Scalable Algorithms for Misinformation Prevention in Social Networks

by

Michael Simpson

Bachelor of Science, University of Victoria, 2011
Master of Science, University of Victoria, 2014

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science



University of Victoria

Scalable Algorithms for Misinformation Prevention in Social Networks

by

Michael Simpson

Bachelor of Science, University of Victoria, 2011
Master of Science, University of Victoria, 2014

Supervisory Committee

---

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Dr. Michael Adams, Outside Member
(Department of Electrical and Computer Engineering)

**Supervisor Committee**

---

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Dr. Michael Adams, Outside Member
(Department of Electrical and Computer Engineering)

## ABSTRACT

This thesis investigates several problems in social network analysis on misinformation prevention with an emphasis on finding solutions that can scale to massive online networks. In particular, it considers two problem formulations related to the spread of misinformation in a network that cover the elimination of existing misinformation and the prevention of future dissemination of misinformation. Additionally, a comprehensive comparison of several algorithms for the feedback arc set (FAS) problem is presented in order to identify an approach that is both scalable and computes a lightweight solution. The feedback arc set problem is of particular interest since several notable problems in social network analysis, including the elimination of existing misinformation, crucially rely on computing a small FAS as a preliminary.

The elimination of existing misinformation is modelled as a graph searching game. The problem can be summarized as constructing a search strategy that will leave the graph clear of any misinformation at the end of the searching process in as few steps as possible. Despite the problem being NP-hard, even on directed acyclic graphs, this thesis presents an efficient approximation algorithm and provides new experimental results that compares the performance of the approximation algorithm to the lower bound on several large online networks. In particular, new scalability goals are achieved through careful algorithmic engineering and a highly optimized preprocessing step.

The minimum feedback arc set problem is an NP-hard problem on graphs that seeks a minimum set of arcs which, when removed from the graph, leave it acyclic.

A comprehensive comparison of several approximation algorithms for computing a minimum feedback arc set is presented with the goal of comparing the quality of the solutions and the running times. Additionally, careful algorithmic engineering is applied for multiple algorithms in order to improve their scalability. In particular, two approaches that are optimized (one greedy and one randomized) result in simultaneously strong performance for both feedback arc set size and running time. The experiments compare the performance of a wide range of algorithms on a broad selection of large online networks and reveal that the optimized greedy and randomized implementations outperform the other approaches by simultaneously computing a feedback arc set of competitive size and scaling to web-scale graphs with billions of vertices and tens of billions of arcs. Finally, the algorithms considered are extended to the probabilistic case in which arcs are realized with some fixed probability and a detailed experimental comparison is provided.

Finally, the problem of preventing the spread of misinformation propagating through a social network is considered. In this problem, a "bad" campaign starts propagating from a set of seed nodes in the network and the notion of a limiting (or "good") campaign is used to counteract the effect of misinformation. The goal is to identify a set of $k$ users that need to be convinced to adopt the limiting campaign so as to minimize the number of people that adopt the "bad" campaign at the end of both propagation processes. *RPS* (Reverse Prevention Sampling), an algorithm that provides a scalable solution to the misinformation prevention problem, is presented. The theoretical analysis shows that *RPS* runs in $O((k+l)(n+m)(\frac{1}{1-\gamma})\log n/\epsilon^2)$ expected time and returns a $(1-1/e-\epsilon)$-approximate solution with at least $1-n^{-l}$ probability (where $\gamma$ is a typically small network parameter). The time complexity of *RPS* substantially improves upon the previously best-known algorithms that run in time $\Omega(mnk \cdot POLY(\epsilon^{-1}))$. Additionally, an experimental evaluation of *RPS* on large datasets is presented where it is shown that *RPS* outperforms the state-of-the-art solution by several orders of magnitude in terms of running time. This demonstrates that misinformation prevention can be made practical while still offering strong theoretical guarantees.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgements

It has been my privilege to work closely with two outstanding mentors. I am very proud of what we have achieved together; thank you both. This dissertation would not have been possible without the support of my loving family.

# Chapter 1

# Introduction

Social networks allow for the widespread distribution of knowledge and information in modern society as they are rapidly becoming a place for people to hear the news and discuss social topics. Information can spread quickly through the network eventually reaching a large audience, especially so for influential users. However, while the ease of information propagation in social networks can be beneficial, it can also have disruptive effects. In recent years, the number of high profile instances of misinformation causing severe real-world effects has risen sharply. These examples range across a number of social media platforms and topics. A series of bogus tweets from a trusted news network referring to explosions at the White House caused immediate and extensive repercussions in the financial markets [35]. During a recent shooting at YouTube's headquarters, and before police managed to secure the area, a wave of misinformation and erroneous accusations were widely disseminated on Twitter causing panic and confusion [74, 40]. Finally, there has been much discussion on the role misinformation and fake news played in the 2016 U.S. presidential election with sites such as Reddit and Facebook being accused of harbouring and spreading divisive content and misinformation [43, 36, 1]. Thus, in order for social networks to serve as a reliable platform for disseminating critical information, it is necessary to

have tools to limit the spread of misinformation. Furthermore, contamination in a network may refer to several scenarios including information propagating through a social network, malware spreading through an online network, or sickness spreading though a population. Since a large fraction of the population today learn of news or events online it is also important to have tools to eliminate, not just minimize, the effects of misinformation. Below we give an overview of the contents for the chapters that make up the thesis.

**Chapter 2.** Previous work related to misinformation in social networks has focused on the task of limiting the spread of misinformation [19, 67, 8, 20] while we study a related problem that seeks to eliminate misinformation, or any kind of contamination, already present in a general network. For a contaminated network, we model the problem in the context of graph searching; a classical game on graphs [75, 16, 26]. In the graph searching game we may think of a network whose edges are contaminated with a gas and the objective is to clean the network with some number of searchers. However, the gas immediately recontaminates cleared edges if its spreading is not blocked by guards at the vertices. The model does not assume knowledge of the location of the gas, yet guarantees its elimination at the end of the search strategy, and assumes an edge is deterministically contaminated, as opposed to probabilistically, which represents the case of a powerful adversary.

In the pioneering work of Brandenburg and Herrmann [18] the dual to the well studied *search number* (the minimum number of searchers required to clear a graph), search time, was introduced as a new cost measure in graph searching. Naturally, we believe it is more important to clear the network as quickly as possible when dealing with a contaminant. Furthermore, until now the theory community has mainly focused on the search number of an undirected graph, but one needs to study the more general case of directed graphs as many real world networks lend themselves to be

modelled as directed.

We study the problem of minimizing the time required to eliminate the contamination in the network given a budget of searchers. We prove that the search time problem is NP-complete even for directed acyclic graphs (DAGs) and introduce an approximation algorithm for clearing DAGs. Furthermore, we propose a method for clearing a network by first reducing it to a DAG which can be cleared by our approximation algorithm. Additionally, we investigate the merits of a split and conquer style strategy and show that our strategy, which instead has searchers staying together as a group, outperforms the (intuitively appealing) split and conquer strategy on a broad class of DAGs. Along the way we prove lower bounds on the time required to search a directed graph and introduce a novel DAG decomposition theorem.

We note that the study of search time is intrinsically more difficult than computing the search number as we can no longer be *strategy oblivious*. That is, when studying the search number, one is only interested in knowing whether some search strategy exists to clear a graph with some number of searchers and thus can be solved through structural properties alone. In contrast, trying to compute the search time of a graph is closely tied to how the strategy actually plays out. The foundation of our approximation algorithm is a modified depth-first search which utilizes a novel stopping condition that allows us to compute strategies that do not allow for any recontamination of edges. Our algorithm produces an edge ordering based on which we construct strategies for an arbitrary number of searchers by partitioning the resulting ordering.

**Chapter 3.** Numerous studies have been carried out on social networks regarding the clearing or limiting of misinformation (cf. [19, 44, 69, 83]). For instance, [83] presents an approximation algorithm for clearing misinformation from a social network that models the network as a general directed graph and hinges on the ability

of making the input graph acyclic first by placing guards on a subset of arcs. Here, it is important to remove as few arcs as possible since such arcs need to be guarded throughout the clearing process. In another perspective on combating misinformation, [19] considers the problem of influence limitation of a bad campaign by selecting a set of seeds from where to start a good (limiting) campaign. This involves determining the eventual influence of the nodes already infected by the bad campaign. Unfortunately, computing influence is #P-hard. Nonetheless, if the network is a directed acyclic graph (DAG), then as shown by [23], computing influence is linear-time solvable. Here as well, placing guards on a subset of arcs, in effect making the graph acyclic, allows for scalable solutions.

Another prominent area of study in Social Network Analysis that can benefit from the elimination of cycles is label propagation [38]. The goal of label propagation is to consider the labels held by a subset of users in the network and attempt to predict the labels of the remaining users. This prediction is useful in determining, for example, the political leanings of users in the network or promising pairings on a dating website. The label propagation technique of choice is Belief Propagation (BP) [77] and it has also been successfully applied in fraud and malware detection. The BP algorithm propagates the information from a small subset of explicitly labeled users throughout the graph by iteratively passing labels between neighbouring users. Unfortunately, BP has known convergence problems in graphs with cycles [38, 82] and the exact criteria for convergence is not known [68]. As a result, alternative avenues to applying BP directly have been explored in which an acyclic graph is first obtained from the original network [4, 9].

The need to obtain an acyclic graph arises in many other applications. Early interest in this problem relates to the ability to simplify the analysis of large-scale feedback systems such as manufacturing processes. Furthermore, the problem has

been studied in the context of online ranking problems (where computing a minimal feedback arc set corresponds to asking for the fewest number of violations against the ranking imposed by the arcs), determining ancestry relations, and certain scheduling problems [34, 47, 79]. Finally, it also plays a prominent role in graph drawing [85].

Thus, for all these areas of research, having the ability to produce acyclic graphs that very closely represent large networks while minimizing the perturbation to the overall structure of the network would be highly advantageous. This goal can be accomplished through the computation of an appropriate *feedback arc set* (FAS).

A feedback arc set of a directed graph $G$ is a subset of arcs $F$ of $G$ such that removing $F$ from $G$ leaves an acyclic graph. Equivalently, a feedback arc set contains at least one arc from each cycle in $G$. The *minimum feedback arc set problem*, a widely studied combinatorial optimization problem on graphs, seeks to minimize the size of the subset of arcs $F$. The decision version of the FAS problem is shown to be NP-complete in Karp's seminal paper [49]. The problem remains NP-hard on several subclasses of graphs including tournaments [2] and Eulerian graphs [78]. Investigation into approximation algorithms for the FAS problem [24, 32, 53] in recent years has led to steady improvements; unfortunately, the approaches that yield good approximation guarantees are unable to scale due to unfavorable running time complexities.

Present day research continues to analyze datasets of increasing size. This requires algorithms for computing solutions to graph problems, such as FAS, to be able to scale to handle massive input graphs. The ability to scale is most relevant in social network analysis and web-related problems where current datasets can reach billions of vertices and tens of billions of arcs [13]. However, despite the importance of the FAS problem and the many efforts devoted to studying it, a complete and clear picture of the problem from an empirical viewpoint still appears elusive with existing empirical studies being incomplete by only considering a restricted version of the problem, for

example [24], or not scaling the considered approaches to real graphs, such as [3, 17].

We carry out an extensive experimental comparison of various algorithms for the FAS problem with runtime complexity $O(n^2)$ or less that have not been compared before in the literature on a full range of dataset sizes. The various approaches we consider occupy the broad classes of sorting-based, traversal-based, and randomized and fall into three running time complexity classes: $O(m+n)$, $O(n \log n)$, and $O(n^2)$. In terms of scalability, we observe that it is the algorithms in the $O(m+n)$ class that can handle graphs of billions of vertices and arcs. However, some algorithms needed careful engineering to best be placed in a complexity class. Our optimizations are discussed further in Section 3.2. In addition, we provide insights into the relative performance of the various approaches and provide global graph properties that indicate when the algorithms perform favourably. Furthermore, we extend our work to the probabilistic case in which arcs are realized with a fixed probability. We adapt our optimized algorithms for the FAS problem to the probabilistic case and conduct experiments on several large datasets. We show that the expected number of arcs in a FAS is usually a small fraction of the expected number of total arcs in the graph.

Our goal is to significantly speed-up the computation of a FAS and scale-up to massive graphs with tens of billions of arcs. Furthermore, we would like to achieve this using only a medium-range machine. In order to make the graph footprint as small as possible, we used *webgraph*, a highly efficient and actively maintained graph compression framework [13].

**Chapter 4.** Budak et al. [19] were the first to formulate the problem of misinformation prevention as a combinatorial optimization problem. By building upon the seminal work of Kempe et al. [52] on *influence maximization* for the single campaign model to handle multiple campaigns ("bad" and "good"), Budak et al. present a greedy approach that provides a $(1 - 1/e - \epsilon)$-approximate solution. Unfortunately

the greedy approach of [19] is plagued by the same scaling issues as [52] when considering large social networks and is further exacerbated by the added complexity of tracking multiple cascades which requires costly shortest paths computations. This leads us to the motivating question for this chapter: Can we find scalable algorithms for the misinformation prevention problem introduced in [19]?

The scalability hurdle for the case of a single campaign was recently resolved by Borgs et al. [15] when the authors made a theoretical breakthrough that fundamentally shifts the way in which we view the influence maximization problem. Their key insight was to reverse the question of "what subset of the network can a particular user influence" to "who could have influenced a particular user". Their sampling method runs in close to linear time and returns a $(1 - 1/e - \epsilon)$-approximate solution with at least $1 - n^{-l}$ probability. In addition, Tang et al. [86] presented a significant advance that improved the practical efficiency of the work by Borgs et al. through a careful theoretical analysis that rids the approach of [15] of a large hidden constant in the runtime guarantee.

Borgs et al. [15] leave open the question whether their techniques can be extended to other influence propagation models. In this work, we resolve the question of [15] for the misinformation prevention problem and achieve scalability in the multi-campaign model. We complement our theoretical analysis with extensive experiments which show an improvement of several orders of magnitude over Budak et al. [19]. Our work can be viewed as a parallel to that of Borgs et al. and Tang et al. in the multi-campaign setting. We present an efficient algorithm for the misinformation prevention problem that provides improved theoretical gaurantees over the existing *Greedy* approach by incorporating an analog to the reverse influence sampling idea.

Since influence in the single campaign setting corresponds to reachability in the network, our solution requires mapping the concept of reachability to an analogous

notion in the multi-campaign model of [19]. In [19] the authors introduce shortest paths as the metric to determine the set of nodes that can be saved from adopting the misinformation (i.e. influenced by the "good" information). Our first contribution is to identify that shortest paths alone are not sufficient in determining the ability to save a particular node in the network and introduce the crucial notion of *blocked* nodes. Additionally, we present a general classification result that captures those types of propagation models that require the notion of blocking and those that do not.

Using our newly defined notion of blocking, we generalize the sampling-based approach of Borgs et al. (later refined by [86]) to the multi-campaign setting. A major contribution of this work, and critical difference from the approach of [15] and [86], is a modified breadth-first search we design to compute the set of nodes that could have saved a particular user from adopting the misinformation. Our novel algorithm handles the notion of blocked nodes when traversing in the transpose graph by utilizing detailed bookkeeping techniques to effectively track those nodes that would become blocked. Furthermore, we are able to adapt optimization ideas from Tang et al. to further improve the scalability of our approach. In particular, we show that our algorithm can efficiently handle graphs with more than 50 million edges.

Finally, we obtain theoretical guarantees on the expected runtime and solution quality for our new approach and show that its expected runtime substantially improves upon the expected runtime of [19]. Additionally, we rule out sublinear algorithms for our problem through a lower bound on the time required to obtain a constant approximation.

## 1.1 Notation

This thesis studies several problems set in the context of graphs. Here, we introduce the terminology and notation common throughout the various chapters.

We consider simple, weakly connected, directed graphs $G = (V, E)$ on $n$ nodes (or vertices) and $m$ edges (or arcs). $G$ has a vertex set $V$ and edge set $E$. We assume there are no self-loops and no multiple edges. A directed graph is considered weakly connected if removing the directions on all edges yields an undirected graph which is connected. For a directed edge $(u, v)$ we refer to $u$ as the start node and $v$ as the end node. Also, we will use the term "digraph" when referring to directed graphs.

# Chapter 2

# Misinformation Removal

In this chapter we revisit work completed in the author's Master's thesis and augment the results with additional algorithmic engineering and new experiments and discussion, as published in [83]. This new work helps achieve scalability to massive social networks. We begin with a brief introduction of the problem and then present the algorithm and new experimental results. We omit the theoretical analysis of the algorithm and the NP-hardness proof and refer the interested reader to [83].

In particular, we consider the problem of clearing contamination spreading through a large network where we model the problem as a graph searching game. The problem can be summarized as constructing a search strategy that will leave the graph clear of any contamination at the end of the searching process in as few steps as possible. Despite the problem being NP-hard, even on directed acyclic graphs, we provide an efficient approximation algorithm. We experimentally observe the performance of our approximation algorithm in relation to the lower bound on several large online networks including Slashdot, Epinions and Twitter.

We start with an overview of information propagation in social networks and the graph searching problem. Next, we introduce the necessary concepts and definitions from graph searching before presenting the lower bound for search time on directed

graphs. Then, we introduce our strategy for clearing general networks and the Plank algorithm. Finally, we provide our updated experimental results and discussion.

## 2.1 Related Work

The task of maximizing the spread of information in a social network is a well studied problem with many works investigating different aspects of the problem [21, 23, 64]. More recently, the problem of limiting the spread of rumours or misinformation in a social network has been studied by [19, 8, 20]. In [8, 20] the problem is posed in terms of competing campaigns while [19] has the misinformation diffusing through a network. All three works are modelled by the Independent Cascade Model: a randomized diffusion process on graphs. However, the location of the misinformation is known and nodes can be inoculated such that once a node takes on the "good" information it will not subsequently adopt the misinformation. While the goal of these works was to limit the spread of misinformation, we believe it is important to investigate how to remove the misinformation from a network in its entirety. Furthermore, the unknown location of the misinformation and the deterministic spreading of contamination in our model captures the case of a stronger adversary.

Several variants of the (undirected) graph searching problem with respect to search number have been studied with varying constraints and adversary behaviour, see e.g. [26, 16, 54, 31]. In addition, it has been shown that the graph searching problem is closely related to several other notable graph parameters such as path-width, cut-width and vertex separation, see e.g. [10, 54, 31]. It was shown by Megiddo et al. [66] that computing the search number is NP-complete on general undirected graphs, but can be computed in linear time on undirected trees. Furthermore, several works [7, 5, 45] have investigated the search number in directed graphs with similar results.

The notion of search time for undirected graphs was introduced by Brandenburg and Herrmann [18]. The authors note that the classical goal of the graph searching game, where the minimal search number is computed, aims to minimize the number of resources used and as such corresponds to space complexity. The authors study the length of a search strategy which corresponds to the time complexity of searching a graph. The authors ask, how fast can a team of $k$ searchers clear a graph (if at all), and conversely how many searchers are needed to search a graph in time $t$. In contrast, search time in undirected graphs has also been investigated by considering the length of path decompositions by [27] in which the authors show that for any fixed $k \geq 4$ computing the minimum length path decomposition is NP-hard and give a polynomial time algorithm for $k < 4$.

## 2.2 Preliminaries

The rules for the graph searching game are as follows: Initially, all edges are contaminated and in the end all edges must be cleared. In a move at each time $t = 1, 2, \ldots$ searchers (or guards) are *first* removed from vertices and then placed on other (and possibly the same) vertices. In a single move some number of searchers can be placed or removed subject to the searcher budget. An edge is *cleared* at time $i$ if both incident nodes have searchers placed on them at the end of time $i$. A cleared edge $e$ is instantaneously *recontaminated* if there is a directed path from a contaminated edge to $e$ without a searcher on any vertex of that path. A *search strategy* is a sequence of moves that results in all edges being cleared at the end. Then the search game is won.

In the following example we show one possible search strategy with four available searchers for the directed graph shown in Figure 2.1. In the first step, searchers are

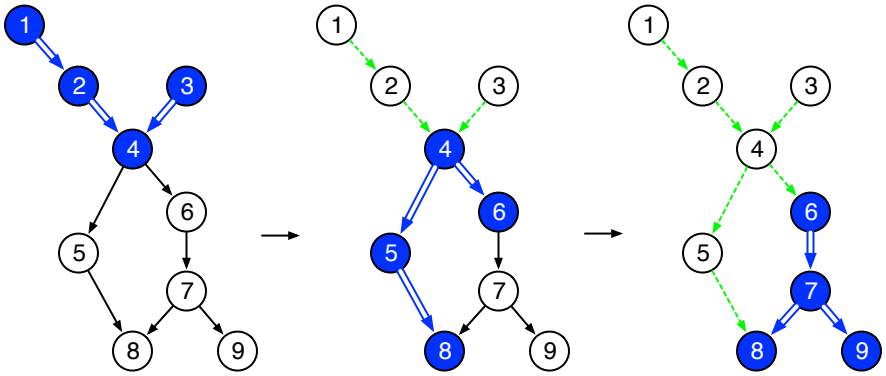



Figure 2.1: An example search strategy

placed on nodes 1, 2, 3, and 4 clearing the three blue (double-wide) edges. In the second step, searchers are removed from nodes 1, 2, and 3 to be placed on nodes 5, 8, and 6. We clear another three edges, and mark cleared edges in green (dotted). Finally, in a third step, we remove searchers from nodes 4 and 5, and place them on nodes 7 and 9. We clear the final three edges in the third step leaving the graph with all its edges cleared.

Our formal definition is similar to that of Brandenburg and Herrmann [18].

**Definition 1.** *A search strategy $\sigma$ on a (connected) digraph $G = (V, E)$ is a sequence of pairs $\sigma = ((E_0, V_0), (E_1, V_1), \ldots, (E_t, V_t))$ such that:*

1. *For $i = 0, \ldots, t$, $E_i \subseteq E$ is the set of cleared edges and $V_i \subseteq V$ is the set of vertices which have searchers placed on them at time $i$. The edges from $E \setminus E_i$ are contaminated.*

2. *(initial state) $E_0 = \emptyset$ and $V_0 = \emptyset$. All edges are contaminated.*

3. *(final state) $E_t = E$ and $V_t = \emptyset$. All edges are cleared.*

4. *(remove and place searchers and clear edges) For $i = 0, \ldots, t-1$ there are sets of vertices $R_i = V_i \setminus V_{i+1}$ and $P_i = V_{i+1} \setminus V_i$ where searchers are removed*

*from the vertices from $R_i$ and then placed at $P_i$. The set of cleared edges is $E_{i+1} = \{(u,v) \in E \mid u,v \in V_{i+1}$; or $(u,v) \in E_i \mid$ there is no unguarded directed path from the end node of a contaminated edge to $u\}$.*

*Let $width(\sigma) = max\{|V_i| \mid i = 0,\ldots,t\}$ and $length(\sigma) = t-1$ be the number of searchers and the number of moves of $\sigma$ respectively. Note that we discard the last move, which only removes searchers.*

While we need the $E_i$ sets above to define how a strategy works, we only need the $V_i$ sets to fully determine a strategy. Therefore, we will often refer to a strategy by only listing its $V_i$ sets.

**Definition 2.** *For a connected digraph $G$ with at least two vertices and integers $s$ and $t$ let search-width$_G(t)$ be the least $width(\sigma)$ for all search strategies $\sigma$ with $length(\sigma) \leq t$ and let search-time$_G(s)$ be the least $length(\sigma)$ for all search strategies $\sigma$ with $width(\sigma) \leq s$.*

In other words, *search-width$_G(t)$* is the least number of searchers that can search $G$ in time at most $t$, and *search-time$_G(s)$* is the shortest time such that at most $s$ searchers can search $G$. Thus, *search-width$_G(t) = s$* implies *search-time$_G(s) \leq t$* and conversely *search-time$_G(s) = t$* implies *search-width$_G(t) \leq s$*.

For a given time $t$, $\sigma$ is *space-optimal* if $width(\sigma) =$ *search-width$_G(t)$* with $length(\sigma) = t$. For a given number of searchers $s$, $\sigma$ is *time-optimal* if $length(\sigma) =$ *search-length$_G(s)$* with $width(\sigma) = s$.

## 2.3 Search-Time Lower Bound

The lower bound for search time on a digraph does not come as easily as the lower bound for undirected graphs of $\left\lceil \frac{n-s}{s-1} \right\rceil + 1$ shown by Brandenburg and Herrmann [18]

since the reasoning used there does not apply to the directed case. That is, a search strategy on a digraph can leave a node unguarded without suffering from recontamination unlike in the undirected case. We follow a completely different avenue to the lower bound.

Given a search strategy $\sigma$ we can construct a set system $S = \{S_1, \ldots, S_t\}$ where each set corresponds to the placement of searchers in a single step of $\sigma$. Thus, $t$ represents the number of steps the strategy requires. We have the following conditions for such a set system to correspond to a valid and complete search strategy.

1. $|S_i| \leq s$
2. If $u, v$ are adjacent nodes in $G$ then there exists an $S_i$ where $u, v \in S_i$

The first condition reflects the fact that we have $s$ searchers to work with while the second condition ensures that every edge in $G$ will be cleared. As a result we have the following fact about $S$

$$\forall i \; \exists j \text{ such that } S_i \cap S_j \neq \emptyset. \tag{2.1}$$

Equation 2.1 comes from condition 2 and the fact that $G$ is connected since a set $S_i$ without an intersection with some other set would constitute a separate connected component violating our assumption of connectedness.

Note, a search strategy will also induce an ordering of $S$, $\Omega$, which dictates how the search strategy unfolds. Notice that every search strategy induces a unique set system while a given set system may correspond to several search strategies depending on the ordering. Next we define the *progress* of a set which will be utilized in the lower bound proof.

**Definition 3.** *The progress of a set $S_i$ in an ordering $\Omega$ is $|\{v \in S_i | v \notin \cup_{j<i} S_j\}|$.*

The progress of a set corresponds to the number of new nodes visited in that step of the corresponding search strategy. The search time lower bound on directed graphs is as follows.

**Theorem 1.** *For every connected digraph $G$ with $|G| = n$ and integer $s$ such that $s$ is at least the search number of $G$ all search strategies require at least $\left\lceil \frac{n-s}{s-1} \right\rceil + 1$ steps to clear $G$.*

*Proof Sketch.* The proof utilizes the set system notion introduced above and considers an arbitrary search strategy $\sigma$ for $G$. First, we construct the corresponding set system $S$ for $\sigma$. Then, we construct a meta-graph on $S$ where each meta-node represents a set $S_i \in S$ and there is an *undirected* edge between two meta-nodes if their corresponding sets have a non-empty intersection. Call the resulting graph $G_S$. Then, we note that equation 2.1 and our assumption of connectedness implies that $G_S$ is connected.

Next, we present a special ordering $\Omega'$ for $S$ by performing a depth-first search of $G_S$. The order in which meta-nodes are visited in the DFS makes up $\Omega'$. This ordering may differ from that of $\sigma$ and is created purely for the proof of bounding the number of sets, $t$. Then, we can bound the progress $\rho$ made by this ordering from above by $s + (t-1)(s-1)$. Furthermore, $\rho$ is bounded below by $n$ as it is a necessary condition that every node in $G$ be visited by a searcher in order to clear all edges of $G$. □

In the next section we present our DFS-based algorithm for searching digraphs.

## 2.4 Our Search Algorithm

### 2.4.1 Searching Digraphs

The graph searching problem is NP-hard even on arborescences (a rooted directed tree where all edges point away from the root) as shown in [83], and so the task of clearing networks, which are general digraphs, is also NP-hard. We present a method for clearing a general digraph which works in two phases. We first compute a *feedback arc set (FAS)* for the network and remove the resulting edges. Formally, an FAS is a set of edges whose removal leaves a graph without cycles. Thus, by doing so, we are left with a DAG which can be cleared by our Plank algorithm given in the next subsection. In an online network the removal of the FAS edges is accomplished by simple software agents which block communication between two users until the search strategy has completed. We emphasize that blocking edges is much less resource intensive than placing searchers since the latter would likely involve clearing the browser or machine of a user, whereas blocking an edge has only "psychological" cost, if any, and it can even go unnoticed by some users. The procedure for searching general digraphs is outlined in Algorithm 1.

**Algorithm 1** Search Digraph

**Input:** The input graph $G$
**Output:** A search strategy $\sigma = (V_1, \ldots, V_t)$
  Compute an FAS for $G$
  Remove the FAS edges from $G$ to create a DAG $G'$ that needs to be cleared
  Run the Plank algorithm on $G'$ to compute a search strategy $\sigma = (V_1, \ldots, V_t)$
  **return** $\sigma = (V_1, \ldots, V_t)$

In the following section we present our Plank algorithm for searching a DAG.

### 2.4.2 Plank Algorithm

Our Plank algorithm works in a depth-first manner with some modifications specific to the graph searching problem. The name comes from a description of how searchers are placed in subsequent steps. Imagine a long plank of wood lying on the ground. We can move this plank by picking up one end until the plank is upright and then letting it drop in the direction we wish to travel. By repeatedly moving in this way we move the plank a distance equal to its length each time. Then, we can think of the plank as $s$ searchers placed adjacently on a graph so that moving the plank corresponds to visiting $s - 1$ new nodes.

The Plank algorithm is a two-phase algorithm for computing its search strategy for a DAG, $G$. In the first step the algorithm computes an edge ordering for $G$, $\Psi$, and in the second step it compiles a search strategy from $\Psi$. In Algorithm 2 below, $mDFS$ refers to a modified depth-first search designed specifically for the Plank algorithm. Our $mDFS$ operates similarly to the $DFS$ algorithm, but with a special stopping condition: we backtrack if the current vertex has an unexplored incoming edge. This ensures we do not allow any recontamination from uncleared incoming edges as our strategy does not leave stationary guards at vertices. The Plank's high level execution proceeds as follows:

1. Run $mDFS$ on $G$ to produce an edge ordering $\Psi$

2. Convert $\Psi$ into a search strategy using $s$ searchers

Now we present the Plank's subalgorithms. First, we have the pseudocode for the $mDFS$ algorithm in Algorithm 2. We assume all nodes in $G$ are initially labelled as unvisited and all edges as unexplored.

Note, in the case that every node in $G$ is not visited in a call to $mDFS$, we continue re-calling the algorithm passing in an unexplored node until there are no

**Algorithm 2** mDFS

**Input:** Input graph $G$ and the current node $v$
**Output:** An edge ordering $\Psi$

```
Ψ ← []
if v has no unexplored incoming edges then
    Label v as visited
    for all edges e in G.outEdges(v) do
        if edge e is unexplored then
            Ψ.append(e)
            Label e as explored
            w ← G.adjacentVertex(v, e)
            Ψ.append(mDFS(G, w))
return Ψ
```

more unexplored nodes in $G$. If there are multiple edge labellings, they are appended together to make a master edge labelling.

Next, we show how to convert the resulting edge labelling, $\Psi$, into a search strategy for $G$ using $s$ searchers (Algorithm 3). In summary, $\Psi$ is traversed adding nodes to the current step in the search strategy until a step has reached $s$ placements. After $\Psi$ has been traversed we will have all the steps which make up the Plank search strategy $\sigma$. This procedure is captured in the pseudocode of Algorithm 3 where $V_c$ represents the nodes present in the current step.

