Clearing Contamination in Large Networks

Michael Simpson, Venkatesh Srinivasan, and Alex Thomo.

Abstract—In this work, we study 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. We show that this problem is NP-hard even on directed acyclic graphs and 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.

Index Terms—Social Networks, Graph Searching, Approximation Algorithms.

1 Introduction

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. In particular, we are interested in studying social networks as they allow for the widespread distribution of knowledge and information in modern society. They are rapidly becoming a place where people go to hear the news and discuss personal and social topics. In turn, the information posted can spread quickly through the network eventually reaching a large audience, especially so for influential users. However, information spread in a social network can have either positive or negative effects. For example, posting about natural disasters or warfare can either help or hinder other users depending on whether the information is accurate or not. In other cases, the information can be strictly detrimental, such as negative rumours about private corporations or people that can even affect the financial markets. Thus, since many people today learn of news or events online it is important to have tools to eliminate, not just minimize, the effects of disinformation. Previous work has focused on the task of limiting the spread of misinformation [1], [2], [3], [4] while we study the stronger model of eliminating disinformation, or any kind of contamination, from a general network.

For a contaminated network, we model the problem in the context of graph searching; a classical game on graphs [5], [6], [7]. 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 [8] 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.

Our main contributions can be summarized as follows.

1) We are the first to investigate the search time of directed graphs.

• *M. Simpson, V. Srinivasan and A. Thomo are with the Department of Computer Science, University of Victoria, Victoria, BC. E-mail: simpsonm@uvic.ca*

2) We prove the search time problem is NP-complete on DAGs.
3) We devise an approximation algorithm for clearing DAGs that also outperforms split and conquer strategies on a broad class of DAGs.
4) We introduce a novel DAG decomposition theorem which we believe is of independent interest.
5) We provide an experimental study of clearing large social networks.

We start with an overview of information propagation in social networks and the graph searching problem in Section 2. In Section 3 we introduce the necessary concepts and definitions from graph searching. Section 4 presents the lower bound for search time on directed graphs. In Section 5 we prove the NP-hardness of the search time problem on DAGs. We introduce our strategy for clearing general networks and the Plank algorithm in Section 6. Section 7 contains our approximation bounds, comparison to the split and conquer strategy, along with our DAG decomposition theorem. Finally, in Section 8 we provide our experimental results.

2 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 [9], [10], [11]. More recently, the problem of limiting the spread of rumours or misinformation in a social network has been studied by [1], [3], [4]. In [3], [4] the problem is posed in terms of competing campaigns while [1] 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. [6], [7], [12], [13]. 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. [12], [13], [14]. It was shown by Megiddo et al. [15] 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 [16], [17], [18] 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 [8]. They 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. They study the length of a search strategy which corresponds to the time complexity of searching a graph. They 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 [19] in which they show that for any fixed $k \geq 4$ computing the minimum length path decomposition is NP-hard and give a polynomial time algorithm for $k < 4$.

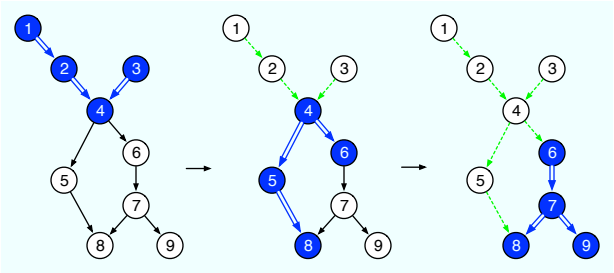


Fig. 1: An example search strategy

3 Preliminaries

We consider the graph searching game on simple, weakly connected, directed graphs $G = (V, E)$ with n nodes, a set of vertices V and a set of edges 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.

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 *re-contaminated* 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 1. In the first step, searchers are 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 [8].

Definition 1. *A search strategy σ 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 σ 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 σ with $length(\sigma) \leq t$ and let search-time$_G(s)$ be the least $length(\sigma)$ for all search strategies σ 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, σ is *space-optimal* if $width(\sigma) =$ *search-width$_G(t)$* with $length(\sigma) = t$. For a given number of searchers s, σ is *time-optimal* if $length(\sigma) =$ *search-length$_G(s)$* with $width(\sigma) = s$.

4 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 $\lceil \frac{n-s}{s-1} \rceil + 1$ shown by Brandenburg and Herrmann [8] 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 σ 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 σ. 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.

$$\forall i \; \exists j \text{ such that } S_i \cap S_j \neq \emptyset \tag{1}$$

Equation 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, Ω, 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 Ω 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.

Now, we present the search time lower bound on directed graphs which utilizes the set system notion.

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 $\lceil \frac{n-s}{s-1} \rceil + 1$ steps to clear G.*

Proof. Assume we are given an arbitrary search strategy σ for G. First, we construct the corresponding set system S for σ. 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, notice that equation (1) and our assumption of connectedness implies that G_S is connected.

Now, we present a special ordering Ω' for S by performing a depth-first search of G_S initialized on any node of G_S. The order in which meta-nodes are visited in the DFS makes up Ω'. This ordering may differ from that of σ and is created purely for the proof of bounding the number of sets, t.

Then, we can bound the progress ρ made by this ordering as follows. The first set visited in Ω' has a progress bounded above by s from condition (1) and the fact that there are no previous sets in Ω'. Then, every subsequent set S_i in Ω' has a progress bounded above by $s - 1$ since, by the DFS style ordering, there will be a set located earlier in Ω' which was connected to S_i, indicating a non-empty intersection. Thus, if there are t sets, the total progress is bounded above by $s + (t-1)(s-1)$. Furthermore, ρ 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.

Thus, we have

$$n \leq \rho \leq s + (t-1)(s-1)$$
$$n - s \leq (t-1)(s-1)$$
$$\frac{n-s}{s-1} \leq t-1$$

Finally, since t must be an integer we have

$$t \geq \left\lceil \frac{n-s}{s-1} + 1 \right\rceil = \left\lceil \frac{n-s}{s-1} \right\rceil + 1$$

Therefore, we have shown that for an arbitrary search strategy, the corresponding set system requires at least $\left\lceil \frac{n-s}{s-1} \right\rceil + 1$ sets and thus any search strategy for G must take at least this number of steps. □

In the next section we prove the hardness of computing the search time of a DAG.

5 HARDNESS FOR DAGS

In this section we prove that computing the search time of a DAG for a given number of searchers is NP-complete. Consider the GRAPH SEARCHING problem as determining the minimum number of steps required to clear an input directed graph G on n nodes with s searchers. The decision version asks if G can be cleared in t steps. To do this we introduce two concepts required for the hardness proof: *B-sections* and the *loss* function. First, consider an *arborescence*, defined as a rooted directed tree with root node v attached to m directed paths $b_1, b_2, \ldots, b_m$ all beginning at v where $|b_i| \geq 1$. We refer to such structures as *B-sections* and a sample B-section can be seen in Figure 2. B-sections will be used in Section 7.3 for our decomposition theorem. Next, we define our *loss* function. We know that a strategy achieving the lower bound with s searchers visits s new nodes in the first step and $s-1$ new nodes in each subsequent step. All strategies can visit s new nodes in the first step. Thus, a searcher placement deviating from the lower bound is one in which $s-1$ new nodes are not visited in a given step. Note, this excludes the final step where there may not be enough nodes left to visit $s-1$ new nodes. For this reason, an alternative definition is a placement in which two or more searchers are left stationary.

