Graph Masking: Maintaining Neighborhoods in Graph Randomization

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

January 1, 2014

Abstract

A k-neighborhood graph of a given graph, G, has an edge between two nodes if there is a path of length k or less between the nodes in G. This paper studies the problem of finding a graph that satisfies a given k-neighborhood graph. We present two algorithms which attempt to solve this problem. The first algorithm greedily adds edges to a graph until no edges can be added without invalidating a k-neighborhood. The second algorithm randomizes a given graph while maintaining k-neighborhoods. This algorithm does not always fully randomize the graph, so we present a better randomizing heuristic that invalidates some k-neighborhoods. All of our algorithms are tested on communication data from LinkedIn. We also run a simulation on this data to determine the minimum k value that can be applied so that edges from an original graph can not be determined from a given k-neighborhood graph.

1 Introduction

Social networking websites provide means to create and maintain meaningful connections between people. Because these websites hold all the data on connections between users, they too hold the tools to suggest future connections; however, this information must be used carefully. Networking websites are trusted with users' private information, and they must take care to not reveal any such information without the users' consent. Any new recommendation service must also reflect this responsibility.

One such recommendation service is the public display of a social network's *k-neighborhood graph*, which shows a connection between two users if there is a path of k or less between these users in the original network. Ideally, users could look at this graph and make new connections based on their k-neighbors. However, it is possible than an adversary could use this information to determine a connection between users that should be kept private.

This paper presents a method to find the minimum k value so that a graph cannot be determined from only its k-neighborhood. In the first section, we present a greedy algorithm which continuously adds edges to a graph until no edges can be added without invalidating the given k-neighborhood. The next section presents an algorithm to randomize a graph while maintaining its k-neighborhood. In section 4, we give a heuristic to better randomize a given graph. The final section uses this algorithm to find the minimum randomizing k value and uses the greedy algorithm to show that edges are properly disguised.

1.1 Relevant Work

There have been some studies on disguising networks of users so that only certain information is revealed. In 2002, Sweeney proposed the *k-anonymity* model, which states that a presentation of data is *k-anonymous* if any given user is indistinguishable from at least $k-1$ other individuals in the presentation [3]. In order to guarantee k-anonymity, Chester and Srivastava gave a method to alter a given network so that the resulting graph is k-anonymous [1]. Later, Chester et. al. study the complexity of this process of k-anonymization and define *t-closeness*, which measures how well an adversary could determine private information from an anonymized graph [2].

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 $G = (V, E)$, $u, v, w \in V$ 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 $i \in [1, l-1]$. If $e'_1 = (v'_0, v'_1)$ 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 Edge Adding Algorithm

The edge-adding algorithm, as shown in figure 2, is a greedy algorithm which find a graph $G = (V, E)$ that approximately satify a given

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 1: Pseudocode for the Edge-Adding Algorithm

Figure 2: 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.

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.

3 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

4, 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*.

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

Figure 3: 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 4: Pseudocode for the label-swapping algorithm.

The label-swapping algorithm, as shown in figure 4 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 3.1); however, it may be possible to determine edges in the original graph from the new graph.

Theorem 3.1. 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

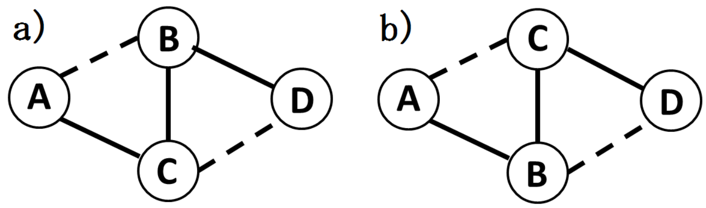

Figure 5: In the above graphs, solid lines represent edges in the original graph and dotted lines represent edges that are only in the 2-Neighborhood graph (Note that all edges in the original graph are necessarily in the 2-Neighborhood graph). a) The vertices B and C are in the same adjacency-group, 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).