**Algorithm 3** Construct Strategy

**Input:** Sequence $\Psi$ and the number of searchers $s$
**Output:** a search strategy $\sigma = (V_1, \ldots, V_t)$

```
σ, V_c, ← ∅
for all edges e in Ψ do
    if nodes(e) not in V_c then
        V_c ← V_c ∪ nodes(e)
    if current step contains s placements then
        σ.append(V_c)
        V_c = ∅
return σ = (V_1, ..., V_t)
```

Furthermore, we introduce an optimized version of the strategy construction al-

gorithm in which the steps that each node participates in during the strategy are recorded. This allows us to determine if, when processing an edge $e = (u, v)$ in $\Psi$, the edge has already been cleared in a previous step of the search strategy and thus can be skipped. In doing this, we eliminate redundantly clearing an edge multiple times leading to shorter search strategies. We can efficiently determine if $e$ has previously been cleared by computing the intersection of the sets of steps that $u$ and $v$ have participated in. This optimization allows us to avoid storing and checking membership in a large set of cleared edges. The optimized strategy construction algorithm is shown in Algorithm 4.

**Algorithm 4** Optimized Construct Strategy

**Input:** Sequence $\Psi$ and the number of searchers $s$
**Output:** a search strategy $\sigma = (V_1, \ldots, V_t)$

```
i ← 0
σ, V_c ← ∅
steps ← new array of n empty sets
for all edges e = (u, v) in Ψ do
    if steps[u] ∩ steps[v] = ∅ then
        if u not in V_c then
            V_c ← V_c ∪ u
            steps[u] ← steps[u] ∪ i
        if v not in V_c then
            V_c ← V_c ∪ v
            steps[v] ← steps[v] ∪ i
    if current step contains s placements then
        σ.append(V_c)
        V_c = ∅
        i ← i + 1
return σ = (V_1, ..., V_t)
```

Finally, we provide proofs for the correctness and asymptotic runtime of the Plank algorithm.

**Theorem 2.** *For any DAG $G$ and budget of searchers $s \geq 2$, Algorithm 2 and Algorithm 3 produce a search strategy $\sigma$ with $s$ searchers for $G$.*

*Proof.* First, note that since Algorithm 2 is a modified DFS it is clear that every edge $e \in E$ is added to the edge ordering $\Psi$. Second, Algorithm 3 places searchers on both endpoints of every edge in $\Psi$. Thus, the resulting strategy $\sigma$ will have cleared every edge at some step $V_i$.

Now, we need to ensure that once an edge $e = (u, v)$ has been cleared it never gets recontaminated. Notice that recontamination can only occur from an edge $e_r$ directed towards $u$. Suppose to the contrary that $e$ is cleared in some step $i$ and $e_r$ has not yet been cleared. Thus, Algorithm 2 would have had to have reached $e$ when $u$ had unexplored incoming edges. Thus, we have a contradiction with line 2 of Algorithm 2 and therefore we will not have any recontaminating edges $e_r$ present when clearing $e$. Finally, since there are no cycles in $G$, we can never have a node with no unexplored incoming edges that then become recontaminating edges resulting in a cleared edge remaining cleared for the rest of $\sigma$.

In conclusion, as long as $s \geq 2$, the resulting search strategy $\sigma$ will leave every edge in $G$ cleared by the end of $\sigma$. □

Algorithm 2 is simply a modified DFS with an alternate stopping condition yielding the same asymptotic runtime as a traditional DFS and Algorithm 3 makes a single pass over the resulting edge ordering.

**Lemma 1.** *For any DAG $G$ and budget of searchers $s$, Algorithm 2 and Algorithm 3 run in time $O(m + n)$.*

## 2.5 Experiments

In this section, we present the results of our experiments, which have the following goals:

- Observe the performance of the Plank strategy in various types of networks.

- Observe how the Plank strategy performs as the number of searchers available increases.
- Observe how the Plank strategy performs as the size of the network grows.

We note that for the majority of our datasets the direction of the edges represents a following/trust relation which we reverse to move to an influence relation. All of our algorithms are implemented in Java and tested on a machine with dual 6 core 2.10GHz Intel Xeon CPUs, 32GB RAM and running Ubuntu 14.04.2.

For the task of computing a FAS, an NP-hard problem, we employ the GreedyFAS heuristic introduced in [30] and remove the resulting edge set from $G$. In particular, we use a highly optimized array-based version as presented in the following chapter which has excellent trade-off between efficiency and FAS size. Chapter 3 provides a full comparison of FAS algorithms that have been carefully engineered to scale to massive web-scale graphs. As an overview, the approach in [30] is a greedy algorithm that computes a vertex sequence from $G$ and returns all leftward arcs from the sequence as an FAS. The algorithm begins with an initial bin sort on a set of vertex classes determined by the degree of each vertex $v$ in $G$ (more specifically the difference between the out-degree and the in-degree denoted $\delta(v)$). In each round of the greedy approach we remove a sink, source, or vertex corresponding to a maximum vertex class and prepend or append the vertex to one of two vertex sequences $s_1$ and $s_2$ depending on its vertex class. As a result, this changes the vertex class of adjacent vertices in $G$. In the end, the two sequences $s_1$ and $s_2$ are concatenated together to produce the final vertex sequence from which the leftward arcs are taken as a FAS. The algorithm runs in $O(n+m)$ time and requires $O(n+m)$ space. The pseudocode for GreedyFAS is presented in Algorithm 5

**Algorithm 5** GreedyFAS

**Input:** Input directed graph $G = (V, E)$
**Output:** Linear arrangement $A$
  $s_1 \leftarrow \emptyset$, $s_2 \leftarrow \emptyset$
  **while** $G \neq \emptyset$ **do**
    **while** $G$ contains a sink **do**
      choose a sink $u$
      $s_2 \leftarrow us_2$
      $G \leftarrow G \setminus u$
    **while** $G$ contains a source **do**
      choose a source $u$
      $s_1 \leftarrow s_1u$
      $G \leftarrow G \setminus u$
    choose a vertex $u$ for which $\delta(u)$ is a maximum
    $s_1 \leftarrow s_1u$
    $G \leftarrow G \setminus u$
  **return** $s = s_1s_2$

### 2.5.1 Online Networks

For each of the networks we run our Plank algorithm on the obtained DAG with $s$ ranging from $0.1 - 1\%$ of the size of the network increasing in 0.1% increments. Then, we plot the ratio of the length of the strategy to the lower bound where we compare the baseline strategy construction algorithm (mDFS) to the optimized version (mDFSO). Furthermore, we compare the performance of the Plank strategy to a generic splitting strategy (mBFSO) generated from a modified BFS with the same alternate stopping condition as our $mDFS$. The splitting strategy sends at least two searchers down as many branches of a section as possible similar to a BFS traversal and is generated with the optimized strategy construction algorithm. Below we give an overview of each of the datasets.

*Wiki-Vote:* First, we look at the Wiki-Vote dataset from [58]. An edge in this network from user A to B indicates that A voted for B to become an administrator on Wikipedia. The FAS computed contained 8% of the network's edges.

*Twitter Retweet Network:* The higgs-retweet network from [28] maps the retweets by users of Twitter during the announcement of the discovery of the Higgs Boson. The network contains an edge from user A to B if A retweeted B. The FAS computed contained 0.1% of the network's edges indicating a very DAG-like structure.

*Email Communication Network:* The email-EU network from [59] was generated using email data from a large European research institution. The network contains an edge from user A to B if A emailed B. The FAS computed contained 14% of the network's edges.

*Signed Epinions:* The signed Epinions trust network from [58] contains an edge from user A to B if A trusts B on the Epinions review site. The FAS computed contained 24% of the network's edges.

*Signed Slashdot:* Next, we look at the signed Slashdot dataset from [58]. An edge in this network from user A to B indicates that B is a friend of A's. The FAS computed contained 8% of the network's edges.

*wiki-Talk:* The wiki-Talk network from [58] contains an edge from user A to B if A has at least once edited a talk page of user B on Wikipedia. The FAS computed contained 10% of the network's edges.

*pokec:* Pokec is the most popular online social network in Slovakia. The pokec network from [58] contains an edge from user A to B if B is a friend of A's. The FAS computed contained 33% of the network's edges. The FAS for the pokec network is the largest among the datasets we consider.

*twitter2010:* The twitter2010 network from [13, 12] contains an edge from user A to B if B is a follower of A on the social network Twitter. The FAS computed contained 22% of the network's edges.

The new scalablility results in this section are from the Pokec and twitter2010 datasets. We show that with our improved preprocessing and strategy construction

| Name | $\|V\|$ | $\|E\|$ | FAS |
|---|---|---|---|
| wiki-Vote | 7,115 | 103,689 | 8% |
| higgs-retweet | 256,491 | 328,132 | .1% |
| email-EuAll | 265,214 | 420,045 | 14% |
| epinions | 131,828 | 841,372 | 24% |
| slashdot | 77,360 | 905,468 | 8% |
| wiki-Talk | 2,394,385 | 5,021,410 | 10% |
| pokec | 1,632,803 | 30,622,564 | 33% |
| twitter2010 | 41,652,230 | 1,468,365,182 | 22% |

Table 2.1: Dataset statistics

the network size that the Plank algorithm can handle improves by orders of magnitude over our previously reported results.

### 2.5.2 Discussion

Figure 2.2 shows the running time in seconds that it took to compute the search strategies for each dataset including the FAS computation. Most run in a matter of seconds, while twitter2010 took approximately 25 minutes. Figure 2.3 shows the approximation ratios of the strategies computed by our Plank algorithm and the splitting strategy. On every network we see the optimized strategy construction algorithm outperforming the baseline version. Furthermore, on all networks except twitter2010 we see that the Plank strategy outperforms the splitting strategy. With regards to twitter2010, we note that our result in [83] comparing the two types of strategies relies on a key structural aspect of the network - namely having long disjoint paths present in the network. However, we expect the twitter network to be very condensed and thus void of long chains of isolated follower relationships as evidenced by the network's ratio of edges-to-nodes.

We can see that the size of the network does not have a strong influence on the resulting approximation ratio. Instead, the structure of the network is the primary factor in determining how large the approximation will be. As described in Section

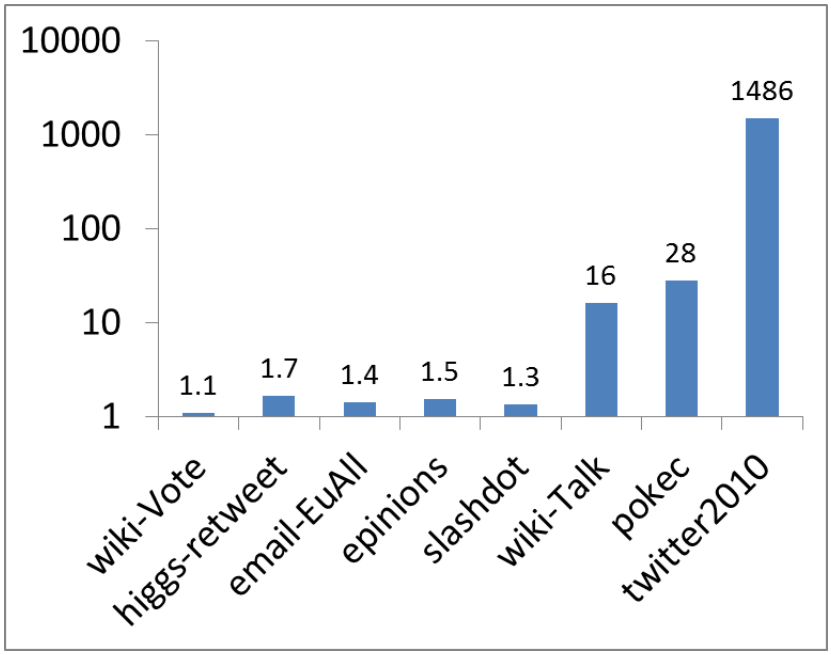
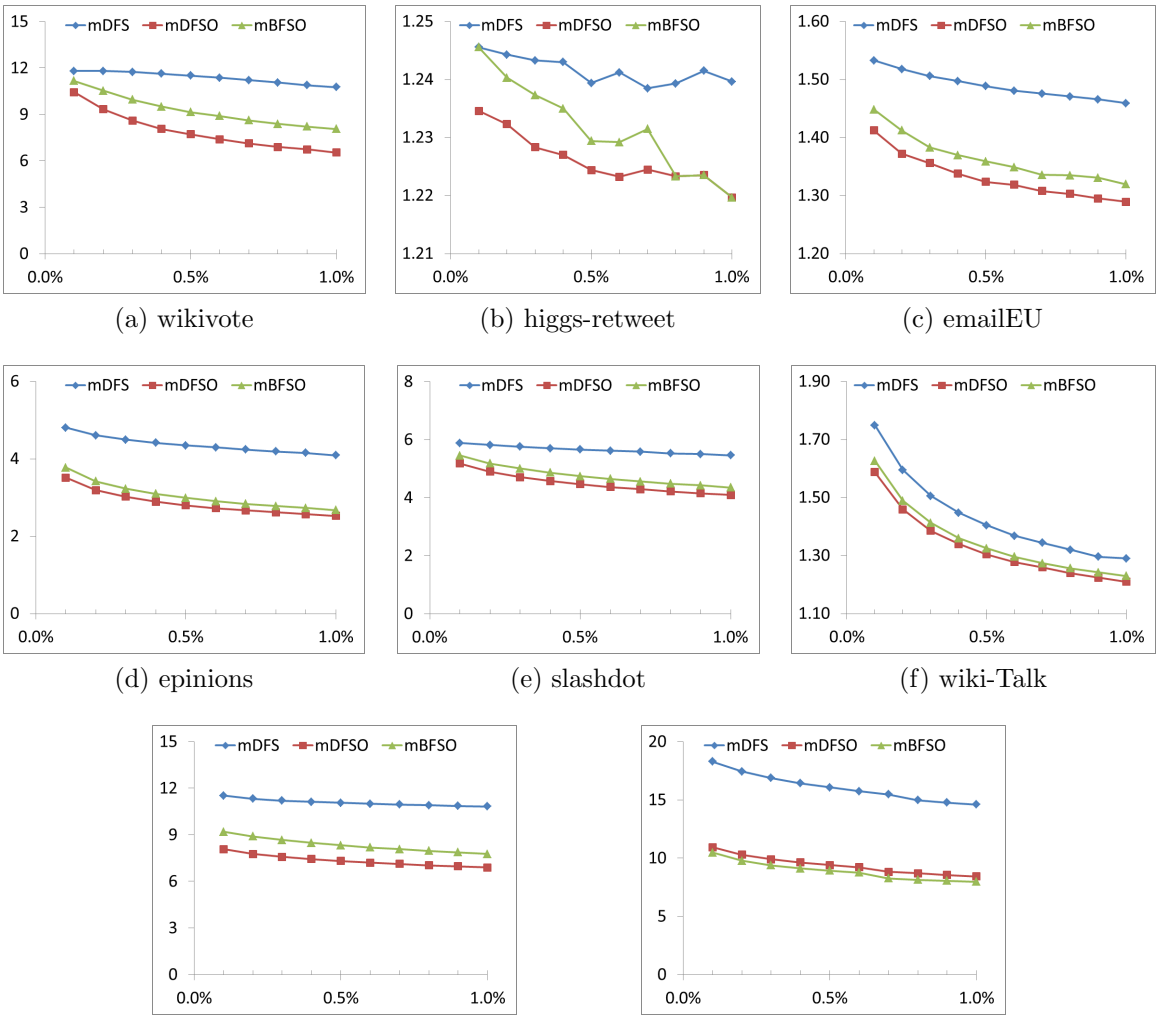



Figure 2.2: Time in seconds.

7.1 of [83], the overlap factor (a density measure of the network), which can often be approximated by $\frac{m}{n}$, drives up the approximation ratio due to high overlap nodes having to be re-visited often in the search strategy. Using this as a bearing, we see that twitter2010, pokec, and wikivote, three networks with large approximation ratios, have an edge-to-node ratio of 15.0, 18.7, and 35.2 respectively.

It is important to remember that the approximation ratio is comparing the strategy lengths to the lower bound and are therefore quite pessimistic. In reality, the optimal strategy achievable for any given network is likely much longer than the lower bound and therefore closer to the length of the strategy computed by our Plank algorithm.

Finally, we also ran experiments with small fixed amounts of searchers as shown in Figure 2.4 to illustrate how our Plank algorithm performs when few searchers are available. We see similar approximation ratios to Figure 2.3, but note that increasing the number of searchers allows for improvement. We outline the reasoning for this




(a) wikivote (b) higgs-retweet (c) emailEU

(d) epinions (e) slashdot (f) wiki-Talk

(g) pokec (h) twitter2010

Figure 2.3: Approximation ratios for real datasets. The horizontal axes represent $s$ ranging from 0.1–1% of the size of the network increasing in 0.1 increments. The vertical axes represent approximation ratios.

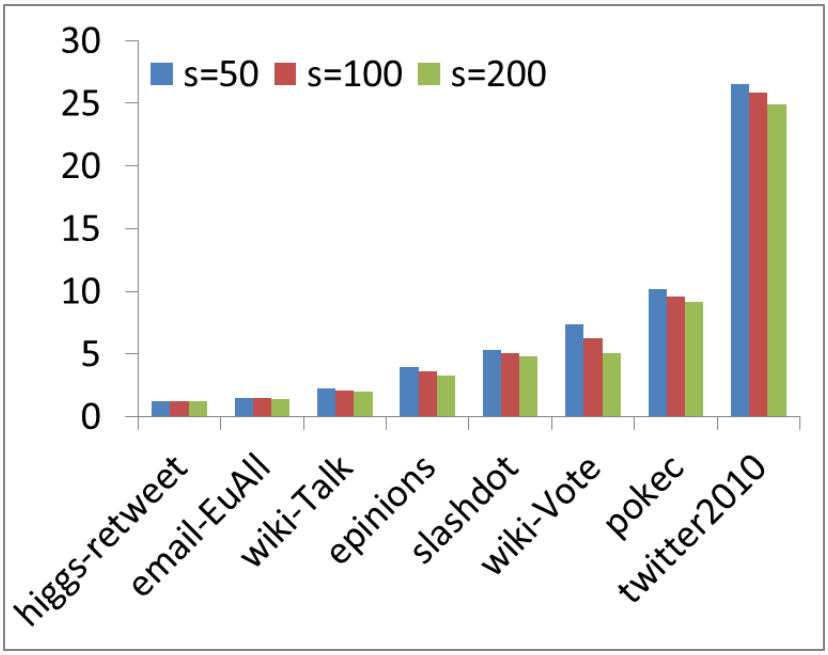



Figure 2.4: Approximation ratio when $s = 50$, $s = 100$, and $s = 200$.

improvement below.

Consider an arbitrary step $V_i$ in a search strategy $\sigma$ using $s$ searchers. The nodes in step $V_i$ were determined by sequential edges taken from the edge ordering. As such, we can consider the edges used to select the $V_i$ as *intended edges*, that is, edges that we intended to clear in the current step. Given $s$ searchers, we will have $s-1$ intended edges. Now, consider the directed clique $K_{V_i}$ on the nodes of $V_i$. Among the $s(s-1)$ edges in $K_{V_i}$, $s-1$ of them are intended. This leaves, $s(s-1)-(s-1) = (s-1)^2$ edges that are potentially edges of $G$. If any of these remaining edges are, in fact, present in $G$, then the current step will clear those edges in addition to the $s-1$ intended edges. We can think of these as *freely cleared* edges. Now, consider what happens as $s$ increases. We get an increasing number of potentially free edges cleared in $V_i$. Therefore, as $s$ increases, the expected number of freely cleared edges increases which leads to more edges cleared in each step of $\sigma$. As a result, we expect the length of search strategies to decrease as $s$ increases. However, at the same time, the rate of

decrease of the approximation ratio will be influenced by the size of the largest clique in $G$ since increasing $s$ beyond this size will limit the number of free edges cleared.

# Chapter 3

# Feedback Arc Set

In this chapter we present the results of [84] where the author considers the minimum feedback arc set problem: an NP-hard problem on graphs that seeks a minimum set of arcs which, when removed from the graph, leave it acyclic. We investigate several approximations for computing a minimum feedback arc set with the goal of comparing the quality of the solutions and the running times. Our investigation is motivated by applications in Social Network Analysis such as misinformation removal and label propagation. We present careful algorithmic engineering for multiple algorithms to improve the scalability of each approach. In particular, two approaches we optimize (one greedy and one randomized) provide a nice balance between feedback arc set size and running time complexity. We experimentally compare the performance of a wide range of algorithms on a broad selection of large online networks including Twitter, LiveJournal, and the Clueweb12 dataset. The experiments reveal that our greedy and randomized implementations outperform the other approaches by simultaneously computing a feedback arc set of competitive size and scaling to web-scale graphs with billions of vertices and tens of billions of arcs. Finally, we extend the algorithms considered to the probabilistic case in which arcs are realized with some fixed probability and provide detailed experimental comparisons.

The contributions of this chapter are:

1. We present algorithmic engineering of several approaches for computing a feedback arc set in order to improve their scalability.

2. We present a thorough experimental study of the known methods for computing feedback arc sets with the potential to scale. For the comparison of these algorithms, we focus on the parameters of running time and feedback arc set size.

3. We provide insights into the relative performance of the considered approaches and present global graph properties that indicate when the algorithms perform favourably.

4. We extend our optimized algorithms for the FAS problem to the probabilistic case and carry out further experimental comparisons.

The layout of the chapter is as follows. Section 3.1 discusses related works. Section 3.2 details each of the algorithms considered and outlines the optimizations proposed. Section 3.3 presents our experimental results and in Section 3.4 we offer a discussion of our results. In Section 3.5 we investigate the probabilistic case. Finally, we conclude in Section 3.6.

## 3.1 Related Work

The task of extracting an acyclic subgraph from an existing social network has been considered by Gupte et al. [42] where a measure of social hierarchy in directed social networks was defined, which extracts an implicit hierarchy from the existing arc directions by computing a social rank for each user. The authors assume that

in a social network, when users connect to other users who are lower in the implicit hierarchy, this causes them social agony. Thus, to infer the ranks of each user, the authors compute the ranking that gives the least possible social agony. The resulting hierarchy can be used to define an acyclic subgraph by only considering those arcs that go from a user with a higher social rank to one with a lower rank.

In other work by [71] the classical problem of influence maximization is considered in the context of belief propagation in Bayesian networks. The author's approach converts social networks, modeled as general digraphs, to DAGs via two different approaches that clip arcs from the graph while retaining important arcs where the predicted impact on influence dissemination is high. The selection of arcs to remove in each approach is based on the influence region of the seed set which can be computed via Dijkstra trees.

Up until now, the theory community has focused on achieving the best possible approximation ratios for the FAS problem with less emphasis on the running time of the resulting algorithms. The first approximation algorithm for the FAS problem was given by Leighton and Rao [57] with an approximation ratio of $O(\log^2 n)$ by using an $O(\log n)$ approximation algorithm for balanced cuts. The authors appealed to linear programming techniques to show that the problem can be solved in polynomial time. This approach was improved by Klein, Stein, & Tardos [55] to a $O(m^2 \log m)$ expected time randomized algorithm. The current best known approximation algorithm, due to [32], for computing the minimum feedback arc set has a ratio $O(\log n \log\log n)$ and runs in $O(n^2 M(n) \log^2 n)$ time, where $M(n)$ denotes the complexity of matrix multiplication.

Despite being NP-complete, the FAS problem is fixed parameter tractable in the number of edges of the minimum feedback arc set. Furthermore, the FAS problem is polynomial-time solvable when $G$ is planar [65, 37], $K_{3,3}$-free [73], and on the classes

of reducible flow graphs [79] and weakly acyclic graphs [41].

Furthermore, [48] showed that the FAS problem is APX-hard, which means that there is a constant $c$, such that, assuming $P \neq NP$, there is no polynomial-time approximation algorithm that always finds an edge set at most $c$ times bigger that the optimal result. The current highest value of $c$ for which such an impossibility result is known is $c = 1.3606$.

Other approximation algorithms for the FAS problem aim to achieve simplicity in their approach. One, due to Demetrescu and Finocchi [25] that applies to graphs with weighted arcs, is purely combinatorial and runs in time $O(mn)$ while achieving an approximation ratio bounded by the length of the longest simple cycle of the graph. Their approach adopts a classical technique for approximating covering problems, the *local-ratio* approach. Roughly speaking, their algorithm attempts to find a compromise between removing light arcs and arcs belonging to a large number of cycles. Unfortunately, a running time complexity of $O(mn)$ makes it unfeasible when applied to web-scale graphs.

Recent work on the FAS problem has led to advances on tournament graphs (see [24] for a survey). The restricted problem admits a polynomial-time approximation scheme due to Kenyon-Mathieu & Schudy [53] that is achieved by applying Kwik-SortFAS once and improving the linear arrangement by repeatedly removing vertices and reinserting them at another position which is akin to the SiftFAS approach. Furthermore, a subexponential fixed parameter algorithm for the weighted version was given by Karpinski & Schudy [50].

In [24] the authors survey the known algorithms for computing a feedback arc set in tournament graphs and introduce a new triangle-deletion approach that, when combined with a powerful local-search technique, outperforms the existing approaches on a variety of datasets.

A body of work also exists that investigates various heuristics for computing a minimum feedback arc set. Saab [80] provides a divide-and-conquer heuristic based on graph partitions and strongly connected components. The divide step aims to produce a balanced partition $G = (V_1, V_2)$ that minimizes the cardinality of the set $P = \{i \to j : i \in V_2, j \in V_1\}$ in order to construct a feedback arc set consisting of the arcs of feedback arcs sets for $V_1$ and $V_2$ together with the arcs in $P$. However, the costly subalgorithms required by this approach preclude it from being considered in this thesis. Finally, there are the heuristics considered in this thesis: GreedyFAS [30], InsertionSortFAS & KwikSortFAS [17], and BergerShorFAS [6].

## 3.2 Algorithms

In this section we outline the scalable approaches we consider for the FAS problem. For most algorithms we provide pseudocode and give the asymptotic running times. Furthermore, we discuss the various optimizations we make to each algorithm.

The algorithm that produced the best FAS size for our largest datasets, but whose running time needed care to bring down to $O(m+n)$, was GreedyFAS due to Eades, Lin & Smyth [30]. A direct implementation of GreedyFAS runs in $O(n^2)$ time, which is impractical for large social and web networks since they are sparse graphs with $m \ll n^2$. To remedy this, we present and incorporate the use of efficient data structures which brings its complexity to $O(m+n)$, thus making it scalable to the largest dataset we consider on tens of billion of arcs. Another algorithm we engineer with efficient data structures is SortFAS of Brandenburg & Hanauer [17]. A direct implementation of it runs in $O(n^3)$ and we bring it down to $O(n^2)$ with our optimized implementation. Finally, we also optimize a randomized algorithm, BergerShorFAS, by Berger & Shor [6], which computes a reasonably small FAS while running in $O(m+n)$ time.

| Algorithm | Running Time | Randomization | Upper bound on FAS size |
|---|---|---|---|
| GreedyFAS | $O(m+n)$ | Deterministic | $\frac{1}{2}\|E\| - \frac{1}{6}\|V\|$ |
| BergerShorFAS | $O(m+n)$ | Randomized | $(\frac{1}{2} - \Omega(1/\sqrt{d_{max}}))\|E\|$ |
| SimpleFAS | $O(m+n)$ | Randomized | $\frac{1}{2}\|E\|$ |
| dfsFAS | $O(m+n)$ | Deterministic | — |
| KwikSortFAS | $O(n \log n)$ | Randomized | — |
| InsertionSortFAS | $O(n^2)$ | Deterministic | — |

Table 3.1: Algorithms evaluated in this chapter.

### 3.2.1 Graph Representation

We assume the vertices are labelled $1 \dots n$. Furthermore, due to the large size of the datasets we consider, our graph data structure uses adjacency lists to maintain neighbour relationships. We make use of the *webgraph* framework [13] which is a highly efficient graph compression framework that allows accessing a graph without fully decompressing it by providing only the required components on the fly. While webgraph facilitates scaling up to large graphs, it also presents implementation challenges as the data structure offered by webgraph is an immutable graph. As such, all the vertex or arc removals in the considered algorithms are implemented as logical removals.

We begin by describing the GreedyFAS approach with the remaining algorithms presented in three groups according to their running times: $O(n^2)$, $O(n \log n)$, and $O(n+m)$. Table 3.1 summarizes the algorithms that we evaluate in this chapter. Additionally, we will illustrate the algorithms using the graph shown in Figure 3.1 on 8 vertices and 13 arcs. Observe that the minimum FAS contains the single arc $(3, 4)$.

### 3.2.2 Equivalent Formulations

The FAS problem has an equivalent formulation called the Linear Arrangement (LA) problem. The LA problem takes as input a directed graph $G$ and outputs an ordering

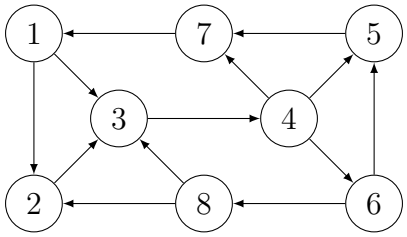



Figure 3.1: An example graph with a minimum FAS of size 1.

of the vertices for which the number of arcs pointing backward from right to left is a minimum. The backward arcs are exactly those arcs that make up a FAS since removing them from $G$ leaves the graph acyclic. We make use of the LA formulation for a subset of the approaches considered.

The dual of the FAS problem is the Maximum Acyclic Subgraph (MAS) problem. The MAS problem takes as input a directed graph $G$ and returns the maximum acyclic subgraph of $G$. Thus, the arcs of $G$ that do not appear in the MAS solution are exactly the arcs that make up a minimum FAS for $G$. The SimpleFAS and BergerShorFAS (see subsections 3.2.4 & 3.2.5) approaches are based on algorithms for the MAS problem.

### 3.2.3 GreedyFAS

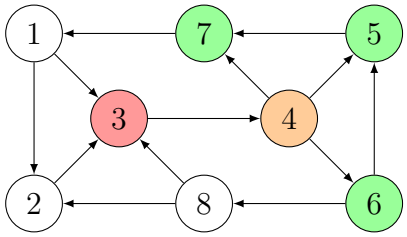
GreedyFAS was introduced by Eades, Lin & Smyth [30] as an efficient approximation algorithm for the feedback arc set problem. For each vertex $u \in V$, let $d^+(u)$ denote the outdegree of $u$ and $d^-(u)$ denote the indegree of $u$. In each iteration of GreedyFAS, the algorithm removes vertices from $G$ that are sources or sinks followed by a vertex $u$ for which $\delta(u) = d^+(u) - d^-(u)$ is currently a maximum. If a vertex $u$ removed from $G$ is a sink it is prepended to a vertex sequence $s_2$, otherwise it is appended to a vertex sequence $s_1$. Once all the vertices of $G$ have been removed, the sequence $s = s_1 s_2$ is returned as a linear arrangement for which the backward arcs make up a feedback arc set. The pseudocode for GreedyFAS is presented in Algorithm 6.

**Algorithm 6** GreedyFAS

**Input:** Input directed graph $G = (V, E)$
**Output:** Linear arrangement $A$
  $s_1 \leftarrow \emptyset$, $s_2 \leftarrow \emptyset$
  **while** $G \neq \emptyset$ **do**
    **while** $G$ contains a sink **do**
      choose a sink $u$
      $s_2 \leftarrow u s_2$
      $G \leftarrow G \setminus u$
    **while** $G$ contains a source **do**
      choose a source $u$
      $s_1 \leftarrow s_1 u$
      $G \leftarrow G \setminus u$
    choose a vertex $u$ for which $\delta(u)$ is a maximum
    $s_1 \leftarrow s_1 u$
    $G \leftarrow G \setminus u$
  **return** $s = s_1 s_2$

The intuition behind this approach is to greedily move all the "sink-like" vertices to the right-side of the ordering and all the "source-like" vertices to the left-side in an attempt to minimize the number of arcs oriented from right to left.