Definition 4. *For a search strategy σ, the loss at step $i < length(\sigma)$ is given by*

$$loss_i(\sigma) = \begin{cases} 0 & \text{if } |V_i \cap (\cup_{j<i} V_j)| \leq 1 \\ |V_i \cap (\cup_{j<i} V_j)| - 1 & \text{otherwise} \end{cases}$$

Then, the loss of σ, $loss(\sigma) = \sum_i loss_i(\sigma)$.

Now, consider the B-section in Figure 2 and the search strategy shown which uses 3 searchers. The blue nodes represent the searcher placements of the current step while green nodes represent already visited nodes. The blue edges indicate edges which are being cleared in the current step while green edges are edges which have been cleared in a previous step. Notice how the search strategy partially clears each of the branches before finishing them off in a single step. The ability to avoid loss when moving to a new branch lay in the strategy's ability to "set up" the number of nodes left in each branch, after partially clearing the branches, as a multiple of $s-1$. This ensured that the clearance of each branch ended exactly at the leaf nodes and did not spill over into the next branch.

Now, we can generalize this idea to capture how a strategy would have to behave to "set up" the branches of a general B-section in a similar fashion in order to achieve zero loss. Consider a B-section with m branches $b_1, \ldots, b_m$ each of length $d_1, \ldots, d_m$ where d_i counts all nodes in b_i other than the branching node. The question of whether or not zero loss can be achieved comes down to whether we can end the clearance of each branch exactly at the branch's final node. Therefore, we are asking whether we can move across the top of the B-section with the s searchers such that after this initial sweep the number of nodes remaining to be cleared in each b_i is a multiple of $s-1$. If this is the case, the branches could then be cleared one at a time with all s searchers with the clearance ending exactly at the last node of each b_i ensuring no loss when moving between branches.

The problem of the initial sweep across the top of the B-section can be phrased as an instance of a BIN PACKING variant. First, notice that there is a single value $0 \leq x_i \leq s-2$ for each branch that makes the number of nodes remaining in b_i a multiple of $s-1$. Thus, we wish to know if we can pack the x_i into bins of size $s-1$ such that each bin is exactly full. The solution to this problem tells us if the B-section can be cleared with zero loss. However, we know that not all B-sections can be cleared with zero loss and we actually want to know the minimum loss achievable. This leads to a variant of the optimization version of BIN PACKING which we wish to solve. In the standard BIN PACKING problem we wish to minimize the number of bins used. Our problem is asking to minimize the number of partially full bins. That is, we want to maximize the number of exactly full bins as each partially full bin represents a loss due to the strategy being forced to revisit a node or carry unused searchers. We refer to this as the EXACT BIN PACKING problem and in the decision version denote the number of allowable partially filled bins by the parameter p. First, we show that the EXACT BIN PACKING problem remains strongly NP-hard even when p is fixed to 0.

Lemma 1. *The EXACT BIN PACKING problem with $p = 0$ is strongly NP-complete.*

Proof. Consider an instance of the decision version of BIN PACKING with items $X = \{x_1, \ldots, x_n\}$, bin size V, and b available bins. Then, let $r = Vb - \sum x_i$. Here, r is the total remaining bin space (regardless of packing) for the BIN PACKING instance.

Now, we construct an instance of the decision version of the EXACT BIN PACKING problem with items $X' = X \cup 1_r$ where 1_n is a set containing n 1's, bin size V, and $p = 0$. Then, the EXACT BIN PACKING instance has a solution iff there is a solution to the instance of BIN PACKING.

If the BIN PACKING instance can be packed with b bins then in the EXACT BIN PACKING instance we have exactly the required number of 1's to fill in the rest of the space leaving all exactly full bins, i.e. $\sum x_i + r = Vb$. However, if the BIN PACKING instance requires more than b bins then there will not be sufficient 1's to fill in the space of additional

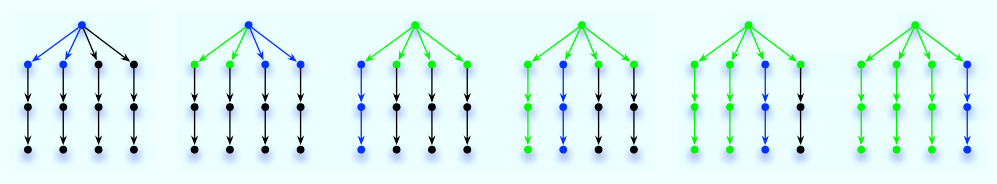


Fig. 2: An example search strategy with zero loss

bins and therefore there will be at least one partially filled bin, i.e. $\sum x_i + r = Vb < V(b+k)$ for some $k > 0$. □

Then, it follows from the above result that the general version of EXACT BIN PACKING with arbitrary p is also strongly NP-hard.

Corollary 1. *The EXACT BIN PACKING problem is strongly NP-complete.*

Now, we formally show the hardness of the graph searching problem on B-sections using the above result. First, we show that the GRAPH SEARCHING problem remains hard even for fixed $t = \lceil \frac{n-s}{s-1} \rceil + 1$ and restricted G.

Before we present the proof we will introduce some facts about the GRAPH SEARCHING problem. First, the search time of a strategy can be computed from the loss as $t = \lceil \frac{n-s+loss}{s-1} \rceil + 1$. Also, recall that the lower bound for clearing a graph is $t_{min} = \lceil \frac{n-s}{s-1} \rceil + 1$. Then, notice that t_{min} can be achieved with a range of losses which depends on the values of n and s. Namely, t_{min} will be achieved by any strategy with $0 \leq loss \leq \left[\lceil \frac{n-s}{s-1} \rceil - \frac{n-s}{s-1} \right](s-1)$. We refer to the upper bound by $loss_{max}$.

Lemma 2. *The GRAPH SEARCHING problem on B-sections with $t = t_{min}$ is NP-complete.*

Proof. Consider an instance of the decision version of EXACT BIN PACKING with items $X = \{x_1, \ldots, x_m\}$, bin size V, and $p = 1$. We construct an instance of GRAPH SEARCHING by transforming the x_i into paths ρ_i of length x_i and attaching each ρ_i to a distinguished branching node β. Additionally, we attach a path b of length $\lceil \frac{\sum_i x_i - V}{V} \rceil V - (\sum_i x_i - V)$ to β. Call the resulting graph G and notice that G is a B-section. Let the GRAPH SEARCHING instance have $s = V + 1$ and $t = \lceil \frac{|b| + \sum x_i - V}{V} \rceil + 1$. Then, the GRAPH SEARCHING instance has a solution iff there is a solution to the instance of EXACT BIN PACKING.

