Graph Masking: Maintaining Neighborhoods in Graph Randomization

Benjamin Caulfield Wesley Miller Malik Magdon-Ismail
Rensselaer Polytechnic Institute
Computer Science Department
Troy, NY 12180

December 19, 2013

Abstract

Given a graph $G(V, E)$ and a natural number k, we define the $k - Neighborhood$ graph of G to be $G_k(V, E_k)$, where the edge (u, v) in E is in E_k if and only if there is a path (u, v) of length less than or equal to k in G. We would like to find a *masking* of G, G', such that G and G' share a $K - neighborhood$ graph, but no edges in G can be determined by studying G' (G' is sufficiently random). This paper provides two heuristic algorithms to solve this problem. The first modifies G to get a new graph which is guaranteed to share a $k - Neighborhood$ with G, but may not be sufficiently random. The second algorithm builds the new graph by continuously adding edges to an originally empty graph. The new graph is sufficiently random from the original graph, but may not share the same $k - Neighborhood$.

1 Introduction

1.1 Motivation

Social networks provide means to create and maintain meaningful connections between people. They foster environments where people can not only connect with people with whom they have a previous relationship, but with entirely new people as well. Without a proper way to introduce people to others with whom they may have relevant connections, there is little room for social networks to grow and provide anything worthwile to their users.

As social networks, such as LinkedIn, hold all the data on connections between their users, they too hold the tools to make suggestions to users on which people to seek out for connections; however, this information must be used carefully. Users put their trust in the social networks that their private information will be kept safe and not disclosed to anyone else, so security is a major factor to be considered while trying to implement methods to

expand the network.

In order to give the users relevant suggestions to expand their personal connections, a construction or product of a subset of the information held by the networks must be given such that it both is helpful to the users and does not infringe on the privacy of others.

1.2 Goal

The ultimate goal of this research is to find a k value such that the network can provide a given user with a list of his/her k-neighbors that satisfy the aforementioned conditions. The list of k-neighbors given must not reveal too much to users to prevent network adversaries from being able to reconstruct the graph to any reasonable extent but must be useful enough to provide helpful information to the users who make appropriate use of the network.

1.3 Approach

This paper describes a method that looks at the connections between users to determine useful information to make available to a given user so that he/she may gain new meaningful connections. This includes a label-swapping algorithm and an edge adding algorithm. Algorithms were also designed to try to test the difficulty of accurately extracting any private information from the information given. All algorithms were evaluated for computational efficiency and data was taken to evaluate them overall.

Definition 1.1. A *graph* is a 2-tuple $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices (nodes) and the set of edges is $E = \{e_1, e_2, ..., e_m\} \subseteq V \times V$. All edges in E are undirected. Unless otherwise stated, when discussing a graph $G = (V, E)$, $v, u, w \in V$ and $e \in E$.

Definition 1.2. A *path* P of length l in G is a sequence of edges in E of the form $e'_1, e'_2, ..., e'_l$ such that $e'_i = (v, u)$ and $e'_{i+1} = (u, w)$ for all $i \in [1, l]$. If $e'_1 = (v'_0, v1')$ and $e'_l = (v'_{l-1}, v'_l)$, then P is a path from v'_0 to v'_l.

Definition 1.3. Let k be a positive integer. The *k-neighborhood* of a node $v \in V$ in a graph $G = (V, E)$, denoted $N_k(v)$ is the set of all $u \in V$ where there exists a path from v to u of length less than or equal to k. The k-neighborhood of G is the graph $N_k(G) = (V, E')$ where $(u, v) \in E'$ iff $v \in N_k(u)$. If $N_k(G) = G'$, we say that G *satisfies* G'.

Definition 1.4. A *masking* of a graph G is a graph G' which satisfies $N_k(G) = N_k(G')$.

Definition 1.5. For an integer k and graph G, we define the *adjacency group* of a node $v \in V$ as the set of all $u \in V$ such that $N_k(v) = N_k(u)$. We can see that adjacency groups are equivalence classes.

2 Label-Swapping Algorithm

In this section, we present the label-swapping algorithm which takes a graph G and yields G', a masking of G.This algorithm, as shown in figure 2, works by altering the original graph while maintaining the same $k - Neighborhood$. This is accomplished by partitioning the vertices of the graph into *adjacency-groups*.