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.

□

4 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 4.1. 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 4.2. 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 $u \in B$.

4.1 Deterministic Merging

Definition 4.3. 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.

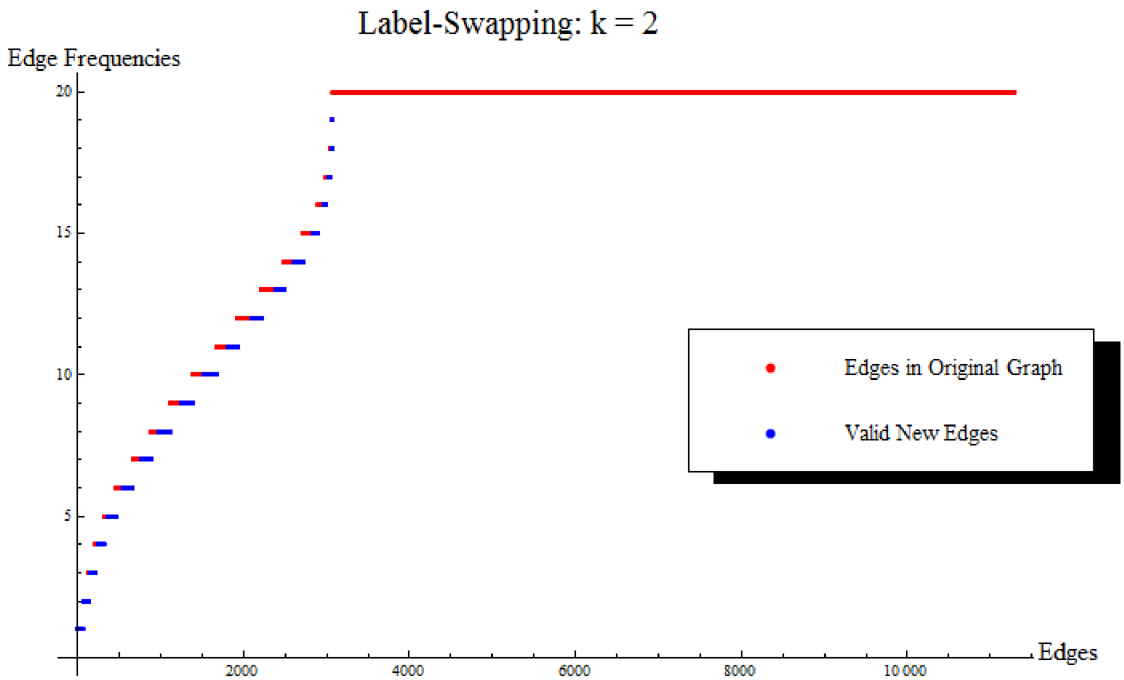

Figure 6: The results from the label-swapping algorithm when run 20 times on the blog data when k=2.

4.2 Non-Deterministic Merging

Definition 4.4. 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.

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

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 ∈ V do
    for all w ∈ W where v ≠ w do
        find Diff(A(v), A(w))
    end for
end for
for all (v, w, d) ∈ V × V × ℤ sorted by d = Diff(A(v), A(w)) where
v ≠ 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 7: Pseudocode for the Deterministic Merging Heuristic Algorithm.

measured by the proportion of edges the algorithm yields that are in G.