Notice that we have chosen the length of b such that $loss_{max} = 0$, therefore $|b| + \sum x_i - V$ is a multiple of V. Also, the chosen $t = t_{min}$. Thus, if the EXACT BIN PACKING problem has a solution then G can be cleared by first placing a searcher at β and on every ρ_i of a bin in a sweep across the top of G. The clearance of b is included in the final step of the strategy. This search strategy has $loss = 0$ and will be able to clear the graph in t_{min} steps.

In the other direction, given that G can be cleared in $t = t_{min}$ steps we show how to obtain a solution to the EXACT BIN PACKING instance by progressively restricting how such a strategy must behave. Again, the structure of G is such that $loss_{max} = 0$ so the strategy clearing G cannot incur any loss. Thus, we can immediately rule out strategies which split into multiple groups; that is, any strategy in which the subgraph induced from searcher placements in a step does not form a connected component (ignoring the directions on edges) as such a step incurs a minimum loss of one. Then, to clear every branch we must leave a guard on β as it is required to clear the first edge in each branch. Thus, since we cannot incur any loss, no node other than β can be revisited in any step else the strategy would not visit $s - 1$ new nodes. Therefore, we cannot partially clear any branch. Then, since each ρ_i has length less than V, the strategy will fully clear some number of branches in every step of the strategy. Now, observe that we have restricted the allowable strategies such that they can only differ from the one described above by a re-ordering of steps. Thus, the ρ_i cleared in each step are placed in a bin and the resulting bins make up the packing. Note, if b had non-zero length we do not include it in the packing and thus get at most one partially full bin. □

From this we get our main result.

Theorem 2. *The GRAPH SEARCHING problem on B-sections is NP-complete.*

Furthermore, hardness on B-sections implies hardness on all its superclasses in the directed setting which includes directed trees, DAGs and all their directed superclasses. Therefore, we see an interesting comparison to computing the search number on undirected graphs where the problem becomes efficiently solvable when we move from general graphs to trees. However, computing the search time does not become efficiently solvable even when restricting the input graph to a B-section. In the following section, despite the hardness of the search time problem, we will introduce an efficient approximation algorithm for searching general digraphs.

6 Our Search Algorithm

6.1 Searching Digraphs

Since the graph searching problem is NP-hard even on B-sections, 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.

6.2 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, Ψ, and in the second step it compiles a search strategy from Ψ. 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 Ψ
2) Convert Ψ 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 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, Ψ, into a search strategy for G using s searchers (Algorithm 3). In summary, Ψ is traversed adding nodes to the current step in the search strategy until a step has reached s placements. After Ψ has been traversed we will have all the steps which make up the Plank search strategy σ. This procedure is captured in the pseudocode of Algorithm 3 where V_c represents the nodes present in the current step.

Algorithm 2 mDFS

Input: Input graph G and the current node v
Output: An edge ordering Ψ
  $\Psi \leftarrow []$
  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**
        $\Psi.append(e)$
        Label e as explored
        $w \leftarrow G.adjacentVertex(v, e)$
        $\Psi.append(mDFS(G, w))$
      end if
    end for
  end if
  return Ψ

Algorithm 3 Construct Strategy

Input: Sequence Ψ and the number of searchers s
Output: a search strategy $\sigma = (V_1, \ldots, V_t)$
  $\sigma, V_c, \leftarrow \emptyset$
  for all edges e in Ψ **do**
    if $nodes(e)$ not in V_c **then**
      $V_c \leftarrow V_c \cup nodes(e)$
    end if
    if current step contains s placements **then**
      $\sigma.append(V_c)$
      $V_c = \emptyset$
    end if
  end for
  return $\sigma = (V_1, \ldots, V_t)$

Furthermore, we introduce an optimized version of the strategy construction algorithm 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 Ψ, 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.

