Tracking Moving Clutches in Streaming Graphs

Sudarshan S. Chawathe*

Computer Science Department
University of Maryland
College Park, Maryland 20742.
chaw@cs.umd.edu

May 2002

Abstract

We address the problem of tracking groups of interacting entities as they move in a graph with vertices representing hosts or locations and edges representing interactions between hosts. The graph of interactions is modeled as a stream of edges (with the arrival of an edge signifying an interaction between the hosts it connects). This problem arises in applications such as tracking groups of fraudulent callers in a telephone network and tracking identities of malicious agents (programs or people) on a data network. We present a formalization of this problem and a streaming solution that uses bounded storage and provides real-time response. Our solution is based on maintaining, at each instant, an approximation of the streaming graph seen so far. We present empirical results to quantify the effectiveness of our solution.

1 Introduction

Long-distance telephone companies need to monitor their networks for fraudulent calls. Empirical evidence indicates that people who make fraudulent calls tend to communicate with other people who make fraudulent calls. This characteristic leads to the formation of groups of people with strong interconnections within the group. We refer to such groups as **clutches** and their members as **agents**. Identifying fraud clutches and tracking their evolution is an important component of fraud detection. By paying careful attention to the members of such clutches, and those closely linked to them, future fraudulent activity can be prevented or detected early.

Identifying fraudulent clutches is an interesting problem, but it is not the focus of this paper. Instead, our focus is on tracking fraud clutches after they have been detected. The key observation here is that we do not have any direct information about the identity of agents

*This work was supported by the National Science Foundation with grants IIS-9984296 (CAREER) and IIS-0081860 (ITR).

initiating a phone call. Calls are identified by the source and destination phone numbers. However, we cannot identify the agents of a clutch naively using the phone numbers they use because they switch phone numbers frequently in an attempt to circumvent blacklisting by the phone companies. In effect, when a phone company detects fraudulent use of a phone number and flags it, the agent moves to another phone number and resumes fraudulent operations from there.

As the agents in a clutch move, the identity of the clutch needs to be updated to reflect the new phone numbers (*hosts*, in general) that the agents are using. Obviously, the fraudulent agents do not inform the phone company of a change in the phone number from which they commit fraud! Instead, the phone company must infer such a change based on records of calls being made on its network. In effect, the problem is one of identifying clutches, which are groups of fraudulent *agents* (people), using information about communications between *hosts* (phone numbers). The mapping between agents and hosts is dynamic and must be inferred from the communications emanating from the hosts.

The basic idea behind inferring agents based on the hosts they use is as follows: If Alice is known to call Bob every Sunday around 7:00PM and Cathy every evening at 10:00PM (and so on), then if we find another phone number with a matching calling pattern, we may wish to infer that Alice may be using that number. If agents in a clutch (say, Alice, Bob, Cathy, and Dan) move one at a time, with plenty of intervening communications, the problem is relatively straightforward (although still computationally intensive due to the large volumes of call data that are involved). Suppose these four people are using phone numbers 1, 2, 3, and 4, respectively. If Alice starts using a new phone number, 101, we can detect the similarity in the calling patterns of 1 and 101 and infer that Cathy is now also operating from 101. Then, if we observe a number 102 which communicates with 101, 3, and 4 in a manner similar to 2, we can infer that (since 101 and 1 are now known to both map to Alice), that 102 may be a new number for Bob.

The problem becomes more complicated if (as is likely to happen in practice), two or more agents simultaneously start using new phone numbers. (By simultaneously, we mean close enough in time that there is not a significant amount of communication between agents in the time interval between the two events.) In our simple example, if Alice and Bob start using 101 and 102, respectively, on the same day, it is more difficult to detect their new numbers. Essentially, we have to reason that 101 is similar in calling pattern to 1 if 102 can be identified with 2 (and similarly in the other direction). When dealing with three or more simultaneous moves of this nature, we need to keep track of how mappings of agents to phone numbers depend on other mappings in general.

Our focus in this paper is on tracking clutches of agents as they move in a (communication) network of hosts, by using the communication patterns. As shall become clear from the general problem statement presented in the next section, this problem has applications in domains other than fraud detection in phone networks. For example, a similar problem arises in data networks, with computer hosts (say, identified using IP addresses) and worms or other malicious programs that may be communicating to coordinate a distributed denial-of-service attack. As another example, consider the banking domain. Here agents are people or organizations that operate accounts, hosts are bank accounts, and the communications of interest are transfers and other transactions between accounts. If a group of ten people

routinely sends money to one bank account and is later observed sending money to another, the common pattern could suggest that the real recipient of the funds has moved to a new account.

In many communication networks, certainly telephone and data networks, the rate at which communications occur is extremely high, making it important to process the data on-the-fly as much as possible. Storing the data and analyzing it offline, while sometimes possible, has serious drawbacks. First, offline detection involves a (typically several hour) delay between the occurrence of an interesting event and its detection. For many applications, a near realtime response is important. Second, even with today's rapidly falling storage prices, the amount of storage needed to simply record all data as it is generated is simply too large. For example, even with AT&T's resources, researchers have found it beneficial to maintain a abbreviated version of the data generated by the network. For these reasons, it makes sense to treat the data obtained from a communication network as *stream data*.