As an illustrative example, consider the execution of GreedyFAS on the graph in Figure 3.1. Initially, there are no sinks or sources so we remove the vertex $u$ for which $\delta(u)$ is a maximum: vertex 4. The vertex and its arcs are removed from $G$ and vertex 4 is appended to $s_1$. After removing vertex 4 from $G$ a sink is created at vertex 3. As such, vertex 3 is removed from $G$ and prepended to $s_2$. Again, a new sink is created at vertex 2. The process continues removing vertices 1, 7, 5, 8, and finally 6 which are all prepended to $s_2$. The resulting sequence is $s = s_1 s_2 = [4, 6, 8, 5, 7, 1, 2, 3]$. Thus, we can extract a feedback arc set of size 1 by observing that this linear arrangement has only a single backward arc from vertex 3 to 4. In Figure 3.2 we illustrate the execution of GreedyFAS in its first iteration when processing vertex 4 in the example graph. We show in red and green the vertices that have their $\delta$ value decremented




$$\delta(3) \leftarrow \delta(3) - 1 \qquad \delta(6) \leftarrow \delta(6) + 1$$
$$\delta(5) \leftarrow \delta(5) + 1 \qquad \delta(7) \leftarrow \delta(7) + 1$$

Figure 3.2: Processing vertex 4 during the execution of GreedyFAS on the example graph.

and incremented, respectively.

Eades, Lin & Smyth discuss the following implementation details for their algorithm. To begin, it is convenient to partition the vertices of $G$ into sources, sinks, and $\delta$-classes as follows:

$$V_{n-2} = \{u \in V \mid d^-(u) = 0;\ d^+(u) > 0\} \tag{3.1}$$
$$V_{-n+2} = \{u \in V \mid d^+(u) = 0\} \tag{3.2}$$
$$V_d = \{u \in V \mid d = \delta(u);\ d^+(u) > 0;\ d^-(u) > 0\} \tag{3.3}$$

with $d \in [-n+3, n-3]$. Then, we can see that every vertex $u \in V$ falls into exactly one of these $2n-3$ classes. For a given directed graph $G$ an initialization phase computes these classes.

Each vertex class is implemented as a bin with the vertices in each class linked together by a doubly-linked list. Then, using the bins, we can recognize a sink, source, or vertex $u$ for which $\delta(u)$ is a maximum. Finally, consider how to form $G \setminus u$. The vertex $u$ itself can be logically removed by eliminating it from its bin list. As a result, every vertex $v$ adjacent to $u$ will either become a sink, a source, or an element of an adjacent bin.

We optimized the GreedyFAS implementation into the following two versions. In the first version, referred to as $dllFAS$, we implement custom doubly-linked lists for the bins in order to directly manipulate the list nodes and alleviate a bottleneck suffered by generic (library) lists. When moving a vertex to an adjacent bin we must first delete the vertex from its current list. In a generic list this deletion requires a linear pass over the list to find the appropriate list node to delete. Only then can we add the vertex to the bin corresponding to its updated vertex class. In the worst case, the size of a bin could be $O(n)$. In order to facilitate moving a vertex to an adjacent bin in $O(1)$ time, we maintain an array containing each individual list node of every vertex in $G$. As such, we are able to manipulate the node's previous and next node references directly and complete the update efficiently regardless of the size of each bin.

In the second version, referred to as $ArrayFAS$, we do away with the doubly-linked lists all together by maintaining three flat arrays that mimic the behaviour of the lists. The first array, $bins$, maintains the tail of bin $i$. That is, at position $i$ in array $bins$ we store the vertex currently at the tail of the doubly-linked list for bin $i$. The remaining two arrays, $prev$ and $next$, contain information regarding the node references for the list node corresponding to vertex $i$. That is, at position $i$ in array $prev$ ($next$) we store the index of the vertex that is before (after) vertex $i$ in $i$'s vertex class bin. ArrayFAS aims to further reduce the space overhead and initialization time required by GreedyFAS. By tightly packing the data structure in flat arrays we allow for better memory management and achieve more efficient execution.

In conclusion, we analyze the complexity of forming $G \setminus u$. The vertex $u$ itself can be logically removed in $O(1)$ time by eliminating it from its bin list. Then, the update for every vertex $v$ adjacent to $u$ moving to an adjacent bin requires $O(1)$ time for each arc incident to $u$. As such, GreedyFAS runs in time $O(m + n)$ and uses

$O(m+n)$ space. It has an approximation guarantee of $\frac{1}{2}|E| - \frac{1}{6}|V|$ but we observe in the experiments that the size of the FAS produced is drastically smaller than the size suggested by the worst-case bound.

**GreedyAbsFAS**

In [24] a variation of GreedyFAS was proposed in which the greedy selection of a vertex is determined by the absolute value of the degree difference, $max(|d^+(u) - d^-(u)|)$, and the vertex is treated as a sink or source depending on whether the outdegree is greater than the indegree or vice versa. Unlike our GreedyFAS implementation where only the current maximum $\delta$-class needs to be tracked, the improved algorithm must maintain both the maximum and minimum $\delta$-classes that are currently occupied in order to efficiently make successive greedy selections for vertex processing. However, the approach did not offer a significant performance improvement as the difference between the FAS size computed by each approach was always less than 1%, often much less.

### 3.2.4 SimpleFAS

SimpleFAS is based on a very simple 2-approximation algorithm for the MAS problem. First, we fix an arbitrary permutation $P$ of the vertices of $G$. Then, we construct two subgraphs $L$ and $R$, containing the arcs $(u, v)$ where $u < v$ in $P$ and those where $u > v$ in $P$, respectively. After this construction, both $L$ and $R$ are acyclic subgraphs of $G$, and at least one of them is at least half the size of the maximum acyclic subgraph. Therefore, we can return $m - max(|L|, |R|)$ as the size of a feedback arc set for $G$. The runtime complexity of SimpleFAS is $O(m+n)$ and the pseudocode is presented in Algorithm 7.

As an illustrative example, consider the execution of SimpleFAS on the graph in

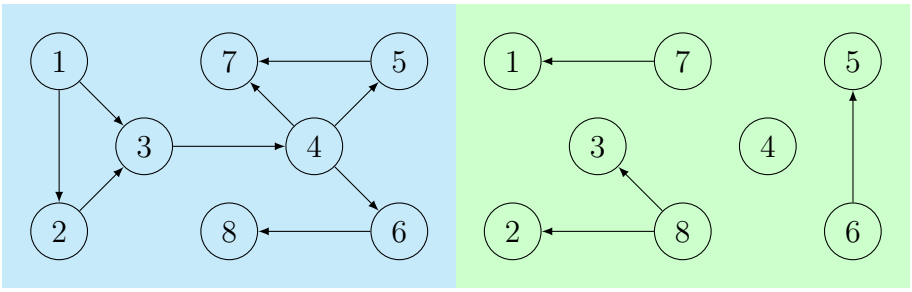



Figure 3.3: The subgraphs $L$ and $R$ resulting from the execution of SimpleFAS on the example graph.

Figure 3.1 with the permutation $[1, 2, 3, 4, 5, 6, 7, 8]$. Then, the construction of $L$ and $R$ yields the subgraphs shown in Figure 3.3. We have 9 arcs in $L$ and 4 arcs in $R$. Thus, we can extract a feedback arc set of size 4 made up of the arcs in subgraph $R$.

**Algorithm 7** simpleFAS

**Input:** Input directed graph $G = (V, E)$
**Output:** The cardinality of a feedback vertex set $\Psi$
Fix an arbitrary permutation of the vertices of $G$ and label them $1 \dots n$
$L \leftarrow \{(u, v) \in E \mid u < v\}$
$R \leftarrow \{(u, v) \in E \mid u > v\}$
**return** $|E| - max(|L|, |R|)$

The runtime complexity of SimpleFAS is $O(m+n)$. Our implementation efficiently permutes the vertices of $G$ using the Fisher-Yates shuffling algorithm [29] on the vertices of $G$.

### 3.2.5 BergerShorFAS

BergerShorFAS is based on an approximation algorithm for the MAS problem due to Berger & Shor [6]. Precisely, their approach builds on the known 2-approximation algorithm for the MAS problem that is presented in SimpleFAS. The algorithm begins by choosing a random permutation $P$ of the vertices of $G$. Then, the vertices are processed in order according to $P$ as follows. When processed, if a vertex has more

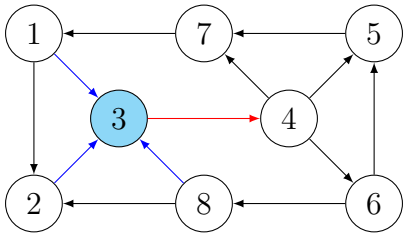
incoming arcs than outgoing ones, the incoming arcs are removed from the graph and added to a set $E'$ and the outgoing arcs are removed from the graph and discarded. If there are at least as many outgoing arcs as incoming arcs, instead the outgoing arcs are added to $E'$ and the incoming arcs are discarded and removed from $G$. When all the vertices have been processed, $G' = (V, E')$ is returned as an acyclic subgraph. Therefore, the arcs from $E \setminus E'$ make up a feedback arc set.

The intuition behind the above approach is as follows. At each step, selecting either the incoming or the outgoing arcs but not both ensures that the resulting graph is acyclic. Moreover, choosing at each step the set of arcs of bigger size, ensures that resulting acyclic graph has a large number of edges.

This randomized approach runs in time $O(m+n)$ and produces an acyclic subgraph containing at least $(1/2 + \Omega(1/\sqrt{d_{max}}))|E|$ arcs, where $d_{max}$ is the maximum vertex degree of $G$. In our experiments, BergerShorFAS far outperforms the worst-case bound provided.

As an illustrative example, consider the execution of Berger-Shor algorithm on the graph in Figure 3.1 with an initial ordering of $[3, 6, 4, 8, 7, 1, 2, 5]$. First, we process vertex 3 and observe that it has one outgoing arc and three incoming arcs. Thus, we add the incoming arcs $(1, 3)$, $(2, 3)$, and $(8, 3)$ to $E'$ before removing them and the outgoing arc $(3, 4)$ from $G$. Next, we process vertex 6 adding the arcs $(6, 5)$ and $(6, 8)$ to $E'$. This process continues adding the arcs $(4, 5)$, $(4, 7)$, $(8, 2)$, $(7, 1)$, and $(1, 2)$ to $E'$. Finally, we can return the arcs $E \setminus E' = [(3, 4), (4, 6), (5, 7)]$ as a feedback arc set of size 3. In Figure 3.4 we illustrate the execution of the algorithm when processing vertex 3 in the example graph $G$.

In Algorithm 8, we give BergerShorFAS, which adapts the above algorithm to compute the feedback arc set, $F$, directly without first computing $E'$. We do this




$$E' \leftarrow E' \cup \{(1,3),(2,3),(8,3)\}$$

Figure 3.4: Processing vertex 3 during the execution of the Berger-Shor algorithm on the example graph.

because maintaining $E'$ is more memory demanding than maintaining $F$.[1]

**Algorithm 8** BergerShorFAS

**Input:** Input directed graph $G = (V, E)$
**Output:** A feedback arc set for $G$

```
Fix an arbitrary permutation P of the vertices of G
F ← ∅
for all vertices v processed in order based on P do
    if inDegree(v) > outDegree(v) then
        F ← F ∪ {(v, u) : u ∈ G.succ(v)}
    else
        F ← F ∪ {(u, v) : u ∈ G.pred(v)}
    E ← E \ ({(v, u) : u ∈ G.succ(v)} ∪
             {(u, v) : u ∈ G.pred(v)})
return F
```

Notice that the BergerShorFAS algorithm manipulates the graph data structure during its execution by deleting arcs. However, since we use the webgraph framework, our graph data structure is in an immutable compressed state and therefore we are unable to delete parts of the graph in this way. Instead, we observe that in each iteration, all the arcs going-to or leaving-from a vertex $v$ are deleted. In essence this amounts to deleting $v$ itself. As such, in each iteration, we label vertices as deleted, but do not physically remove them from graph. We consider only those vertices that

[1] In applications, once a feedback arc set $F$ is computed, using the webgraph API, we can generate a new immutable compressed graph not containing the arcs in $F$ in only a single pass over the original graph $G$ while consulting $F$. This process needs memory mainly to hold $F$.

have not been labelled as deleted in the execution of the algorithm.

Furthermore, instead of marking both vertices and arcs as deleted, we only mark when a vertex is deleted and infer if an arc is deleted by checking to see if either endpoint has been labelled as deleted. In this way, we are able to achieve an equivalence with only a single auxiliary data structure of size $O(n)$ as opposed to two data structures of size $O(n)$ and $O(m)$, respectively, if we labelled arcs as well. We maintain the status of a vertex, deleted or present, via a bit set to minimize the space overhead.

### 3.2.6 dfsFAS

The hallmark algorithm for graph traversals, depth-first search (DFS), can be used to compute a feedback arc set. Removing all the *back arcs* of a depth-first traversal ensures the resulting graph is acyclic. We follow the standard vertex colouring approach to identifying the back arcs. The runtime complexity of dfsFAS is $O(m + n)$.

Note, if the number of back arcs exceeds half the number of arcs in $G$, we can instead return the complementary arcs as a FAS since the back arcs of a DFS cannot contain a cycle. Recall that in the DFS ordering a back arc points from a vertex with a higher label to a smaller one. Therefore, by transitivity, we cannot have a cycle among back arcs.

Observe that DFS has the ability to output a FAS as a side effect of its execution, but at no point in the execution does it make any intelligent decisions which act to minimize the resulting FAS size. As such, the SimpleFAS and dfsFAS approaches serve as baseline algorithms in our experiments.

As an illustrative example, consider the execution of dfsFAS on the graph in Figure 3.1 initialized at vertex 1 as shown in Figure 3.5. The arcs are coloured blue for tree arcs, red for back arcs, green for forward arcs and orange for cross arcs.

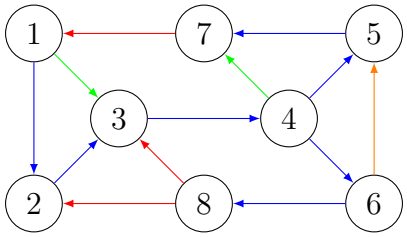



Figure 3.5: The execution of dfsFAS on the example graph and the resulting arc colourings.

Note, neighbours are visited in ascending order. We see that after the execution of a depth-first traversal there are 3 back arcs which we return as a feedback arc set for $G$.

Initially each vertex is coloured `WHITE` to indicate they have not yet been discovered. The process begins by selecting one vertex $u$ from the graph and colouring it `GREY` to indicate that the vertex has been discovered, but not yet finished. For each neighbour, $w$, of $u$ that is coloured `WHITE` the procedure moves to $w$ recursively. Finally, if no `WHITE` neighbours of $u$ exist, then $u$ is coloured `BLACK` and the procedure returns to $u$'s ancestor. During the exploration of $u$'s neighbours, if a `GREY` node is encountered then the arc is labelled as a back arc. The pseudocode is presented in Algorithm 9.

**Algorithm 9** dfsFAS

**Input:** The current node $v$
  $colour(v) \leftarrow GREY$
  **for all** vertices $w$ in $G.successors(v)$ **do**
    **if** $colour(w) = WHITE$ **then**
      $dfsFAS(G, w)$
    **else if** $colour(w) = GREY$ **then**
      label $(v, w)$ as a back edge
  $colour(v) \leftarrow BLACK$

Our dfsFAS implementation maintains a colour array to store the colour of each vertex as an `int` array with 0 representing `WHITE` and 1 and 2 representing `GREY` and

`BLACK`, respectively. The use of an `int` array minimizes the space overhead and allows for quick colour look-up. The runtime complexity of dfsFAS is $O(m + n)$.

### 3.2.7 KwikSortFAS

The KwikSortFAS heuristic, originally introduced by Ailon et al. [2] as a 3-approximation algorithm for the FAS problem on tournaments, was later extended by Brandenburg & Hanauer [17] as a heuristic for general directed graphs.

Brandenburg and Hanauer [17] extend several classical sorting algorithms to heuristics for the FAS problem using the underlying idea that the vertices of a directed graph $G$ can be treated as items to be sorted into a favourable linear arrangement based on the number of back arcs induced. These sorting-based approaches are applied to an initial linear arrangement and output a sorted linear arrangement.

The motivation for KwikSortFAS is based on the classical sorting algorithm Quicksort. The algorithm uses the 3-way partition variant of Quicksort due to it being highly adaptive in the case of sorting with many equal keys. In our application, two vertices without an arc connecting them are treated as equal when making a comparison and thus we have many equal items. Our implementation follows the optimized version due to Sedgewick and Wayne [81] which only uses $O(\log n)$ additional space.

The application of Quicksort to the vertices of $G$ is implemented as follows. Given a starting linear arrangement, we move the vertices to the left or to the right relative to a random pivot element based on whether there is an arc to or from the pivot. The algorithm then proceeds recursively. We put the pivot vertex and the vertices equal to it, with no arc from or to the pivot, in the middle and recurse on the left, middle, and right subsets. Note that unlike in the case of sorting numbers, we need to recurse on the middle since these vertices, though equal to the pivot, could have arcs between them and hence may not be equal to each other. When ties must be

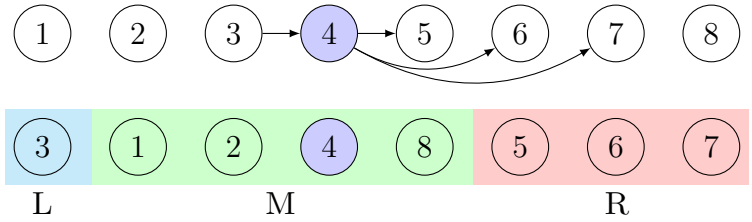



Figure 3.6: Initial recursive step of KwikSortFAS on the example.

broken, in the case where the remaining vertices are disconnected, their order is left unaltered. The pseudocode for KwikSortFAS is presented in Algorithm 10.

As an illustrative example, consider the execution of KwikSortFAS on the graph in Figure 3.1 with an initial ordering of $[1, 2, 3, 4, 5, 6, 7, 8]$. In the first level of recursion we randomly select vertex 4 as the pivot. This places vertex 3 to the left, vertices 1, 2, 4, and 8 in the middle and vertices 5, 6, and 7 to the right. In Figure 3.6 we illustrate the execution of the KwikSortFAS algorithm on the initial arrangement to show how the recursion proceeds. In this first step we see how the vertices are split into the three categories less than (L), equal to (E), and greater than (G) the pivot determined by the arcs incident on the pivot vertex 4. Next we recurse on each part.

We first recurse on the left, which only contains a single vertex and thus returns immediately. Next we recurse on the middle and randomly select vertex 2 as the pivot. This places vertices 1 and 8 to the left and vertices 2 and 4 in the middle with the right left empty. We recurse to the left and leave vertices 1 and 8 both in the middle as there are no arcs between them. The same is true when we recurse to the middle with vertices 2 and 4. Finally, we recurse to the right and place vertex 5 to the left and vertices 6 and 7 in the middle. The resulting sorted ordering after the recursive calls return is $[3, 1, 8, 2, 4, 5, 6, 7]$. Thus, we can extract a feedback arc set of size 6 by observing that this linear arrangement has six backward arcs.

The runtime complexity of $O(n \log n)$ for KwikSortFAS assumes that arc membership can be tested in constant time, i.e. when the graph is represented with an

**Algorithm 10** KwikSortFAS

**Input:** Linear arrangement $A$, vertex $lo$, vertex $hi$
  **if** $lo < hi$ **then**
    $lt \leftarrow lo$, $gt \leftarrow hi$, $i \leftarrow lo$
    $p \leftarrow$ random pivot in range $[lo, hi]$
    **while** $i \leq gt$ **do**
      **if** arc $(i, p)$ exists **then**
        $swap(lt, i)$
        $lt \leftarrow lt + 1$, $i \leftarrow i + 1$
      **else if** arc $(p, i)$ exists **then**
        $swap(i, gt)$
        $gt \leftarrow gt - 1$
      **else**
        $i \leftarrow i + 1$
    $KwikSortFAS(A, lo, lt - 1)$
    **if** at least one swap was made **then**
      $KwikSortFAS(A, lt, gt)$
    $KwikSortFAS(A, gt + 1, hi)$

adjacency matrix. However, given that our graph data structure uses an adjacency list we must search the adjacency list in order to test for the presence of an arc. Conveniently, the adjacency list for each vertex is kept in sorted order and therefore we can utilize a binary search to test for the presence of an arc in $G$. With an adjacency list representation, the runtime complexity becomes $O(n \log n \log(d_{max}))$ where $d_{max}$ is the maximum vertex degree in $G$.

Furthermore, since KwikSortFAS is randomized, each run may yield a different result. To this end, as presented in [17], we also consider KwikSort200FAS which runs KwikSortFAS 200 times on random initial linear arrangements and takes the best result.

### 3.2.8 InsertionSortFAS

We include the following definitions that apply to the Insertion Sort based algorithms introduced in [17].

**Definition 4.** *An algorithm A for the linear arrangement problem is* monotone *if the number of back arcs of the linear arrangement output by A is never greater than the number of back arcs of the initial linear arrangement.*

Therefore, an algorithm for the linear arrangement problem that is *monotone* can be run repeatedly using the output of one execution as the input to the next in order to improve the resulting feedback arc set.

**Definition 5.** *For a monotone algorithm A the convergence number is the number of runs of A that strictly improve the result. The resulting algorithm is denoted* $A^*$.

### SortFAS

SortFAS is equivalent to sorting by insertion for the linear arrangement problem. The vertices are processed in order according to an ordering of the vertices, $(v_1 \dots v_n)$. In the $i$-th iteration $v_i$ is inserted at the optimal position among the already sorted set of the first $i-1$ vertices. In case of a tie the leftmost position is taken. The optimal position is defined as the position with the least number of backward arcs induced by $v_i$. Notice that only the arcs between $v_i$ and the first $i-1$ vertices are relevant in the $i$-th iteration of SortFAS.

As an illustrative example, consider the execution of SortFAS on the graph in Figure 3.1 with an initial ordering of $[1, 2, 3, 4, 5, 6, 7, 8]$. In the first iteration, the single vertex 1 is trivially sorted. In the second iteration, vertex 2 is inserted to the right of vertex 1, i.e. its position is unchanged, as there is an arc from vertex 1 to 2 and thus swapping their locations would induce a backward arc. This behaviour continues in iterations 3 through 5. Then, in iteration 6 we insert vertex 6 in between vertices 4 and 5 as this placement induces 0 backward arcs. In iteration 7 we again leave vertex 7 in place. Finally, in iteration 8, we insert vertex 8 in between vertices 1 and 2 which reduces the number of backward arcs induced from 2 to 1. The resulting

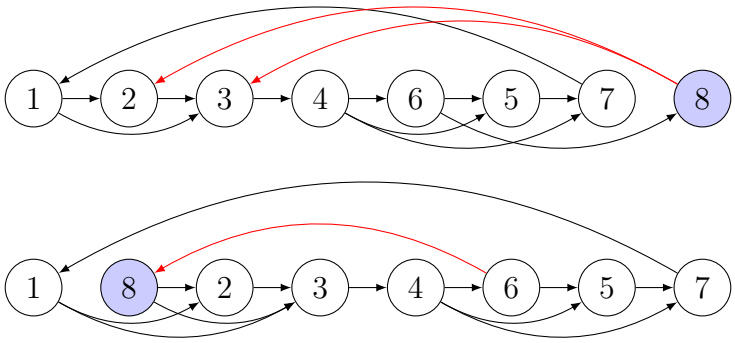



Figure 3.7: Iteration 8 of SortFAS on the example graph.

sorted arrangement is $[1, 8, 2, 3, 4, 6, 5, 7]$. Thus, we can extract a feedback arc set of size 2 by observing that this linear arrangement has two backward arcs: $(7, 1)$ and $(6, 8)$. In Figure 3.7 we show the execution of SortFAS in iteration 8 to provide some insight into the selection of the position that induces the least number of backward arcs. In this iteration, we see that vertex 8 in its original location induced two backward arcs, $(8, 2)$ and $(8, 3)$ shown in red, and after it is inserted at position 2, it only induces a single backward arc, $(6, 8)$.

Now, while the runtime of a traditional Insertion Sort is $O(n^2)$, we must take care in the analysis when considering sorting graph vertices. In particular, in a standard Insertion Sort, we repeatedly compare the item at position $i$ to the left until a smaller value is reached. The comparison done at each stage is typically a simple arithmetic operation, as in the case of integer values, but requires more thought when comparing two vertices in $G$. A naive approach is to consider inserting $v_i$ at each of the possible $i - 1$ locations. However, this would require a pass over the first $i - 1$ vertices to count the number of back arcs induced for each of the $i - 1$ possible locations. The resulting sum is $\sum_{i=1}^{n}(i-1)^2 \in O(n^3)$.

Instead, we engineer a more sophisticated approach that can identify the optimal location of $v_i$ using a single pass over the $i - 1$ possible locations. We begin by initializing a counter variable to zero. Then, for each possible location, $j$, we determine

if there is an arc from $v_i$ to $v_j$ and from $v_j$ to $v_i$. We increment, or decrement, the counter variable if the arc from $v_i$ to $v_j$, or $v_j$ to $v_i$, is present, respectively. This process assumes that $v_i$ and $v_j$ will swap locations and keeps tracks via the counter variable whether or not any current arcs between $v_i$ and $v_j$ would switch direction by inserting $v_i$ at position $j$. For example, if there is currently a backward arc from $v_i$ to $v_j$ then, if $v_i$ is inserted at position $j$, the arc will become a forward arc. Thus, incrementing and decrementing the counter variable indicates the potential loss or gain of a backward arc. Therefore, we can identify the minimum value achieved by the counter variable and record the value of $j$ for which this minimum was achieved to use as the optimal location to insert $v_i$ at since the minimum value of the counter variable corresponds to the location which induces the least number of backward arcs. The pseudocode for SortFAS is presented in Algorithm 11.

**Algorithm 11** SortFAS

```
Input: Linear arrangement A
  for all vertices v in A do
      val ← 0, min ← 0, loc ← position of v
      for all positions j from loc − 1 down to 0 do
          w ← vertex at position j
          if arc (v, w) exists then
              val ← val − 1
          else if arc (w, v) exists then
              val ← val + 1
          if val ≤ min then
              min ← val, loc ← j
      insert v at position loc
```

The above implementation of SortFAS has a runtime complexity of $O(n^2)$ under the assumption that arc membership can be tested in constant time. With an adjacency list implementation the runtime complexity of SortFAS becomes $O(n^2 \log(d_{max}))$. Furthermore, SortFAS is monotone which yields the additional algorithm SortFAS*.

**SiftFAS**

SiftFAS, similar to SortFAS, is equivalent to two-sided Insertion Sort for the linear arrangement problem since we can place a vertex $v$, not only in the first $i-1$ positions, but on either side of $v$'s current location. Here, in the $i$-th iteration, $v_i$ is inserted at the optimal position across the entire current linear arrangement and in the case of a tie the leftmost position is taken. Additionally, the same technique used for SortFAS to determine the optimal location to insert $v_i$ can be utilized to ensure the running time is equivalent to that of a traditional two-sided insertion sort.

SiftFAS also has a runtime complexity of $O(n^2)$ and is monotone which yields the additional algorithm SiftFAS*. Similiar to SortFAS, the runtime complexity given for SiftFAS assumes constant time arc testing. For an adjacency list representation, the runtime complexity is $O(n^2 \log(d_{max}))$.

## 3.3 Experiments

In this section, we present our experimental results which have the following results:

- Within each of the three classes of algorithms, we observe an overall trend showing a trade off between scalability and quality.
- We observe an approximate maximum scalability of 300K arcs for the $O(n^2)$ algorithms, 3.5M arcs for the $O(n \log n)$ algorithms, and 50B arcs for the $O(m+n)$ algorithms.
- We achieve approximate FAS sizes of 3-20%, 23-40%, and 11-17% for the best algorithms of the $O(m+n)$, $O(n \log n)$, and $O(n^2)$ runtime categories, respectively.

- GreedyFAS and BergerShorFAS provide the best balance between scalability and quality. GreedyFAS is the algorithm that produces always either the smallest or a very close second smallest FAS size while being a fast algorithm in general. In particular our array implementation of it scales to the biggest dataset we consider, clueweb12, with more than 42 billion arcs.

All of our algorithms are implemented in Java (available at `https://github.com/stamps/FAS`) and tested on a machine with dual 6 core 2.10GHz Intel Xeon CPUs, 32GB RAM and running Ubuntu 14.04.2.

The network statistics for all of the datasets we consider are shown in Table 3.2. We obtained the datasets from Laboratory of Web Algorithmics.[2] We divide the datasets by horizontal lines according to their size, small (S), medium (M), large (L), and extra-large (XL).

The algorithms are summarized in Table 3.3. The last column gives the sizes of the datasets that each algorithm can handle. The SimpleFAS and dfsFAS approaches serve as baseline algorithms in our experiments.

The measures of effectiveness in our experiments are the *FAS size* defined as the number of arcs in a FAS output by a particular algorithm and the *algorithm efficiency* measured in running time. Our goal is to keep both parameters as small as possible.

### 3.3.1 Small Datasets

The small datasets we consider are the *word_assoc* network with 10,617 vertices and 72,172 arcs, and the *enron* dataset with 69,244 vertices and 276,143 arcs.

First, we plot the size of the FAS computed by each of the algorithms considered in Figures 3.8a and 3.8b. We observe that dfsFAS, SimpleFAS, and the KwikSort-based algorithms perform the worst compared to the rest of the algorithms. GreedyFAS

[2] `http://law.di.unimi.it/datasets.php`

| Name | $\|V\|$ | $\|E\|$ | Size (Gb) |
|---|---|---|---|
| word_assoc | 10,617 | 72,172 | 0.235 |
| enron | 69,244 | 276,143 | 0.721 |
| uk-2007 | 100,000 | 3,050,615 | 1.764 |
| cnr-2000 | 325,557 | 3,216,152 | 3.328 |
| uk-2002 | 18,520,486 | 298,113,762 | 220.945 |
| arabic-2005 | 22,744,080 | 639,999,458 | 342.399 |
| uk-2005 | 39,459,925 | 936,364,282 | 514.289 |
| webbase-2001 | 118,142,155 | 1,019,903,190 | 1,207.959 |
| twitter-2010 | 41,652,230 | 1,468,365,182 | 5,286.142 |
| clueweb12 | 978,408,098 | 42,574,107,469 | 23,830.734 |

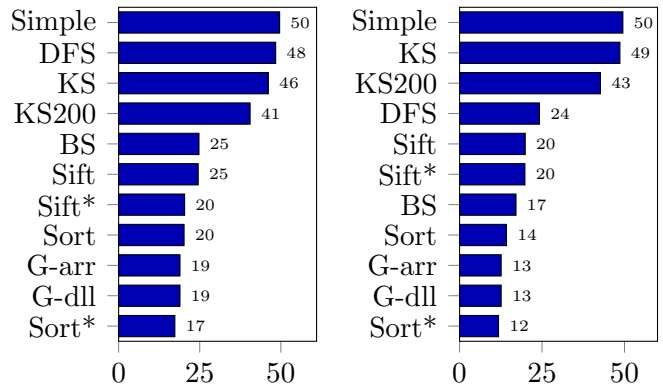
Table 3.2: Dataset Statistics

| Algorithm | Abbrev. | Section | Complexity | Type | Dataset-Size |
|---|---|---|---|---|---|
| SortFAS | Sort | 3.2.8 | $O(n^2)$ | Sorting | S |
| SortFAS* | Sort* | 3.2.8 | $O(n^2)$ | Sorting | S |
| SiftFAS | Sift | 3.2.8 | $O(n^2)$ | Sorting | S |
| SiftFAS* | Sift* | 3.2.8 | $O(n^2)$ | Sorting | S |
| KwikSortFAS | KS | 3.2.7 | $O(n \log n)$ | Sorting | S, M |
| KwikSortFAS200 | KS200 | 3.2.7 | $O(n \log n)$ | Sorting | S, M |
| GreedyFAS (dll) | G-dll | 3.2.3 | $O(m+n)$ | Greedy | S, M, L |
| GreedyFAS (array) | G-arr | 3.2.3 | $O(m+n)$ | Greedy | S, M, L, XL |
| SimpleFAS | Simple | 3.2.4 | $O(m+n)$ | Randomized | S, M, L, XL |
| BergerShorFAS | BS | 3.2.5 | $O(m+n)$ | Randomized | S, M, L, XL |
| dfsFAS | DFS | 3.2.6 | $O(m+n)$ | Traversal | S, M, L, XL |

Table 3.3: Algorithms considered

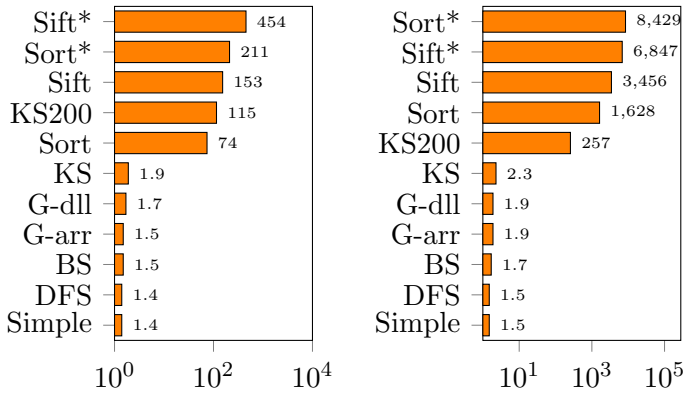
and Insertion Sort-based algorithms perform well, with feedback arc sets in the range 17-25% for *word_assoc* and 12-20% for *enron*. Both SortFAS* and SiftFAS* required 4 iterations to converge. We see that even though the best algorithm with respect to the size of FAS is SortFAS*, GreedyFAS is very close. BergerShorFAS computed a feedback arc set that was 25% and 17% of the size of the network for *word_assoc* and *enron*, respectively.