Definition 2.1. A *swapping* on the adjacency-group A in V is a bijection from A onto itself, where each vertex is mapped to a randomly chosen vertex in A. A swapping on V is the union of swappings on each adjacency-group. The application of a swapping on a vertex v is denoted $Swap(v)$.

Theorem 2.1. *Applying a swapping to a graph, G, will yield a graph, G', with the same k-Neighborhood graph as G.*

Proof. Let G'_k be the k-Neighborhood graph of G' and G_k be the k-Neighborhood graph of G. Assume G'_k is not equal to G_k. Then (i) G'_k contains an edge not in G_k or (ii) G_k contains an edge not in G'_k.

i) Let (u, v) be an edge in G'_k that's not in G_k. Since G' was formed by *swappings* on G, u must have some label x and v must have some label y in G, where x and y were in the adjacency groups of u and v, respectively, and (x, y) is in G_k. But, since u and x are in the same adjacency group, and (x, y) is in G_k, then (u, y) must be in G_k. Since y and v are in the same adjacecy-group and (u, y) is in G_k, then (u, v) is in G_k, and our assumption that G'_k has an edge that is not in G_k must be false.

ii) Let (u, v) be an edge in G_k that is not in G'_k. Let the nodes labeled x and y in G be given the labels u and v, respectively, in G'. Therefore, u and x share an adjacency group, as do v and y. Since (u, v) is in G_k and u and x share an adjacency group, then (x, v) is in G_k. Likewise, since v and y share an adjacency group and (x, v) is in G_k, then (x, y) is in G_k. This implies that (u, v) must be in G'_k. Therefore, our assumption that G_k has an edge that is not in G'_k is false.

Because (i) and (ii) are false, we can conclude that G_k equals G'_k.

□

Definition 2.2. The *sorting criteria* used to order the adjacency groups is defined as follows. A given adjacency group a is *less than* another given adjacency group b if and only if a and b share the first $n - 1$ nodes, where n is a positive integer, and either a does not have an n^{th} node while b does, or they both have an n^{th} node, but $a_n < b_n$. A given adjacency group a is *greater than* another given adjacency group b if and only if $b < a$.

Definition 2.3. A *kTree* is a binary tree used to sort a set of adjacency groups based on the sorting criteria. When an adjacency group a is inserted into the kTree, it is sent to the root node. If the node is empty, a occupies its space; if it is not empty, a is compared to the one occupying the node, b. If $a < b$, then a is send to the left child of the node, and if $a > b$, then a is sent to the right child. This comparison process continues until a is placed within a previously empty node or a node is found that whose contents are equal to

a. When necessary, the kTree returns a sorted list of the adjacency groups by recursively passing up from each node a list containing the contents retrieved from that node's left child, its own contents, and the contents from its right child.

Input: Set of Adjacency Groups A
Output: Ordered List of Adjacency Groups A'
  Declare kTree T
  for all $a \in A$ **do**
    Insert a into T
  end for
  Extract sorted list of adjacency groups from T as A'
  return A'

Figure 1: Pseudocode for the kSort Algorithm.

Input: Graph $G = (V, E)$, Integer k
Output: Graph $G' = (V, E')$
  for all $v \in V$ **do**
    calculate $N_k(v)$
  end for
  Apply ksort algorithm to find adjacency groups
  for all $A \in AdjacencyGroups$ **do**
    find a valid swapping on A
  end for
  for all $(u, v) \in E$ **do**
    add edge $(Swap(u), Swap(v))$ to E'
  end for
  return $G' = (V, E')$

Figure 2: Pseudocode for the label-swapping algorithm.

The label-swapping algorithm, as shown in figure 2 works by finding all maximal adjacency groups and applying a random swapping to each one. The new graph formed by applying these swappings must have the same k-Neighborhood as the original graph (see Theorem 1); however, it may be possible to determine edges in the original graph from the new graph.

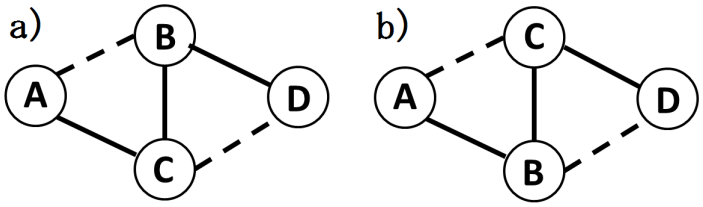

Figure 3: In the above graphs, solid lines represent edges in the original graph and dotted lines represent edges that are only in the 2-Neighborhood graph (Note that all edges in the original graph are necessarily in the 2-Neighborhood graph). a) The vertices B and C are in the same adjacency group, while A and D are each in adjacency groups of size 1. b) The result of applying a swapping to the adjacency group containing B and C, with the 2-Neighborhood graph remaining the same as the graph in (a).