We make two main contributions in this paper. The first is the formulation and formalization of the problem of *tracking moving clutches*, informally described above. This problem is interesting from both applications and research viewpoints. The second is a solution to this problem for streaming data. Our solution adapts to the available storage and provides near realtime response.

In the next section, we describe the problem more precisely, develop some of the ideas used in the rest of the paper, and present a formal problem definition. In Section 3, we describe our method for detecting moving clutches in streaming graphs. Section 4 presents an experimental evaluation of our method. We discuss related work in Section 5 and conclude in Section 6.

2 Problem Development

The general setting for our problem is a communication network composed of **hosts** and **links**. Intuitively, hosts are entities that can be unambiguously identified (see below) by the communication network. Examples of hosts are phone numbers in telephone networks and IP or MAC addresses in a data network. Links between hosts represent network interaction. For example, in a telephone network, each phone call can be modeled as a link from the calling host to the called host. Link **labels** (annotations) are used to model the salient characteristics of the interaction (e.g., length of a phone call, number of packets sent on a TCP connection). Consider the graph obtained by mapping hosts to vertices and links to edges. We refer to this graph as the **call graph**. (More precisely, this mapping yields a multigraph; see below.) In what follows, we shall often drop the distinction between a vertex in this graph and the host it represents (and, similarly, between an edge in the graph and the link it represents).

The call graph is continually changing as new interactions between hosts are recorded. In its purest form, the call graph is a large, continually growing, historical database. In the applications of interest (e.g., detecting fraudulent hosts in telephone networks, detecting malicious agents in a data network) the rate at which edges are added to the call graph is extremely high. Storing and analyzing the database in its entirety is not practicable. Further, a timely response to fraudulent activity is of particular importance. (Detecting

clutches of fraudulent callers from a year ago is unlikely to yield much benefit, since the agents typically do not use a phone number for very long.) Given these characteristics of the data, it is best viewed as *stream data*. Specifically, we model our data as a stream of graph edges. We assume that we are not interested in hosts that do not interact with others (degree-0 vertices in the graph); therefore, we can infer the vertices from the edges.

The notion of unambiguous identification of hosts requires some explanation as it depends on the level at which a network is modeled. For example, IP addresses can be *spoofed* using well-known techniques [Bel89]. However, if we model the network at a higher level of abstraction (say, at the TCP level), we assume that IP addresses are reliable. In effect, we assume the standard layered network architecture in which each layer must trust the one below it. It is possible to apply the techniques we describe in this paper to multiple levels by changing the real-world entities to which hosts are mapped. However, the focus of this paper is efficient detection of moving clutches in a single layer; therefore we do not develop the multi-layer idea further. Further, we present our ideas for an undirected call graph for ease of presentation only; they apply almost unchanged to directed graphs.

As described, the collection of hosts and links forms a labeled, weighted *multigraph* [Bol98]; that is, there may be multiple edges between a pair of vertices (e.g., one corresponding to each phone call between the hosts the vertices represent). For ease of presentation, our description below assumes that this multigraph has been mapped to a graph by a method such as thresholding on edge weights: We create an edge between two vertices only when the collection of links between the corresponding hosts meets some criterion. Our description does not depend on any particular mapping from multigraphs to graphs, although a natural one for telephone networks is the aggregate cost or connection time of links between a pair of hosts exceeding a threshold. Thus, instead of an edge representing an individual interaction (e.g, phone call) between hosts, it represents a *significant* interaction (e.g., several phone calls in one week, costing more than $100).

In more detail, we are given an infinite data stream consisting of a sequence of edges: $P = (e_i)_{i=1}^{\infty} = \{e_1, e_2, \ldots\}$, with $e_i = (u_i, v_i)$. Let P_{lm}, $l \leq m$, denote the restriction of P to edges e_l through e_m (inclusive): $P_{lm} = (e_i)_{i=l}^{m}$. We shall identify time with the indices of the stream P. Thus, at **time** k, the sequence P_{1k} has been seen while the remaining sequence $P_{k+1,\infty}$ is unavailable (as it denotes edges that become available in the future). It is also useful view the sequence $(P_{1k})_{k=1}^{\infty}$ as a sequence of graphs $(G_k)_{k=1}^{\infty}$, where G_k is the graph induced by the edges in P_{1k}. We are also given a **cutoff index** c. Finally, we are given a collection of **fraud clutches** $\mathcal{S} = \{S_1, S_2, \ldots, S_f\}$, where each graph S_i is a subgraph of G_{1c}. On receiving the k'th edge e_k, we are required to detect all subgraphs G of P_{1k} that **match** (as explained below) a clutch $S \in \mathcal{S}$. (Note that the we use the term clutch to refer to an identifiable group of communicating agents that we wish to monitor. *Fraud* clutches are clutches that have been specially flagged. That is, not all clutches are fraud clutches.)