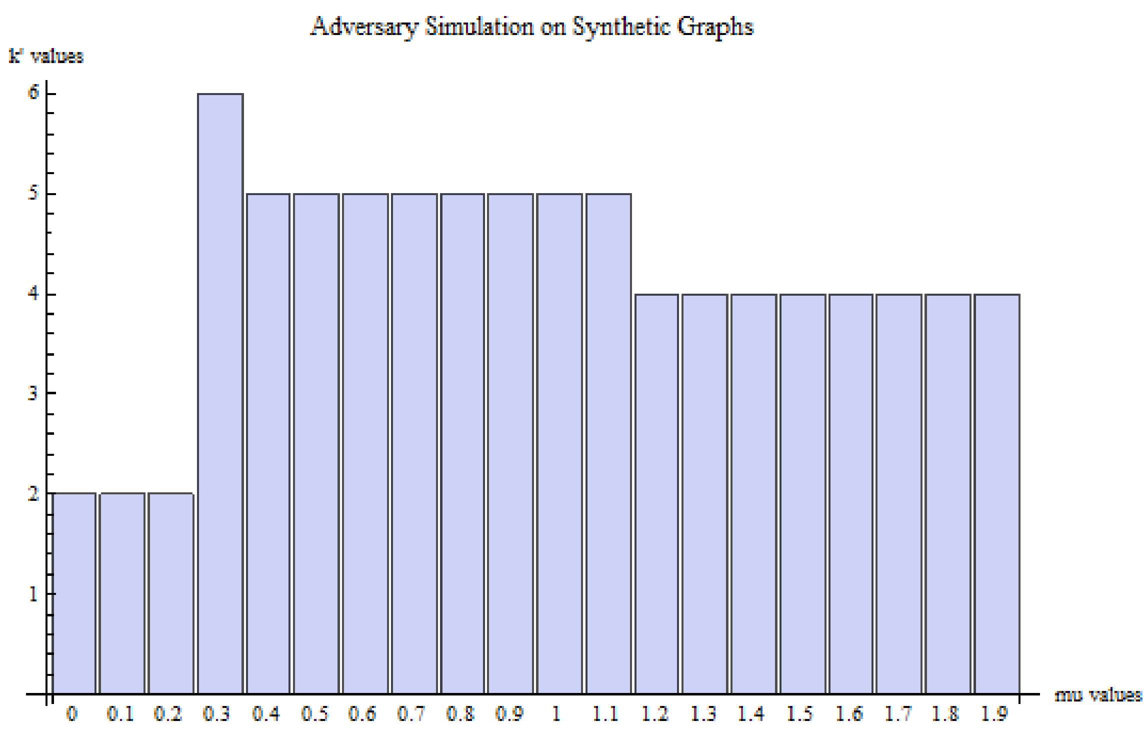

Figure 12: Returned k' values for synthetic graphs of different mu values.