Second, we plot the runtimes achieved by each algorithm in Figures 3.9a and 3.9b. Note that the runtime plots use a logarithmic scale, illustrating how much faster the non-sorting based algorithms execute. Each of the sorting based algorithms, with the




(a) word_assoc (pct) (b) enron (pct)

Figure 3.8: FAS size for small datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.




(a) word_assoc (sec) (b) enron (sec)

Figure 3.9: Running time for small datasets.

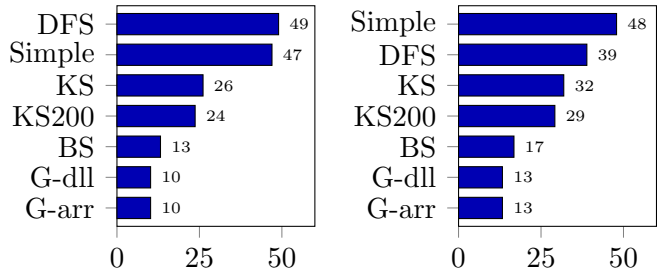
exception of KwikSortFAS, required at least 70 seconds to complete on the *word_assoc* network. The difference is even more severe on the *enron* network where dfsFAS, SimpleFAS, BergerShorFAS, KwikSortFAS, and GreedyFAS each complete in about 2 seconds followed by a jump up to 257 seconds for KwikSort200FAS, 1,628 for SortFAS, 3,456 for SiftFAS, 6,847 seconds for SiftFAS* with 2 iterations, and 8,429 seconds for SortFAS* with 5 iterations.

In summary, on the small datasets, the feedback arc set of minimum size was computed by SortFAS*, followed closely by GreedyFAS. On the other hand, the latter

was more than one order of magnitude faster than the former. As such, GreedyFAS provides simultaneously a good quality FAS and a small running time. Overall, the quickest algorithms were the suite of non-sorting algorithms in addition to the single pass KwikSortFAS algorithm.

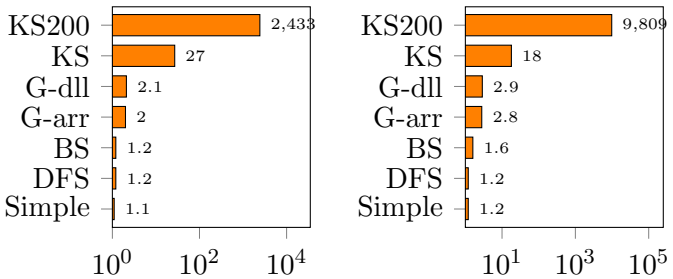
### 3.3.2 Medium Datasets

The medium datasets we consider are the *uk-2007* network containing 100,000 vertices and 3,050,615 arcs and the *cnr-2000* network containing 325,557 vertices and 3,216,152 arcs. On the medium sized datasets, the SortFAS and SiftFAS algorithms and their converging versions fail to complete in a reasonable amount of time and are therefore omitted from the following plots.




(a) uk-2007 (pct) (b) cnr-2000 (pct)

Figure 3.10: FAS size for medium datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.




(a) uk-2007 (sec) (b) cnr-2000 (sec)

Figure 3.11: Running time for medium datasets.

First, we plot the size of the FAS computed by each of the algorithms considered in Figures 3.10a and 3.10b. We see that the dfsFAS, SimpleFAS, and the KwikSort based algorithms perform the worst. For the *uk-2007* network, dfsFAS and SimpleFAS compute a feedback arc set in the range 47-50% of the size of the network, while the KwikSort based algorithms compute feedback arc sets in the range $24 - 26\%$. In contrast, GreedyFAS and BergerShorFAS achieve impressively small feedback arc sets of 10% and 13%, respectively. We observe very similar results for the *cnr-2000* network.

Second, we plot the runtimes achieved by each algorithm in Figures 3.11a and 3.11b. Again, we employ a logarithmic scale to best illustrate the substantial gap in runtimes among the tested algorithms.

We observe that the KwikSort-based algorithms are from one to several orders of magnitude slower than the non-sorting-based algorithms. The latter all completed in under 3 seconds on both networks with dfsFAS, SimpleFAS, and BergerShorFAS each about a second quicker than the GreedyFAS implementations.

In summary, on the medium datasets, as we move beyond the capabilities of the Insertion Sort-based algorithms, we observe the feedback arc set of minimum size being computed by GreedyFAS, followed by BergerShorFAS. With respect to runtimes, dfsFAS, SimpleFAS, and BergerShorFAS narrowly beat GreedyFAS, with all algorithms running in a very reasonable amount of time.

### 3.3.3 Large and Extra-Large Datasets

The largest, or web-scale, datasets we consider are *uk-2002*, *arabic-2005*, *uk-2005*, *webbase-2001*, *twitter-2010*, and *clueweb12*. On these datasets, the KwikSort-based algorithms fail to complete in a reasonable amount of time and are therefore omitted from the following plots, leaving us with only those algorithms that run linear in the

number of arcs and vertices of the graph.

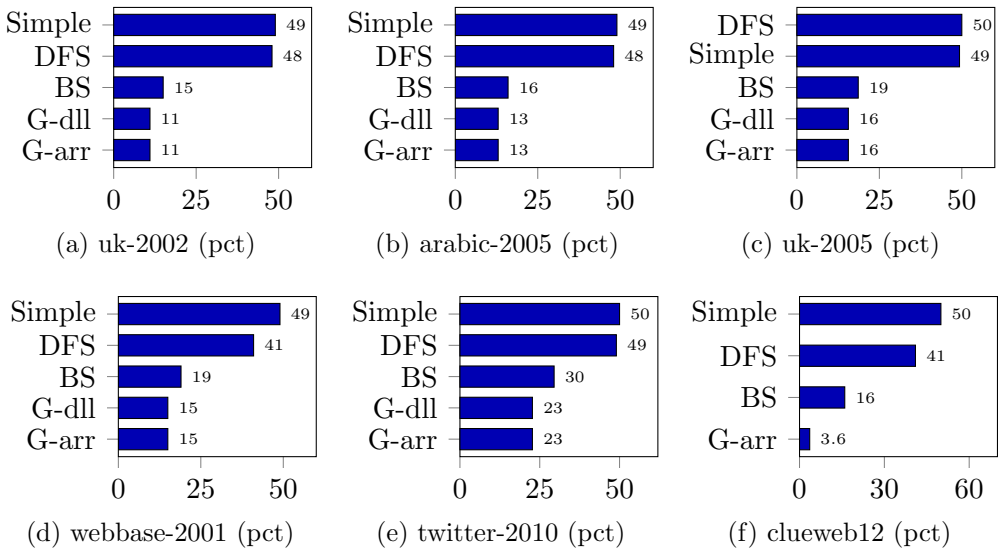



Figure 3.12: FAS size for large and extra-large datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.

We plot the size of the FAS computed by each of the algorithms considered in Figure 3.12 and the runtimes achieved by each algorithm in Figure 3.13. On these datasets we observe a consistent trade-off between running time and feedback arc set size. dfsFAS and SimpleFAS complete the quickest across all the datasets, followed closely by BergerShorFAS and then by GreedyFAS.

On *clueweb12* because of its massive size, only the array-based implementation of GreedyFAS could run with the available memory.

GreedyFAS trades running time for feedback arc set quality. It outperforms all of the tested algorithms on every dataset with respect to the FAS size. The dfsFAS and SimpleFAS algorithms produce unacceptably large feedback arc sets that are approximately 50% of the size of the network while GreedyFAS is around 15% on average. Furthermore, GreedyFAS beats its closest competitor, BergerShorFAS, by about 4%.

Remarkably, on *clueweb12*, a dataset of more than 42 billion arcs, GreedyFAS

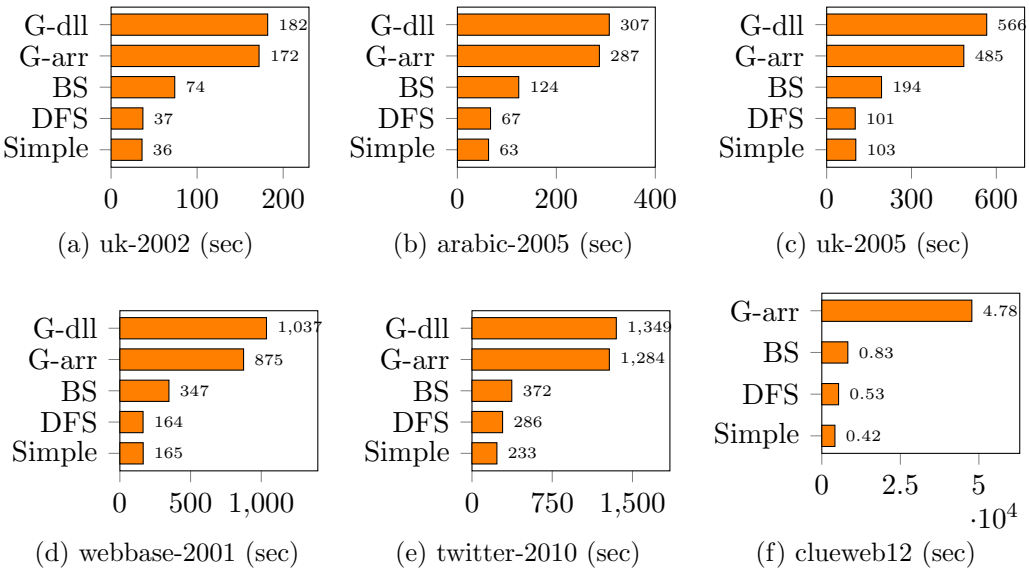



Figure 3.13: Running time for large and extra-large datasets.

achieves an impressive 3.6% FAS size, compared to BergerShorFAS with 16%. Being able to scale greedyFAS to such an extra-large dataset is a significant contribution of this chapter.

## 3.4 Discussion

In this section, we summarize and discuss the most important results of the presented experiments. In particular, we provide insight into the expected and actual performance of the algorithms both independently and relative to each other.

First, we discuss when we expect each approach to perform well independently and provide global graph properties that suggest when a particular approach will perform well. Both the GreedyFAS and BergerShorFAS approaches benefit when there are many source and sink-like vertices present in $G$. Therefore, indication of when they will perform well can be extracted from the $\delta$-values of the top-$k$ vertices. Recall that during the execution of BegerShorFAS the vertices are processed in a

| Name | $F_{topk}$ | $s \leq 0.1$ | $s \in (0.1, 0.4)$ | $s \geq 0.4$ | $f \leq 0.1$ | $f \in (0.1, 0.4)$ | $f \geq 0.4$ | L-$\lvert M \rvert$ |
|---|---|---|---|---|---|---|---|---|
| word_assoc | 19.40 | 0.3208 | 0.6792 | 0.0 | 0.5402 | 0.3552 | 0.1348 | 0.4789 |
| enron | 74.30 | 0.3035 | 0.4682 | 0.2298 | 0.8577 | 0.0924 | 0.0557 | 0.5156 |
| uk-2007 | 63.42 | 0.8250 | 0.1690 | 0.0060 | 0.5242 | 0.3730 | 0.1277 | 0.3750 |
| cnr-2000 | 62.40 | 0.4104 | 0.3853 | 0.2061 | 0.3779 | 0.3590 | 0.3001 | 0.5059 |
| uk-2002 | 60.98 | 0.7086 | 0.1664 | 0.1253 | 0.4088 | 0.4242 | 0.2048 | - |
| arabic-2005 | 79.94 | 0.5414 | 0.1296 | 0.3292 | 0.4303 | 0.4359 | 0.1671 | - |
| uk-2005 | 41.98 | 0.2748 | 0.4491 | 0.2769 | 0.3969 | 0.4238 | 0.2146 | - |
| webbase-2001 | 60.16 | 0.5291 | 0.2749 | 0.1977 | 0.3846 | 0.3723 | 0.2872 | - |
| twitter-2010 | 98.23 | 0.2049 | 0.3683 | 0.4271 | 0.3143 | 0.4365 | 0.2913 | - |
| clueweb12 | 59.19 | 0.4920 | 0.3025 | 0.2058 | 0.5249 | 0.3675 | 0.1299 | - |

Table 3.4: Additional Dataset Statistics. The fraction (in %) of total arcs accounted for by the top-$k$ vertices is given by $F_{topk}$ while the skew of the top-$k$ vertices is given by $s$. The fraction of vertices added to the FAS in each iteration of BS is given by $f$. Finally, L-$|M|$ represents the fraction of recursive iterations in KS that have $|M| > 0.9n$.

random order. However, since the top-$k$ vertices have such large neighbourhoods, we expect the neighbourhood of a top-$k$ vertex $u$ when it is processed to be similar to its initial neighbourhood. It is unlikely that a significant fraction of $u$'s neighbours have been removed from $G$ by the time $u$ is processed due to the sheer number of them. Therefore, we can look at the $\delta$-values of the top-$k$ vertices as an indication of whether or not a large number of arcs will be added to the FAS from such vertices. Note that for the networks considered, the top 1% of vertices by degree comprise, on average, 60% of the arcs of the entire network with a peak of 98% for twitter-2010. Thus, if the $\delta$-values for these vertices are favorable ($|\delta| \gg 0$) then we can expect a small fraction of the top-$k$ vertices arc's to be included in the FAS. In Table 3.4 we show the percentage of total arcs accounted for by the top 1% of vertices and the distribution of their *skews* defined as the lesser of the indegree and outdegree of a vertex $u$ as a fraction of $u$'s total degree. Note, a small skew corresponds to a large absolute $\delta$-value and a very source/sink-like vertex.

Unlike traditional numerical sorting problems, where there is a total ordering on the data, a difficulty in applying sorting techniques to the FAS problem is a lack

of transitivity which sorting algorithms are designed to exploit. In fact, real-world networks are far from exhibiting a total ordering because of their sparsity (typically $m \ll n^2$). Therefore, we end up applying these sorting techniques in a setting that not only lacks a total ordering and transitivity, but is far from exhibiting such properties. Instead, we would expect the sorting-based approaches to perform best on graphs that exhibit nearly total orderings which necessitates a much higher density. Consider the KwikSortFAS algorithm, in which the order of equal vertices is left unaltered. We can expect that in sparse graphs there will be many vertices determined to be equal in each iteration which can lead to poor performance since large subsections of the ordering will not be modified in a meaningful way. This fact is highlighted in Table 3.4 where the fraction of iterations of KwikSortFAS for which the size of the equal items is greater than 90% of the total vertices in $G$ is presented.

Unfortunately, for traversal-based approaches there are no global graph properties we can point to that would indicate the resulting performance. This is simply due to the fact that the traversal-based approaches have the ability to output a FAS as a side effect of their execution, but at no point in their execution do they make any intelligent decisions which act to minimize the resulting FAS size.

Second, we will discuss how we expect the various approaches to perform relative to each other and compare this to the experimental results. As mentioned previously, BergerShorFAS and GreedyFAS both function around $\delta$-values, but GreedyFAS has a distinct advantage in that it updates the $\delta$-values during execution. This leads to better decision making by ensuring the processing of the most sink and source-like vertices first. Furthermore, notice that BergerShorFAS always adds some number of arcs to the FAS for non-source/sink vertices. In contrast, GreedyFAS has the potential to avoid adding arcs to the FAS depending on the previous vertices added to the vertex ordering. That is, GreedyFAS may append a non-source/sink vertex $u$

to the ordering for which all outgoing arcs point to vertices to the right of $u$ in the ordering and all incoming arcs point to vertices to the left of $u$. Thus, we expect GreedyFAS to outperform BergerShorFAS which is exactly what we observe in our experiments.

In comparing GreedyFAS and the sorting-based approaches, notice that GreedyFAS can be considered a 2-sided selection sort. However, GreedyFAS relies less on the transitivity property since it makes local greedy choices. Therefore, in sparse networks, we expect GreedyFAS to outperform the other sorting-based approaches which happens to be the case with the exception of the SortFAS* algorithm. Furthermore, while GreedyFAS builds up an ordering from scratch, the other sorting-based approaches manipulate an existing ordering. The resulting pitfall is that the sorting-based approaches can get stuck in local minima.

Finally, we investigated the effect of two graph properties related to social networks: power-law degree distribution and the small-world phenomenom. We constructed synthetic networks using iGraph in the R programming language on 10,000 vertices according to the preferential attachment model [14] for power-law degree distribution and the Watts-Strogatz model [88] for the small-world effect. In the Watts-Strogatz model we vary the re-wiring probability from $5 - 25\%$ and observe a linear decrease in the size of the FAS output by both GreedyFAS and BergerShorFAS. The FAS computed by GreedyFAS ranges from $39 - 27\%$ and $43 - 36\%$ for BergerShorFAS. Observe that such a decrease lines up with our intuition since re-wiring leads to larger $|\delta|$-values for the vertices of $G$. In the preferential attachment model we did not observe any meaningful difference across a wide range of parameter values.

In summary, the only algorithms that were able to scale to large and extra-large graphs were those that ran in time linear in the number of vertices and arcs of the graph. Furthermore, within the complexity class of linear algorithms the GreedyFAS

algorithm was unable to scale with a direct implementation of the original algorithm and was only able to complete in a reasonable amount of time due to our careful algorithmic engineering. As such, any new method for computing a feedback arc set that aspires to scale should be developed with a linear runtime complexity.

With regard to approximation ratios, each of the algorithms with performance guarantees computed feedback arc sets well below the guaranteed size. Therefore, it is important not to disregard a proposed heuristic based on its worst-case approximation ratio alone. Additionally, it does not appear that randomization (i.e. BergerShorFAS, simpleFAS, KwikSortFAS) provides a distinct advantage over deterministic approaches with no real pattern to the results based on randomization alone.

Interestingly, unlike the results of [17] regarding the sorting-based approaches, we found that the SortFAS algorithm computed a smaller FAS on the small datasets than the SiftFAS algorithm. However, the experiments from [17] make use of synthetic graphs which are not guaranteed to match the network structure of the real graphs and the different network structure may play a role in the performance of each approach. It is an open problem to understand how the structure of the network influences the performance of different approaches to the FAS problem.

## 3.5 The Probabilistic Case

Traditionally, much of the investigation into the FAS problem from the practitioner's perspective has focused on the unweighted case. However, large datasets often contain information that is uncertain (probabilistic) in nature. For example, in large social networks, the arc probability may denote the accuracy of a *link prediction* [62], or the influence of one user on another, e.g., in *viral marketing* [52]. Uncertainty can also be injected intentionally for obfuscating the identity of users for privacy reasons

[11]. Our confidence in such relations is commonly quantified using probability, and we say that the relation exists with a probability of existence $p$. In this section, we consider *probabilistic graphs* (also called uncertain graphs), whose arcs are labeled with a probability of existence. Probabilistic graphs have been used extensively in modeling, for example, communication networks, social networks, protein interaction networks, and regulatory networks in biological systems.

More formally, let $\mathcal{G} = (V, E, p)$ be a probabilistic graph, where $p : E \to [0, 1]$ is the function that assigns a probability of existence to each arc $e \in E$. Following the literature (cf. [56, 60]), we assume that the existence of different arcs are mutually independent events. A probabilistic graph is a probability distribution over $2^m$ deterministic graphs, each of which is a subgraph of the directed graph $(V, E)$. The set of possible deterministic graphs is called the set of "possible worlds" of $\mathcal{G}$. In a probabilistic graph, the expected number of arcs in a possible world $G = (V, E_G)$ of $\mathcal{G}$ is given by $\mathbb{E}(|E_G|) = \sum_{e \in E} p(e)$. Thus, removing a feedback arc set from a probabilistic graph $\mathcal{G}$ ensures that all possible worlds of $\mathcal{G}$ will be acyclic with the exact weight associated with a particular world's feedback arc set depending on the exact composition of arcs that are realized. Then, we can compute the expected number of arcs in a FAS $F$ as $\mathbb{E}(|F|) = \sum_{e \in F} p(e)$.

Traditionally, much of the investigation into the FAS problem from the practitioner's perspective has focused on the unweighted case while the work in the theory community addresses approximation guarantees in both the unweighted and weighted case. However, recent work on social network analysis, such as the seminal work by Kleinberg et al. [52] relating to influence maximization in a social network, naturally lends itself to representing the network by a weighted graph. In this context, a social network can be modeled as a directed graph with the users represented by the vertices of $G$. Then, each arc $(u, v)$ has an associated weight, $p_{u,v} \in [0, 1]$, which is used to

model the direct influence $u$ has on $v$.

### 3.5.1 Algorithms

We investigate adaptations of the most promising class of algorithms from the previous section to apply to probabilistic graphs since, to the best of our knowledge, there are no specialized algorithms for the weighted feedback arc set problem in the literature that run in linear time. We consider those $O(m+n)$ algorithms that present the opportunity to take arc probabilities into consideration during execution. For example, we do not consider SimpleFAS since it has no opportunity to take actions based on arc probabilities and therefore the execution on a probabilistic graph would not differ. On the other hand, we consider versions of the GreedyFAS and BergerShorFAS algorithms for probabilistic graphs.

#### Probabilistic Greedy (pG)

Recall that in the standard GreedyFAS approach we compute a delta class, $\delta(u)$, for each vertex $u \in G$. In the unweighted case, by definition, $\delta(u)$ only takes on integer values and therefore we have an exact expression for the number of possible $\delta$-classes. However, in the probabilistic case, a natural extension to the original approach would be to compute $\delta(u)$ as $w^+(u) - w^-(u)$ where $w^+(u)$ and $w^-(u)$ are the sum of the out and in-probabilities of $u$, respectively. As such, $\delta(u)$ gives the expected difference of the outdegree from the indegree of $u$.

Unfortunately, the above adaptation of $\delta(u)$ leads to the set of possible $\delta$-class values as being any real value in the range $[3-n, n-3]$. Instead, we introduce the concept of an *approximate $\delta$-class* where we maintain the exact $\delta(u)$ value for each vertex $u \in G$, but place $u$ in the approximate $\delta(u)$-class computed by $\lfloor w^+(u) - w^-(u) \rfloor$. Using the concept of approximate $\delta$-classes we are able to maintain the logic of the

original GreedyFAS. However, we must take care when deleting a vertex $v$ from $G$. In GreedyFAS, a neighbour $v$ of $u$ has its $\delta$-class incremented or decremented when forming $G \setminus u$, whereas in the probabilistic case we only move $v$ to an adjacent $\delta$-class if the probability on the arc $(u, v)$ changes the exact $\delta(v)$ value enough to cross the threshold into a different approximate $\delta$-class.

**Probabilistic Berger-Shor (pBS)**

The probabilistic version of BergerShorFAS follows a natural extension by altering its decision function for updating set $F$ to incorporate the probabilities on the arcs. Here, when processing a vertex $u$, if the expected number of outgoing arcs (as computed by the sum of probabilities of outgoing arcs) is greater than or equal to the sum of the probabilites of the incoming arcs, then the incoming arcs are removed from $G$ and added to $F$ while the outgoing arcs are removed from $G$ and discarded. If the sum of the probabilities of the incoming arcs is greater, instead the incoming arcs are discarded and the outgoing arcs are added to $F$. After processing each vertex, the arcs in $F$ form a probabilistic FAS in which the expected number of arcs for a possible world is $\mathbb{E}(|F|) = \sum_{e \in F} p(e)$.

**Probabilistic DFS (pDFS)**

Finally, we consider a depth-first version of a Priority-First Search. We alter the standard depth-first traversal so as to visit neighbours in decreasing order of arc probabilities. That is, when processing a vertex $u$, the algorithm attempts to recursively visit the unvisited neighbour $v$ for which $p_{u,v}$ is a maximum first followed by progressively less probable arcs. This natural extension attempts to include as many high probability arcs as possible in the discovery of the DFS tree. By visiting neighbours in this way, the back arcs encountered should be less probable on average

and thus it will bear a lesser contribution in the expected size of a feedback arc set for a possible world.

### 3.5.2 Experiments

The measures of effectiveness in the probabilistic case are *expected FAS size* defined as the expected number of arcs in a FAS output by a particular probabilistic FAS algorithm and the efficiency measured in running time, both of which we aim to minimize.

Due to the increased storage size of a probabilistic graph from the requirement of storing arc probabilities in addition to the network structure, we consider the medium datasets and a subset of the large datasets. We construct probabilistic versions of the datasets by randomly assigning probabilities in the range $[0, 1]$ to the arcs of each graph. In Figure 3.14 we plot the ratio of the expected size of the FAS to the expected number of arcs in a possible world, i.e. the sum of the probabilities of the FAS against the sum of the probabilities of the arcs in the graph, while Figure 3.15 shows the running times.

We observe that the adapted algorithms for GreedyFAS [3] and BergerShorFAS see an improvement in terms of the expected FAS size for several datasets compared to the unweighted case. For example, for GreedyFAS, the improvement is around 8% ($= 1 - 12/13$) on cnr-2000 and arabic-2005, 7% ($= 1 - 15/16$) on webbase-2001, and 6% ($= 1 - 15/16$) on uk-2005. This shows that when the algorithms' logic is adapted to take into consideration the arc probabilities the resulting quality improves in terms of expected FAS size compared to the unweighted case. The additional information available leads to the algorithms choosing feedback arc sets that contain low probability arcs resulting in a smaller expected value. Interestingly, the opposite

[3] We have extended the array version of GreedyFAS.

is true of the dfsFAS adaptation in which it performs slightly worse than in the unweighted case.

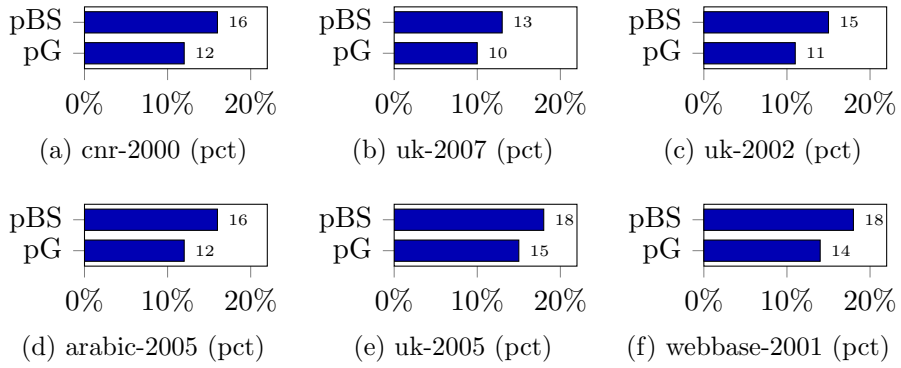



Figure 3.14: Expected FAS sizes for probabilistic versions of algorithms. The resulting expected FAS size is given as a percentage of the total number of expected arcs in the corresponding graph.

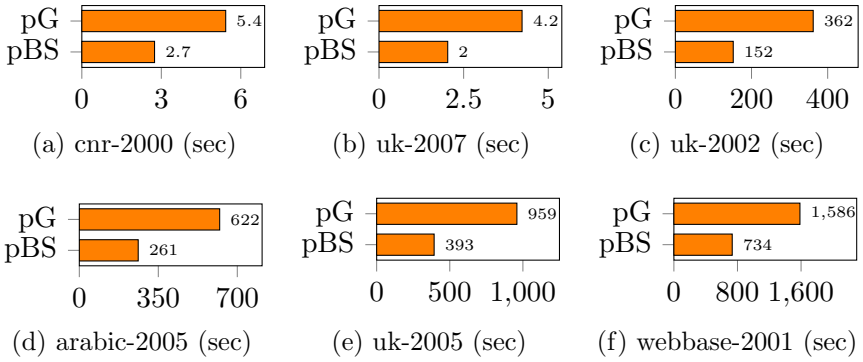



Figure 3.15: Running time for probabilistic versions of algorithms.

## 3.6 Conclusions

To conclude, this chapter presented a thorough experimental investigation into the FAS problem. We presented highly optimized implementations of GreedyFAS, BergerShorFAS, and the sorting-based heuristics. Within each of the three complexity classes of algorithms we observe an overall trend showing a trade off between scal-

ability and quality. We observe an approximate maximum scalability of 300K arcs for the $O(n^2)$ algorithms, 3.5M arcs for the $O(n \log n)$ algorithms, and 50B arcs for the $O(m + n)$ algorithms. We achieve approximate FAS sizes of 3-20%, 23-40%, and 11-17% for the best algorithms of the $O(m + n)$, $O(n \log n)$, and $O(n^2)$ runtime categories, respectively. GreedyFAS and BergerShorFAS provide the best balance between scalability and quality. GreedyFAS is the algorithm that produces always either the smallest or a very close second smallest FAS size while being a fast algorithm in general. In particular our G-arr implementation scales to the biggest dataset we consider, clueweb12, with more than 42 billion arcs.

In addition, we can look to the skew of the top-$k$ vertices as an indication of when the GreedyFAS and BergerShorFAS algorithms are expected to perform favourably. For the sorting-based approaches, we can use the sparsity of $G$ as an indication of the expected performance.

# Chapter 4

# Misinformation Prevention

In this chapter we consider misinformation propagating through a social network and study the problem of its prevention. In this problem, a "bad" campaign starts propagating from a set of seed nodes in the network and we use the notion of a limiting (or "good") campaign to counteract the effect of misinformation. The goal is to identify a set of $k$ users that need to be convinced to adopt the limiting campaign so as to minimize the number of people that adopt the "bad" campaign at the end of both propagation processes.

This chapter presents *RPS* (Reverse Prevention Sampling), an algorithm that provides a scalable solution to the misinformation prevention problem. Our theoretical analysis shows that *RPS* runs in $O((k+l)(n+m)(\frac{1}{1-\gamma})\log n/\epsilon^2)$ expected time and returns a $(1-1/e-\epsilon)$-approximate solution with at least $1-n^{-l}$ probability (where $\gamma$ is a typically small network parameter). The time complexity of *RPS* substantially improves upon the previously best-known algorithms that run in time $\Omega(mnk \cdot POLY(\epsilon^{-1}))$. We experimentally evaluate *RPS* on large datasets and show that it outperforms the state-of-the-art solution by several orders of magnitude in terms of running time. This demonstrates that misinformation prevention can be made practical while still offering strong theoretical guarantees.

In summary, the contributions of this chapter are:

1. We introduce the concept of blocked nodes that fully captures the necessary conditions for preventing the adoption of misinformation in the multi-campaign model.