We now make more precise the idea of **finding a match** to a clutch. An initial idea is to say a that a graph G matches a clutch S if G is isomorphic to S (in a graph-theoretical sense). There are, however, two problems with such a definition. First, since the definition is based on only the interconnections among the hosts of a clutch, it is likely to give rise to many false positive matches (i.e., matches that do not represent the agents in the matched clutch). For example, the calling pattern of a clutch consisting of seven hosts is likely to

match, just by chance, that of some group of seven friends that is completely unrelated to the clutch. Of course, given only the information in the call graph, no method can eliminate such false positive matches while still detecting some matches. (An adversary could pick a call graph with several subgraphs isomorphic to a clutch.) However, this definition would essentially guarantee many false positives. For example, a clutch isomorphic to K_5 will match any group of five friends who call each other frequently. The second problem with defining matches based purely on isomorphism is tractability. Subgraph isomorphism is a celebrated NP-hard problem even in a simplified, offline (non-streaming) setting [GJ90]. A streaming version of it that includes memory and processing constraints is not likely to admit an effective solution.

We therefore amend our working definition as follows. A graph G matches a clutch S if G is isomorphic to S *and* G and S have at least one vertex in common. Intuitively, this definition is likely to result in fewer false positive matches because although subgraphs isomorphic to a clutch may be common, the number of such subgraphs that share a vertex with the clutch is likely to be small. More precisely, we are making a trade-off between false positives and false negatives; by insisting on at least one common vertex, we will miss detecting moves of clutches in which all agents in the clutch simultaneously move to new hosts.

We make one more amendment to our working definition of matching graphs to account for the likelihood of entities that move communicating with new entities not previously in the clutch. (For example, in a telephone network, a fraudulent caller may graduate to a broker who recruits new fraudulent callers. In a computer network, a worm preparing for a distributed denial-of-service attack may communicate with newly infected hosts.) We therefore permit G to include edges not present in S. More precisely, the notion of a match is explicated by the following definitions:

Following standard terminology, the **neighborhood** of a vertex u is the set of vertices v such that edge (u, v) exists. Intuitively, we say that a vertex u neighborhood-dominates vertex u' in a graph if u's neighborhood is a superset of the neighborhood of u'. However, we modify the superset test by allowing some vertices to be mapped to others by a mapping m' that parameterizes the neighborhood-domination relationship. More precisely, we have the following:

Definition 1 (Neighborhood-domination for vertices) Given graphs $G = (V, E)$ and $G' = (V', E')$, and a partial one-to-one mapping $m : V \to V'$, a vertex $v \in V$ neighborhood-dominates $v' \in V'$ (under m) if $m'(n(v)) \supseteq n(v')$, where $m' : U \cup U' \to U \cup U'$ is the mapping obtained by extending m to be the identity on vertices not mapped by m, and to sets of vertices in the natural manner: $m'(\{u_1, u_2, \ldots, u_n\}) = \{m'(u_1), m'(u_2), \ldots, m'(u_n)\}$. If v dominates v' under a mapping m, we write $v \sim_m v'$. We use the notation $\{x/x', y/y', \ldots\}$ for a mapping that maps x to x', y to y', and so on. We refer to x/x' as a **substitution**.

The neighborhood domination relation is extended to graphs in the natural manner:

Definition 2 (Neighborhood-domination for graphs) A graph $G = (V, E)$ neighborhood-dominates a graph $G' = (V', E')$ under a partial mapping $m : V \to V'$ if $\forall v \in V : v \sim_m m(v)$. We write $G \sim_m G'$.

The size of a mapping is the number of vertices it maps nontrivially (to a vertex other than the vertex itself). The size of the smallest mapping that satisfies the conditions in the definition of neighborhood domination is a measure of dissimilarity. We specify graphs with small dissimilarity using the following definition:

Definition 3 (t-matching) A graph G t-matches a graph G' if there exists a partial mapping m' of size at most t such that $G \sim_m G'$.

Recall our earlier discussion on requiring that a fraud clutch and a subgraph deemed its match have at least one common vertex:

Definition 4 (ot-Matching) A graph $G = (V, E)$ ot-matches a graph $G' = (V', E')$ if G t-matches G' and $V \cap V' \neq \emptyset$. A graph $G = (V, E)$ ot-matches a graph $G' = (V', E')$ if G

Using the idea of an ot-matching, we can now state our problem definition formally as follows:

Definition 5 (Formal Problem Definition) Given an infinite stream of edges $P = (e_i)_{i=1}^{\infty}$, an integer c (called the old-vertex cutoff), a collection of graphs $\mathcal{S} = \{S_1, S_2, \ldots, S_f\}$ such that each S_i is a subgraph of P_{1c}, and a threshold t, find, for each position k in the stream, the subgraphs G of P_{ck} that ot-match some $S \in \mathcal{S}$ without using any information from $P_{k+1,\infty}$, and using only a fixed amount M of storage.