Input: Integer max_diff, Graph $G = (V, E)$, Adjacency Groups A

```
current_diff = 0
total_changes = 0
for all u ∈ 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 v ∈ V do
        if v < u or v has been altered then
            continue
        end if
        if diff(u, v) < min_diff and diff(u, v) ≠ 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 ≠ −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 8: Pseudocode for the Non-Deterministic Merging Heuristic Algorithm.

References

[1] Chester, S., Srivastava, G. (2011). 1 social network privacy for attribute disclosure attacks. Advances in Social Networks Analysis and Mining.

[2] Chester, S., Kapron, B., Srivastava, G., Venkatesh, S. (2013). Complexity of social network anonymization. (2nd ed., Vol. 3, pp. 151-166). Vienna: Springer.

[3] Sweeney, L. (2002). k -anonymity: a model for protecting privacy. International Journal on Uncertainty, 10(5), 557-570.

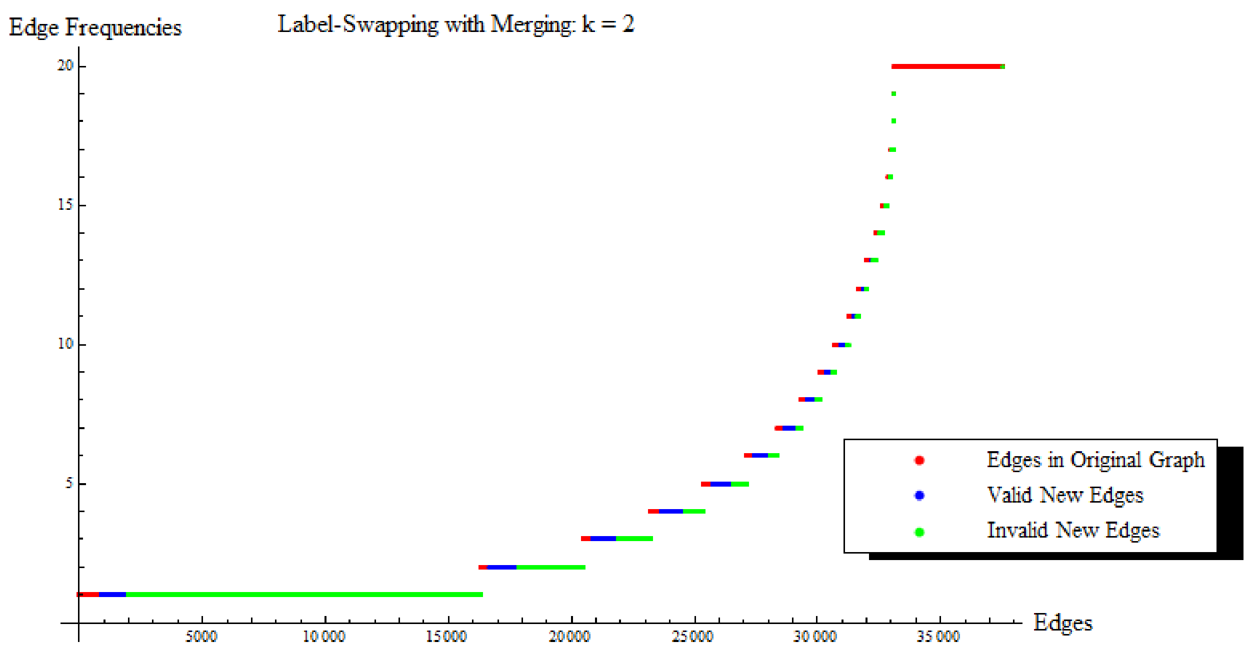

Figure 9: The results from the label-swapping algorithm when run 20 times on the blog data when k=2. Adjacency groups differing by 5 or less were merged.

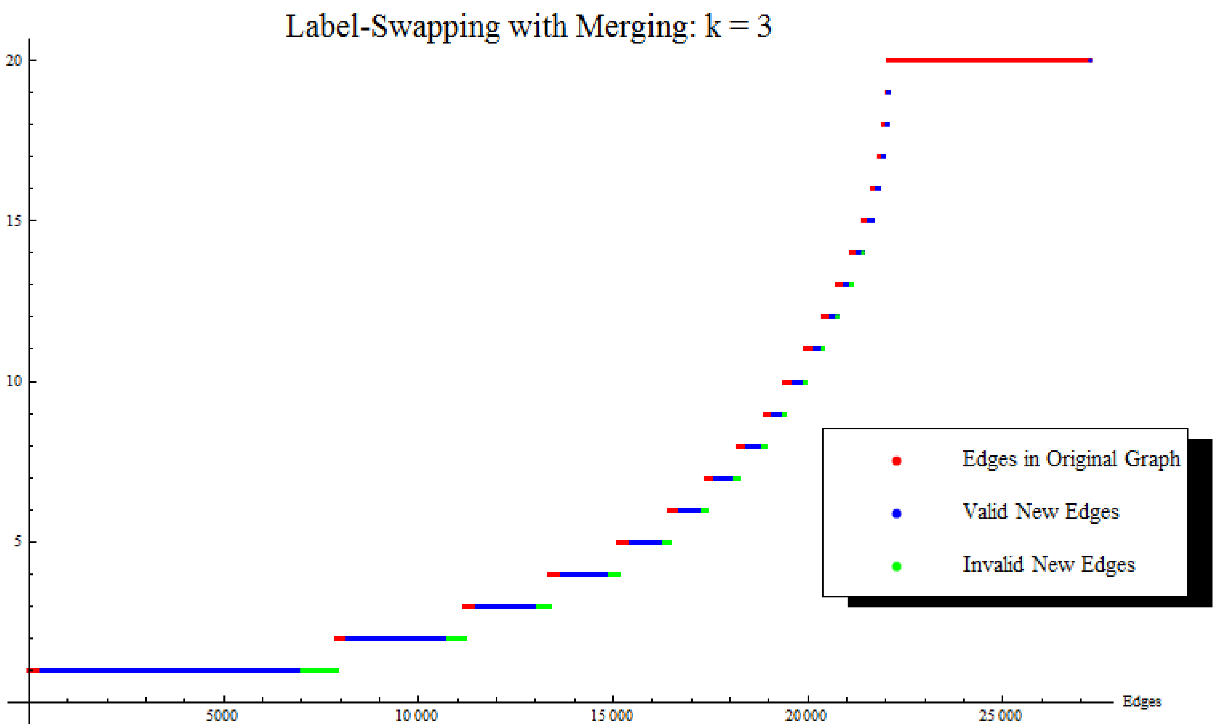

Figure 10: The results from the label-swapping algorithm when run 20 times on the blog data when k=3. Adjacency groups differing by 5 or less were merged.

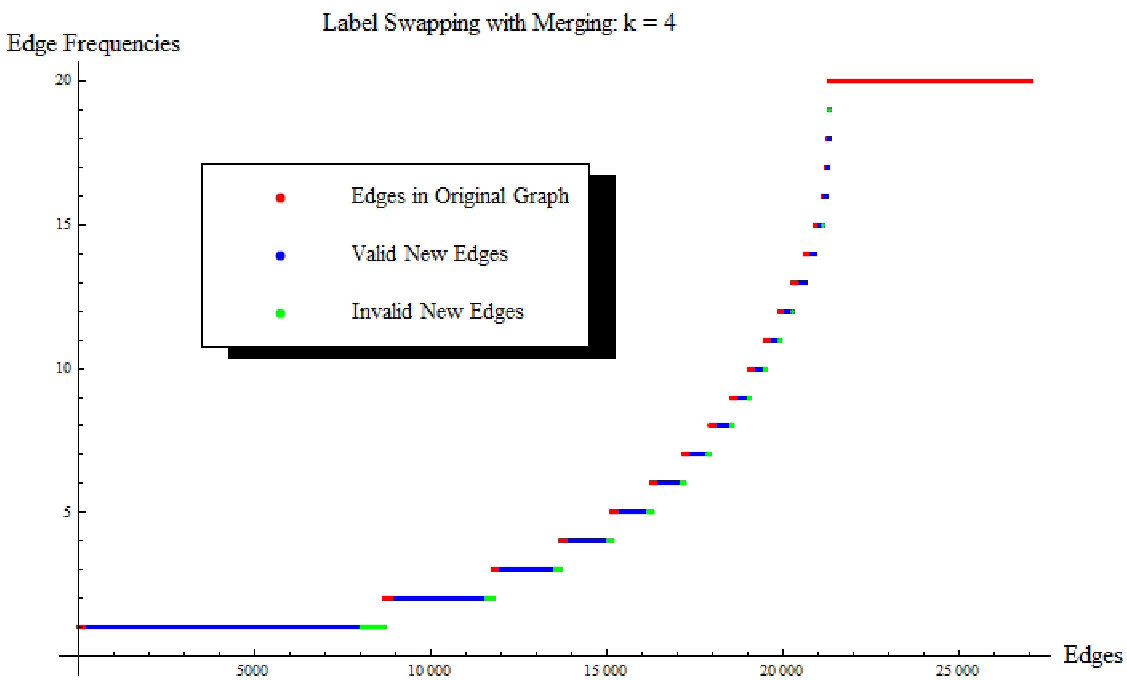

Figure 11: The results from the label-swapping algorithm when run 20 times on the blog data when k=4. Adjacency groups differing by 5 or less were merged.