2. We design and implement a novel BFS-based algorithm for computing the set of nodes that could save a particular user from adopting the misinformation.

3. We propose a misinformation prevention approach that returns a $(1 - 1/e - \epsilon)$-approximate solution with high probability in the multi-campaign model and show that its expected runtime substantially improves upon the expected runtime of the algorithm of Budak et al. [19].

4. We give a lower bound of $\Omega(m + n)$ on the time required to obtain a constant approximation for the misinformation prevention problem.

5. We experiment with large datasets and show that our algorithm gives an improvement of several orders of magnitude over Budak et al. [19] and can efficiently handle graphs with more than 50 million edges.

We begin with an overview of the related work before the preliminaries section that defines the diffusion model, problem definition, notation, and new definitions. Next, we present our misinformation prevention approach and describe the novel algorithm for computing the set of nodes that could save a given user from adopting the misinformation. Finally, we introduce two new lower bounds followed by our experimental results.

## 4.1 Related Work

There exists a large body of work on the Influence Maximization problem first proposed by Kempe et al. [52]. The primary focus of the research community has been related to improving the practical efficiency of the *Greedy* approach under the Independent Cascade (IC) or Linear Threshold (LT) propagation models. These works fall into two categories: heuristics that trade efficiency for approximation guarantees [46, 87] and practical optimizations that speed up *Greedy* while retaining the approximation guarantees [58, 23, 39]. Leskovec et al. [58] present algorithmic optimizations to *Greedy* that reduces the runtime up to 700-fold by avoiding the evaluation of the expected influence for many node sets, while [23, 39] provide further refinements to the approach of [58]. Despite these advancements, the ability of *Greedy* to scale to web-scale networks remained unfeasible.

The breakthrough work of Borgs' et al. [15] brought the first asymptotic runtime improvements while maintaining the $(1 - 1/e - \epsilon)$-approximation guarantees with their *reverse influence sampling* technique. Furthermore, the authors prove their approach is near-optimal under the IC model. Tang et al. [86] presented practical and theoretical improvements to the approach of Borgs' et al. that lead to the elimination of a large hidden constant in the runtime and novel heuristics that result in up to 100-fold improvements to the runtime of their *TIM* approach.

Related to the ongoing work in identifying influential users in a network, there has been a significant amount of work on incorporating the spread of competing campaigns [8, 63, 76, 44] and the spread of misinformation and rumours [19, 33, 72, 61] by augmenting or extending the IC or LT propagation models. In particular, the work of [19] and [44] best capture the idea of preventing the spread of misinformation set in multi-campaign versions of the IC and LT models respectively since both aim to

minimize the number of users that end up adopting the misinformation. Unfortunately, despite both objective functions proving to be monotone and submodular, the *Greedy* solution used in [19] is plagued by the same scaling issues as [52] when considering large social networks and is further exacerbated by the added complexity of tracking multiple cascades which requires costly shortest paths computations. Meanwhile, the heuristics introduced in [44] improve the scalability but lack the theoretical guarantees of the *Greedy* solution.

The work most related to ours is [63] where the authors extend the *reverse influence sampling* technique of [15] to competing campaigns (such as two competing products) as opposed to misinformation prevention. However, their work differs from ours in an important way: the authors use the COIC model of [19] (as opposed to the MCIC model) where the edge probabilities are campaign oblivious. That is, no matter which information reaches a node, it is propagated with a single fixed probability for both campaigns. Importantly, the COIC model does not require the concept of *blocking* defined in this thesis. Since all probabilities are the same, shortest paths suffice for determining which campaign a node adopts. Therefore the work of [63] does not require the new definitions we introduce to close the shortest paths gap in the analysis of [19] for the MCIC model. Budak et al. note that this alternative model does not capture the notion of misinformation as well as the MCIC model, but instead that it is better suited for the spread of multiple competing campaigns. Finally, the scalability of their approach is orders of magnitude less than ours as their largest experiments are on a network with only 500,000 edges.

## 4.2 Preliminaries

In this section, we formally define the multi-campaign diffusion model, the eventual influence limitation problem presented by Budak et al. [19], and present an overview of Kempe et al. and Borgs et al.'s work [52, 15] on the influence maximization problem.

### 4.2.1 Diffusion Model

Let $C$ (for "bad $C$ampaign") and $L$ (for "$L$imiting") denote two influence campaigns in a social network $\mathcal{G}$. Furthermore, assume that each directed edge $e$ in $\mathcal{G}$ is associated with propagation probabilities $p_C(e), p_L(e) \in [0, 1]$. Further, let $G$ be the underlying unweighted graph (ignoring edge probabilities). Given $\mathcal{G}$, the Multi-Campaign Independent Cascade model (MCIC) of Budak et al. [19] considers a time-stamped influence propagation process as follows:

1. At timestamp 1, we *activate* selected sets $A_C$ and $A_L$ of nodes in $\mathcal{G}$ for campaigns $C$ and $L$ respectively, while setting all other nodes *inactive*.
2. If a node $u$ is first activated at timestamp $i$ in campaign $C$ (or $L$), then for each directed edge $e$ that points from $u$ to an inactive neighbour $v$ in $C$ (or $L$), $u$ has $p_C(e)$ (or $p_L(e)$) probability to activate $v$ at timestamp $i + 1$. After timestamp $i + 1$, $u$ cannot activate any node.
3. In the case when two or more nodes from different campaigns are trying to activate $v$ at a given time step we assume that the "good information" (i.e. campaign $L$) takes effect.
4. Once a node becomes activated in one campaign, it never becomes inactive or changes campaigns.

In this work we adopt the activation policy of Budak et al. [19] that has the limiting campaign take precedence when simultaneous activations occur. However, we note that in He et al. [44] the authors consider the opposite policy where the misinformation succeeds in the case of a tie-break. We observe that the tie-break policy does not affect the correctness of our approach, but acts as an interesting model parameter that can vary based on perceived real world information adoption likeliness.

### 4.2.2 Eventual Influence Limitation

It is important to consider the context in which the influence limitation problem is posed in order to select an appropriate objective function. Possible objective functions outlined by Budak et al. include "saving" as many nodes as possible, limiting the lifespan of the "bad" campaign and maximizing the effect of the "good" campaign in the presence of the "bad" campaign.

A natural objective function, as outlined in [19], is "saving" as many nodes as possible. That is, we seek to minimize the number of nodes that end up adopting campaign $C$ when the propagation process is complete. This is referred to as the *eventual influence limitation problem (EIL)*.

Let $A_C$ and $A_L$ be the set of nodes from which campaigns $C$ and $L$ start, respectively. Let $I(A_C)$ be the set of nodes that are activated in campaign $C$ in the absence of $L$ when the above propagation process converges (the set of nodes that would accept campaign $C$ if there was no limiting campaign) and $\pi(A_L)$ be the size of the subset of $I(A_C)$ that campaign $L$ prevents from adopting campaign $C$. We refer to $A_L$ and $A_C$ as the *seed sets*, $I(A_C)$ as the *influence* of campaign $C$, and $\pi(A_L)$ as the *prevention* of campaign $L$. The nodes that are prevented from adopting campaign $C$ are referred to as *saved*. Note that $\pi(A_L)$ is a random variable that depends on the

edge probabilities that each node uses in determining out-neighbors to activate.

Now, notice that the above process has the following equivalent description. We can interpret $\mathcal{G}$ as a distribution over unweighted directed graphs, where each edge $e$ is independently realized with probability $p_C(e)$ (or $p_L(e)$). If we realize a graph $g$ according to this probability distribution, then we can associate the set of saved nodes in the original process with the set of nodes which campaign $L$ reaches before campaign $C$ during the diffusion process in $g \sim \mathcal{G}$. We will make use of this alternative formulation of the MCIC model throughout the chapter.

Budak et al. [19] present a simplified version of the problem that captures the idea that it may be much easier to convince a user of the truth than a falsehood such as cases where photographic evidence can be presented or when a transcript exists. Specifically, the information from campaign $L$ is accepted by users with probability 1 ($p_L(e) = 1$ if edge $e$ exists and $p_L(e) = 0$ otherwise) referred to as the *high effectiveness property*. In [19] it is shown that even with these restrictions EIL with the high effectiveness property is NP-hard. Interestingly, with the high effectiveness property, the prevention function is submodular and thus the greedy approach yields approximation guarantees. Budak et al. study and obtain results for this version of the EIL problem and is the problem that we consider here.

Further, as in [19], we assume that the spread of influence for campaign $C$ starts from a single node $v_c$, referred to as the *adversary node*. However, note that our approach can be easily extended to the case where $C$ starts from a set of nodes. Additionally, our approach works when we allow campaign $C$ to be detected with delay $\Delta$ at which time campaign $L$ is initiated.

**Problem Statement.** Given $\mathcal{G}$, seed set $A_C$, and a positive integer $k$, the eventual influence limitation (EIL) problem asks for a size-$k$ seed set $A_L$ maximizing the value of $\mathbb{E}[\pi(A_L)]$.

### 4.2.3 Influence Maximization Problem

In the following we outline two approaches to the well studied *influence maximization problem (IM)*. This problem is posed in the popular Independent Cascade model (IC) which, unlike the MCIC model, only considers a single campaign. The goal here is to compute a seed set $S_{IM}$ of size $k$ that maximizes the influence of $S_{IM}$ in $\mathcal{G}$.

#### Kempe et al.'s Greedy Approach

Kempe et al.'s approach [52] to the IM problem (referred to as *Greedy* in the following) builds up the seed set $S_{IM}$ in $k$ iterations by adding one node to $S_{IM}$ in each round. In each iteration it adds into $S_{IM}$ the node $u$ that gives the largest marginal increase in $\mathbb{E}[I(S_{IM})]$. That is,

$$u = \underset{v \in V}{\operatorname{argmax}} \Big( \mathbb{E}[I(S \cup \{v\})] - \mathbb{E}[I(S)] \Big).$$

However, the computation of $\mathbb{E}[I(S_{IM})]$ is #P-hard. Therefore, the node $u$ leading to the largest marginal increase in influence must be estimated via a sufficient number of Monte Carlo simulations. In [51] it is shown that if we take a large number $r \in \Omega(POLY(\frac{n}{\epsilon}))$ of measurements in the estimation of each $\mathbb{E}[I(S_{IM})]$, then, with high probability, *Greedy* yields a $(1 - 1/e - \epsilon)$-approximate solution in the IC model. In general, *Greedy* achieves the same approximation ratio under any cascade model where $\mathbb{E}[I(S)]$ is a submodular function of $S$.

Note, if we let $R_H(S)$ be the set of nodes in a graph $H$ that are *reachable* from a set $S$ (a node $v$ in $H$ is reachable from $S$ if there exists a directed path in $H$ that starts from a node in $S$ and ends at $v$), then $I(S_{IM}) = R_g(S_{IM})$ for a fixed $g \sim \mathcal{G}$. To explain, suppose that we flip a coin for each edge $e$ in $G$, and remove the edge with $1 - p(e)$ probability and let $g$ be the resulting graph. Kempe et al. prove that

the expected size of $R_g(S_{IM})$ equals $\mathbb{E}[I(S_{IM})]$. Therefore, to estimate $\mathbb{E}[I(S_{IM})]$, we can first generate multiple instances of $g$, then measure $R_g(S_{IM})$ on each instance, and finally take the average measurement as an estimation of $\mathbb{E}[I(S_{IM})]$.

Despite *Greedy*'s simplicity, its ability to scale to large social networks is severely restricted by its $\Omega(mnk \cdot POLY(\epsilon^{-1}))$ time complexity. As a result, *Greedy* runs for days even when the network contains only a few thousand nodes [22]. Specifically, it runs in $k$ iterations, each of which requires estimating the expected influence of $O(n)$ node sets. In addition, each estimation of expected influence requires $r$ Monte Carlo simulations, and each simulation needs $O(m)$ time.

### Borgs et al.'s Method

The primary cause of *Greedy*'s unfavourable runtime can be observed by considering the algorithm's first iteration of execution. The goal is to identify a single node in $G$ with the maximum expected influence. Without any prior knowledge of the expected influence of the nodes of $G$ we would have to evaluate $\mathbb{E}[I(v)]$ for every node $v$ in $G$. Thus, we see that the overhead of the first iteration alone would be $O(mnr)$. Unfortunately, since *Greedy* requires estimating the expected spread of $O(kn)$ node sets, most of those $O(kn)$ estimations are wasted since, in each iteration of *Greedy*, we are only interested in the node set with the largest increase in expected influence.

Borgs et al. [15] propose a novel method for solving the IM problem under the IC model that avoids the limitations of *Greedy*. We follow the convention of [86] and refer to the method of [15] as *Reverse Influence Sampling (RIS)*. To explain how *RIS* works, Tang et al. [86] introduce the following definitions:

**Definition 6** (Reverse Reachable Set)**.** *The reverse reachable (RR) set for a node $v$ in $g \sim \mathcal{G}$ is the set of nodes that can reach $v$. (That is, for each node $u$ in the RR set, there is a directed path from $u$ to $v$ in $g$.)*

**Definition 7** (Random RR Set)**.** *A random RR set is an RR set generated on an instance of $g \sim \mathcal{G}$, for a node selected uniformly at random from $g$.*

By definition, if a node $u$ appears in an RR set generated for a node $v$, then $u$ can reach $v$ via a certain path in $G$. As such, $u$ should have a chance to activate $v$ if we run an influence propagation process on $G$ using $\{u\}$ as the seed set. The connection between RR sets and node activations is formalized in the following lemma.

**Lemma 2.** [15] *For any seed set $S$ and node $v$, the probability that an influence propagation process from $S$ can activate $v$ equals the probability that $S$ overlaps an RR set for $v$.*

Note that an RR set for a node $v$ is generated by sampling a $g \sim \mathcal{G}$. Based on this result, Borgs' et al. *RIS* algorithm runs in two steps

1. Generate random RR sets from $\mathcal{G}$ until a threshold on the total number of steps taken has been reached.

2. Consider the maximum coverage problem of selecting $k$ nodes to cover the maximum number of RR sets generated. Use the standard greedy algorithm for the problem to derive a $(1 - 1/e)$-approximate solution $S_k^*$. Return $S_k^*$ as the seed set to use for activation.

The rationale behind *RIS* is as follows: if a node $u$ appears in a large number of RR sets it should have a high probability to activate many nodes under the IC model; hence, $u$'s expected influence should be large. As such, we can think of the number of RR sets $u$ appears in as an estimator for $u$'s expected influence. By the same reasoning, if a size-$k$ node set $S_k^*$ covers most RR sets, then $S_k^*$ is likely to have the maximum expected influence among all size-$k$ node sets in $\mathcal{G}$ leading to a good solution to the IM problem. Therefore, the primary goal is to show that we have a

*good* estimator for $u$'s expected influence. The main contribution of Borgs et al. is an analysis of their proposed threshold-based approach: the authors allow *RIS* to keep generating RR sets, until the total number of nodes and edges examined during the generation process reaches a pre-defined threshold $\Gamma$. The authors show that when $\Gamma$ is set to $\Theta((m+n)k\log n/\epsilon^2)$, *RIS* runs in time $O((m+n)k\log n/\epsilon^2)$, and it returns a $(1-1/e-\epsilon)$-approximate solution to the IM problem with at least constant probability.

Finally, the authors provide an algorithm that amplifies the success probability to at least $1-n^{-l}$, by increasing $\Gamma$ by a factor of $l$, and repeating *RIS* for $\Omega(l\log n)$ times. Their algorithm carefully controls the number of random RR sets generated in Step 1 of *RIS* in order to strike a fine balance between efficiency and accuracy. This balance allows *RIS* to avoid estimating the expected spreads of a large number of node sets resulting in a solution substantially more efficient than *Greedy*.

## 4.3 New Definitions

In this section we introduce new definitions that are crucial to our approach. Importantly, we present and discuss a gap in the work of Budak et al. that requires closing in order for our approach to be possible.

Given set $A_L$ of vertices and (unweighted) directed graph $g \sim \mathcal{G}$, write $cl_g(A_L)$ for the set of nodes closer to $A_L$ than $A_C = \{v_c\}$ in $g$. That is, a node $w \in cl_g(A_L)$ if there exists a node $v$ such that $v \in A_L$ and $|SP_G(v,w)| \leq |SP_g(v_c,w)|$ where $SP_H(v,w)$ denotes a shortest path from node $v$ to $w$ in graph $H$. Similarly, $SP_H(S,w)$ for a set $S$ denotes the shortest path from any node $v \in S$ to $w$ in graph $H$. When $g$ is drawn from $\mathcal{G}$ this is a necessary, but not sufficient[1], condition for the set of nodes *saved*

[1]In Budak et al.'s work, the set of nodes closer to $A_L$ than $A_C$ is established as a necessary and sufficient condition to *save* a node in the MCIC model, but we note that this should be revised to include the *blocking* condition due to a gap in the proof of Claim 1 in [19].

| Notation | Description |
|---|---|
| $\mathcal{G}$ | a social network represented as a weighted directed graph $\mathcal{G}$ |
| $G$, $G^T$ | the underlying unweighted graph $G$ and its transpose $G_T$ constructed by reversing the direction of each edge |
| $n$, $m$ | the number of nodes and edges in $\mathcal{G}$ respectively |
| $k$ | the size of the seed set for misinformation prevention |
| $C$, $L$ | the misinformation campaign $C$ and the limiting campaign $L$ |
| $p_C(e)$, $p_L(e)$ | the propagation probability on an edge $e$ for campaigns $C$ and $L$ respectively |
| $\pi(S)$ | the prevention of a node set $S$ in a misinformation propagation process on $\mathcal{G}$ (see Section 4.3) |
| $\omega(R)$, $\omega_\pi(R)$ | the number of edges considered in generating an RRC set and that originate from nodes in an RRC set $R$ (see Equation 4.1) |
| $\kappa(R)$ | see Equation 4.10 |
| $\mathcal{R}$ | the set of all RRC sets generated by Algorithm 12 |
| $\mathcal{F}_\mathcal{R}(S)$ | the fraction of RRC sets in $\mathcal{R}$ that are covered by a node set $S$ |
| $EPT$ | the expected width of a random RRC set |
| $EXP_C$ | the expected influence of campaign $C$ |
| $OPT_L$ | the maximum $\pi(S)$ for any size-$k$ seed set $S$ |
| $KPT$ | a lower bouund of $OPT_L$ established in Section 4.4.2 |
| $\lambda$ | see Equation 4.5 |

Table 4.1: Frequently used notation.

by $A_L$. We also require that the nodes in $cl_g(A_L)$ not be *blocked* by the diffusion of campaign $C$ in $g$.

**Definition 8** (Blocked Nodes)**.** *A node $w \in cl_g(A_L)$ is* blocked *and cannot be saved by $A_L$ if for every path $p$ from $A_L$ to $w$ there exists a node $u$ such that $|SP_g(v_c, u)| < |SP_G(A_L, u)|$.*

Let $bl_g(A_L)$ be the set of blocked nodes for $A_L$. Conceptually, the nodes in $bl_g(A_L)$ are cutoff because some node on the paths from $A_L$ is reached by campaign $C$ before $L$ which stops the diffusion of $L$.

To help illustrate the concept of blocked nodes, consider the graph presented in Figure 4.1. Assume that the solid lines are *live* edges that make up graph $g \sim \mathcal{G}$ in the influence propagation process. Then, the dotted lines are edges that were not

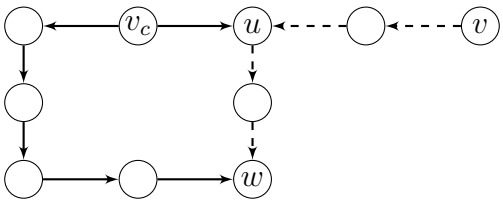



Figure 4.1: An example illustrating the concept of blocked nodes.

realized for campaign $C$. The adversary campaign $C$ starts from $v_c$ while the limiting campaign $L$ starts from $v$.

Observe that $|SP_G(v, w)| = 4$ and $|SP_g(v_c, w)| = 5$. However, $w$ cannot be saved in the resulting cascade since at timestamp 1 the node $u$ will adopt campaign $C$. This intersects the shortest path from $v$ to $w$ and therefore campaign $L$ will not be able to reach node $w$ since a node never switches campaigns. Thus, we say that node $w$ is *blocked* by $C$.

The key observation that lead to our definition of blocked nodes is that the shortest path condition must hold along the entire path. This observation was missed by [19] in the MCIC model and by [44] in their multi-campaign extension of the LT model.

Importantly, the approach of [19] can be recovered with a modified proof for Claim 1 and Theorem 4.2 of [19]. The statements must include the notion of blocked nodes in their *inoculation graph* definition, but a careful inspection shows that their objective function remains submodular after this inclusion. As a result, the greedy approach of [19] still provides the stated approximation guarantees and also allows us to incorporate the ideas of [15] in our solution (as they require a submodular objective function).

Next, we formally define the prevention, $\pi(A_L)$, which corresponds to the number of nodes saved by $A_L$. That is, $\pi(A_L) = |R_g(v_c) \cap (cl_g(A_L) \setminus bl_g(A_L))|$. We write $\mathbb{E}[\pi(A_L)] = \mathbb{E}_{g\sim\mathcal{G}}[\pi(A_L)]$ for the expected prevention of $A_L$ in $\mathcal{G}$.

We refer to the set of nodes that could have saved $u$ as the *saviours* of $u$. A node

$w$ is a candidate saviour for $u$ if there is a directed path from $w$ to $u$ in $G$ (i.e. reverse reachability). Then, $w$ is a saviour for $u$ subject to the additional constraint that $w$ would not be cutoff by the diffusion of $A_C$ in $g$. That is, a candidate saviour $w$ would be cutoff and cannot be a saviour for $u$ if for every path $p$ from $w$ to $u$ there exists a node $v_b$ such that $|SP_g(v_c, v_b)| < |SP_G(w, v_b)|$. We refer to the set of candidate saviours for $u$ that are cutoff as $\tau_g(u)$. Thus, we can define the saviours of $u$ as the set $R_{G^T}(u) \setminus \tau_g(u)$. Therefore, we have:

**Definition 9** (Reverse Reachability without Cutoff Set)**.** *The reverse reachability without cutoff (RRC) set for a node $v$ in $g \sim \mathcal{G}$ is the set of saviour nodes of $v$, i.e. the set of nodes that can save $v$. (That is, for each node $u$ in the RRC set, $u \in R_{G^T}(v) \setminus \tau_g(v)$.) Note, if $v \notin I(v_c)$ then we define the corresponding RRC set as empty since $v$ is not eligible to be saved.*

**Definition 10** (Random RRC Set)**.** *A random RRC set is an RRC set generated on an instance of $g \sim \mathcal{G}$, for a node selected uniformly at random from $g$.*

Finally, let $OPT_L = max_{S:|S|=k}\{\mathbb{E}[\pi(S)]\}$ be the maximum expected prevention of a set of $k$ nodes.

The above example motivates the following classification of diffusion models that captures the importance of our newly defined notion of blocked nodes. Recall, the triggering model is an influence propagation model that generalizes the IC and LT models. It assumes that each node $v$ is associated with a triggering distribution $T(v)$ over the power set of $v$'s incoming neighbors. An influence propagation process under the triggering model works as follows: (1) for each node $v$, take a sample from $T(v)$ and define the sample as the triggering set of $v$, then (2) in the subsequent timestamp, if an active node appears in the triggering set of $v$, then $v$ becomes active. The propagation process terminates when no more nodes can be activated. To understand why the

triggering model captures the IC model as a special case, consider that we assign a triggering distribution to each node $v$, such that each of $v$'s incoming neighbors independently appears in $v$'s triggering set with $p(e)$ probability, where $e$ is the edge from the neighbor to $v$. It can be verified that influence maximization under this distribution is equivalent to that under the IC model.

Observe that the aspect of the MCIC model that enabled the existence of blocked nodes is that the two campaigns are allowed to propagate along disjoint sets of edges. This leads to our classification statement for triggering models.

**Theorem 3** (Model Classification)**.** *Any influence propagation model under the triggering model that allows the influence of different campaigns to propagate along disjoint edge sets requires the notion of blocking.*

## 4.4 Reverse Prevention Sampling

This section presents our misinformation prevention method, *Reverse Prevention Sampling (RPS)*, that applies the *RIS* approach to the EIL problem presented in the MCIC model of Budak et al. [19]. Our approach borrows ideas from the work of Tang et al. [86] which improves the practical efficiency of Borgs et al. approach. Our algorithm returns a $(1 - \frac{1}{e} - \epsilon)$-approximation to the EIL problem, with success probability $1 - n^{-l}$, in time $O((k + l)(m + n)(\frac{1}{1-\gamma}) \log n/\epsilon^2)$. At a high level, *RPS* consists of two steps.

1. **Parameter Estimation.** Compute a lower-bound for the maximum expected prevention among all possible size-$k$ seed sets for $A_L$ and then use the lower-bound to derive a parameter $\theta$.

2. **Node Selection.** Sample $\theta$ random RRC sets from $\mathcal{G}$ to form a set $\mathcal{R}$ and

then compute a size-$k$ seed set $S_k^*$ that covers a large number of RRC sets in $\mathcal{R}$. Return $S_k^*$ as the final result.

In [15] it is noted that since a threshold is used as the stopping condition for the generation of random RR sets, the resulting set of samples is correlated. The result is that some nodes may be over-represented during the generation step leading to a sub-optimal solution. This correlation issue was initially remedied by setting the threshold to a large enough value that lead to a $\epsilon^{-3}$ term in the asymptotic runtime of *RIS*, but an updated analysis by Borgs et al. showed that the $\epsilon^{-2}$ factor could be recovered at the cost of a larger hidden constant.

Conversely, instead of using a threshold-based approach to indirectly control the number of samples, the node selection step of *RPS* samples a *pre-decided* number $\theta$ of random RRC sets following the approach of Tang et al. [86]. By generating a pre-decided number of random RRC sets we ensure the resulting samples are independent given $\theta$ and avoid having to deal with the correlation issue.

We briefly remark on the difficulty of deriving $\theta$ as it must be larger than a particular threshold necessary for ensuring the correctness of *RPS*, but the threshold depends on the optimal prevention, which is an unknown quantity. In our parameter estimation step we derive a $\theta$ value for *RPS* that is above the threshold but also allows for practical efficiency.

In the rest of this section we first identify the conditions necessary for the node selection phase of *RPS* to return a good solution and then describe the the parameter estimation phase in detail. Table 4.1 provides a quick reference to the frequently used notation. Finally, due to space constraints, we omit some proofs and only include those that help build the intuition behind our approach or aid the exposition. The remaining proofs can be found in the full version[2].

[2] `https://github.com/stamps/misinformation_prevention/blob/master/ScalableMisinformationPrevention.pdf`

### 4.4.1 Node Selection

The pseudocode of *RPS*'s node selection step is presented in Algorithm 12. Given $\mathcal{G}$, $k$, $A_C$, and a constant $\theta$ as input, the algorithm stochastically generates $\theta$ random RRC sets, accomplished by repeated invocation of the prevention of misinformation process, and inserts them into a set $\mathcal{R}$. Next, the algorithm follows a greedy approach for the *maximum coverage problem* to select the final seed set. In each iteration, the algorithm selects a node $v_i$ that covers the largest number of RRC sets in $\mathcal{R}$, and then removes all those covered RRC sets from $\mathcal{R}$. The $k$ selected nodes are put into a set $S_k^*$, which is returned as the final result.

**Algorithm 12** NodeSelection($\mathcal{G}$,$k$,$A_C$,$\theta$)

1: $\mathcal{R} \leftarrow \emptyset$
2: Generate $\theta$ random RRC sets and insert them into $\mathcal{R}$.
3: Initialize a node set $S_k^* \leftarrow \emptyset$
4: **for** $i = 1,\ldots,k$ **do**
5: Identify the node $v_i$ that covers the most RRC sets in $\mathcal{R}$
6: Add $v_i$ into $S_k^*$
7: Remove from $\mathcal{R}$ all RRC sets that are covered by $v_i$
8: **return** $S_k^*$

Lines 4-8 in Algorithm 12 correspond to a standard greedy approach for a *maximum coverage problem*. The problem is equivalent to maximizing a submodular function with cardinality constraints for which it is well known that a greedy approach returns a $(1 - 1/e)$-approximate solution in linear time [70].

**RRC set generation.** Line 2 generates $\mathcal{R}$ by repeated simulation of the misinformation prevention process. The generation of each random RRC set is implemented as two breadth-first searches (BFS) on $\mathcal{G}$ and $G^T$ respectively. The first BFS on $\mathcal{G}$ corresponds to a standard randomized BFS and computes the influence set of $A_C$. The second BFS on $G^T$ is an important algorithmic contribution of this work that is essential for the *RPS* approach. This novel bounded-depth BFS with pruning

carefully tracks which nodes will become blocked and is described in detail below.

For completeness, we first describe the standard randomized BFS on $\mathcal{G}$: given the seed node $v_c$ of campaign $C$, we create an empty queue, and then flip a coin for each outgoing edge $e$ of $v_c$ in $\mathcal{G}$; with $p_C(e)$ probability, we retrieve the node $u$ at which $e$ ends, and we put $u$ into the queue. Subsequently, we iteratively extract the node $v'$ at the top of the queue, and examine each outgoing edge $e'$ of $v'$ in $G$; if $e'$ ends at an unvisited node $u'$, we add $u'$ into the queue with $p(e')$ probability. This iterative process terminates when the queue becomes empty. The set of nodes traversed in this manner is equivalent to $I(v_c)$ for $g \sim \mathcal{G}$, due to deferred randomness. Note that in each step of the above BFS we record at each node $w$ the distance from $v_c$ to $w$, denoted $D(w)$, for use in the second step.

The set of nodes traversed by the standard randomized BFS on $\mathcal{G}$ is equivalent to $I(v_c)$ for $g \sim \mathcal{G}$, due to deferred randomness. Note that in each step of the BFS we record at each node $w$ the distance from $v_c$ to $w$, denoted $D(w)$, for use in the second step.

Given a randomly selected node $u$ in $G$, observe that in order for $u$ to be able to be saved we require $u \in I(v_c)$. Therefore, if the randomly selected node $u \notin I(v_c)$ then we return an empty RRC set. On the other hand, if $u \in I(v_c)$, we have $D(u) = |SP_g(v_c, u)|$ as a result of the above randomized BFS which indicates the maximum distance from $u$ that candidate saviour nodes can exist. We run a second BFS from $u$ in $G^T$ to depth $D(u)$ to determine the saviour nodes for $u$ by carefully pruning those nodes that would become blocked.

The *bounded-depth BFS with pruning* on $G^T$, presented in Algorithm 13, takes as input a source node $u$, the maximum depth $D(u)$, and a directed graph $G^T$. Algorithm 13 utilizes special indicator values associated with each node $w$ to account for potential cutoffs from $v_c$. Each node $w$ holds a variable, $\beta(w)$, which indicates the

distance beyond $w$ that the BFS can go before the diffusion would have been cutoff by $v_c$ in $g$. The $\beta$ value for each node $w$ is initialized to $D(w)$. In each round, the current node $w$ has an opportunity to update the $\beta$ value of each of its successors only if $\beta(w) > 0$. For each successor $z$ of $w$, we assign $\beta(z) = \beta(w) - 1$ if $\beta(z) = \texttt{null}$ or if $\beta(z) > 0$ and $\beta(w) - 1 < \beta(z)$. In this way, each ancestor of $z$ will have an opportunity to apply a $\beta$ value to $z$ to ensure that if any ancestor has a $\beta$ value then so will $z$ and furthermore, the $\beta$ variable for $z$ will be updated with the smallest $\beta$ value from its ancestors. We terminate the BFS early if we reach a node $w$ with $\beta(w) = 0$.

