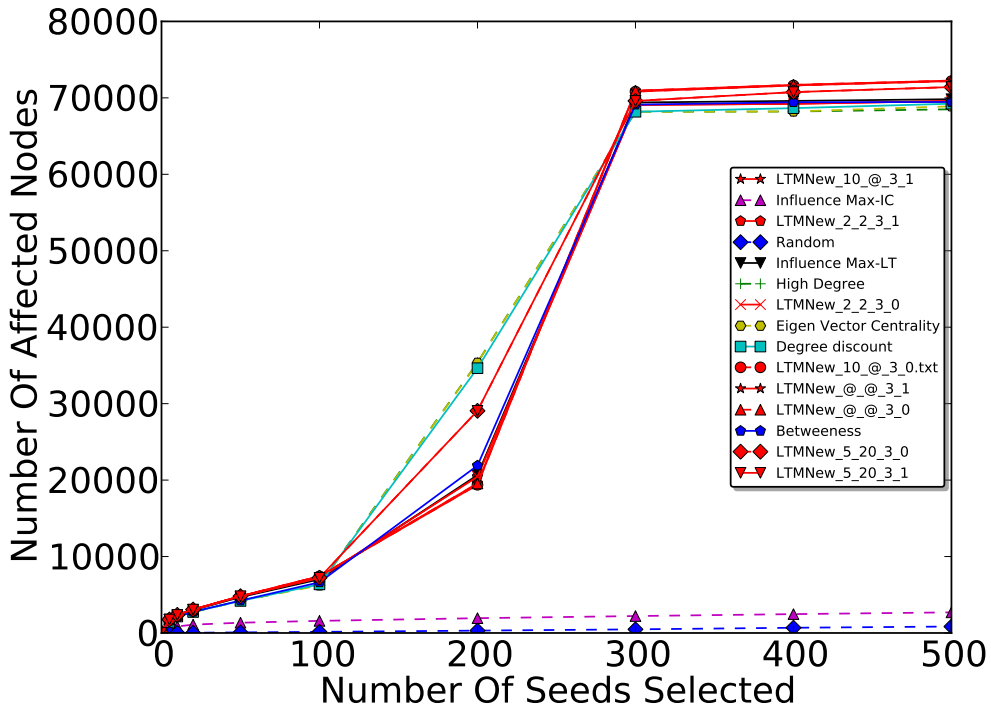


Figure A.56: Final outbreak Results for Slashdot graph with random threshold between 0.1 and 0.9

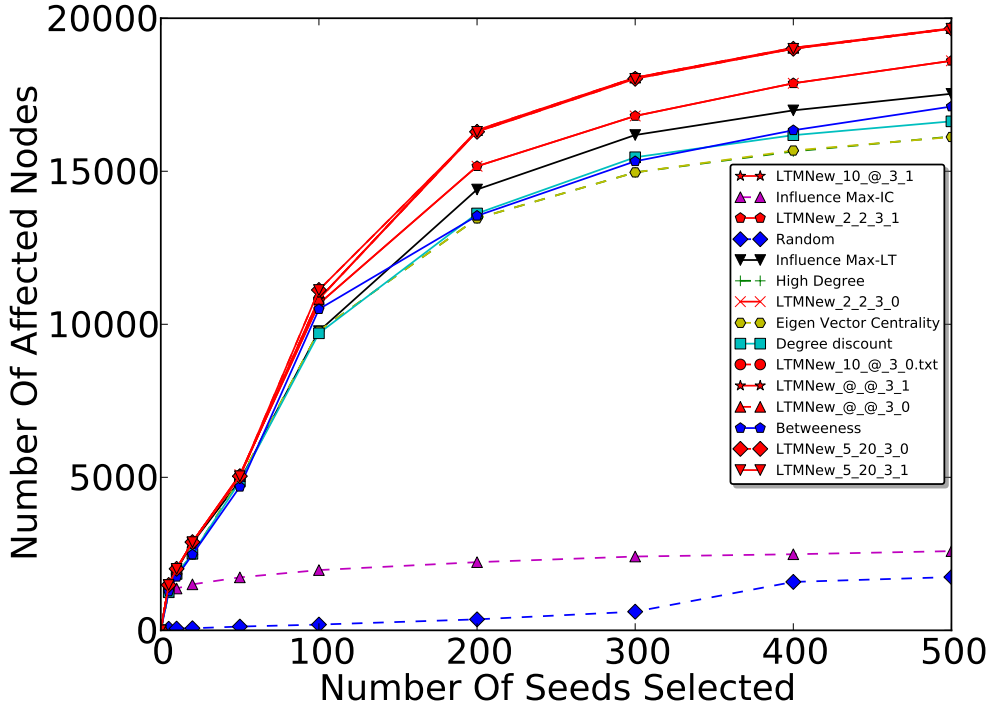


Figure A.57: Final outbreak Results for Twitter graph with random threshold between 0.1 and 0.9