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

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

Proof. First, note that since Algorithm 2 is a modified DFS it

Algorithm 4 Optimized Construct Strategy

Input: Sequence Ψ 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
        end if
        if v not in V_c then
            V_c ← V_c ∪ v
            steps[v] ← steps[v] ∪ i
        end if
    end if
    if current step contains s placements then
        σ.append(V_c)
        V_c = ∅
        i ← i + 1
    end if
end for
return σ = (V_1, ..., V_t)
```

is clear that every edge $e \in E$ is added to the edge ordering Ψ. Second, Algorithm 3 places searchers on both endpoints of every edge in Ψ. Thus, the resulting strategy σ 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 σ.

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

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

7 Analysis

7.1 Approximation Bounds

In this section we will show that the Plank strategy is a $(3 + f_O)$-approximation algorithm for searching DAGs, where f_O is an instance determined parameter, and motivate its performance on typical DAGs.

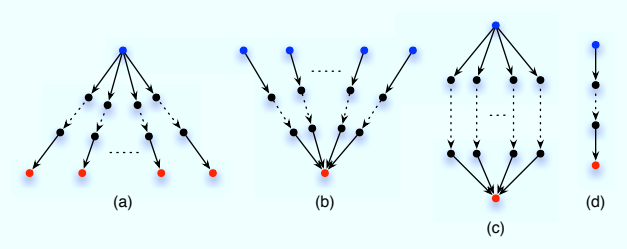


Fig. 3: A sample (a) B-section (b) R-section (c) D-section and (d) P-section

First we introduce some definitions to be used in the following proofs. We define four types of DAGs referred to as *sections*. We have already seen the definition of B-sections in Section 4 (Fig. 3(a)). Second are sections that resemble B-sections, except that the direction of each edge is reversed. That is, the structure is the same as a B-section, but with all branches directed towards a distinguished root which we refer to as *R-sections* (Fig. 3(b)). Next, we have sections which look like diamonds, or *D-sections* (Fig. 3(c)). These sections have a start node, two or more node disjoint branches, and an end node with branches originating at the start node and ending at the end node. Finally, we have simple directed paths, or *P-sections* (Fig. 3(d)). Note, the blue and red nodes mark the top and bottom nodes of a section respectively.

We prove in section 7.3 that any DAG can be decomposed into sections of the above four types and assume this holds for the remainder of the analysis.

To begin, we first prove an approximation bound for zero-overlap DAGs and then modify the bound to include the full range of DAGs. We define the *overlap* of a node v by

$$overlap(v) = \begin{cases} r & \text{if } v \text{ is a top/bottom node in} \geq 3 \text{ sections} \\ 0 & \text{else} \end{cases}$$

Where r is the total number of sections for which v is a top or bottom node.

Then, the overlap of a DAG G, denoted Ω, is defined as $\Omega = \sum_{u \in V} overlap(u)$. A DAG is said to be a *zero-overlap* DAG if $\Omega = 0$ and indicates a DAG in which each section overlaps with at most one other section. The following analysis assumes a zero-overlap DAG.

We bound the number of steps required by the Plank strategy by bounding the loss measure we introduced in Section 4. To that end, we consider the loss the Plank strategy can achieve when taking an arbitrary step in its clearance. We have four cases for how the strategy moves between steps: (1) the strategy remains within a single section, (2) the strategy moves from an unfinished section to another unfinished section, (3) the strategy moves from an unfinished or finished section to a previously unvisited section or (4) the strategy finishes clearing a section and returns to a partially cleared section. We investigate these cases in four claims below.

Claim 1. *A step taken by the Plank strategy described by Case 1 can incur a loss of no more than 2.*

Proof. Recall that the Plank strategy will move across branches of a section one at a time. Thus, when moving between branches in a B/R-section the strategy will revisit the top/bottom node of the section. Thus, the strategy will incur a loss if the previous branch was not cleared in a single step. On the other hand, when moving between branches of a D-section the Plank strategy will revisit both the top and bottom nodes incurring a loss of two if the previous branch was only partially cleared. Finally, a P-section trivially cannot incur a loss. □

Claim 2. *A step taken by the Plank strategy described by Case 2 can incur a loss of no more than 3.*

Proof. When moving to another unfinished section θ_2, we may revisit nodes that connect the sections as well as any nodes already visited in θ_2. Thus, we have a worst case loss of 3 in the situation where both θ_1 and θ_2 are D-sections with overlapping top nodes. Here, we revisit the bottom node of θ_1, the top node of each section and the bottom node of θ_2. In contrast, moving to a downstream section can only incur a worst case loss of 1, when the bottom node is revisited, as every other node is visited for the first time. □

Claim 3. *A step taken by the Plank strategy described by Case 3 can incur a loss of no more than 2.*

Proof. When moving from a section θ_1 to a new section θ_2, besides the nodes connecting sections, every node is being visited for the first time. Thus, we again have a worst case loss of 2, in the situation where θ_1 and θ_2 have overlapping top nodes. In contrast, moving to a downstream section can only incur a worst case loss of 1, when the bottom node is revisited, as every other node is visited for the first time. □

Claim 4. *A step taken by the Plank strategy described by Case 4 can incur a loss of no more than $\lceil \frac{s}{2} - 1 \rceil$.*

Proof. When returning to a B-section θ_B, we incur a loss of 1 by returning to the branching node. Then, consider the case where θ_B is entirely cleared with the available searchers and the strategy must again move to a new section or return to another B-section. Here, moving to a new section would incur no extra loss as the new section would be downstream from θ_B. However, the strategy could continue clearing B-sections and returning up to more partially cleared B-sections incurring a loss each time this occurs. The number of times the strategy could return to a B-section is bounded by the number of searchers available, s, and the minimum size of the portion of the B-section left to be cleared, 2. Thus, the Plank strategy could take a single step which incurs a loss of $\lceil \frac{s}{2} - 1 \rceil$ as the sections which are revisited must have at least one node other than the branching node not yet visited. An analogous situation occurs when returning to an R-section. Note, a strategy will not return to an upstream section while clearing a D-section and thus do not come up in Case 3 steps. □