Note that the maximum amount of storage available is part of the problem definition. That is, we need solutions that work well over a range of available storage sizes. Implicit in the definition is the requirement that each match be detected as soon as possible (i.e., at the point in the stream at which there is enough information to confirm the match). This timeliness requirement may not be critical for all applications. For example, in detecting fraud in telephone networks, a delay of up to several days may be acceptable, in which case some of the processing can be done offline. However, our focus is on problems that require online detection. For example, for detecting suspicious connections on a data network in order to prevent a distributed network attack, we need real-time or near real-time response time.

The problem definition does not explicitly mention the rate at which the stream of edges must be consumed. However, it is clear that the ability to process the stream rapidly (high throughput) is important. For many applications, the required throughput rates preclude any substantial number of disk accesses. Thus, the M in the problem definition is of the order of RAM sizes (few Gigabytes), not disk sizes.

Since the stream of edges is infinite, any solution that is based on storing all data will eventually run out of space, assuming bounded space. In practice, even if we assume space is infinite (as we can buy additional storage over time), the amount of storage that can be accessed quickly remains bounded. Thus, a high processing throughput requires storing only part of the edge stream seen so far. Any solution that does not store all the information in the data stream for eternity cannot find all matches. (For example, a solution based on storing only the edges seen in the last seven days will miss detecting a match with a subgraph

whose edges appear spaced more than a week apart. It is easy to observe that, no matter what scheme is chosen for deleting some of the edges seen so far, an adversary can pick a sequence of edges that includes a match that cannot be detected.) Thus, strictly speaking, the problem as stated admits solutions only for stream indices up to a certain limit (based on the amount of storage, M). In practice, we are rarely interested in detecting matches for eternity. Instead, the stream history of interest is moved forward in time. In our problem definition, this history moving operation corresponds to moving the origin (index 1) of the stream forward in time. Although such considerations are obviously necessary, they are not the focus of this paper. Our assumption is that they occur rarely and that the runtime performance depends primarily on solution for a fixed historical origin. We therefore do not discuss them further in this paper.

3 Clutch Tracking Method

Our method for tracking clutches is based on two main ideas. The first is a case analysis of matching subgraphs of the call graph, given the requirements of ot-matching described in Section 2. The second is an incremental, lossy compression method for reducing the storage requirements of the call graph seen so far.

3.1 Case analysis of matching

Let $i(x)$ denote the index of the first occurrence of vertex x in the stream P. That is, $i(x)$ is the smallest stream index i such $e_i = (x, y)$ or $e_i = (y, x)$. We use the indices of first occurrence to classify vertices into three sets: (1) The set of **old vertices** consists of vertices that first no later than the old-vertex cutoff c; that is, $N_o = \{x : i(x) \in [1, c]\}$. (2) The set of **new vertices** consists of vertices that first occur after the old-vertex cutoff but before the current index k; that is, $N_n = \{x : i(x) \in [c + 1, k - 1]\}$. (3) Finally, the set of **unseen vertices** consists of vertices that first occur at or beyond the current index; that is $N_u = \{x : i(x) \in [k, \infty]\}$. Given our problem formulation, old vertices that are not also clutch vertices can be ignored for the purposes of detecting moving clutches. We therefore assume that the set of old vertices is identical to the set of clutch vertices, N_s. (If detecting moving clutches is the only application of interest, all information about non-clutch old vertices can be deleted. In the more likely scenario where such information has other uses, we simply preprocess the old vertex data to filter out the unnecessary vertices.)

We now describe the actions performed on the arrival of an edge $e = (u, v)$ on the edge stream. Since each endpoint of e belongs to one of the three sets described above (old, new, and unseen), we have six cases to analyze. We present these below.

Case 1: u unseen; v old.

Since u has not been seen before, the current edge $e = (u, v)$ is the only edge incident on it (so far). Further, the common vertex requirement implies that u can only match some vertex in the neighborhood of v. Now since $n(u) = \{v\}$, a vertex $u' \in n(v)$ matches u if and only if $n(u') = \{v\}$. In other words, we need only examine the neighborhood of v for degree-1 vertex, all of which match u.

```
if X' = ∅ then return m; fi
if r < ||X'|| then return ⊥; fi
for x' ∈ X' do
    M'_x := {x ∈ n(w) : ||n(x') − m(n(x))|| ≤ r};
    found := false;
    for x ∈ M'_x do
        m' := nMatch(r − 1, n(x') − m(n(x)), x, {x/x'} ∪ m);
        if m ≠ ⊥ then begin
                            found := true;
                            r := r + ||m'|| − ||m||;
                            m := m';
                       end
        fi;
    end
    if found = false then return ⊥; fi
end
return m;
```

Figure 1: Pseudo-code for function *nMatch*

Case 2: u unseen; v new.
In this case, both u and v must be mapped to old vertices. Therefore, if the matching threshold t is smaller than two, no match is possible. If $t \geq 2$, we seed the search for matching vertices using u: Since the degree of u is one, we search over the degree-1 vertices in the clutch sets in $\mathcal{S}$. For each such vertex u' with $n(u') = \{v'\}$, we try to generate a match by mapping u to u' and v to v'. The matching condition for mapping u to u' (viz., $m(n(u)) \supseteq n(u')$) is satisfied since $m(n(u)) = m(\{v\}) = \{v'\} = n(u')$. However, we need to check the matching condition for mapping v to v'. In particular, all vertices in the neighborhood of v' must be mapped to some vertices (recursively, satisfying the mapping conditions). This recursive expansion of a mapping to satisfy the mapping conditions is outlined as function *nMatch* below (and will be useful in other cases too).