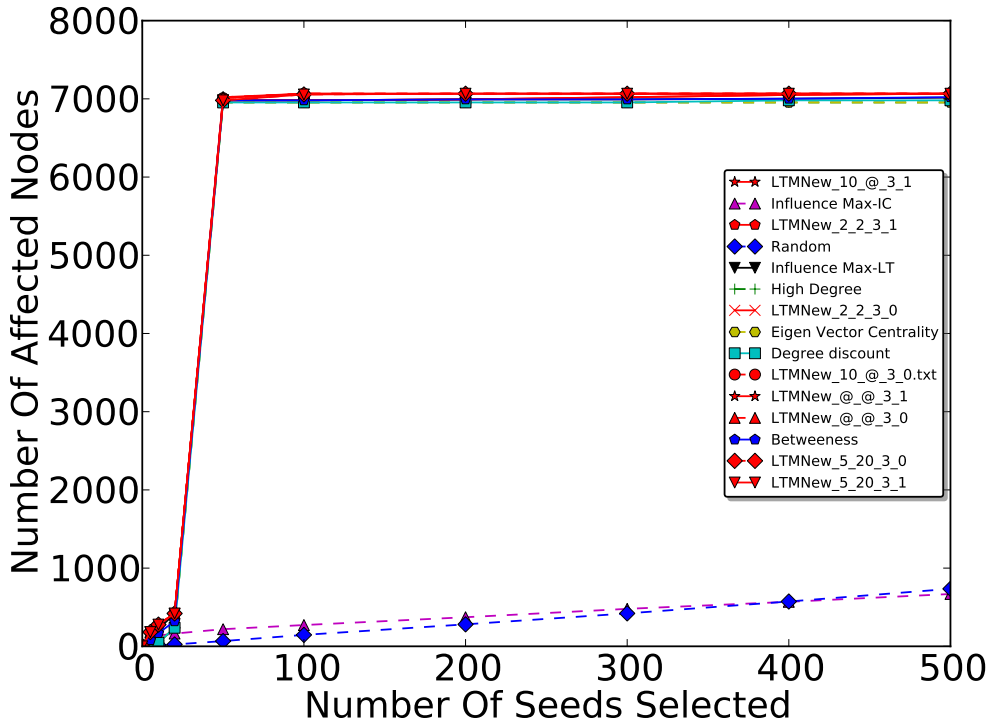


Figure A.58: Final outbreak Results for Wiki graph with random threshold between 0.1 and 0.9

Appendix B

Software

B.1 Graph input format

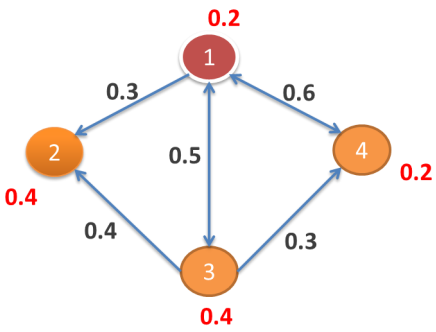


Figure B.1: Figure showing a graph with 4 nodes and associated edge weight and threshold values

The graph input for all the influence maximization algorithms must be in a specific format. The algorithms will only accept the graphs in following format. The format for file is

Line 1 : NodeId< *space* >NumberOfNeighbors<space>Threshold
Line 2 to Line NumberOfNeighbors+1 : <space><space>NodeId<space>EdgeWeight

The above format is repeated for all the nodes in the graph. The node id must be an integer(supported long integers) and the edge weights and thresholds are double decimal values. For example, for the graph shown in the figure B.1. The input graph file will be :
1<space>3<space>0.2
<space><space>2<space>0.3
<space><space>3<space>0.5
<space><space>4<space>0.6
2<space>0<space>0.4
3<space>3<space>0.4
<space><space>1<space>0.5
<space><space>2<space>0.4
<space><space>4<space>0.3
4<space>2<space>0.2
<space><space>1<space>0.6
<space><space>3<space>0.3

Note that each directed edge is provided as a input. In case of an undirected graph, two directed edges are assigned in opposite direction between two nodes.