Figure 4.2 captures the primary scenarios encountered by Algorithm 13 when initialized at node $u$. The enclosing dotted line represents the extent of the influence of campaign $C$ for the current influence propagation process. First, notice that if the BFS moves away from $v_c$, as in the case of node $z$, that, once we move beyond the influence boundary of $v_c$, there will be no potential for cutoff. As such, the BFS is free to traverse until the maximum depth $D(u)$ is reached. On the other hand, if the BFS moves towards (or perpendicular to) $v_c$ then we must carefully account for potential cutoff. For example, when the BFS reaches $v$, we know the distance from $v_c$ to $v$: $D(v) = SP_g(v_c, v)$. Therefore, the BFS must track the fact that there cannot exist saviours at a distance $D(v)$ beyond $v$. In other words, if we imagine initializing a misinformation prevention process from a node $w$ such that $SP_G(v, w) > D(v)$ then $v$ will adopt campaign $C$ before campaign $L$ can reach $v$. Therefore, at each out-neighbour of $v$ we use the knowledge of $D(v)$ to track the distance beyond $v$ that saviours can exist. This updating process tracks the smallest such value and is allowed to cross the enclosing influence boundary of campaign $C$ ensuring that all potential for cutoff is tracked.

Finally, we collect all nodes visited during the process (including $u$), and use them

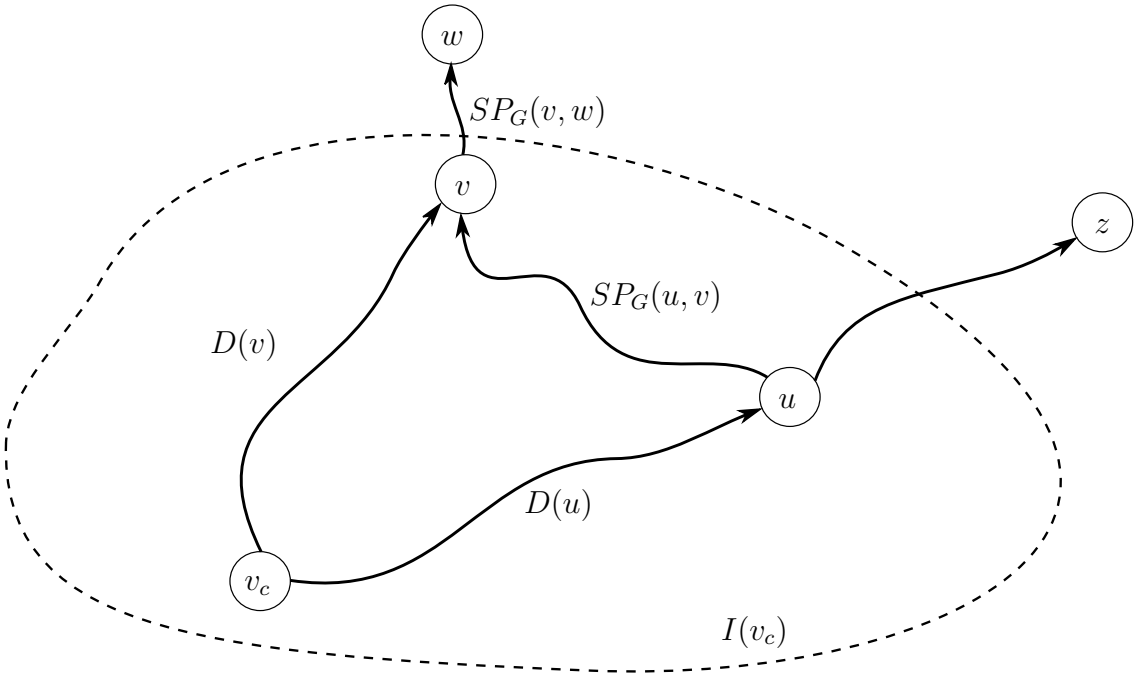



Figure 4.2: An overview of the primary scenarios encountered by Algorithm 13.

to form an RRC set. The runtime of this procedure is precisely the sum of the degrees (in $G$) of the nodes in $I(v_c)$ plus the sum of the degrees of the nodes in $R_{G^T}(u) \setminus \tau(u)$.

**Performance Bounds.** In this section we first define a quantity $EPT$ that captures the expected number of edges traversed when generating a random RRC set. After that, we define the expected runtime of $RPS$ in terms of $EPT$ and the parameter $\theta$. Finally, we define a threshold for $\theta$ that ensures the correctness of $RPS$.

First, we refer to the instance of $I(v_c)$ used in computing an RRC set $R$ as $M_R$. Then, we define the *width* of an RRC set $R$, denoted as $\omega(R)$, as the number of edges in $G$ that point to nodes in $R$ plus the number of edges in $G$ that originate from nodes in $M_R$. That is,

$$\omega(R) = \sum_{u \in M_R} outdegree_G(u) + \sum_{v \in R} indegree_G(v). \tag{4.1}$$

Furthermore, we define the *subwidth* of an RRC set $R$, denoted as $\omega_\pi(R)$, as the

**Algorithm 13** generateRRC($u$, $D(u)$, $G^T$)

```
1: let Q be a queue
2: u.depth = 0
3: Q.enqueue(u)
4: label u as discovered
5: while Q is not empty do
6:     w ← Q.dequeue()
7:     if w.depth = D(u) OR β(w) = 0 then
8:         continue
9:     for all nodes z in G^T.adjacentEdges(w) do
10:        if β(w) > 0 AND β(z) > 0 then
11:            if β(w) − 1 < β(z) then
12:                β(z) ← β(w) − 1
13:        else if β(w) > 0 then
14:            β(z) ← β(w) − 1
15:        if z is not labelled as discovered then
16:            z.depth = w.depth + 1
17:            Q.enqueue(z)
18:            label z as discovered
```

number of edges in $G$ that point to nodes in $R$. That is $\omega_\pi(R) = \sum_{v \in R} indegree_G(v)$. Also, we define the *prewidth* of an RRC set $R$, denoted as $\omega(M_R)$, as the number of edges in $G$ that originate from nodes in $M_R$. That is $\omega(M_R) = \sum_{u \in M_R} outdegree_G(u)$.

Let $EPT$ be the expected width of a random RRC set, where the expectation is taken over the randomness in $R$ and $M_R$, and observe that Algorithm 12 has an expected runtime of $O(\theta \cdot EPT)$. This can be observed by noting that $EPT$ captures the expected number of edge traversals required to generate a random RRC set since an edge is only considered in the propagation process (either of the two BFS's) if it points to a node in $R$ or originates from a node in $M_R$. The remainder of the section aims to derive a value for $\theta$ that minimizes the expected runtime of Algorithm 12 while ensuring the quality of the solution returned.

We establish a connection between RRC sets and the prevention process on $\mathcal{G}$. That is, the prevention of a set of nodes $S$ is precisely $n$ times the probability that a

node $u$, chosen uniformly at random, has a saviour from $S$:

**Lemma 3.** *For any seed set $S$ and any node $v$, the probability that a prevention process from $S$ can save $v$ equals the probability that $S$ overlaps an RRC set for $v$.*

*Proof.* Let $S$ be a fixed set of nodes, and $v$ be a fixed node. Suppose that we generate an RRC set $R$ for $v$ on a graph $g \sim \mathcal{G}$. Let $\rho_1$ be the probability that $S$ overlaps with $R$ and let $\rho_2$ be the probability that $S$, when used as a seed set, can save $v$ in a prevention process on $\mathcal{G}$. By Definition 9, if $v \in I(v_c)$ then $\rho_1$ equals the probability that a node $u \in S$ is a saviour for $v$. That is, $\rho_1$ equals the probability that $G$ contains a directed path from $u \in S$ to $v$ and $u \notin \tau(v)$ and 0 if $v \notin I(v_c)$. Meanwhile, if $v \in I(v_c)$ then $\rho_2$ equals the probability that a node $u \in S$ can save $v$ (i.e. $v \in (cl_g(u) \setminus bl_g(u))$) and 0 if $v \notin I(v_c)$. It follows that $\rho_1 = \rho_2$ due to the equivalence between the set of saviours for $v$ and the ability to save $v$. □

For any node set $S$, let $F_{\mathcal{R}}(S)$ be the fraction of RRC sets in $\mathcal{R}$ covered by $S$. Then, based on Lemma 3, we can prove that the expected value of $n \cdot F_{\mathcal{R}}(S)$ equals the expected prevention of $S$ in $\mathcal{G}$.

**Corollary 1.** $\mathbb{E}[n \cdot F_{\mathcal{R}}(S)] = \mathbb{E}[\pi(S)]$

*Proof.* Observe that $\mathbb{E}[F_{\mathcal{R}}(S)]$ equals the probability that $S$ intersects a random RRC set, while $\mathbb{E}[\pi(S)]/n$ equals the probability that a randomly selected node can be saved by $S$ in a prevention process on $\mathcal{G}$. By Lemma 3, the two probabilities are equal, leading to $\mathbb{E}[n \cdot F_{\mathcal{R}}(S)] = \mathbb{E}[\pi(S)]$. □

Corollary 1 implies that we can estimate $\mathbb{E}[\pi(S)]$ by estimating the fraction of RRC sets in $\mathcal{R}$ covered by $S$. The number of sets covered by a node $v$ in $\mathcal{R}$ is precisely the number of times we observed that $v$ was a saviour for a randomly selected node $u$. We can therefore think of $n \cdot F_{\mathcal{R}}(S)$ as an estimator for $\mathbb{E}[\pi(S)]$. Our primary task is

to show that it is a *good* estimator. Using Chernoff bounds, we show that $n \cdot F_{\mathcal{R}}(S)$ is an accurate estimator of any node set $S$'s expected prevention, when $\theta$ is sufficiently large:

**Lemma 4.** *Suppose that $\theta$ satisfies*

$$\theta \geq (8 + 2\epsilon)n \cdot \frac{l \log n + \log \binom{n}{k} + \log 2}{OPT_L \cdot \epsilon^2}. \tag{4.2}$$

*Then, for any set $S$ of at most $k$ nodes, the following inequality holds with at least $1 - n^{-l}/\binom{n}{k}$ probability:*

$$\left| n \cdot F_{\mathcal{R}}(S) - \mathbb{E}[\pi(S)] \right| < \frac{\epsilon}{2} \cdot OPT_L. \tag{4.3}$$

*Proof.* Let $\rho$ be the probability that $S$ overlaps with a random RRC set. Then, $\theta \cdot F_{\mathcal{R}}(S)$ can be regarded as the sum of $\theta$ i.i.d. Bernoulli variables with a mean $\rho$. By Corollary 1,

$$\rho = \mathbb{E}[F_{\mathcal{R}}(S)] = \mathbb{E}[\pi(S)]/n.$$

Then, we have

$$\begin{aligned} &Pr\left[ |n \cdot F_{\mathcal{R}}(S) - \mathbb{E}[\pi(S)]| \geq \frac{\epsilon}{2} \cdot OPT_L \right] \\ &\quad = Pr\left[ |\theta \cdot F_{\mathcal{R}}(S) - \rho\theta| \geq \frac{\epsilon\theta}{2n} \cdot OPT_L \right] \\ &\quad = Pr\left[ |\theta \cdot F_{\mathcal{R}}(S) - \rho\theta| \geq \frac{\epsilon \cdot OPT_L}{2n\rho} \cdot \rho\theta \right] \end{aligned} \tag{4.4}$$

Let $\delta = \epsilon \cdot OPT_L/(2n\rho)$. By the Chernoff bounds, Equation 4.2, and the fact that $\rho = \mathbb{E}[\pi(S)]/n \leq OPT_L/n$, we have

$$\begin{aligned}\text{r.h.s. of Eqn. 4.4} &< 2exp\Big(-\frac{\delta^2}{2+\delta}\cdot\rho\theta\Big)\\ &= 2exp\Big(-\frac{\epsilon^2\cdot OPT_L^2}{8n^2\rho+2\epsilon n\cdot OPT_L}\cdot\theta\Big)\\ &\leq 2exp\Big(-\frac{\epsilon^2\cdot OPT_L^2}{8n\cdot OPT_L+2\epsilon n\cdot OPT_L}\cdot\theta\Big)\\ &= 2exp\Big(-\frac{\epsilon^2\cdot OPT_L}{(8+2\epsilon)\cdot n}\cdot\theta\Big)\leq\frac{1}{\binom{n}{k}\cdot n^l}.\end{aligned}$$

Therefore, the lemma is proved. □

Based on Lemma 4, we prove that if Eqn. 4.2 holds, Algorithm 12 returns a $(1-1/e-\epsilon)$-approximate solution with high probability by a simple application of Chernoff bounds.

**Theorem 4.** *Given a $\theta$ that satisfies Equation 4.2, Algorithm 12 returns a $(1-1/e-\epsilon)$-approximate solution with at least $1-n^{-l}$ probability.*

*Proof.* Let $S_k$ be the node set returned by Algorithm 12, and $S_k^+$ be the size-$k$ node set that maximizes $F_{\mathcal{R}}(S_k^+)$ (i.e., $S_k^+$ covers the largest number of RRC sets in $\mathcal{R}$). As $S_k$ is derived from $\mathcal{R}$ using a $(1-1/e)$-approximate algorithm for the maximum coverage problem, we have $F_{\mathcal{R}}(S_k)\geq(1-1/e)\cdot F_{\mathcal{R}}(S_k^+)$. Let $S_k^\circ$ be the optimal solution for the EIL problem on $\mathcal{G}$, i.e. $\mathbb{E}[\pi(S_k^\circ)]=OPT_L$. We have $F_{\mathcal{R}}(S_k^+)\geq F_{\mathcal{R}}(S_k^\circ)$, which leads to $F_{\mathcal{R}}(S_k)\geq(1-1/e)\cdot F_{\mathcal{R}}(S_k^\circ)$.

Assume that $\theta$ satisfies Equation 4.2. By Lemma 4, Equation 4.3 holds with at least $1-n^{-l}/\binom{n}{k}$ probability for any given size-$k$ node set $S$. Thus, by the union bound, Equation 4.3 should hold simultaneously for all size-$k$ node sets with at least $1-n^{-l}$ probability. In that case, we have

$$\begin{aligned}\mathbb{E}[\pi(S_k)] &> n \cdot F_{\mathcal{R}}(S_k) - \epsilon/2 \cdot OPT_L \\ &\geq (1 - 1/e) \cdot n \cdot F_{\mathcal{R}}(S_k^+) - \epsilon/2 \cdot OPT_L \\ &\geq (1 - 1/e) \cdot n \cdot F_{\mathcal{R}}(S_k^\circ) - \epsilon/2 \cdot OPT_L \\ &\geq (1 - 1/e) \cdot (1 - \epsilon/2) \cdot OPT_L - \epsilon/2 \cdot OPT_L \\ &> (1 - 1/e - \epsilon) \cdot OPT_L.\end{aligned}$$

Thus, the theorem is proved. □

An important consideration is that, since $OPT$ is unknown, we cannot set $\theta$ directly from Equation 4.2. For simplicity, we define

$$\lambda = (8 + 2\epsilon)n \cdot \left(l \log n + \log \binom{n}{k} + \log 2\right) \cdot \epsilon^{-2} \tag{4.5}$$

and rewrite Equation 4.2 as

$$\theta \geq \lambda / OPT_L. \tag{4.6}$$

The following section details how to derive a $\theta$ which not only satisfies Equation 4.2, but also leads to an $O((k+l)(m+n)(\frac{1}{1-\gamma}) \log n/\epsilon^2)$ expected runtime for Algorithm 12.

### 4.4.2 Parameter Estimation

Recall that the expected runtime complexity of Algorithm 12 is $O(\theta \cdot EPT)$, where $EPT$ captures the expected number of coin tosses required to generate a random RRC set. Our objective is to identify a $\theta$ that makes $\theta \cdot EPT$ reasonably small, while still ensuring $\theta \geq \lambda / OPT_L$. First, we define a probability distribution $\mathcal{V}^*$ over the nodes

in $G$, such that the probability mass for each node is proportional to its indegree in $G$. Let $v^*$ be a random variable following $\mathcal{V}^*$ and recall that $M_R$ is a random instance of $I(v_c)$.

**Lemma 5.** $\frac{m}{n} \cdot \mathbb{E}[\pi(\{v^*\})] = EPT - \mathbb{E}[\omega(M_R)]$, *where the expectation of* $\pi(\{v^*\})$ *and* $\omega(M_R)$ *is taken over the randomness in* $v^*$ *and the prevention process.*

*Proof.* Let $R$ be a random RRC set, $M_R$ be the random instance of $I(v_c)$ used to compute $R$, and $p_R$ be the probability that a randomly selected edge from $G$ points to a node in $R$. Then, $EPT = \mathbb{E}[\omega(M_R)] + \mathbb{E}[p_R \cdot m]$, where the expectation is taken over the random choices of $R$ and $M_R$.

Let $v^*$ be a sample from $\mathcal{V}^*$ and $b(v^*, R)$ be a boolean function that returns 1 if $v^* \in R$, and 0 otherwise. Then, for any fixed $R$, $p_R = \sum_{v^*}(\Pr[v^*] \cdot b(v^*, R))$. Now consider that we fix $v^*$ and vary $R$. Define $p_{v^*,R} = \sum_R(\Pr[R] \cdot b(v^*, R))$ so that by Lemma 3, $p_{v^*,R}$ equals the probability that a randomly selected node can be saved in a prevention process when $\{v^*\}$ is used as a seed set. Therefore, $\mathbb{E}[p_{v^*,R}] = \mathbb{E}[\pi(\{v^*\})]/n$. This leads to

$$\begin{aligned}
\mathbb{E}[p_R] &= \sum_R (\Pr[R] \cdot p_R) \\
&= \sum_R (\Pr[R] \cdot \sum_{v^*} (\Pr[v^*] \cdot b(v^*, R))) \\
&= \sum_{v^*} (\Pr[v^*] \cdot \sum_R (\Pr[R] \cdot b(v^*, R))) \\
&= \sum_{v^*} (\Pr[v^*] \cdot p_{v^*,R}) = \mathbb{E}[p_{v^*,R}] = \mathbb{E}[\pi(\{v^*\})]/n.
\end{aligned}$$

Resulting in

$$\begin{aligned} EPT &= \mathbb{E}[\omega(M_R)] + m \cdot \mathbb{E}[p_R] \\ &= \mathbb{E}[\omega(M_R)] + \frac{m}{n} \cdot \mathbb{E}[\pi(\{v^*\})]. \end{aligned}$$

This completes the proof. □

Lemma 5 shows that if we randomly sample a node from $\mathcal{V}^*$ and calculate its expected prevention $p$, then on average we have $p = \frac{n}{m}(EPT - \mathbb{E}[\omega(M_R)])$. This implies that $\frac{n}{m}(EPT - \mathbb{E}[\omega(M_R)]) \leq OPT_L$, since $OPT_L$ is the maximum expected prevention of any size-$k$ node set.

Recall that the expected runtime complexity of Algorithm 12 is $O(\theta \cdot EPT)$. Now, suppose we are able to identify a parameter $t$ such that $t = \Omega(\frac{n}{m}(EPT - \mathbb{E}[\omega(M_R)]))$ and $t \leq OPT_L$. Then, by setting $\theta = \lambda/t$, we can guarantee that Algorithm 12 is correct, since $\theta \geq \lambda/OPT_L$, and has an expected runtime complexity of

$$O(\theta \cdot EPT) = O\Big(\frac{\lambda}{t} \cdot EPT\Big) = O\left(\frac{\lambda \cdot EPT}{\frac{n}{m}(EPT - \mathbb{E}[\omega(M_R)])}\right). \tag{4.7}$$

Furthermore, we can define a ratio $\gamma \in (0, 1)$ which captures the relationship between $\mathbb{E}[\omega(M_R)]$ and $EPT$ by writing $\mathbb{E}[\omega(M_R)] = \gamma EPT$. Then we can rewrite Equation 4.7 as

$$O\Big(\frac{m}{n}\Big(\frac{1}{1-\gamma}\Big)\lambda\Big) = O((k+l)(m+n)(1/(1-\gamma))\log n/\epsilon^2). \tag{4.8}$$

Note that $\gamma$ is an instance-based parameter not present in the work of Tang et al, but arises from the MCIC model. See Section 4.5 for a detailed discussion of $\gamma$.

**Computing $t$.** Ideally, we seek a $t$ that increases monotonically with $k$ to mimic the behaviour of $OPT_L$. Suppose we take $k$ samples $e_i = (v_i, w_i)$ with replacement over

a uniform distribution on the edges in $G$, and use the $v_i$'s to form a node set $S^*$. Let $KPT$ be the mean of the expected prevention of $S^*$ over the randomness in $S^*$ and the prevention process. Due to the submodularity of the prevention function, it can be verified that $KPT$ increases with $k$ and

$$\frac{n}{m}\Big(EPT - \mathbb{E}[\omega(M_R)]\Big) \leq KPT \leq OPT_L. \tag{4.9}$$

Additionally,

**Lemma 6.** *Let $R$ be a random RRC set and $\omega_\pi(R)$ be the subwidth of $R$. Define*

$$\kappa(R) = 1 - \left(1 - \frac{\omega_\pi(R)}{m}\right)^k. \tag{4.10}$$

*Then, $KPT = n \cdot \mathbb{E}[\kappa(R)]$, where the expectation is taken over the random choices of $R$.*

*Proof.* Let $S^*$ be a node set formed by the $v_i$ from $k$ samples $e_i = (v_i, w_i)$ over a uniform distribution on the edges in $G$, with duplicates removed. Let $R$ be a random RRC set, and $\alpha_R$ be the probability that $S^*$ overlaps with $R$. Then, by Corollary 1,

$$KPT = \mathbb{E}[\pi(S^*)] = \mathbb{E}[n \cdot \alpha_R].$$

Consider that we sample $k$ times over a uniform distribution on the edges in $G$. Let $E^*$ be the set of edges sampled, with duplicates removed. Let $\alpha'_R$ be the probability that one of the edges in $E^*$ points to a node in $R$. It can be verified that $\alpha'_R = \alpha_R$. Furthermore, given that there are $\omega_\pi(R)$ edges in $G$ that point to nodes in $R$, $\alpha'_R = 1 - (1 - \omega_\pi(R)/m)^k = \kappa(R)$. Therefore,

$$KPT = \mathbb{E}[n \cdot \alpha_R] = \mathbb{E}[n \cdot \alpha'_R] = \mathbb{E}[n \cdot \kappa(R)],$$

which proves the lemma. □

Lemma 6 shows we can estimate $KPT$ by first computing $n \cdot \kappa(R)$ on a set of random RRC sets and taking the average of the resulting measurements. However, as dictated by Chernoff bounds, if we want to obtain an estimate of $KPT$ with $\delta \in (0, 1)$ relative error with at least $1 - n^{-l}$ probability then the number of samples required is $\Omega(ln \log n \cdot \delta^{-2}/KPT)$. That is, the number of measurements required to estimate $KPT$ depends on $KPT$ itself. This issue is also encountered in [86] and we are able to resolve it by mimicking their adaptive sampling approach, which dynamically adjusts the number of measurements based on the observed values of $\kappa(R)$, under the MCIC model.

**Algorithm 14** KptEstimation($\mathcal{G}$,$k$,$A_C$)

1: **for** $i = 1$ to $\log_2 n - 1$ **do**
2: Let $c_i = (6l \log n + 6 \log(\log_2 n)) \cdot 2^i$
3: Let $sum = 0$
4: **for** $j = 1$ to $c_i$ **do**
5: Generate a random RRC set $R$
6: $\kappa(R) = 1 - (1 - \frac{\omega_\pi(R)}{m})^k$
7: $sum = sum + \kappa(R)$
8: **if** $sum/c_i > 1/2^i$ **then**
9: **return** $KPT^* = n \cdot sum/(2 \cdot c_i)$
10: **return** $KPT^* = 1$

**Estimating *KPT*.** Algorithm 14 presents the sampling approach for estimating $KPT$. The high level idea is as follows. Since $KPT$ is an unknown quantity, we begin with the assumption that it takes on the value $n/2$. Then, we compute an estimate for the expected value of $\kappa(R)$ based on a relatively few number of samples. Chernoff bounds allow us to determine if the computed value of $KPT = n \cdot \kappa(R)$ is a good estimator and, if so, the algorithm terminates. However, if the estimate is much smaller than $n/2$ we apply the standard doubling approach and generate an increased number of samples to determine if $KPT$ takes on a value close to half the

initial estimate. The algorithm continues computing estimates for $KPT$, based on an increasing number of samples, and comparing to values that halve in size until the error bounds dictated by Chernoff bounds indicate we have reached a suitably precise estimation of $KPT$.

More specifically, in each iteration (Lines 2-7), the goal of Algorithm 14 is to compute the average value of $\kappa(R)$ over $c_i$ randomly generated RRC sets from $\mathcal{G}$. As described in Lemma's 7 and 8 below, the $c_i$ are chosen carefully such that if the average computed for $\kappa(R)$ over the $c_i$ samples is greater than $2^{-i}$ then we can conclude that we have a good estimate for $KPT$ with high probability. More precisely, that the expected value of $\kappa(R)$ is at least half of the average computed. Conversely, if the average computed is too small then Chernoff bounds imply that we have a bad estimate for $KPT$ and the algorithm proceeds to the next iteration.

In the case that the true value of $KPT$ is very small, the algorithm will terminate in the $\log_2 n$-th iteration and return $KPT^* = 1$, which equals the smallest possible $KPT$ (since each node in the seed set can always save itself assuming $INF_C \geq k$). As we will show in the next section, $KPT^* \in [KPT/4, OPT_L]$ holds with a high probability. Thus, setting $\theta = \lambda / KPT^*$ ensures Algorithm 12 is correct and achieves the expected runtime complexity in Equation 4.8.

**Performance Bounds.** Proving the correctness and demonstrating bounds on the runtime for Algorithm 14 requires a careful analysis of the algorithm's behaviour. As shown below, we make use of two lemmas to prove that the algorithm's estimate of $KPT^*$ is close to $KPT$.

Let $\mathcal{K}$ be the distribution of $\kappa(R)$ over random RRC sets in $\mathcal{G}$ with domain $[0, 1]$. Let $\mu = KPT/n$, and $s_i$ be the sum of $c_i$ i.i.d. samples from $\mathcal{K}$, where $c_i$ is defined as $c_i = (6l \log n + 6 \log(\log_2 n)) \cdot 2^i$. Chernoff bounds give

**Lemma 7.** *If $\mu \leq 2^{-j}$, then for any $i \in [1, j-1]$,*

$$Pr\left[\frac{s_i}{c_i} > \frac{1}{2^i}\right] < \frac{1}{n^l \cdot \log_2 n}. \tag{4.11}$$

*Proof.* Let $\delta = (2^{-i} - \mu)/\mu$. By the Chernoff bounds,

$$\begin{aligned} Pr\left[\frac{s_i}{c_i} > 2^{-i}\right] &\leq exp\left( - \frac{\delta^2}{2+\delta} \cdot c_i \cdot \mu \right) \\ &= exp(-c_i \cdot (2^{-i} - \mu)^2/(2^{-i} + \mu)) \\ &\leq exp(-c_i \cdot 2^{-i-1}/3) = \frac{1}{n^l \cdot \log_2 n}. \end{aligned}$$

This completes the proof. □

By Lemma 7, if $KPT \leq 2^{-j}$, then Algorithm 14 is very unlikely to terminate in any of the first $j-1$ iterations. This prevents the algorithm from outputting a $KPT^*$ too much larger than $KPT$.

**Lemma 8.** *If $\mu \geq 2^{-j}$, then for any $i \geq j+1$,*

$$Pr\left[\frac{s_i}{c_i} > \frac{1}{2^i}\right] > 1 - \left(\frac{1}{n^l \cdot \log_2 n}\right)^{2^{i-j-1}}. \tag{4.12}$$

*Proof.* Let $\delta = (\mu - 2^{-i})/\mu$. By the Chernoff bounds,

$$\begin{aligned} Pr\left[\frac{s_i}{c_i} \leq 2^{-i}\right] &\leq exp\left( - \frac{\delta^2}{2} \cdot c_i \cdot \mu \right) \\ &= exp(-c_i \cdot (\mu - 2^{-i})^2/(2 \cdot \mu)) \\ &\leq exp(-c_i \cdot \mu/8) < \left(\frac{1}{n^l \cdot \log_2 n}\right)^{2^{i-j-1}}. \end{aligned}$$

This completes the proof. □

By Lemma 8, if $KPT \leq 2^{-j}$ and Algorithm 14 enters iteration $i > j+1$, then it will terminate in the $i$-th iteration with high probability. This ensures that the algorithm does not output a $KPT^*$ that is too much smaller than $KPT$.

Based on Lemmas 7 and 8, we prove the following theorem for the correctness and expected runtime of Algorithm 14.

**Theorem 5.** *When $n \geq 2$ and $l \geq 1/2$, Algorithm 14 returns $KPT^* \in [KPT/4, OPT_L]$ with at least $1 - n^{-l}$ probability, and runs in $O(l(m+n)(1/(1-\gamma))\log n)$ expected time. Furthermore, $\mathbb{E}[\frac{1}{KPT^*}] < \frac{12}{KPT}$.*

*Proof.* Assume that $KPT/n \in [2^{-j}, 2^{-j+1}]$. We first prove the accuracy of the $KPT^*$ returned by Algorithm 14.

By Lemma 7 and the union bound, Algorithm 14 terminates in or before the $(j-2)$-th iteration with less than $n^{-l}(j-2)/\log_2 n$ probability. On the other hand, if Algorithm 14 reaches the $(j+1)$-th iteration, then by Lemma 8, it terminates in the $(j+1)$-th iteration with at least $1 - n^{-l}/\log_2 n$ probability. Given the union bound and the fact that Algorithm 14 has at most $\log_2 n - 1$ iterations, Algorithm 14 should terminate in the $(j-1)$-th, $j$-th, or $(j+1)$-th iteration with a probability at least $1 - n^{-l}(\log_2 n - 2)/\log_2 n$. In that case, $KPT^*$ must be larger than $n/2 \cdot 2^{-j-1}$, which leads to $KPT^* > KPT/4$. Furthermore, $KPT^*$ should be $n/2$ times the average of at least $c_{j-1}$ i.i.d. samples from $\mathcal{K}$. By the Chernoff bounds, it can be verified that

$$Pr[KPT^* \geq KPT] \leq n^{-l}/\log_2 n.$$

By the union bound, Algorithm 14 returns, with probability at least $1 - n^{-l}$ probability, $KPT^* \in [KPT/4, KPT] \subseteq [KPT/4, OPT_L]$.