Algorithm 3.1 (Function nMatch)
Input:
r: maximum number of elements that can be added to the mapping m
X': set of vertices to be matched
w: anchor; the mate of each vertex in X' must be in $n(w)$
m: current mapping (which is extended by at most r elements)
Output:
m': mapping (superset of m) such that $\forall x \in X' : \exists\{x/x'\} \in m'$; $\perp$ if no such mapping exists.
Method: The psuedo-code is displayed in Figure 1. The code iterates over each vertex x' that is to be matched. If it fails to find a match for any one, it returns null immediately. For matching a vertex x', the simple case is when there is a vertex x in w's neighborhood such that the m maps x's neighborhood to a superset of x'. However, if a vertex x in w's neighborhood

does not meet this requirement, we cannot dismiss it since it may be possible to map some of the vertices in x's neighborhood to those in x' neighborhood, by extending the given mapping m. Determining whether the vertices in x's neighborhood can be mapped in this manner is accomplished by making recursive calls to nMatch. The rest of the code is arithmetic to keep track of how many more substitutions we can add to m, based on the original limit r. As an optimization, we skip exploring vertices x whose mapped neighborhoods differ from the neighborhood of u' by an amount that exceeds r (since the unifying these neighborhoods would require extending m by more than r, which is not permitted).

Using the function *nMatch*, the expansion of the mapping as required in Case 2 can be accomplished as follows:

for $S \in \mathcal{S}$ **do**
  for $u' \in S$ **do**
    if $n(u') = \{v'\}$ **then** $nMatch(r, n(v') - m(n(v)), v, \{u/u', v/v'\})$; **fi**;
  end;
end;

Case 3: u unseen; v unseen.

In this case, the subgraph of the current call graph induced by $\{u, v\}$ (with the single edge (u, v)) is disconnected from the rest of the call graph. Given the common vertex requirement in the problem definition, this edge cannot lead to any matches yet. (However, a match using this edge at a later point in the stream is possible: as edges are added to the graph, u and v may become connected to a clutch.)

Case 4: u new; v old.

This case is similar to Case 1 in that since v is an old vertex that can be used to seed the search for a matching. However, unlike Case 1, the neighborhood of u contains vertices other than v. (The degree of u is at least two since its classification as new implies the occurrence of an earlier edge incident on u.) Therefore, in mapping u to a vertex u' in the neighborhood of v, we need to ensure that the neighborhoods of u and u' satisfy the mapping condition: $m(n(u)) \supseteq n(u')$. The nMatch function is used to expand the mapping m as needed to satisfy this condition. More precisely, we evaluate $nMatch(t - 1, n(u') - m(n(u)), v, \{u/u', v/v'\})$ for each $u' \in n(v)$.

Case 5: u new; v new.

In this case, both u and v have occurred before, and their neighborhoods contain several vertices in general. We must find a pair of adjacent vertices u' and v' in some clutch that match u and v, respectively. That is, we must find a mapping that simultaneously matches the neighborhoods of both u and v. Put another way, we must extend m, if needed, to map the vertices in the neighborhood of u' to vertices in u's neighborhood, and similarly for v' and v. These tasks are accomplished by calls to the nMatch function. We pass the mapping as modified by one call to the other in order to make sure we find a consistent mapping.

for $S \in \mathcal{S}, v' \in S$ **do**
  for $u' \in n(v') \cap N_o$ **do**
    $m' := nMatch(t-2, n(u') - m(n(u)), u, \{u/u', v/v'\}$;
    if $m' \neq \perp$ **then begin**
      $m := nMatch(t - \|m'\|, n(u') - m(n(u)), u, \{u/u', v/v'\}$;
    end;
  fi
  end;
end;

Case 6: u old; v old.

Since both u and v are already old (hence, clutch) vertices, this edge cannot give rise to any new matches. However, the neighborhoods of u and v are updated to include the other. These updated neighborhoods will be used in future matches. With such a neighborhood update, it is possible that matches detected earlier are no longer valid. For example, a match that mapped u and v to vertices u' and v', where the edge (u, v) does not belong to the call graph, will no longer be valid. However, given the streaming, online nature of the problem, we do not consider the issue of retroactively invalidating matches made earlier. Further, if u and v belong to different clutches this edge may suggest that the clutches be merged. Such considerations are orthogonal to the problem of detecting moving clutches and we do not consider them further in this paper. For the problem at hand, the only action required is the updating of the neighborhoods of u and v.