B.2 Influence maximization by Chen et al. for independent cascade model, linear threshold model and threshold difference greedy algorithm

The influence maximization by Chen et al. for independent cascade model based on [58], linear threshold model based on [20] and the threshold difference greedy algorithm presented in this thesis are available in a single package.

Programming language: Java
Package jar path :
svn.vbi.vt.edu/svn/ndssl-collab/Papers/anand-work/Softwares/java/InfMax_Algorithm_Package.jar
Source code path:
svn.vbi.vt.edu/svn/ndssl-collab/Papers/anand-work/Softwares/java/MaxIndependentCasc
Execution Command :
java -jar InfMax_Algorithm_Package.jar Command.txt
Execution Command with memory specification :
java -Xms1000M -Xmx30000M -jar InfMax_Algorithm_Package.jar Command.txt

The InfMax_Algorithm_Package.jar is the name of the jar package available in the package jar path. The *Command.txt* is the name of the file which contains the set of statements that provides input for the execution. Each line in the file Command.txt corresponds to a separate execution. Each line in the Command.txt file contains a set of arguments separated by a <space>.

Argument 1: Name of graph file
Argument 2: Name of output file
Argument 3: Propagation probability limit
Argument 4: Number of seed nodes required
Argument 5: Type of Algorithm. Specify 1 for Influence maximization by Chen et al. for independent cascade model. Specify 2 for Influence maximization by Chen et al. for linear threshold model. Specify 3 for threshold difference greedy algorithm.
Argument 6: Debug mode On/Off. Specify 1 to set debug mode as On.
Arguement 7: Number of threads

Example for determining 500 seed nodes using influence maximization by Chen et al. for independent cascade model for cit-hepph graph with 0.5 as propagation probability limit, 16 threads :
inp-cit-hepph.giant.uel cit-hepph-opt.txt 0.5 500 1 0 16

Example for determining 500 seed nodes using influence maximization by Chen et al. for linear threshold model for cit-hepph graph with 0.5 as propagation probability limit, 16 threads :
inp-cit-hepph.giant.uel cit-hepph-opt.txt 0.5 500 1 0 16

The arguments below are only required if argument 5 is specified as 3, i.e the following arguments are required only for threshold difference greedy algorithm.

Argument 8: Value for d_1
Argument 9: Value for d_2
Argument 10: Value for d_3
Argument 11: Flag to take d_3 into account. Set this value as 1 to **not** consider value of d_3.
Argument 12 : Target set selection factor value.

The Argument 3 is not taken into consideration for the threshold difference greedy algorithm. Example for determining 500 seed nodes using threshold difference greedy algorithm for cit-hepph graph, 16 threads and values for parameters set as $d_1 = 10$, $d_2 = 100$, $d_3 = 3$,

flag to take d_3 into account not set and target set selection factor as 0.5:
inp-cit-hepph.giant.uel cit-hepph-opt.txt 0.5 500 1 0 16 10 100 3 0 0.5

B.3 Node Selection Using High Degree Heuristic, Degree Discount Heuristic, Betweenness Centrality, Eigen Vector Centrality, Random Heuristic

Programming Language : Python
Code Path : svn.vbi.vt.edu/svn/ndssl-collab/Papers/anand-work/Softwares/python/generate-seeds.py
Execution command :
python generate-seeds.py FileOrFolderName option seedCount

The FileOrFolderName is the path to a folder which contains the graphs. A single graph file can also be provided as input in FileOrFolderName. The option determines the heuristic needed to be executed. The option values are :

Betweenness centrality **1**
Eigen vector centrality **2**
High Degree heuristic **3**
Random heuristic **4**
Degree Discount heuristic **5**

The seedCount is the number of seed nodes to be produced. For example in order to find 200 seed nodes using the degree discount heuristic for all the graphs in folder input_graph, the execution command is :
python file.py input_graph 5 200