Next, we analyze the expected runtime of Algorithm 14. Recall that the $i$-th iteration of the algorithm generates $c_i$ RRC sets, and each RRC set takes $O(EPT)$

expected time. Given that $c_{i+1} = 2 \cdot c_i$ for any $i$, the first $j+1$ iterations generate less than $2 \cdot c_{j+1}$ RRC sets in total. Meanwhile, for any $i' \geq j+2$, Lemma 8 shows that Algorithm 14 has at most $n^{-l \cdot 2^{i'-j-1}} / \log_2 n$ probability to reach the $i'$-th iteration. Therefore, when $n \geq 2$ and $l \geq 1/2$, the expected number of RRC sets generated after the first $j+1$ iterations is less than

$$\sum_{i'=j+2}^{\log_2 n - 1} \left( c_{i'} \cdot \left( \frac{1}{n^l \cdot \log_2 n} \right)^{2^{i-j-1}} \right) < c_{j+2}.$$

Hence, the expected total number of RRC sets generated by Algorithm 14 is less than $2c_{j+1} + c_{j+2} = 2c_{j+2}$. Therefore, the expected time complexity of the algorithm is

$$\begin{aligned}
O(c_{j+2} \cdot EPT) &= O(2^j l \log n \cdot EPT) \\
&= O\Big(2^j l \log n \cdot \Big(1 + \frac{m}{n}\Big) \cdot \Big(KPT + INF_C\Big)\Big) \\
&= O\Big(2^j l \log n \cdot \Big(1 + \frac{m}{n}\Big) \cdot KPT \cdot \Big(1 + \frac{INF_C}{KPT}\Big)\Big) \\
&= O\Big(2^j l \log n \cdot (m+n) \cdot 2^{-j} \cdot \Big(1 + \frac{INF_C}{KPT}\Big)\Big) \\
&= O\Big(l \log n \cdot (m+n) \cdot \Big(1 + \frac{INF_C}{KPT}\Big)\Big).
\end{aligned}$$

Here we used Equation 4.9 in the second equality. Now, we can write $KPT = \gamma' \cdot INF_C$, where $\gamma' \in (0, 1]$, and observe that $\gamma \cdot INF_C = \mathbb{E}[\pi(\{v^*\})] \leq KPT = \gamma' \cdot INF_C$. Therefore, $\gamma \leq \gamma'$ and $\frac{INF_C}{KPT} = \frac{INF_C}{\gamma' \cdot INF_C} = \frac{1}{\gamma'} \leq \frac{1}{\gamma}$. This gives,

$$O\Big(l \log n \cdot (m+n) \cdot \Big(1 + \frac{INF_C}{KPT}\Big)\Big) = O\Big(l \log n \cdot (m+n) \cdot \Big(1 + \frac{1}{\gamma}\Big)\Big).$$

Finally, we show that $\mathbb{E}[1/KPT^*] < 12/KPT$. Observe that if Algorithm 14

terminates in the $i$-th iteration, it returns $KPT^* \geq n \cdot 2^{-i-1}$. Let $\zeta_i$ denote the event that Algorithm 14 stops in the $i$-th iteration. By Lemma 8, when $n \geq 2$ and $l \geq 1/2$, we have

$$\begin{aligned}
\mathbb{E}[1/KPT^*] &= \sum_{i=1}^{\log_2 n - 1} \left(2^{i+1}/n \cdot \Pr[\zeta_i]\right) \\
&< \sum_{i=j+2}^{\log_2 n - 1} \left(2^{i+1}/n \cdot \left(n^{-l \cdot 2^{i-j-1}} / \log_2 n\right)\right) + 2^{j+2}/n \\
&< (2^{j+3} + 2^{j+2})/n \leq 12/KPT.
\end{aligned}$$

This completes the proof. □

**Summary of steps.** In summary, our *RPS* algorithm works as follows. Given $G$, $k$, $A_C$, and two parameters $\epsilon$ and $l$, *RPS* first feeds $G$, $k$, and $A_C$ as input to Algorithm 14, and obtains a number $KPT^*$ in return. After that, *RPS* computes $\theta = \lambda / KPT^*$, where $\lambda$ is as defined in Equation 4.5 and is a function of $k$, $l$, $n$, and $\epsilon$. Finally, *RPS* gives $G$, $k$, $A_C$, and $\theta$ as input to Algorithm 12, whose output $S_k^*$ is the final result of the prevention maximization.

By Theorems 4 and 5, Equation 4.8, and the union bound, *RPS* runs in $O((k + l)(m + n)(1 + 1/\gamma) \log n / \epsilon^2)$ expected time, and returns a $(1 - 1/e - \epsilon)$-approximate solution with at least $1 - 2 \cdot n^{-l}$ probability. This success probability can easily be increased to $1 - n^{-l}$, by scaling $l$ up by a factor of $1 + \log 2 / \log n$.

### 4.4.3 Improved Parameter Estimation

This section presents a heuristic for improving the practical performance of *RPS*, without affecting its asymptotic guarantees, by improving the estimated lower bound on $OPT_L$.

**Algorithm 15** RefineKPT($\mathcal{G}$,$k$,$A_C$, $KPT^*$, $\epsilon'$)

1: Let $\lambda' = (2+\epsilon')ln\log n \cdot (\epsilon')^{-2}$.
2: Let $\theta' = \lambda'/KPT^*$.
3: Generate $\theta'$ random RRC sets; put them into a set $\mathcal{R}'$.
4: Initialize a node set $S'_k = \emptyset$.
5: **for** $i = 1$ to $k$ **do**
6: Identify node $v_i$ that covers the most RRC sets in $\mathcal{R}'$.
7: Add $v_i$ to $S'_k$.
8: Remove from $\mathcal{R}'$ all RRC sets that are covered by $v_i$.
9: Let $f$ be the fraction of the original $\theta'$ RRC sets that are covered by $S'_k$.
10: Let $KPT' = f \cdot n/(1+\epsilon')$
11: **return** $KPT^+ = max\{KPT', KPT^*\}$

The $KPT^*$ output by Algorithm 14 largely determines the efficiency of *RPS*. If $KPT^*$ is close to $OPT_L$, then $\theta = \lambda/KPT^*$ is small and Algorithm 12 only needs to generate a relatively small number of RRC sets. However, if $KPT^* \ll OPT_L$ then the efficiency of Algorithm 12 degrades rapidly and, in turn, so does the overall performance of *RPS*.

To remedy this issue, we add an intermediate step before Algorithm 12 to refine $KPT^*$ into a potentially tighter lower-bound of $OPT_L$. The intuition behind this heuristic is to generate a reduced number $\theta'$ of random RRC sets, placing them into a set $\mathcal{R}'$, and then apply the greedy approach (for the maximum coverage problem) on $\mathcal{R}'$ to obtain a good estimator for the maximum expected prevention in $\mathcal{R}'$. Thus, we can use the estimation as a good lower-bound for $OPT_L$.

Algorithm 15 shows the pseudo-code of the intermediate step. It first generates $\theta'$ random RRC sets and invokes the greedy approach for the maximum coverage problem on $\mathcal{R}'$ to obtain a size-$k$ node set $S'_k$. Algorithm 15 computes the fraction $f$ of RRC sets that are covered by $S'_k$ so that, by Corollary 1, $f \cdot n$ is an unbiased estimation of $\mathbb{E}[\pi(S'_k)]$. We set $\theta'$ based on the $KPT^*$ output by Algorithm 14 to a reasonably large number to ensure that $f \cdot n < (1+\epsilon') \cdot \mathbb{E}[\pi(S'_k)]$ occurs with at least $1 - n^{-l}$ probability. Based on this, Algorithm 15 computes $KPT' = f \cdot n/(1+\epsilon')$

scaling $f \cdot n$ down by a factor of $1 + \epsilon'$ to ensure that $KPT' \leq \mathbb{E}[\pi(S'_k)] \leq OPT_L$. The final output of Algorithm 15 is $KPT^+ = max\{KPT', KPT^*\}$. Below we give a lemma that shows the theoretical guarantees of Algorithm 15.

**Lemma 9.** *Given that* $\mathbb{E}[\frac{1}{KPT^*}] < \frac{12}{KPT}$*, Algorithm 15 runs in* $O(l(m+n)(1/(1-\gamma))\log n/(\epsilon')^2)$ *expected time. In addition, it returns* $KPT^+ \in [KPT^*, OPT_L]$ *with at least* $1 - n^{-l}$ *probability, if* $KPT^* \in [KPT/4, OPT_L]$.

*Proof.* We first analyze the expected time complexity of Algorithm 15. Observe that the expected time complexity of Lines 1-3 of Algorithm 15 is $O(\mathbb{E}[\frac{\lambda'}{KPT^*}] \cdot EPT)$, since they generate $\frac{\lambda'}{KPT^*}$ random RRC sets, each of which takes $O(EPT)$ expected time to generate. By Theorem 5, $\mathbb{E}[\frac{1}{KPT^*}] < \frac{12}{KPT}$. In addition, by Equation 4.9, $EPT \leq \frac{m}{n}(KPT + INF_C)$. Therefore,

$$
\begin{aligned}
&O\left(\mathbb{E}\left[\frac{\lambda'}{KPT^*}\right] \cdot EPT\right) \\
&\quad = O\left(\frac{\lambda'}{KPT} \cdot EPT\right) \\
&\quad = O\left(\frac{\lambda'}{KPT} \cdot \left(1 + \frac{m}{n}\right) \cdot \left(KPT + INF_C\right)\right) \\
&\quad = O\left(\frac{\lambda'}{KPT} \cdot \left(1 + \frac{m}{n}\right) \cdot KPT \cdot \left(1 + \frac{INF_C}{KPT}\right)\right) \\
&\quad = O(l(m+n)(1+1/\gamma)\log n/(\epsilon')^2).
\end{aligned}
$$

On the other hand, Lines 4-12 run in time linear to the total size of the RRC sets in $\mathcal{R}'$, i.e. the set of all RRC sets generated in Lines 1-3 of Algorithm 15. Given that the expected total size of the RRC sets in $\mathcal{R}'$ should be no more than $O(l(m+n)(1+1/\gamma)\log n)$, Lines 4-12 of Algorithm 15 have an expected time complexity of $O(l(m+n)(1+1/\gamma)\log n)$. Therefore, the expected time complexity of Algorithm 15 is $O(l(m+n)(1+1/\gamma)\log n/(\epsilon')^2)$.

Next, we prove that Algorithm 15 returns $KPT^+ \in [KPT^*, OPT_L]$ with high probability. First, observe that $KPT^+ \geq KPT^*$ trivially holds, as Algorithm 15 sets $KPT^+ = max\{KPT', KPT^*\}$, where $KPT'$ is derived in Line 11 of Algorithm 15. To show that $KPT^+ \in [KPT^*, OPT_L]$, it suffices to prove that $KPT' \leq OPT_L$.

By Line 11 of Algorithm 15, $KPT' = f \cdot n/(1+\epsilon')$, where $f$ is the fraction of RRC sets in $\mathcal{R}'$ that is covered by $S'_k$, where $\mathcal{R}'$ is a set of $\theta'$ random RRC sets, and $S'_k$ is a size-$k$ node set generated from Lines 4-9 in Algorithm 15. Therefore, $KPT' \leq OPT_L$ if and only if $f \cdot n \leq (1+\epsilon') \cdot OPT_L$.

Let $\rho'$ be the probability that a random RRC set is covered by $S'_k$. By Corollary 1, $\rho' = \mathbb{E}[\pi(S'_k)]/n$. In addition, $f \cdot \theta'$ can be regarded as the sum of $\theta'$ i.i.d. Bernoulii variables with a mean $\rho'$. Therefore, we have

$$\begin{aligned}
&\Pr[f \cdot n > (1+\epsilon') \cdot OPT_L] \\
&\quad \leq \Pr\left[n \cdot f - \mathbb{E}[\pi(S'_k)] > \epsilon' \cdot OPT_L\right] \\
&\quad = \Pr\left[\theta' \cdot f - \theta' \cdot \rho' > \frac{\theta'}{n} \cdot \epsilon' \cdot OPT_L\right] \\
&\quad = \Pr\left[\theta' \cdot f - \theta' \cdot \rho' > \frac{\epsilon' \cdot OPT_L}{n \cdot \rho'} \cdot \theta' \cdot \rho'\right]. \qquad (4.13)
\end{aligned}$$

Let $\delta = \epsilon' \cdot OPT_L/(n\rho')$. By the Chernoff bounds, we have

$$\begin{aligned}
\text{r.h.s. of Eqn. 4.13} &\leq \exp\left(-\frac{\delta^2}{2+\delta}\cdot\rho'\theta'\right) \\
&= \exp\left(-\frac{\epsilon'^2\cdot OPT_L^2}{2n^2\rho' + \epsilon' n\cdot OPT_L}\cdot\theta'\right) \\
&\leq \exp\left(-\frac{\epsilon'^2\cdot OPT_L^2}{2n\cdot OPT_L + \epsilon' n\cdot OPT_L}\cdot\theta'\right) \\
&= \exp\left(-\frac{\epsilon'^2\cdot OPT_L}{(2+\epsilon')\cdot n}\cdot\frac{\lambda'}{KPT^*}\right) \\
&\leq \exp\left(-\frac{\epsilon'^2\cdot\lambda'}{(2+\epsilon')\cdot n}\right) \leq \frac{1}{n^l}.
\end{aligned}$$

Therefore, $KPT' = f\cdot n/(1+\epsilon') \leq OPT_L$ holds with at least $1-n^{-l}$ probability. This completes the proof. □

Note that the time complexity of Algorithm 15 is smaller than that of Algorithm 14 by a factor of $k$, since the former only needs to accurately estimate the expected prevention of one node set (i.e. $S'_k$), whereas the latter needs to ensure accurate estimations for $\binom{n}{k}$ node sets simultaneously.

**Summary of steps.** In summary, we integrate Algorithm 15 into *RPS* and obtain an improved solution (referred to as $RPS^+$) as follows. Given $\mathcal{G}$, $k$, $A_C$, $\epsilon$, and $l$, we first invoke Algorithm 14 to derive $KPT^*$. After that, we feed $\mathcal{G}$, $k$, $A_C$, $KPT^*$, and a parameter $\epsilon^*$ (as defined in [86]) to Algorithm 15, and obtain $KPT^+$ in return. Then, we compute $\theta = \lambda/KPT^+$, where $\lambda$ is as defined in Equation 4.5. Finally, we run Algorithm 12 with $\mathcal{G}$, $k$, $A_C$, and $\theta$ as the input and get the output $S^*_k$ as the final result of prevention maximization.

By Theorems 4 and 5, Equation 4.8, and the union bound, *RPS* runs in $O((k+l)(m+n)(1/(1-\gamma))\log n/\epsilon^2)$ expected time and it can be verified that when $\epsilon' \geq \epsilon/\sqrt{k}$, $RPS^+$ has the same time complexity as *RPS*. Furthermore, $RPS^+$ returns a

$(1 - 1/e - \epsilon)$-approximate solution with at least $1 - 3n^{-l}$ probability and the success probability can be increased to $1 - n^{-l}$ by scaling $l$ up by a factor of $1 + \log 3 / \log n$.

Finally, we note that the time complexity of $RPS^+$ is near-optimal (up to the instance-specific factor $\gamma$) under the MCIC model, as it is only a $(\frac{1}{1-\gamma}) \log n$ factor larger than the $\Omega(m + n)$ lower-bound proved in Section 4.5 (for fixed $k$, $l$, and $\epsilon$).

## 4.5 Lower Bounds

**Comparison with *Greedy*.** As mentioned previously, *Greedy* runs in $O(kmnr)$ time, where $r$ is the number of Monte Carlo samples used to estimate the expected prevention of each node set. Budak et al. do not provide a formal result on how $r$ should be set to achieve a $(1 - 1/e - \epsilon)$-approximation ratio in the MCIC model; instead, the authors only point out that when each estimation of expected prevention has $\epsilon$ related error, *Greedy* returns a $(1 - 1/e - \epsilon)$-approximate solution for a certain $\epsilon'$ [19]. In the following lemma, we present a more detailed characterization of the relationship between $r$ and *Greedy*'s approximation ratio in the MCIC model:

**Lemma 10.** Greedy *returns a* $(1-1/e-\epsilon)$*-approximate solution with at least* $1-n^{-l}$ *probability, if*

$$r \geq (8k^2 + 2k\epsilon) \cdot n \cdot \frac{(l+1)\log n + \log k}{\epsilon^2 \cdot OPT_L}. \tag{4.14}$$

*Proof.* Let $S$ be any node set that contains no more than $k$ nodes in $G$, and $\xi(S)$ be an estimation of $\mathbb{E}[\pi(S)]$ using $r$ Monte Carlo steps. We first prove that, if $r$ satisfies Equation 4.14, then $\xi(S)$ will be close to $\mathbb{E}[\pi(S)]$ with a high probability.

Let $\mu = \mathbb{E}[\pi(S)]/n$ and $\delta = \epsilon OPT_L/(2kn\mu)$. By the Chernoff bounds, we have

$$
\begin{aligned}
&\Pr\left[|\xi(S) - \mathbb{E}[\pi(S)]| > \frac{\epsilon}{2k} OPT_L\right] \\
&\quad = \Pr\left[\left| r \cdot \frac{\xi(S)}{n} - r \cdot \frac{\mathbb{E}[\pi(S)]}{n} \right| > \frac{\epsilon}{2kn} \cdot r \cdot OPT_L \right] \\
&\quad = \Pr\left[\left| r \cdot \frac{\xi(S)}{n} - r \cdot \frac{\mathbb{E}[\pi(S)]}{n} \right| > \delta \cdot r \cdot \mu \right] \\
&\quad < 2\exp\left( - \frac{\delta^2}{2+\delta} \cdot r \cdot \mu \right) \\
&\quad = 2\exp\left( - \frac{\epsilon^2}{(8k^2 + 2k\epsilon) \cdot n} \cdot r \cdot \mu \right) \\
&\quad = 2\exp((l+1)\log n + \log k) \\
&\quad = \frac{1}{k \cdot n^{l+1}}.
\end{aligned} \tag{4.15}
$$

Observe that, given $\mathcal{G}$, $k$, and $A_C$ *Greedy* runs in $k$ iterations, each of which estimates the expected prevention f at most $n$ node sets with sizes no more than $k$. Therefore, the total number of node sets inspected by *Greedy* is at most $kn$. By Equation 4.15 and the union bound, with at least $1 - n^{-l}$ probability, we have

$$
|\xi(S') - \mathbb{E}[\pi(S')]| \leq \frac{\epsilon}{2k} OPT_L \tag{4.16}
$$

for all those $kn$ node sets $S'$ simultaneously. In what follows, we analyze the accuracy of *Greedy*'s output, under the assumption that for any node set $S'$ considered by *Greedy*, it obtains a sample of $\xi(S')$ that satisfies Equation 4.15. For convenience, we abuse notation and use $\xi(S')$ to denote the aforementioned sample.

Let $S_0 = \emptyset$, and $S_i \quad (i \in [1, k])$ be the node set selected by *Greedy* in the $i$-th iteration. We define $x_i = OPT_L - \pi(S_i)$, and $y_i(v) = \pi(S_{i-1} \cup \{v\}) - \pi(s_{i-1}))$ for any node $v$. Let $v_i$ be the nodes that maximizes $y_i(v_i)$. Then, $y_i(v_i) \geq x_{i-1}/k$ must hold; otherwise, for any size-$k$ node set $S$, we have

$$\begin{aligned}\pi(S) &\leq \pi(S_{i-1}) + \pi(S \setminus S_{i-1}) \\ &\leq \pi(S_{i-1}) + k \cdot y_i(v_i) \\ &< \pi(S_{i-1} + x_{i-1}) = OPT_L\end{aligned}$$

which contradicts the definition of $OPT_L$.

Recall that, in each iteration of *Greedy*, it add into $S_{i-1}$ the node $v$ that leads to the largest $\xi(S_{i-1} \cup \{v\})$. Therefore,

$$\xi(S_i) - \xi(S_{i-1}) \geq \xi(S_i \cup \{v\}) - \xi(S_{i-1}). \tag{4.17}$$

Combining Equations 4.16 and 4.17, we have

$$\begin{aligned}x_{i-1} - x_i &= \pi(S_i) - \pi(S_{i-1}) \\ &\geq \xi(S_i) - \frac{\epsilon}{2k} OPT_L - \xi(S_{i-1}) + \big(\xi(S_{i-1}) - \pi(S_{i-1})\big) \\ &\geq \xi(S_{i-1} \cup \{v_i\}) - \xi(S_{i-1}) - \frac{\epsilon}{2k} OPT_L + \big(\xi(S_{i-1}) - \pi(S_{i-1})\big) \\ &\geq \pi\big(S_{i-1} \cup \{v_i\}\big) - \pi(S_{i-1}) - \frac{\epsilon}{k} OPT_L \\ &\geq \frac{1}{k} x_{i-1} - \frac{\epsilon}{k} OPT_L. \end{aligned} \tag{4.18}$$

Equation 4.18 leads to

$$
\begin{aligned}
x_k &\leq \big(1-\frac{1}{k}\big)\cdot x_{k-1} + \frac{\epsilon}{k}OPT_L \\
&\leq \big(1-\frac{1}{k}\big)^2 \cdot x_{k-2} + \big(1+\big(1-\frac{1}{k}\big)\big)\cdot\frac{\epsilon}{k}OPT_L \\
&\leq \big(1-\frac{1}{k}\big)^k \cdot x_0 + \sum_{i=0}^{k-1}\big(\big(1-\frac{1}{k}\big)^i \cdot \frac{\epsilon}{k}OPT_L\big) \\
&= \big(1-\frac{1}{k}\big)^k \cdot OPT_L + \big(1-\big(1-\frac{1}{k}\big)^k\big)\cdot\epsilon\cdot OPT_L \\
&\leq \frac{1}{e}\cdot OPT_L - \big(1-\frac{1}{e}\big)\cdot\epsilon\cdot OPT_L.
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\pi(S_k) &= OPT_L - x_k \\
&\leq (1-1/e)\cdot(1-\epsilon)\cdot OPT_L \\
&\leq (1-1/e-\epsilon)\cdot OPT_L.
\end{aligned}
$$

Thus, the lemma is proved. □

Assume that we know $OPT_L$ in advance and set $r$ to the smallest value satisfying the above inequality, in *Greedy*'s favour. In that case, the time complexity of *Greedy* is $O(k^3lmn^2\epsilon^{-2}\log n/OPT_L)$. Towards comparing *Greedy* to RPS, we show the following upper bound on the value of $\gamma$:

**Claim 1.** $\gamma \leq \frac{n}{n+1}$

*Proof.* Let $M_R$ be the random instance of $I(v_c)$ used to compute $R$. Furthermore, let us assume that $|M| \geq 2$ so that at least one non-seed node is influenced by campaign $C$. Then, from Lemma 5 and the definition of $\gamma$ we have

$$\frac{1}{\gamma} = 1 + \frac{m}{n} \cdot \frac{\mathbb{E}[\pi(\{v^*\})]}{\mathbb{E}[\omega(M_R)]}.$$

Then, observe that the expected number of nodes saved by $v^*$ is at least $\Pr[v^* \in M_R]$. That is, if $v^* \in M_R$ then campaign $L$ can save at least one node, namely $v^*$ itself. Giving

$$\begin{aligned}\Pr[v^* \in M_R] &= \sum_{v \in M_R} \frac{deg(v)}{m} \\ &\geq \sum_{v \in M_R} \frac{1}{m} \\ &= \frac{|M_R|}{m}.\end{aligned}$$

Therefore, $\frac{\mathbb{E}[\pi(\{v^*\})]}{\mathbb{E}[\omega(M_R)]} \geq \frac{1}{m}$. Then, we have $\frac{1}{\gamma} \geq 1 + \frac{m}{n} \cdot \frac{1}{m} = 1 + \frac{1}{n}$. Thus, we get

$$\gamma \leq \frac{n}{n+1},$$

which proves the claim. □

Claim 1 shows that the expected runtime for *RPS* is at most $O((k+l)mn\epsilon^{-2} \log n)$. As a consequence, given that $OPT_L \leq n$, the expected runtime of *Greedy* is always more than the expected runtime of *RPS*. In practice, we observe that for typical social networks $OPT_L \ll n$ and $\frac{1}{1-\gamma} \ll n + 1$ resulting in superior scalability of *RPS* compared to *Greedy*. Table 4.3 confirms that $\frac{1}{1-\gamma} \ll n + 1$ on our small datasets.

**A Lower Bound for EIL.** In the theorem below, we provide a lower bound on the time it takes for any algorithm to compute a $\beta$-approximation for the EIL problem given uniform node sampling and an adjacency list representation. Thus, we rule out the possibility of a sublinear time algorithm for the EIL problem for an arbitrary $\beta$.

**Theorem 6.** *Let* $0 < \epsilon < \frac{1}{10e}$, $\beta \leq 1$ *be given. Any randomized algorithm for EIL that returns a set of seed nodes with approximation ratio* $\beta$*, with probability at least* $1 - \frac{1}{e} - \epsilon$*, must have a runtime of at least* $\frac{\beta(m+n)}{24 \min\{k, 1/\beta\}}$.

The proof of Theorem 6 uses Yao's Minmax Lemma for the performance of Las Vegas (LV) randomized algorithms on a family of inputs [89]. Precisely, the lemma states that the least expected cost of deterministic LV algorithms on a distribution over a family of inputs is a lower bound on the expected cost of the optimal randomized LV algorithm over that family of inputs. We build such an input family of lower bound graphs via the use of a novel gadget.

*Proof.* Throughout the proof we assume all edge probabilities for both campaigns are 1.

Note first that for a graph consisting of $p = n/2$ connected pairs for which each pair contains a node $u \in A_C$, an algorithm must return at least $\beta k$ nodes to obtain an approximation ratio of $\beta$. Doing so in at most $\beta^2 n/2$ queries requires that $2\beta k \leq \beta^2 n$, which implies $2k/\beta \leq n$. We can therefore assume $2k/\beta \leq n$.

The proof will use Yao's Minmax Principle for the performance of Las Vegas (LV) randomized algorithms on a family of inputs CITE. The lemma states that the least expected cost of deterministic LV algorithms on a distribution over a family of inputs is a lower bound on the expected cost of the optimal randomized LV algorithm over that family of inputs. Define the cost of the algorithm as 0 if it returns a set of seed nodes with approximation ratio better than $\beta$ and 1 otherwise. As the cost of an algorithm equals its probability of failure, we can think of it as a LV algorithm.

Assume for notational simplicity that $\beta = 1/T$ where $T$ is an integer. We will build a family of lower bound graphs, one for each value of $n$ (beginning from $n = 1 + T$); each graph will have $m \leq n$, so it will suffice to demonstrate a lower bound of $\frac{n}{12T \min\{k, T\}}$.

We now consider the behaviour of a deterministic algorithm $A$ with respect to the uniform distribution on the constructed family of inputs. For a given value $T$ the graph would be made from $k$ components of size $2T$ and $p = \frac{n-2kT}{2}$ connected pairs (recall that $2kT = 2k/\beta \leq n$). Specifically, the $k$ components of size $2T$ are structured as follows: for each component there is a *hub* node $v_h$ that is connected to $2T-2$ leaf nodes and a node $u \in A_C$. Furthermore, each of the $p$ pairs also contains one node $u \in A_C$. If algorithm $A$ returns seed nodes from $l$ of the $k$ components of size $2T$, it achieves a total prevention of $l \cdot (2T-1) + (k-l)$ since choosing either the hub node $v_h$ or any leaf node will result in saving all $2T-1$ eligible nodes in the component. Thus, to attain an approximation factor better than $\beta = \frac{1}{T}$, we must have $l \cdot (2T-1) + (k-l) \geq \frac{1}{T} \cdot k \cdot (2T-1)$, which implies $l \geq \frac{k}{2T}$ for any $T > 1$.

Suppose $k > 12T$. The condition $l \geq \frac{k}{2T}$ implies that at least $\frac{k}{2T}$ of the large components must be queried by the algorithm, where each random query has probability $\frac{k\cdot(2T-1)}{n-(p+k)} \geq \frac{kT}{n}$ of hitting a large component. If the algorithm makes fewer than $\frac{n}{6T^2}$ queries, then the expected number of components hit is $\frac{n}{6T^2} \cdot \frac{kT}{n} = \frac{k}{6T}$. Chernoff bounds then imply that the probability of hitting more than $\frac{k}{2T}$ components is no more than $e^{-\frac{k}{6T}\cdot 2/3} \leq \frac{1}{e^{4/3}} < 1 - \frac{1}{e} - \epsilon$, a contradiction.

If $k \leq 12T$ then we need that $l \geq 1$, which occurs only if the algorithm queries at least one of the $k \cdot (2T-1)$ vertices in the large components. With $\frac{n}{k\cdot(2T-1)}$ queries, for $n$ large enough, this happens with probability less than $\frac{1}{e} - \epsilon$, a contradiction.

We conclude that, in all cases, at least $\frac{n}{24T\min\{k,T\}}$ queries are necessary to obtain an approximation factor better than $\beta = \frac{1}{T}$ with probability at least $1 - \frac{1}{e} - \epsilon$, as required.

By Yao's Minmax Principle this gives a lower bound of $\Omega(\frac{n}{24T\min\{k,T\}})$ on the expected performance of any randomized algorithm, on at least one of the inputs.

Finally, the construction can be modified to apply to non-sparse networks. For any

$d \leq n$, we can augment our graph by overlaying a $d$-regular graph with exponentially small weight on each edge. This does not significantly impact the prevention of any set, but increases the time to decide if a node is in a large component by a factor of $O(d)$ (as edges must be traversed until one with non-exponentially-small weight is found). Thus, for each $d \leq n$, we have a lower bound of $\Omega(\frac{nd}{24T\min\{k,T\}})$ on the expected performance of $A$ on a distribution of networks with $m = nd$ edges. □

## 4.6 Experiments

In this section, we present our experimental results. All of our algorithms are implemented in C++ (available at `https://github.com/stamps`) and tested on a machine with dual 6 core 2.10GHz Intel Xeon CPUs, 128GB RAM and running Ubuntu 14.04.2.

**Datasets.** The network statistics for all of the datasets we consider are shown in Table 4.2. We obtained the datasets from Laboratory of Web Algorithmics.[3] We divide the datasets by horizontal lines according to their size, small (S), medium (M), and large (L).

**Propagation Model.** We consider the MCIC model (see Section 2.1) of Budak et al. We set the propagation probability of each edge $e$ as follows: we first identify the node $v$ that $e$ points to, and then set $p(e) = 1/i$, where $i$ denotes the in-degree of $v$. This setting of $p(e)$ is widely adopted in prior work [23, 22, 46, 87].

**Parameters.** Unless otherwise specified, we set $\epsilon = 0.1$ in our experiments. We set $l$ in a way that ensures a success probability of $1 - 1/n$. For *Greedy*, we set the number of Monte Carlo steps to $r = 10000$, following the standard practice in the literature. Note that this choice of $r$ is to the advantage of *Greedy* because the value

[3] `http://law.di.unimi.it/datasets.php`

of $r$ required to achieve the same theoretical guarantees as *RPS* in our experiments is always much larger than 10000. In each of our experiments, we repeat each run ten times and report the average result. For simplicity, we set $\Delta = 0$ since the drop off in accuracy of the *Greedy* approach ([19]) as $\Delta$ increases is well documented by Budak et al.

We are interested in simulating the misinformation prevention process when the bad campaign $C$ has a sizable influence on the network to best demonstrate how the techniques could be used in real world settings. That is, we believe the scenario in which we are attempting to prevent the spread of misinformation when the bad campaign has the ability to influence a large fraction of the network to be more relevant than when only a very small number of users would adopt the bad campaign. Towards this end, we first compute the *top-k* influential vertices for each network and then randomly select the seed set $A_C$ from the *top-1* and *top-5* vertices for each experiment. This process ensures the misinformation has a large potential influence in the network.

The focus of our experiments is *algorithm efficiency* measured in runtime where our goal is to demonstrate the superior performance of *RPS* compared to *Greedy*. Meanwhile, we observed that the *algorithm accuracy* (measured in percentage of nodes saved) of *RPS* matches *Greedy* very closely. Precisely, we observe that, consistent with the results reported in [19], *RPS* quickly approaches a maximal expected prevention value as $k$ increases across all datasets. This is natural since both *RPS* and *Greedy* are maximizing a submodular objective function in a greedy fashion. The novelty of *RPS* addresses the scalability hurdle in a similar sense to Borgs et al. [15] in relation to Kempe et al. [52]. For a detailed comparison of the accuracy of *Greedy* compared to a number of natural heuristics we refer the interested reader to [19].