3.2 Neighborhood Store

We store the graph G_k (determined from the stream P_{1k} seen at time k) by storing the neighborhoods of each vertex. As described above, retrieval of vertex's neighborhoods is an operation frequently used by our method for detecting moving clutches; therefore we organize the neighborhoods using a hash table keyed by vertex identifiers. We refer to this collection of indexed neighborhoods as the **neighborhood store**. However, given the fixed storage requirement in the problem definition (Section 2), a simple strategy that simply records all neighborhoods is not feasible. We limit the amount of storage required for the neighborhood store by compressing the store when it gets too large. Our compression method, described below, reduces the number of neighborhoods in half.

The basic idea behind this compression method is to replace the neighborhoods of two vertices with a single neighborhood that is an approximation of both. We have a few variations based on the manner in which vertices to be merged are selected, and the manner in which neighborhoods are merged. We discuss these below: If the vertices being merged have identical neighborhoods, the merging process consists of simply replacing the neighborhood of one with a pointer to the neighborhood of the other. Further, this merging results in no loss of information and yields space savings proportional to the size of the neighborhoods being merged. Unfortunately, most of the vertices that are merged do not have identical neighborhoods and we need to select a common neighborhood that, after compression, will be used as the neighborhood for both vertices. We consider two simple options for this common neighborhood: the intersection and the union of the individual neighborhoods. When

individual neighborhoods are replaced by their intersection, we risk missing matches since neighbors of one vertex that are not neighbors of the other will be absent from the neighborhoods of both vertices after the compression step. In the extreme case, if the neighborhoods being merged are disjoint, the merging process leaves both vertices with empty neighborhoods. Analogously, replacing individual neighborhoods with their union can lead to many spurious matches because the merging step in effect adds the vertices that are in the neighborhood of a vertex to the neighborhood of its partner, irrespective of whether they occur in the partner's neighborhood.

Suppose we are merging neighborhoods by substituting the union of the neighborhoods for both. That is, when vertices x and y are merged, $n(x)$ and $n(y)$ are replaced by $n(x) \cup n(y)$. (We use the term merging of vertices to mean merging their neighborhoods.) Using the union instead of $n(x)$ may result in the use of the spurious edges in $n(y) - n(x)$. We refer to these as erroneous edges. We use the normalized number of erroneous edges introduced for $n(x)$, i.e., $(n(y) - n(x))/n(x)$ to determine the desirability of merging x and y, from x's point of view. The similar quantity from y's side is $(n(x) - n(y))/n(y)$. Thus, accounting for both x and y, we have the following metric for the normalized error of merging the neighborhoods of x and y with their union:

$$e_u(x, y) = \frac{1}{2} \cdot \left(\frac{n(y) - n(x)}{n(x)} + \frac{n(x) - n(y)}{n(y)} \right)$$

A similar reasoning for the case in which we use the intersection of neighborhoods as the replacement for the individual neighborhoods indicates $(n(x) - n(y))/n(x)$ as the relative error for x and $(n(y) - n(x))/n(y)$ as the relative error for y, yielding the following as the metric for normalized error of merging using intersection:

$$e_i(x, y) = \frac{1}{2} \cdot \left(\frac{n(x) - n(y)}{n(x)} + \frac{n(y) - n(x)}{n(y)} \right)$$

The above error functions range from 0 (no error) to 1 (maximum error). We use $s_u(x, y) = 1 - e_u(x, y)$ and $s_i(x, y) = 1 - e_i(x, y)$ as similarity measures.

Given the potential for information loss when neighborhoods are merged, pairing up vertices in a manner that minimizes such loss during compression is important. We can formulate this subproblem as a maximum-weight matching problem in the complete graph consisting of the vertices of the neighborhood store. In this *auxiliary graph* (which is distinct from the call graph), there is an edge connecting each vertices pair; the weight of this edge is the similarity of the two vertices. A solution of this matching problem yields a pairing of of vertices that results in the minimum loss of information, as measured by the metrics e_u and e_i. Although this reduction to the maximum weight matching problem is conceptually attractive, it suffers from a serious practical drawback. In contrast to bipartite weighted matching, which admits very efficient solutions, nonbipartite weighted matching algorithms run in roughly $O(n^3)$ time, where n denotes the number of vertices [Law76, Gab73]. (Algorithms that improve slightly on this asymptotic complexity exist, but they are complex and unlikely to yield implementations that run faster than the $O(n^3)$ algorithm on typical instances of the problem.)