3 Merging Heuristics

Unfortunately, although the label-swapping algorithm yields perfectly satisfying graphs, it often doesn't sufficiently disguise a given graph. In this section, we present a heuristic, called adjacency group merges (or simply merges) that further randomizes a given graph, but is not guaranteed to maintain the same k-neighborhood. There are two versions of this heuristic that we developed, deterministic merges and non-deterministic merges.

Definition 3.1. A *merge* or *merging* of a graph's adjacency groups combines similar adjacency groups into larger groups containing the union of their nodes based on their *difference*.

Definition 3.2. The *difference* between adjacency groups A and B is the size of the symmetric difference of $N_k(v)$ and $N_k(u)$, for $v \in A$ and $u \in B$.

3.1 Deterministic Merging

Definition 3.3. A *deterministic adjacency group merge* maps every pair of nodes to the differencence value between their adjacency groups and merges adjacency groups of the first n that have not been involved in a previous merge.

Input: graph $G = (V, E)$, k-neighborhoods $K = \{k_1, k_2, ...\}$, adjecency groups $A = \{a_1, a_2, ...\}$, limit L
Output: adjecency groups A'
for all $v \in V$ **do**
for all $w \in W$ where $v \neq w$ **do**
find $Diff(A(v), A(w))$
end for
end for
for all $(v, w, d) \in V \times V \times \mathbb{Z}$ sorted by $d = Diff(A(v), A(w))$ where $v \neq w$ **do**
$n = 0$
end for
if $A(v)$ has not been merged and $A(w)$ has not been merged **then**
$merge(A(v), A(w))$
mark $A(v)$ as merged
mark $A(w)$ as merged
$n++$
if $n = L$ **then**
break
end if
end if

Figure 4: Pseudocode for the Deterministic Merging Heuristic Algorithm.

3.2 Non-Deterministic Merging

Definition 3.4. A *non-deterministic adjacency group merge* merges all of the pairs of randomly selected adjacency groups that have not been involved in a previous merge and have a *difference* value less than or equal to a prescribed cutoff value.

Input: Integer max_diff, Graph $G = (V, E)$, Adjacency Groups A
  $current\_diff = 0$
  $total\_changes = 0$
  for all Randomly Selected $u \in V$ **do**
    if u has been altered **then**
      continue
    end if
    Integer $min\_diff = max\_diff + 1$
    Integer $min\_pos = -1$
    Integer $min\_diff2 = max\_diff + 1$
    Integer $min\_pos2 = -1$
    for all Randomly Selected $v \in V$ **do**
      if $v < u$ or v has been altered **then**
        continue
      end if
      if $diff(u, v) < min\_diff$ and $diff(u, v) \neq 0$ and u does not share an adjacency group with v **then**
        $min\_diff = diff(u, v)$
        $min\_pos = v$
      end if
    end for
    if $min\_pos \neq -1$ **then**
      $total\_changes++$
      mark u as altered
      mark min_pos as altered
      $current\_diff += min\_diff$
      merge the two adjacency groups belinging to u and min_pos
    end if
  end for

Figure 5: Pseudocode for the Non-Deterministic Merging Heuristic Algorithm.

4 Edge Adding Algorithm

Input: Integer k, k-neighborhood graph $G = (V, E)$
Output: Graph $G' = (V, E')$
  $wList = E$
  while $wList \neq \emptyset$ **do**
    select and remove some $(u, v) \in wList$
    perform a BFS of length k from $u \in V$
    if BFS reaches $x \in V$ such that $x \notin N_k(u)$ **then**
      skip
    end if
    perform a BFS of length k from $v \in V$
    if BFS reaches $x \in V$ such that $x \notin N_k(v)$ **then**
      skip
    end if
    add (u, v) to E'
  end whilereturn G'

Figure 6: Pseudocode for the Edge-Adding Algorithm

Figure 7: An example of the edge-adding algorithm. a) The 3-neighborhood graph for a given input. Each edge in this graph is added to the potential-edge list at the start of the algorithm. b) The solid lines represent edges that will be in the graph the algorithm returns (edge (B,D) was the last edge added). The dotted lines are remaining edges in the potential-edge list. After (B,D) was added, there became a 2-path between A and D. Since E is not adjacent to A in the 3-neighborhood graph, the edge (D,E) was removed from the potential-edge list.