Now, we can divide an arbitrary Plank strategy into steps adhering to Case 1, 2, 3, or 4. Thus, w.l.g. we can investigate the approximation ratios for steps of each type to arrive at an overall approximation ratio. We group Case 1, 2, and 3 steps together as *Type 1* steps while Case 4 steps are referred to as *Type 2* steps.

Lemma 4. *The Type 1 steps have an approximation ratio of no more than* 3.

Proof. Given s searchers, consider k steps incurring a loss of 3. Then, the total number of nodes from Type 1 steps, n, is at least $(k-1)(s-4)+s+2$ for $s \geq 6$. We only consider $s \geq 6$ because 5 or less searchers cannot enter into a pattern which incurs a loss of 3 for successive steps. In the case of $s = 5$ we can only enter into a pattern having loss 2 between steps and therefore have $n \geq (k-1)(s-3)+s+2$. Furthermore, in the case of $s = 4$ we can only incur a loss of 2 for a single step in a specific D-section on 5 nodes where it is easy to verify by hand that the approximation ratio remains less than 3. For the $s \geq 6$ expression, we visit $s-4$ new nodes in each step except the last step where we may run out of nodes left to visit in which case 2 additional nodes is a minimum and similarly for $s = 5$ where we visit $s-3$ nodes in each step. Notice, the additional s comes from the fact that all strategies visit s nodes in the first step and incur no loss.

Then, the loss is bounded by $3k$ or $\frac{3(n-6)}{s-4}$ for $s \geq 6$ and $2k$ or $\frac{2(n-5)}{s-3}$ for $s = 5$. Now, we can compute an approximation ratio by comparing the lower bound $\lceil \frac{n-s}{s-1} \rceil + 1$ to the expression $\lceil \frac{n-s+loss}{s-1} \rceil + 1$. First, in the case for $s = 5$ we have,

$$\left\lceil \frac{n - s + \frac{2(n-5)}{s-3}}{s-1} \right\rceil + 1 \leq \frac{ns - n + s^2 - 5s - 4}{(s-3)(s-1)} \tag{2}$$

Then

$$\frac{\lceil \frac{n-s+\frac{2(n-5)}{s-3}}{s-1} \rceil + 1}{\lceil \frac{n-s}{s-1} \rceil + 1} \leq \frac{ns - n + s^2 - 5s - 4}{(n-1)(s-3)} \tag{3}$$

Where (3) is bounded above by 2 for $s \leq \frac{n+3}{2}$. Note, we only consider the case where $s \leq \frac{n+3}{2}$ since when $s > \frac{n+3}{2}$ all n nodes will be cleared in 2 steps as no more than 3 nodes will remain stationary between steps.

Second, in the case for $s \geq 6$ we have,

$$\left\lceil \frac{n - s + \frac{3(n-6)}{s-4}}{s-1} \right\rceil + 1 \leq \frac{ns - n + 2s^2 - 10s - 10}{(s-4)(s-1)} \tag{4}$$

Then

$$\frac{\lceil \frac{n-s+\frac{3(n-6)}{s-4}}{s-1} \rceil + 1}{\lceil \frac{n-s}{s-1} \rceil + 1} \leq \frac{ns - n + 2s^2 - 10s - 10}{(n-1)(s-4)} \tag{5}$$

Where (5) is bounded above by 3 for $6 \leq s \leq \frac{n+4}{2}$. Note, we only consider the case where $s \leq \frac{n+4}{2}$ since when $s > \frac{n+4}{2}$ all n nodes will be cleared in 2 steps as no more than 4 nodes will remain stationary between steps.

Therefore, an arbitrary number of Type 1 steps has an approximation ratio of no more than 3. □

Lemma 5. *The Type 2 steps have an approximation ratio of no more than* 2.

Proof. Given s searchers, consider k steps incurring a loss of $\lceil \frac{s}{2} - 1 \rceil$. Then, notice that the upper bound on the number of steps required by a strategy on zero-overlap DAGs is $n -$

$s+1$ as we visit at least one new node in each step. Thus, $k \leq n-s+1$ giving a loss bounded above by $k\lceil \frac{s}{2}-1 \rceil \leq \frac{ns-s^2+s}{2}$ since $\lceil \frac{s}{2}-1 \rceil \leq \frac{s}{2}$. Now, notice that $\frac{ns-s^2+s}{2} \leq \frac{n}{2}$ for $s \leq n$ which holds for all search strategies. Therefore, we can compute the approximation ratio as,

$$\left\lceil \frac{n-s+\frac{n}{2}}{s-1} \right\rceil + 1 \leq \frac{n-s+\frac{n}{2}}{s-1} + 2 = \frac{3n+2s-4}{2(s-1)} \tag{6}$$

Then

$$\frac{\left\lceil \frac{n-s+\frac{n}{2}}{s-1} \right\rceil + 1}{\left\lceil \frac{n-s}{s-1} \right\rceil + 1} \leq \frac{\frac{3n+2s-4}{2(s-1)}}{\frac{n-1}{s-1}} = \frac{3n+2s-4}{2(n-1)} \tag{7}$$

Where (7) is bounded above by 2 for $s \leq \frac{n-1}{2}$. Note, we only consider the case where $s \leq \frac{n-1}{2}$ since when $s > \frac{n-1}{2}$ the number of nodes remaining after the first step is less than $\frac{n}{2}$ and therefore the loss cannot exceed this value.

Therefore, an arbitrary number of Type 2 steps has an approximation ratio of no more than 2. □

Thus, we get the following approximation bounds for the Plank strategy on zero-overlap DAGs.

Lemma 6. *The Plank algorithm is a 3-approximation algorithm for computing the search time of a zero-overlap DAG.*

Proof. We consider an arbitrary instance of a Plank strategy. The steps of the strategy are all of Type 1 or 2. Then, the proof follows directly from Lemma's 4 and 5. □

In practice, the number of searchers will often be much less than the size of the DAG, $s \ll n$, in which case (3) $\approx 1 + O(\frac{2}{s}) + O(\frac{s}{n})$, (5) $\approx 1 + O(\frac{3}{s}) + O(\frac{s}{n})$, and (7) $\approx \frac{3}{2} + O(\frac{s}{n})$. Furthermore, the structure of a DAG required to produce an approximation ratio for (7) of $\frac{3}{2} + O(\frac{s}{n})$ is extremely artificial and would not show up in a large fraction of DAGs. In general, we expect the approximation ratio to closely resemble $1 + O(\frac{s}{n})$. Therefore, proving the usefulness of the Plank algorithm for typical zero-overlap DAGs.

Now, we must modify the bound for DAGs with nonzero overlap. The overlap Ω of a DAG can be viewed as a rough estimation of the density of the digraph. As such, DAGs move progressively towards resembling directed complete bipartite graphs (with all edges directed from one partition to the other) as Ω increases. We take a conservative route and add to the bound of 3 for zero-overlap DAGs an *overlap factor*, f_o. The f_o factor upper bounds the number of steps required to clear the number of possible edges incident on the overlapping nodes. It is defined as,

$$f_o = \left(\frac{\Omega}{n-1}\right) \tag{8}$$

and can often be approximated by $\frac{m}{n}$. Thus, combining the possible loss in zero-overlap DAGs and the potential loss in DAGs with overlap yields an approximation ratio that holds for all DAGs of $3 + f_o$.

Theorem 4. *The Plank algorithm is a $(3 + f_o)$-approximation algorithm for computing the search time of a DAG.*

Note, as we provide a lower bound on the length of a search strategy that is independent of the structure of the input DAG, our f_o factor may take on large values for highly overlapping DAGs when the length of the Plank strategy, in reality, may not be far off the instance-optimal solution.

7.2 Comparison to Splitting Strategies

Another natural candidate for graph searching would be a BFS style strategy which we investigate next. We show that the DFS style strategy, our Plank algorithm, outperforms the BFS style strategies on a broad class of DAGs. We refer to BFS style strategies as *splitting strategies* and define them as follows.