In addition to the high running time of weighted matching algorithms, these algorithms have $O(n^2)$ space requirements on dense graphs (since every edge must be read). Using such an algorithm to compress a neighborhood would thus require working space that is quadratic in the size of the neighborhood. If we avoid using external memory, then the need for this working storage in RAM effectively reduces the maximum permissible size of a compressed neighborhood quadratically. As a result, the neighborhood store will have to be compressed more aggressively. Since, in general, the greater the compression, the higher the information lost, this option is not attractive. An alternative strategy is to use external memory as working storage for the compression algorithm. However, such an external memory algorithm for weighted matching is orders of magnitudes slower than RAM-based algorithms and will cause a drastic reduction in the rate at which the data stream can be processed. More precisely, using such an algorithm for compression would result in unacceptably long stalls in the stream processing while the neighborhood store is being compressed using external memory. The weighted matching problem admits efficient greedy 1/2-approximation algorithms (guaranteed to give a solution of weight at least half the weight of the optimal solution) [Pre99]. Computationally, this algorithm is simpler than the exact matching algorithm. However, the space requirement is still quadratic in the size of the neighborhood store.

An alternative to the greedy algorithm is a heuristic that pairs a vertex u with the first vertex v (in some order) such that $s_u(u, v) \geq s$ (for some threshold $s \in [0, 1]$) or the vertex most similar to u (if no vertex passes the similarity threshold check). This method has the attractive property of requiring only a small, constant amount of working storage. This method also potentially avoids examining a large number of edges in the auxiliary graph; as a result, its running time can be sub-quadratic. Further, if each vertex is paired with a vertex that passes the similarity threshold check, then the resulting pairing of vertices provides an upper bound on the information lost due to merging vertices (based on the lower bound on the similarity). However, it does not provide any guarantees relative to the optimal solution. Further, the similarity threshold must be carefully chosen in order to avoid quadratic running time. We evaluate these methods in Section 4.

After neighborhoods have been merged in this manner, all the (new) neighborhoods are updated by replacing references to the unmerged vertices with references to the merged vertices (since the unmerged vertices are now indistinguishable based on neighborhoods). This process results in a reduction in the size of neighborhoods because multiple vertices are potentially replaced by the same vertex (representing all of them, and mapping to the common neighborhood).

We now describe a simple improvement to this method (which we have incorporated in our implementation). Since the definition of an ot-matching requires a vertex in common with a fraud clutch, vertices that are at a distance greater than t (the mapping size threshold) from the nearest fraud vertex cannot be part of a match. This information can thus allow us to prune the search process in the nMatch function. We simply maintain with each vertex a record of the distance from the vertex to the nearest fraud vertex. This information can be easily updated as new edges arrive. (For example, if an edge connects a new vertex to an old vertex the new vertex's distance is set to 1; if it connect two new vertices, we update the larger distance to be no more than 1 plus the smaller distance, etc.)

To summarize, when the space used by the neighborhood store exceeds a limit (based on the parameter M in the problem definition), our method compresses it by merging pairs of vertices, thus reducing in half the number of stored neighborhoods. Over time, as new data is added, the neighborhood store will grow and again exceed the storage limit, at which point it is compressed again. Note that in this compression step, the merged neighborhoods generated by the earlier compression get compressed again, while newly added data is only compressed once. As this process continues, with repeated compressions, older data is compressed (exponentially) more than newer data. This bias of representing recent data more accurately than older data is advantageous given the greater importance of detecting matches in recent data. Finally, we do not compress any neighborhoods belonging to fraud clutches. The reason for this decision is twofold: First, since the fraud clutches are fixed (and typically few in number), storing them as is does not impose a large overhead. Second, it is important to represent them accurately because any error in their representation will affect all subsequent operations (unlike errors in the neighborhoods of other vertices, which remain localized).

4 Experimental Evaluation

We have implemented our methods for detecting moving clutches using the JDK 1.3.1 environment. Our experiments were performed on a Pentium III based workstation with 512 MB or RAM, running the Redhat 7.1 distribution of GNU/Linux (kernel version 2.4.2) and build 1.3.1-b24, mixed mode, of the Java HotSpot Server virtual machine.

4.1 Dataset Generation

The streams of edges used as input data for evaluating our method are based on an underlying random graph model, which we now describe: Our data generation program first generates a large random graph. The edges of this graph are ordered randomly and stored in a list which forms the basis for the data stream. In choosing a model for the random graphs in our experiments, our main goal was to make a trade off between a simple model that is easy to interpret and a complex model that better reflects application characteristics.

The literature in graph theory focuses mainly on two random graph models: $G(n,p)$ and $G(n,m)$. In the $G(n,p)$ model, a random graph of n vertices is obtained as follows: Consider K_n, the complete graph of order n. For each of the $n(n-1)/2$ edges of K_n, a biased coin having p as the probability of head is tossed; this edge is in the random graph iff the result of the toss is heads. The $G(n,m)$ model generates graphs of order n and size m assuming that all such graphs are equi-probable. The simplicity of these models permits many elegant results. Unfortunately, neither model is realistic for the applications of interest. In fact, the highly skewed nature of a calling graph and the existence of clutches of vertices is the basis for the problem we study.