The edge-adding algorithm, as shown in figure 7, is a greedy algorithm which find a graph $G = (V, E)$ that approximately satify a given k-neighborhood graph, $G_k = (V, E')$. The algorithm works to find a graph whose k-neighborhood is at least a subgraph of G_k. It begins by adding the edges of G_k into a working list. We know any satisfying graph must be a subgraph of G_k, as every edge of a graph must be included in the k-neighborhood of that graph. Therefore, we want to find a subset of our working list that satisfies G_k. This algorithm works by iteratively adding edges from the working list to the new graph. At each iteration, a random edge (u, v) is selected and removed from the working list. A breadth-first search of length k is run from both u and v in the current graph, G. If the search (say, from u) reaches a node that is not adjacent to u in G_k, then we know adding (u, v) to G would invalidate the graph and the edge is discarded. If no such node is found, then (u, v) is added to G. This process continues until the working list is empty. This algorithm runs in $O(|E| * d^k)$ time, where d is the maximum node density. As social networking graphs are typically sparse, this algorithm runs in near-linear time.

5 Adversary Simulation

This section presents a simulated contest between a social networking website publishing neighborhoods and an adversary looking to determine existing edges from these neighborhoods. The website, knowing the original social network, uses the label-swapping algorithm multiple times and tracks the frequency each edge appears (edges that never occur in an output of label-swapping are ignored). For some $\epsilon, \delta \in [0, 1]$, test the proportion of edges that occur with a frequency in $[0.5-\delta, 0.5+\delta]$. If that proportion is less than ϵ, increment k and repeat the process. If k reaches some set maximum (say 6), stop the process: k has become too large for the k-neighborhoods to hold any meaningful information. If the proportion is greater than ϵ, set $k' = k$. Figure 5 shows the determined k' values for various μ values. We belive that k' is the minimum k value to sufficiently disguise the given graph.

To test this theory, we pass the k'-neighborhood of G to the edge-adding algorithm and attempt to reconstruct G. The sucess of this attempt is measured by the proportion of edges the algorithm yields that are in G.

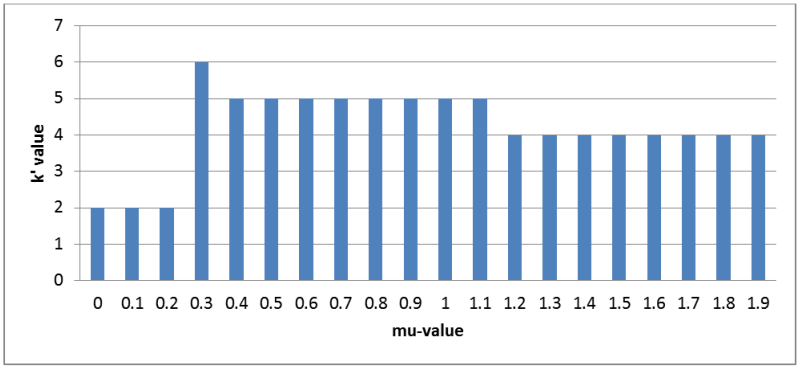

Figure 8: Returned k' values for synthetic graphs of different mu values.

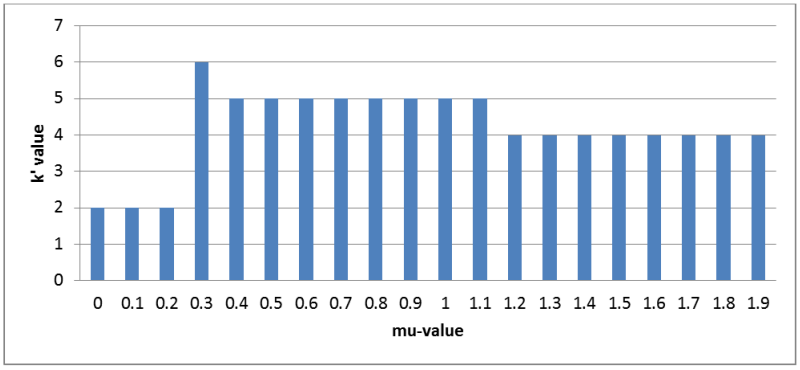

Figure 9: Output of edge-adding algorithm run with corresponding k-values.