Definition 5. *The class of splitting strategies are all search strategies which send at least two searchers down as many branches of a section as possible.*

The way in which a splitting strategy distributes the searchers over the branches is arbitrary, but the key point is that such a strategy tries to split as much as possible, mimicking a BFS. As with the Plank algorithm, splitting strategies do not move passed nodes with unexplored incoming edges to avoid recontamination. Alternatively, we can think of splitting strategies as split and conquer style strategies.

For the sake of brevity, we give a high level description of our results regarding splitting strategies. We begin by proving the Plank strategy outperforms all splitting strategies on each type of section individually. The proof follows from a direct comparison of the loss required by any splitting strategy to the upper bound on the loss possible by the Plank strategy.

Now, the fact that any DAG can be decomposed into sections of our four types, which we prove in the next section, allows us to observe that the loss due to the Plank algorithm will be the same in its clearance of decomposed sections within a DAG as if they were being cleared in isolation conditioned on the length of the section's branches. The condition required is that the B, R, and D-sections contain branches of length s or greater which we refer to as having *large* sections. Thus we are able to show the following result.

Theorem 5. *The Plank strategy outperforms all splitting strategies in clearing DAGs with large B, R, and D-sections with any number of searchers.*

7.3 Decomposing a DAG

We claim that a DAG can be decomposed into sections of our four types. Formally, we define a *valid* section-decomposition as follows.

Definition 6. *Given a DAG G a section-decomposition Δ is valid if and only if it consists of sections $\theta_i = (V_i, E_i)$ of type B, R, D, or P such that $\bigcup_i V_i = V$, $\bigcup_i E_i = E$ and $E_i \cap E_j = \emptyset$ for all i, j. Additionally, sections may only overlap on top and bottom nodes.*

A *merge* operation takes two valid sections and combines them to form a new valid section. Formally, given two sections $\theta_1 = (V_1, E_1)$ and $\theta_2 = (V_2, E_2)$, $merge(\theta_1, \theta_2) =$

$(V_1 \cup V_2, E_1 \cup E_2)$. We outline the possible merge operations in the below table.

Components	Possible Merge Outcome
P	B, R, D, P
B	B
R	R
D	D
P, B	B
P, R	R
P, D	D
B, R	D
P, B, R	D

Next, we define an ordering among valid section-decompositions.

Definition 7. *We say $\Delta_1 < \Delta_2$ if Δ_1, Δ_2 are valid section-decompositions and Δ_1 can be obtained from Δ_2 by some number of merge operations.*

Then, we can define a minimality property for section-decompositions.

Definition 8. *A section-decomposition Δ is minimal if $\neg\exists\ \Delta'$ such that $\Delta' < \Delta$.*

Finally, we show how to compute a minimal section-decomposition for any DAG. Consider the following procedure on a topological ordering Γ of a DAG G. In the first phase we will move through Γ one node at a time. Starting at the current node v we will traverse Γ for each outgoing edge of v until we reach a node with multiple incoming edges or zero or multiple outgoing edges. This sequence of nodes will be appended to a list η. Phase one is presented in the pseudocode of Algorithm 5.

Algorithm 5 Phase one of the minimal decomposition algorithm

Input: the topological ordering Γ
Output: the list η
  $\eta \leftarrow \emptyset$
  for all nodes $v \in \Gamma$ **do**
    for all outgoing edges e of v **do**
      $u \leftarrow e.destination$
      $seq \leftarrow \{v, u\}$
      while u has exactly one outgoing edge e_o **do**
        $u \leftarrow e_o.destination$
        append u to seq
      end while
      append seq to η
    end for
  end for

After this phase, each edge will be in a unique sequence in η. In a second phase, for each sequence λ in our list we will combine λ with other sequences located after λ in η which have not already been designated to a section to create a section of one of the four types. Once η has been traversed each edge of G will be in a unique section.

The way in which we combine sequences is as follows. Consider two sequences λ_1, λ_2 made up of nodes $u_1, \ldots, u_{m_1}$ and $v_1, \ldots, v_{m_2}$ respectively. We proceed through a series of possible scenarios. First, if $u_1 = v_1$ and $u_{m_1} = v_{m_2}$ we combine λ_1 and λ_2 into a D-section. Second, if $u_1 = v_1$ we combine λ_1 and λ_2 into a B-section. Third, if $u_{m_1} = v_{m_2}$ we combine λ_1 and λ_2 into a R-section. Finally, if the previous three scenarios fail to be met, we leave λ_1 as a P-section. Phase two is captured in the pseudocode of Algorithm 6. Note that we refer to the first and last nodes in a sequence λ by λ_s and λ_e respectively.

Algorithm 6 Phase two of the minimal decomposition algorithm

Input: the list η
Output: a collection of sections of type B, R, D, and D
  for all sequences $\lambda \in \eta$ **do**
    collect all unclaimed sequences $\alpha \in \eta$ such that $\lambda_s = \alpha_s$ in a list L_1
    if $L_1 \neq \emptyset$ **then**
      collect all unclaimed sequences $\beta \in L_1$ such that $\lambda_e = \beta_e$ in a list L_2
      if $L_2 \neq \emptyset$ **then**
        create a D-section from L_2 and λ
        mark L_2 and λ as claimed
      else
        create a B-section from L_1 and λ
        mark L_1 and λ as claimed
      end if
      continue
    end if
    collect all unclaimed sequences $\gamma \in \eta$ such that $\lambda_e = \gamma_e$ in a list L_3
    if $L_3 \neq \emptyset$ **then**
      create a R-section from L_3 and λ
      mark L_3 and λ as claimed
      continue
    end if
    create a P-section from λ
    mark λ as claimed
  end for

Theorem 6. *For any DAG G, Algorithm 5 and Algorithm 6 produce a minimal decomposition Δ.*

Proof. Suppose there exists a section-decomposition $\Delta' < \Delta$. Thus, there exists two or more sections in Δ which can be merged. Without loss of generality, suppose there are only two sections θ_1, θ_2 which can be merged. Consider the sequences $\lambda_1, \ldots, \lambda_n$ and $\mu_1, \ldots, \mu_n$ that were combined to make θ_1 and θ_2 respectively. Then, it is easy to see that regardless of which λ_i or μ_i appeared first in η, Algorithm 6 would have created a section with all the λ_i and μ_i in the same iteration. Thus, there cannot be two or more sections which can be merged and therefore there is no $\Delta' < \Delta$. □

8 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.
- Study how the Plank strategy performs as we vary size, number of searchers, and network structure on random DAGs.