**Running-time Results.** First, we plot the runtimes of *Greedy* and *RPS* for a

| Name | $\|V\|$ | $\|E\|$ | Average degree |
|---|---|---|---|
| nethept | 15,229 | 62,752 | 4.1 |
| word_assoc | 10,617 | 72,172 | 6.8 |
| dblp-2010 | 326,186 | 1,615,400 | 6.1 |
| cnr-2000 | 325,557 | 3,216,152 | 9.9 |
| ljournal-2008 | 5,363,260 | 79,023,142 | 28.5 |

Table 4.2: Dataset Statistics

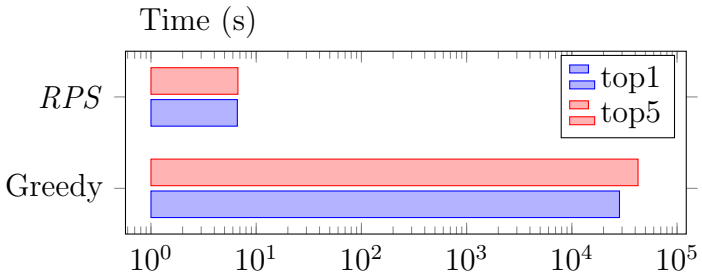



Figure 4.3: Runtimes comparison between *RPS* and *Greedy* for the wordassociation-2011 dataset.

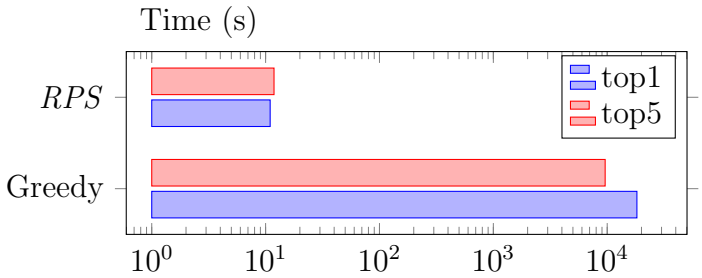



Figure 4.4: Runtimes comparison between *RPS* and *Greedy* for the nethept dataset.

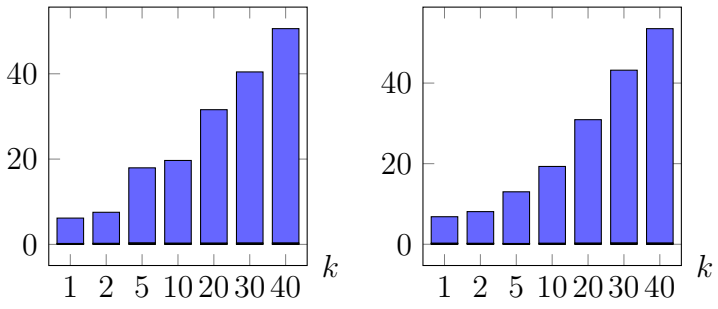



Figure 4.5: Breakdown of computation time (s) for the word_assoc datasets. Blue stack corresponds to Algorithm 12, red to Algorithm 15, and green (which is almost invisible) to Algorithm 14. Left: top1 and right: top5.

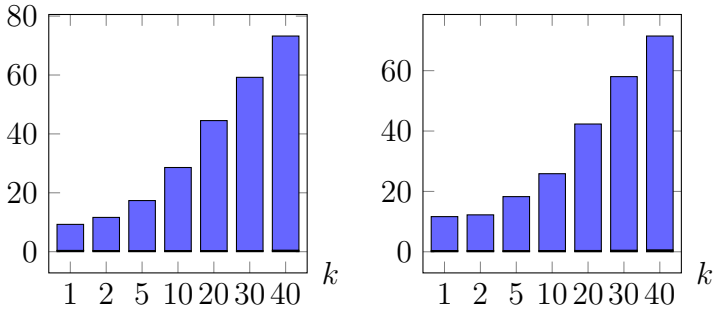



Figure 4.6: Breakdown of computation time (s) for nethept dataset. Blue stack corresponds to Algorithm 12, red to Algorithm 15, and green (which is almost invisible) to Algorithm 14. Left: top1 and right: top5.

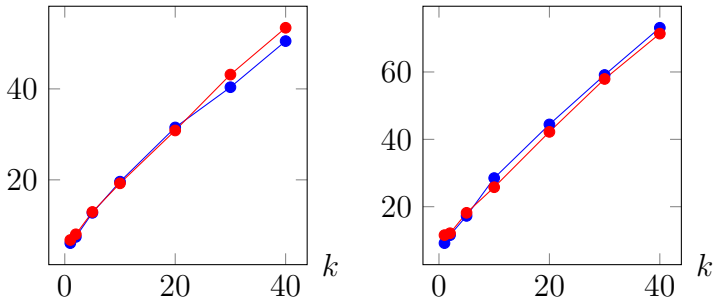



Figure 4.7: Runtimes (s) for small datasets. word_assoc on the left and nethept on the right with blue for top1 and red for top5.

single seed in Figures 4.3 & 4.4 and observe that *RPS* provides a significant improvement of several orders of magnitude over *Greedy*. Note, we only compare *RPS* to *Greedy* on the smallest networks due to the substantial runtime required for *Greedy*. Furthermore, for similar scaling issues of *Greedy*, we restrict our comparison to $k = 1$. However, since both approaches scale linearly with $k$ we can conclude that *RPS* offers a tremendous runtime improvement over the approach of [19].

Next, we show the total runtimes (Figures 4.7, 4.10, 4.9, 4.11) and computation breakdowns (Figures 4.5, 4.6, 4.13, 4.12, 4.8) for each dataset. We observe that the vast majority of the computation time is spent on generating the RRC sets for $\mathcal{R}$. Furthermore, the amount of time spent on refining $KPT$ increases across all datasets though remains a small fraction of the overall runtime. As expected, the time spent computing $KPT^*$ remains a very small fraction of total computation time due to the small number of iterations of Algorithm 14 and only takes up a relatively large fraction of the computation time on the cnr-2000 top5 dataset (Figure 4.13). The density of the cnr-2000 network leads to larger RRC sets that results in a larger fraction of time spent on computing $KPT^*$ and refining $KPT^+$.

We now compare the runtime trends of our results for the EIL problem to those of [86] for the IM problem. Tang et al. report that, when $k$ increases, the runtime of their approach (*TIM*) tends to decrease before eventually increasing. The authors explain this by considering the breakdown of the computation times required by each algorithm in *TIM*. The authors observe that the computation time is mainly incurred by their analog to Algorithm 1 (the node selection phase) which is primarily determined by the number $\theta$ of RR sets that need to be generated. The authors of [86] have $\theta = \lambda / KPT^+$, where $\lambda$ is analogous to ours, and $KPT^+$ is a lower-bound on the optimal influence of a size-$k$ node set. In both the IC and MCIC models, the analogs of $\lambda$ and $KPT^+$ increase with $k$, and it happens that for the IM problem,

| | word_assoc | | nethept | |
|---|---|---|---|---|
| k | top1 | top5 | top1 | top5 |
| 1 | 23.4471 | 25.9804 | 48.3194 | 48.619 |
| 10 | 24.8521 | 26.6518 | 60.5 | 43.1875 |
| 20 | 24.7509 | 25.2607 | 57.0111 | 61.4167 |
| *max* | 10,618 | 10,618 | 15,230 | 15,230 |

Table 4.3: $\frac{1}{1-\gamma}$ values for small datasets.

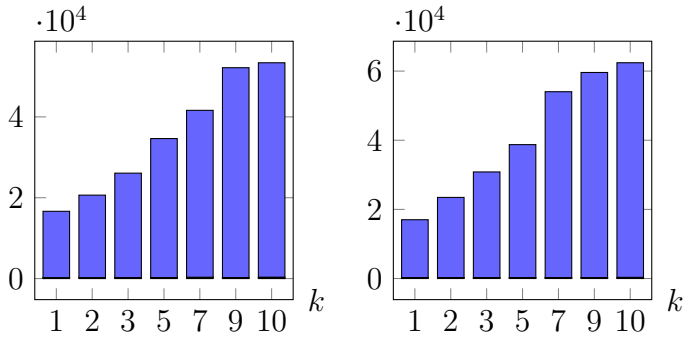



Figure 4.8: Breakdown of computation time (s) for ljournal-2008. Blue stack corresponds to Algorithm 12, red to Algorithm 15, and green to Algorithm 14. Left is top1 and right is top5.

Tang et al. observe that the increase of $KPT^+$ is more pronounced than that of $\lambda$ for smaller values of $k$, which leads to the decrease in *TIM*'s runtime.

On the contrary, for the EIL problem, the increase of $KPT^+$ does not dominate to a point that the runtime of *RPS* decreases as $k$ increases. Instead, we see a linear increase in runtime as $k$ increases for all the networks considered (Figures 4.7, 4.10, 4.9, 4.11). To explain, consider how $KPT^+$ grows in each setting. In the MCIC model we see that $KPT^+$ rapidly approaches its maximal value which corresponds to the growth of $KPT^+$ plateauing much sooner. In contrast, in the IC model, the analogous $KPT^+$ value continues to grow at a significant rate for a wider range of $k$ values since the ceiling for the maximal influence is not tied to a second campaign, as it is in the MCIC model. As such, the influence estimates do not level off as quickly. This translates to the growth of $KPT^+$ outpacing the growth of $\lambda$.

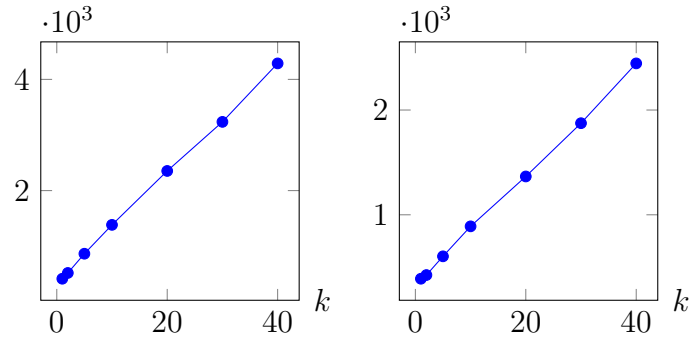



Figure 4.9: Runtimes (s) for the dblp dataset. Left: top1 and right: top5.

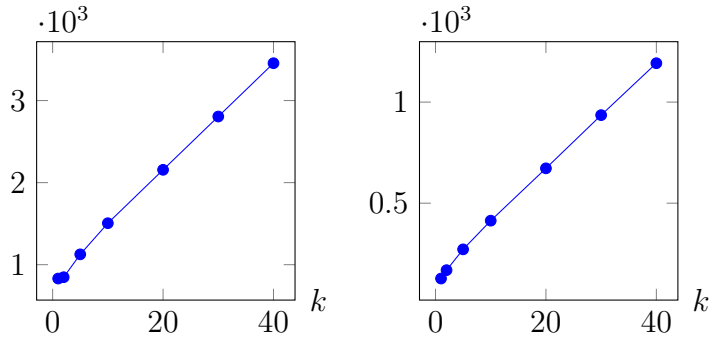



Figure 4.10: Runtimes (s) for the cnr dataset. Left: top1 and right: top5.

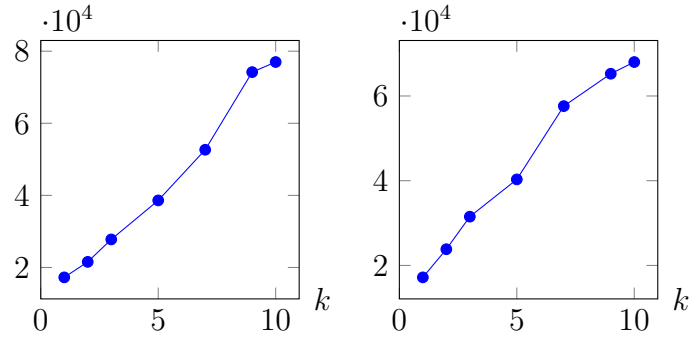



Figure 4.11: Runtimes (s) for large dataset. Listed left to right: ljournal-2008 top1, ljournal-2008 top5.

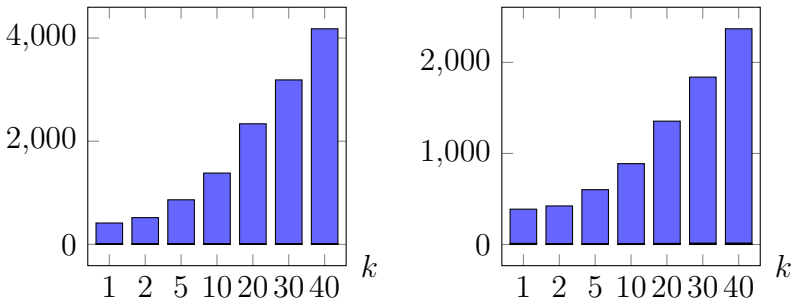



Figure 4.12: Breakdown of computation time (s) for the dblp dataset. Blue stack corresponds to Algorithm 12, red to Algorithm 15, and green to Algorithm 14. Leftt: top1 and right: top5.

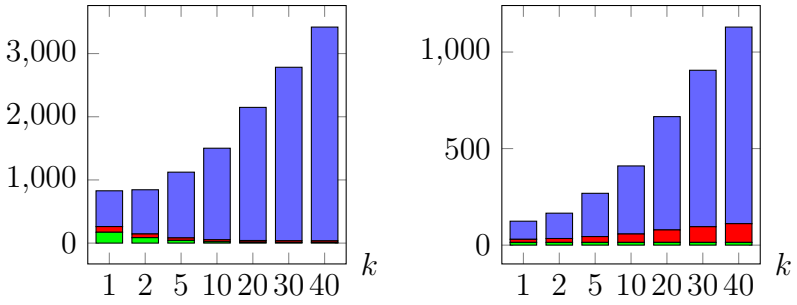



Figure 4.13: Breakdown of computation time (s) for the cnr dataset. Blue stack corresponds to Algorithm 12, red to Algorithm 15, and green to Algorithm 14. Left: top1 and right: top5.

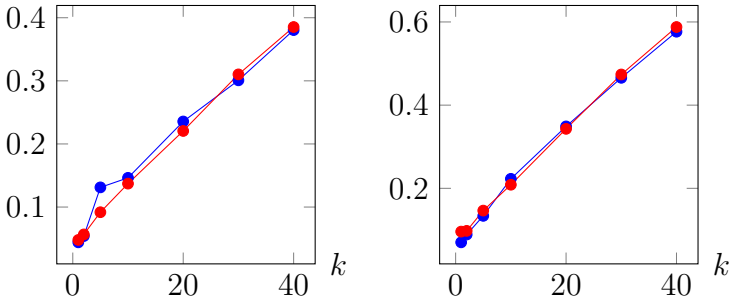



Figure 4.14: Memory consumption (Gb) for the small datasets. Blue corresponds to top1 and red to top5. Listed left to right: word_assoc, nethept.

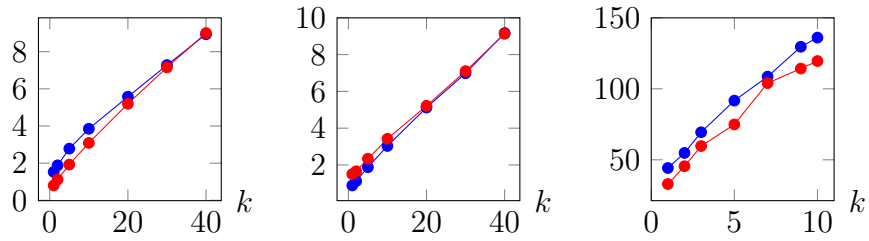



Figure 4.15: Memory consumption (Gb) for the medium and large datasets. Blue corresponds to top1 and red to top5. Listed left to right: cnr, dblp, ljournal.

**Memory Consumption.** Our final set of experiments monitors the memory consumption required to store the RRC set structure $\mathcal{R}$. We observe that the size of $\mathcal{R}$ for the EIL problem is larger than that required by the IM problem. In the "hypergraph" nomenclature due to Borgs et al. $\mathcal{R}$ is viewed as a hypergraph with each RRC set in $\mathcal{R}$ corresponding to a hyperedge. We observe that the hyperedges generated for the IM problem are non-empty in every iteration of the algorithm. Additionally, each hyperedge has relatively small size. The result is that the hypergraph generated for the IM problem is very dense, but each hyperedge is relatively "light" (i.e. it contains few nodes).

In contrast, in each iteration of *RPS* we have a substantial probability to produce an empty RRC set, since we require that a randomly selected node is in the randomized BFS tree resulting from the influence propagation process initialized at $A_C$. Even though our seed sets for campaign $C$ are chosen at random from the *top-k* sets, and thus are highly influential (resulting in a large randomized BFS tree), the probability of overlap with a randomly selected node is still small due to the vast size of the networks considered. These empty RRC sets are necessary for the computation of the expected prevention to be accurate, but results in a hypergraph that differs significantly from that of the IM problem.

In particular, since the *generateRRC* algorithm is a deterministic BFS (with spe-

cialized stopping conditions to account for cutoff nodes) it reaches a much larger fraction of the network. Therefore, while there are far fewer non-empty hyperedges generated, they are much larger in size: often on the order of half the network. Thus, the resulting hypergraph is sparse, but contains very "heavy" edges.

These two opposing metrics, a dense hypergraph with "light" hyperedges versus a sparse hypergraph with "heavy" hyperedges, result in the latter requiring more memory to store. Memory consumption statistics are shown in Figures 4.14 & 4.15. We observe that, while the size of the RRC sets when considering top5 seed nodes will on average be smaller, the increased number of samples required results in more memory usage on average for top5 seeds sets. This separation is most pronounced on the massive ljournal-2008 dataset. Despite a larger memory requirement compared to the single campaign setting we show that our approach has the ability to scale far beyond what was achieved by Budak et al. and provides orders of magnitude improvement for the runtime.

## 4.7 Future Work

In future work we plan to apply the techniques introduced for *RPS* to the multi-campaign LT model as introduced in [44] in order to have solutions in the two most prominent influence propagation models. Additionally, we are motivated to further reduce the $\gamma$ factor of *RPS* by investigating if we can sample directly from the non-empty RRC sets and avoid unnecessary simulation the spread of the bad campaign.

# Chapter 5

# Conclusions

**Misinformation Removal.** We studied the problem of minimizing the time required to eliminate misinformation propagating through a social network given a budget of searchers. Since the search time problem is NP-complete, even for directed acyclic graphs, we introduced an updated approximation algorithm for clearing networks that allows for greater scalability. We investigated how a natural split and conquer style strategy compares to our strategy and found that our approach outperforms the (intuitively appealing) split and conquer strategy on a broad range of networks. Furthermore, the experimental analysis showed that our general strategy for clearing a network, which uses our Plank algorithm as a subroutine, was able to achieve scalability to very large online networks, on the order of billions of edges.

**Feedback Arc Set.** Our investigation of the feedback arc set problem resulted in a thorough experimental comparison of several algorithms on a range of datasets. We presented highly optimized implementations of GreedyFAS, BergerShorFAS, and the sorting-based heuristics. Within each of the three complexity classes of algorithms we observed an overall trend showing a trade off between scalability and quality. We observed an approximate maximum scalability of 300K arcs for the $O(n^2)$ algorithms, 3.5M arcs for the $O(n \log n)$ algorithms, and 50B arcs for the $O(m+n)$ algorithms. We

achieved approximate FAS sizes of 3-20%, 23-40%, and 11-17% for the best algorithms of the $O(m+n)$, $O(n \log n)$, and $O(n^2)$ runtime categories, respectively. We conclude that GreedyFAS and BergerShorFAS provide the best balance between scalability and solution quality. GreedyFAS is the algorithm that produces always either the smallest or a very close second smallest FAS size while being a fast algorithm in general. In particular our G-arr implementation scales to the biggest dataset we consider, clueweb12, with more than 42 billion arcs.

In addition, we presented global graph properties that signal the expected performance of several algorithm for the FAS problem. In particular, we can look to the skew of the top-$k$ vertices as an indication of when the GreedyFAS and BergerShorFAS algorithms are expected to perform favourably. For the sorting-based approaches, we can use the sparsity of $G$ as an indication of the expected performance. Finally, we presented a detailed discussion of the reasons behind the relative performance observed between the approaches considered.

**Misinformation Prevention.** We presented *RPS*, a novel and scalable approach to the EIL problem for studying the prevention of misinformation propagating through a social network. We gave proofs of correctness and a detailed running-time analysis of our approach. Furthermore, we experimentally verified the performance of our algorithm on a collection of large social networks and observed a significant improvement over the state-of-the-art *Greedy* approach. Finally, we provided additional theoretical results in the form of two lower bounds: one on the running-time requirement for any approach to solve the EIL problem and another of the number of Monte Carlo simulations required by *Greedy* to return a correct solution with high probability. We follow up the second lower bound result by showing that it implies that the expected runtime of *RPS* is always less than the expected runtime of *Greedy*.

# Bibliography

[1] Bob Abeshouse. Troll factories, bots and fake news: Inside the wild west of social media, 2018. https://www.aljazeera.com/blogs/americas/2018/02/troll-factories-bots-fake-news-wild-west-social-media-180207061815575.html, Last accessed on 2018-04-18.

[2] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 2008.

[3] Ali Baharev, Hermann Schichl, Arnold Neumaier, and TOBIAS Achterberg. An exact method for the minimum feedback arc set problem. *University of Vienna*, 10:35–60, 2015.

[4] Reuven Bar-Yehuda, Ann Becker, and Dan Geiger. Randomized algorithms for the loop cutset problem. *JAIR*, 2000.

[5] János Barát. Directed path-width and monotonicity in digraph searching. *Graphs and Combinatorics*, 22(2):161–172, 2006.

[6] Bonnie Berger and Peter W. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In *SODA'90*, 1990.

[7] Dietmar Berwanger, Anuj Dawar, Paul Hunter, Stephan Kreutzer, and Jan ObdráLek. The dag-width of directed graphs. *J. Comb. Theory Ser. B*, 102(4):900–923, July 2012.

[8] Shishir Bharathi, David Kempe, and Mahyar Salek. Competitive influence maximization in social networks. In *WINE'07*, pages 306–311, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Bozhena Bidyuk and Rina Dechter. Cutset sampling for bayesian networks. *JAIR*, 2007.

[10] D. Bienstock. Graph searching, path-width, tree-width and related problems. *DIMACS Ser. in Discrete Mathematics and Theoretical Computer Science*, 5:33–49, 1991.

[11] Paolo Boldi, Francesco Bonchi, Aristides Gionis, and Tamir Tassa. Injecting uncertainty in graphs for identity obfuscation. *Proceedings of the VLDB Endowment*, 2012.

[12] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[13] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*, 2004.

[14] Béla Bollobás, Christian Borgs, Jennifer Chayes, and Oliver Riordan. Directed scale-free graphs. In *SODA'03*, 2003.

[15] Christian Borgs, Michael Brautbar, Jennifer T. Chayes, and Brendan Lucier. Influence maximization in social networks: Towards an optimal algorithmic solution. *CoRR*, abs/1212.0884, 2012.

[16] Richard Borie, Craig Tovey, and Sven Koenig. Algorithms and complexity results for graph-based pursuit evasion. *Auton. Robots*, 31(4):317–332, November 2011.

[17] Franz J Brandenburg and Kathrin Hanauer. Sorting heuristics for the feedback arc set problem. Technical report, University of Passau, Germany, 2011.

[18] Franz J. Brandenburg and Stephanie Herrmann. Graph searching and search time. In *SOFSEM 2006: Theory and Practice of Computer Science*, volume 3831 of *Lecture Notes in Computer Science*, pages 197–206. 2006.

[19] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *WWW'11*, 2011.

[20] Tim Carnes, Chandrashekhar Nagarajan, Stefan M. Wild, and Anke van Zuylen. Maximizing influence in a competitive social network: a follower's perspective. In *ICEC '07*, pages 351–360, New York, NY, USA, 2007. ACM.

[21] Wei Chen, Laks V. S. Lakshmanan, and Carlos Castillo. *Information and Influence Propagation in Social Networks*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[22] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *KDD '09*, pages 199–208, New York, NY, USA, 2009. ACM.

[23] Wei Chen, Yifei Yuan, and Li Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM'10*, 2010.

[24] Tom Coleman and Anthony Wirth. Ranking tournaments: Local search and a new algorithm. *J. Exp. Algorithmics*, 2010.

[25] Camil Demetrescu and Irene Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 2003.

[26] Nick D. Dendris, Lefteris M. Kirousis, and Dimitrios M. Thilikos. Fugitive-search games on graphs and related parameters. In *WG '94*, pages 331–342, London, UK, UK, 1995. Springer-Verlag.

[27] Dariusz Dereniowski, Wieslaw Kubiak, and Yori Zwols. Minimum length path decompositions. *CoRR*, abs/1302.2788, 2013.

[28] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The anatomy of a scientific rumor. *Scientific Reports*, 3, 01 2013.

[29] Richard Durstenfeld. Algorithm 235: random permutation. *Communications of the ACM*, 7(7):420, 1964.

[30] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Inf. Process. Lett.*, 1993.

[31] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Inf. Comput.*, 113(1):50–79, August 1994.

[32] Guy Even, Joseph (Seffi) Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In *IPCO'95*, 1995.

[33] Lidan Fan, Zaixin Lu, Weili Wu, Bhavani Thuraisingham, Huan Ma, and Yuanjun Bi. Least cost rumor blocking in social networks. In *Distributed Computing*

*Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 540–549. IEEE, 2013.

[34] Merrill M. Flood. Exact and heuristic algorithms for the weighted feedback arc set problem: A special case of the skew-symmetric quadratic assignment problem. *Networks*, 20(1):1–23, 1990.

[35] Peter Foster. 'bogus' ap tweet about explosion at the white house wipes billions off us markets, 2018. `https://www.telegraph.co.uk/finance/markets/10013768/Bogus-AP-tweet-about-explosion-at-the-White-House-wipes-billions-off-US-markets.html`, Last accessed on 2018-04-18.

[36] Peter Foster. Facebook's failure: did fake news and polarized politics get trump elected?, 2018. `https://www.theguardian.com/technology/2016/nov/10/facebook-fake-news-election-conspiracy-theories`, Last accessed on 2018-04-18.

[37] András Frank. How to make a digraph strongly connected. *Combinatorica*, 1981.

[38] Wolfgang Gatterbauer, Stephan Günnemann, Danai Koutra, and Christos Faloutsos. Linearized and single-pass belief propagation. *PVLDB*, 2015.

[39] Amit Goyal, Francesco Bonchi, Laks V. S. Lakshmanan, and Suresh Venkatasubramanian. On minimizing budget and time in influence propagation over social networks. *Social Netw. Analys. Mining*, 3(2):179–192, 2013.

[40] Chris Graham. Youtube employee's twitter account hacked to spread fake news during attack, 2018. `https://www.telegraph.co.uk/technology/2018/04/04/youtube-employees-twitter-account-hacked-spread-fake-news-attack/`, Last accessed on 2018-04-18.

[41] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 1985.

[42] Mangesh Gupte, Pravin Shankar, Jing Li, S. Muthukrishnan, and Liviu Iftode. Finding hierarchy in directed online social networks. In *WWW '11*, pages 557–566, New York, NY, USA, 2011. ACM.

[43] Laura Hautala. Reddit was a misinformation hotspot in 2016 election, study says, 2018. `https://www.cnet.com/news/reddit-election-misinformation-2016-research/`, Last accessed on 2018-04-18.

[44] Xinran He, Guojie Song, Wei Chen, and Qingye Jiang. *Influence Blocking Maximization in Social Networks under the Competitive Linear Threshold Model*, pages 463–474. 2012.

[45] Paul Hunter and Stephan Kreutzer. Digraph measures: Kelly decompositions, games, and orderings. *Theor. Comput. Sci.*, 399(3):206–219, June 2008.

[46] Kyomin Jung, Wooram Heo, and Wei Chen. Irie: Scalable and robust influence maximization in social networks. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 918–923. IEEE, 2012.

[47] M. Junger. Polyhhedral combinatorics and the acyclic subgraph problem. *Research and Exposition in Mathematics*, 7, 1985.

[48] Viggo Kann. *On the approximability of NP-complete optimization problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992.

[49] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer US, 1972.

[50] Marek Karpinski and Warren Schudy. Faster algorithms for feedback arc set tournament, kemeny rank aggregation and betweenness tournament. In *ISAAC'10*, 2010.

[51] David Kempe. Structure and dynamics of information in networks. *Lecture Notes*, 2011.

[52] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *KDD'03*, 2003.

[53] Claire Kenyon-Mathieu and Warren Schudy. How to rank with few errors. In *STOC'07*, 2007.

[54] M Kirousis and C H Papadimitriou. Searching and pebbling. *Theor. Comput. Sci.*, 47(2):205–218, November 1986.

[55] Philip Klein, Serge Plotkin, Clifford Stein, and Eva Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput.*, 1994.

[56] George Kollios, Michalis Potamias, and Evimaria Terzi. Clustering large probabilistic graphs. *TKDE*, 2013.

[57] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *FOCS'88*, 1988.

[58] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *WWW '10*, pages 641–650, New York, NY, USA, 2010. ACM.

[59] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[60] Rong-Hua Li, Jeffrey Xu Yu, Rui Mao, and Tan Jin. Recursive stratified sampling: A new framework for query evaluation on uncertain graphs. *TKDE*, 2016.

[61] 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 *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 657–666, New York, NY, USA, 2013. ACM.

[62] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *JASIST'07*, 2007.

[63] Yishi Lin and John CS Lui. Analyzing competitive influence maximization problems with partial information: An approximation algorithmic framework. *Performance Evaluation*, 91:187–204, 2015.

[64] Bo Liu, Gao Cong, Dong Xu, and Yifeng Zeng. Time constrained influence maximization in social networks. In *ICDM*, pages 439–448, 2012.

[65] C. L. Lucchesi and D. H. Younger. A minimax theorem for directed graphs. *J. London Math. Soc*, 1978.

[66] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *J. ACM*, 35(1):18–44, January 1988.

[67] Dominic Meier, Yvonne Anne Oswald, Stefan Schmid, and Roger Wattenhofer. On the windfall of friendship: inoculation strategies on social networks. In *EC '08*, pages 294–301, New York, NY, USA, 2008. ACM.

[68] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[69] Seth A Myers and Jure Leskovec. Clash of the contagions: Cooperation and competition in information diffusion. In *ICDM'12*, 2012.

[70] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical programming*, 14(1):265–294, 1978.

[71] Huy Nguyen and Rong Zheng. *Influence Spread in Large-Scale Social Networks – A Belief Propagation Approach*, pages 515–530. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[72] Nam P. Nguyen, Guanhua Yan, My T. Thai, and Stephan Eidenbenz. Containment of misinformation spread in online social networks. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 213–222, New York, NY, USA, 2012. ACM.

[73] Zeev Nutov and Michal Penn. on the integral dicycle packings and covers and the linear ordering polytope. *Discrete Applied Mathematics*, 1995.

[74] Maya Oppenheim. Youtube shooting: Twitter and facebook explodes with misinformation and hoaxes, 2018. `https://www.independent.co.uk/news/world/americas/youtube-shooting-fake-news-twitter-facebook-identity-illegal-immigrant-hoax-misinformation-a8287946.html`, Last accessed on 2018-04-18.

[75] T.D. Parsons. Pursuit-evasion in a graph. *Theory and Applications of Graphs*, pages 426–441, 1976.

[76] Nishith Pathak, Arindam Banerjee, and Jaideep Srivastava. A generalized linear threshold model for multiple cascades. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 965–970. IEEE, 2010.

[77] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[78] Kévin Perrot and Van Trung Pham. Np-hardness of minimum feedback arc set problem on eulerian digraphs and minimum recurrent configuration problem of chip-firing game. *CoRR*, 2013.

[79] Vijaya Ramachandran. Finding a minimum feedback arc set in reducible flow graphs. *J. Algorithms*, 1988.

[80] Youssef Saab. A fast and effective algorithm for the feedback arc set problem. *Journal of Heuristics*, 2001.

[81] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[82] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 2008.

[83] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *TKDE*, 2016.

[84] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[85] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, Feb 1981.

[86] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 75–86, New York, NY, USA, 2014. ACM.

[87] Chi Wang, Wei Chen, and Yajun Wang. Scalable influence maximization for independent cascade model in large-scale social networks. *Data Mining and Knowledge Discovery*, 25(3):545–576, 2012.

[88] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[89] Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 222–227. IEEE, 1977.