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 an FAS, an NP-hard problem, we employ a heuristic introduced in [20] and remove the resulting edge set from G. The approach in [20] 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 in G (more specifically the difference between the out-degree and the in-degree). The bins for each vertex class are implemented as a doubly-linked list. 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 order to ensure the algorithm is efficient the nodes of the doubly-linked lists are moved to adjacent bins by manipulating the node references directly. 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 $O(m)$ time and requires $O(m)$ space.

8.1 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 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 [21]. An edge in this network from user A to B indicates that A voted for B to become an administrator on Wikipedia. The wiki-vote dataset contains 7,115 nodes and 103,689 edges. The FAS computed contained 8% of the network's edges.

Twitter Retweet Network: The higgs-retweet network from [22] 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 network contains 256,491 nodes and 328,132 edges. 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 [23] 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 network contains 265,214 nodes and 420,045 edges. The FAS computed contained 14% of the network's edges.

Signed Epinions: The signed Epinions trust network from [21] contains an edge from user A to B if A trusts B on the Epinions review site. The signed Epinions dataset contains 131,828 nodes and 841,372 edges. The FAS computed contained 24% of the network's edges.

Signed Slashdot: Next, we look at the signed Slashdot dataset from [21]. An edge in this network from user A to B indicates that B is a friend of A's. The signed Slashdot dataset contains 77.350 nodes and 905,468 edges. The FAS computed contained 8% of the network's edges.

wiki-Talk: The wiki-Talk network from [21] contains an edge from user A to B if A has at least once edited a talk page of user B on Wikipedia. The wiki-Talk dataset contains 2,394,385 nodes and 5,021,410 edges. 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 [21] contains an edge from user A to B if B is a friend of A's. The pokec dataset contains 1,632,803 nodes and 30,622,564 edges. The FAS computed contained 33% of the network's edges.

twitter2010: The twitter2010 network from [24], [25] contains an edge from user A to B if B is a follower of A on the social network Twitter. The twitter2010 dataset contains 41,652,230 nodes and 1,468,365,182 edges. The FAS computed contained 22% of the network's edges.

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 1: Dataset statistics

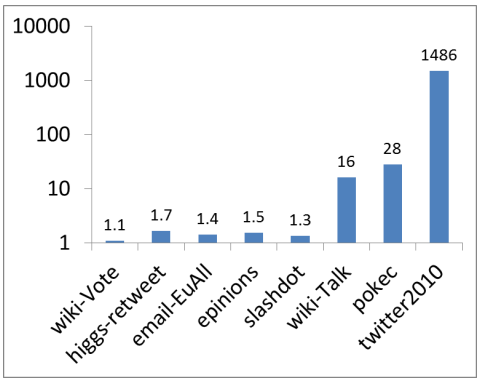
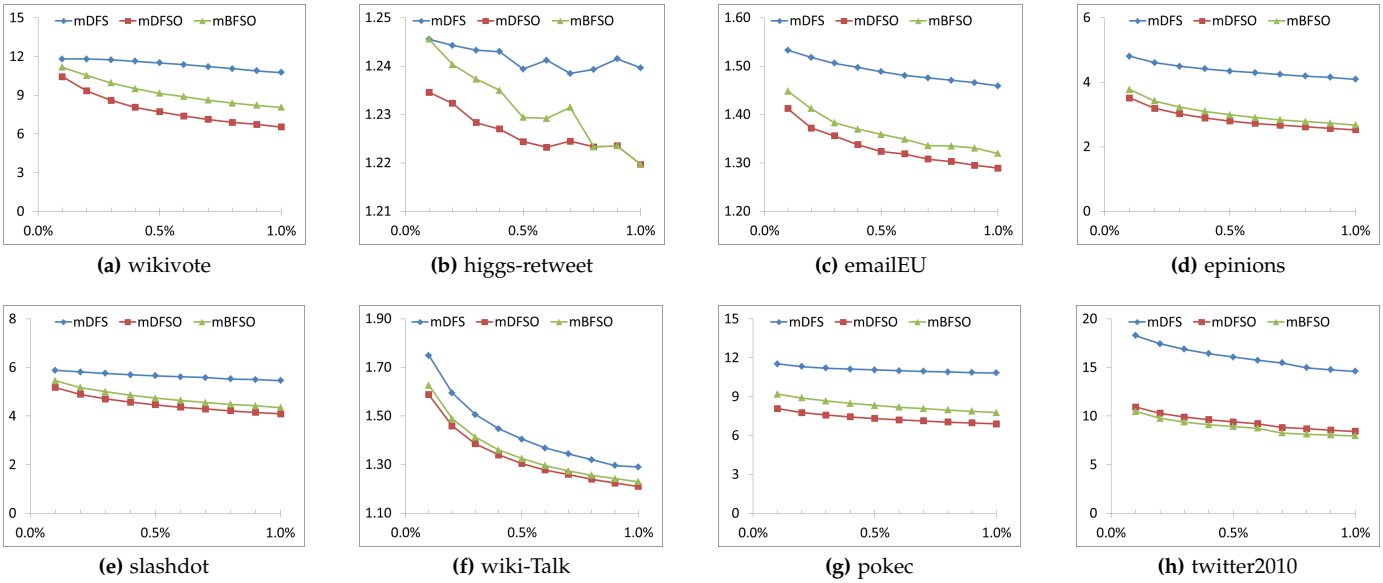


Fig. 4: Time in seconds.

8.2 Discussion

Figure 4 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.



(a) wikivote **(b)** higgs-retweet **(c)** emailEU **(d)** epinions

(e) slashdot **(f)** wiki-Talk **(g)** pokec **(h)** twitter2010

Fig. 5: 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 5 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 from section 7.2 comparing the two types of strategies relies on a key structural aspect of the network - namely having *large* sections. However, we expect the twitter network to be very condensed and thus void of long chains of 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 we described in Section 7.1, the overlap factor, 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 6 to illustrate how our Plank algorithm performs when few searchers are available. We see similar approximation ratios to Figure 5, but note that increasing the number of searchers allows for improvement. We outline the reasoning for this improvement below.

Consider an arbitrary step V_i in a search strategy σ 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 σ. 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.

8.3 Random DAGs

Next, we consider the individual parameters of the system and investigate how the approximation ratio is affected as they are varied. For these tests, we generate random DAGs similar to the Erdös-Rényi model except we predetermine an ordering of the nodes, $(1 \dots n)$, in the DAG and then randomly add edges from node i to j with probability p provided i comes before j in the ordering. We generate five random DAGs for each data point and average the results.

First, we study how the approximation ratio behaves as the size of the network is increased. We fix $p = \frac{1}{n}$ and run the tests for $s = 10, 25, 50$. Figure 7 shows the resulting approximation ratios as the network size increases from

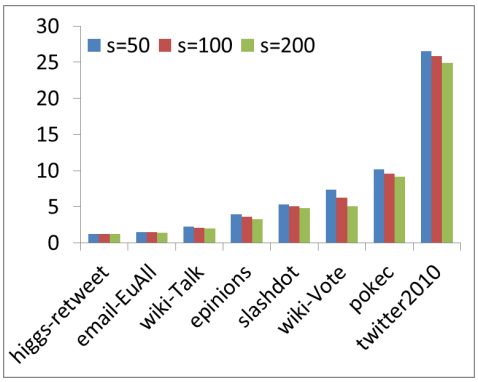


Fig. 6: Approximation ratio when $s = 50$, $s = 100$, and $s = 200$.

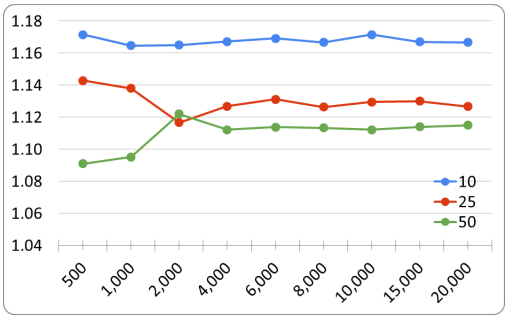


Fig. 7: Effect of increasing network size on the approximation ratio for $s = 10$, $s = 25$, and $s = 50$. The horizontal axes represent n ranging from 500–20,000. The vertical axes represent approximation ratios.

1,000 to 20,000 nodes. We observe the ratios remain nearly constant as the network size is increased.

Next, we look at how the approximation ratio behaves as the number of searchers is increased. We fix $p = \frac{1}{n}$ and run the tests for $n = 5{,}000$, $n = 10{,}000$, and $n = 20{,}000$. Figure 8 shows the resulting approximation ratios as the number of searchers increases from 0.2% to 2% of the network size. We see that the approximation ratio decreases as the number of searchers increases.

Futhermore, we produce random DAGs according to the Barabási-Albert model [26] to replicate the power law

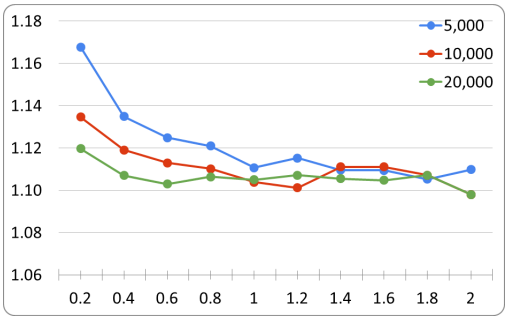


Fig. 8: Effect of increasing searchers on the approximation ratio for $n = 5,000$, $n = 10,000$, and $n = 20,000$. The horizontal axes represent s ranging from 0.2-2% of the size of the network increasing in 0.2 increments. The vertical axes represent approximation ratios.

structure exhibited in many online social networks. The Barabási-Albert model takes three parameters n, m, and m_0. The graph begins with m_0 isolated nodes. New nodes are added to the graph one at a time until we have a graph with n nodes. Each new node is connected to $m \leq m_0$ existing nodes with a probability that is proportional to the number of edges that the existing nodes already have. We direct new edges from existing nodes to new nodes to maintain a DAG structure. We run the Plank algorithm on each DAG with $n = 20{,}000$ and s ranging from $0.5 - 3\%$ of the size of the network increasing in 0.25% increments.

In Figure 9 we have $m = m_0 = 2$ and see a steady decrease in approximation ratio. Then, in Figure 10 we investigate the effects of adding an additional preferential node where we observe a similar decrease in approximation ratio. Finally, in Figure 11 we look at a Barabási-Albert DAG in which there are 6 preferential nodes and 3 links are added with each new node in which a decreasing approximation ratio is also observed as the number of searchers (modestly) increases. In each case we observe good approximation ratios.

9 Conclusion

In this work we perform an extensive study of the problem of eliminating contamination spreading through a network. Specifically, we study the related graph searching problem which we prove is NP-hard even on DAGs and therefore an exact algorithm is infeasible for large networks. Consequently, we introduce a novel approximation algorithm for clearing DAGs which we incorporate into a procedure for clearing general digraphs. We experimentally test our algorithm on several large online networks and observe good performance in relation to the lower bound. Furthermore, we explore various parameters of the graph searching problem on random DAGs and discover the search time is unaffected by network size, yet significantly decreases with modest increases in searcher allocation.

References

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

[2] D. Meier, Y. A. Oswald, S. Schmid, and R. Wattenhofer, "On the windfall of friendship: inoculation strategies on social networks," in *Proceedings of the 9th ACM Conference on Electronic Commerce*. New York, NY, USA: ACM, 2008, pp. 294–301. [Online]. Available: http://doi.acm.org/10.1145/1386790.1386836

[3] S. Bharathi, D. Kempe, and M. Salek, "Competitive influence maximization in social networks," in *WINE'07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 306–311. [Online]. Available: http://dl.acm.org/citation.cfm?id=1781894.1781932

[4] T. Carnes, C. Nagarajan, S. M. Wild, and A. van Zuylen, "Maximizing influence in a competitive social network: a follower's perspective," in *ICEC '07*. New York, NY, USA: ACM, 2007, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/1282100.1282167

[5] T. Parsons, "Pursuit-evasion in a graph," *Theory and Applications of Graphs*, pp. 426–441, 1976.

[6] R. Borie, C. Tovey, and S. Koenig, "Algorithms and complexity results for graph-based pursuit evasion," *Auton. Robots*, vol. 31, no. 4, pp. 317–332, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1007/s10514-011-9255-y

[7] N. D. Dendris, L. M. Kirousis, and D. M. Thilikos, "Fugitive-search games on graphs and related parameters," in *WG '94*. London, UK, UK: Springer-Verlag, 1995, pp. 331–342. [Online]. Available: http://dl.acm.org/citation.cfm?id=647675.731680

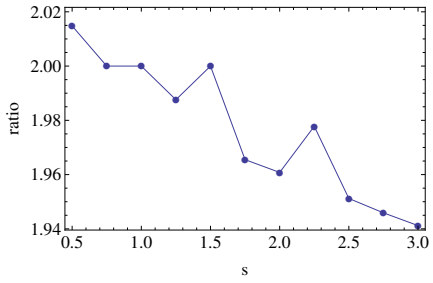


Fig. 9: $m = m_0 = 2$.

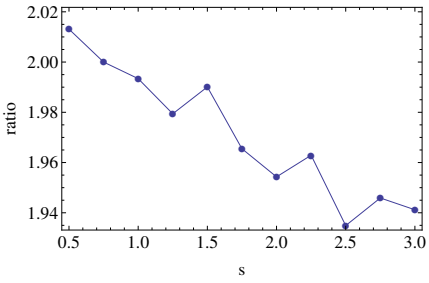


Fig. 10: $m = 2$, $m_0 = 3$.

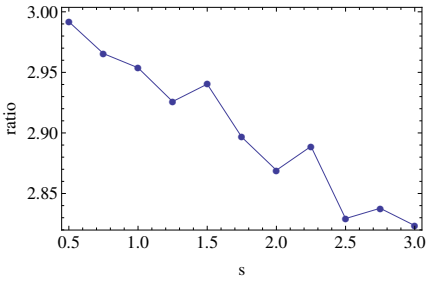


Fig. 11: $m = 3$, $m_0 = 6$.

[8] F. J. Brandenburg and S. Herrmann, "Graph searching and search time," in *SOFSEM 2006: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, 2006, vol. 3831, pp. 197–206.

[9] W. Chen, L. V. S. Lakshmanan, and C. Castillo, *Information and Influence Propagation in Social Networks*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[10] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," in *ICDM'10*, 2010. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2010.118

[11] B. Liu, G. Cong, D. Xu, and Y. Zeng, "Time constrained influence maximization in social networks," in *ICDM*, 2012, pp. 439–448.

[12] M. Kirousis and C. H. Papadimitriou, "Searching and pebbling," *Theor. Comput. Sci.*, vol. 47, no. 2, pp. 205–218, Nov. 1986. [Online]. Available: http://dl.acm.org/citation.cfm?id=21559.21567

[13] J. A. Ellis, I. H. Sudborough, and J. S. Turner, "The vertex separation and search number of a graph," *Inf. Comput.*, vol. 113, no. 1, pp. 50–79, Aug. 1994. [Online]. Available: http://dx.doi.org/10.1006/inco.1994.1064

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

[15] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, "The complexity of searching a graph," *J. ACM*, vol. 35, no. 1, pp. 18–44, Jan. 1988. [Online]. Available: http://doi.acm.org/10.1145/42267.42268

[16] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, and J. ObdráLek, "The dag-width of directed graphs," *J. Comb. Theory Ser. B*, vol. 102, no. 4, pp. 900–923, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jctb.2012.04.004

[17] J. Barát, "Directed path-width and monotonicity in digraph searching," *Graphs and Combinatorics*, vol. 22, no. 2, pp. 161–172, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00373-005-0627-y

[18] P. Hunter and S. Kreutzer, "Digraph measures: Kelly decompositions, games, and orderings," *Theor. Comput. Sci.*, vol. 399, no. 3, pp. 206–219, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.02.038

[19] D. Dereniowski, W. Kubiak, and Y. Zwols, "Minimum length path decompositions," *CoRR*, vol. abs/1302.2788, 2013. [Online]. Available: http://arxiv.org/abs/1302.2788

[20] P. Eades, X. Lin, and W. F. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Inf. Process. Lett.*, 1993. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(93)90079-O

[21] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting positive and negative links in online social networks," in *WWW '10*. New York, NY, USA: ACM, 2010, pp. 641–650. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772756

[22] M. D. Domenico, A. Lima, P. Mougel, and M. Musolesi, "The anatomy of a scientific rumor," *Scientific Reports*, vol. 3, 01 2013. [Online]. Available: http://arxiv.org/abs/1301.2952

[23] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1217299.1217301

[24] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW'04*, 2004.

[25] P. Boldi, M. Rosa, M. Santini, and S. 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.

[26] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: http://www.sciencemag.org/cgi/content/abstract/286/5439/509

PLACE PHOTO HERE

Michael Simpson received his B.Sc. and M.Sc. degrees in Computer Science from the University of Victoria in 2011 and 2014 respectively. His research interests include social network analysis, graph theory, and approximation algorithms.

PLACE PHOTO HERE

Venkatesh Srinivasan received his B.Eng from the Birla Institute of Technology and Science in 1994 and his Ph.D. from the Tata Institute of Fundamental Research in 2000. His research interests are in theoretical computer science. In particular, the connection between computational complexity and other branches of computer science.

PLACE PHOTO HERE

Alex Thomo received his B.Sc. from the University of Piraeus, Athens in 1996 and his M.Sc. and Ph.D. degrees from Concordia University in 2001 and 2003 respectively. His research interests include algorithms for big graphs, social networks, and privacy.