Therefore, our random graph model starts by generating clutches of densely connected vertices, distinguishing between fraud and regular clutches. Cross-clutch edges are added with a lower probability than the in-clutch edges. More precisely, a chosen number of fraud clutches is generated using the $G(n,p)$ model. The edge probability p is fixed, while the

Parameter	Typical Values
Number of fraud clutches	10–100
Number of regular clutches	40–400
Minimum number of vertices per clutch	5–15
Maximum number of vertices per clutch	15–25
Intra-clutch edge probability	0.2–0.8
Inter-clutch edge probabilities	0.2–0.8
Minimum number of edges crossing regular clutches	0–3
Maximum number of edges crossing regular clutches	5–10
Minimum number of edges between clutches and regular clutches	0–3
Maximum number of edges between clutches and regular clutches	5–10
Probability that a fraud clutch moves	0.5
Maximum number of vertices moved per clutch	4

Figure 2: Data generation parameters

number of vertices, n, is selected uniformly randomly from a fixed range. Regular clutches are generated in an analogous manner. Recall that we assume that there are no edges connected vertices of different fraud clutches. However, edges that connect two regular clutches or a regular clutch to a fraud clutch do exist. These are inserted between randomly selected qualifying vertices with a fixed probability. The main parameters guiding this generation process are summarized in Figure 2. After the graph is generated, we randomly select some of the fraud clutches for moves. For each selected clutch, the vertices to be moved are selected at random. For each vertex that is to be moved, we generate a new vertex that has the same neighborhood as the original. The last two parameters in Figure 2 control this behavior in the obvious manner. Finally, all the edges in the graph thus obtained are permuted to a random order, which is the order in which they appear in the test streams.

In our experimental setup, the fraud clutches generated by the above method were passed from the data generation phase to the detection phase. The experiments were all run to completion on the input data stream. That is, when numbers are quoted as totals (e.g., the total number of compressions in an experimental trial), they refer to the sum over the entire data stream generated by the above process. In all experiments except those specifically studying variations in the data generation parameters, the same (randomly generated) dataset was used for all trials. Note that our main interest is in the steady-state behavior of our method, with throughput and size of working storage being the most important considerations. Since the data stream is assumed infinite in the applications (e.g., a stream of phone calls), the total length of the stream used in our experiments is not of much significance. However, some properties of the random graph underlying the stream are significant to establish baselines (e.g., in the discussion of errors due to compression below).

For the experiments, described below, related to Figures 3, 4 and 5, we used the following dataset parameters: 20 fraud clutches, 100 regular clutches, clutch sizes in $[5, 20]$ (uniformly random), inter-clutch edge probability 0.5, number of cross-clutch edges in each clutch in $[0, 5]$ (uniformly random). Further, each fraud clutch was moved with probability 0.5. The

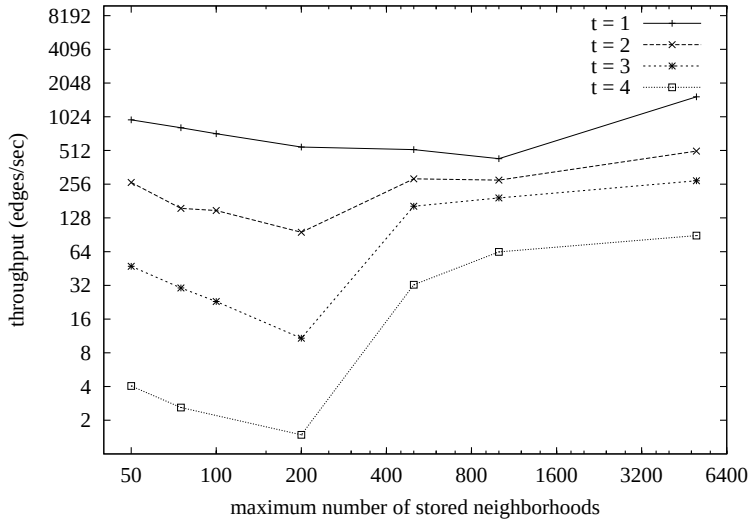

Figure 3: Effect of maximum transition size (t)

number of vertices moved per group was in $[1, 4]$ (uniformly random).

Figure 3 summarizes some results of our experiments studying the effect of two important parameters on the throughput of our method: The first is the maximum number of neighborhoods stored by our method. Recall that by bounding this parameter, we can control the amount of working space required for our method. The second parameter is that maximum size t of a mapping discovered by our method. Recall that our method looks for moved clutches that have at most t vertices different from the known clutch. The four lines suggest the effects of increasing the size of the neighborhood store on throughput for values one through four of t. The throughput is measured as the average number of edges from the data stream processed per second, and the neighborhood store size is measured in the number of neighborhoods stored. Note that both axes of Figure 3 have a logarithmic scale. We note that throughput falls rapidly with increasing values of t. This behavior is to be expected, given the hardness of graph isomorphism.

Figure 3 also indicates that as the neighborhood store is made larger, throughput falls at first, then rises, and finally levels off. This trend is interesting, because at first glance it may appear that increasing the size of the neighborhood store should simply reduce throughput because of the potentially larger amount of data that must be searched when processing an edge. A larger neighborhood store means fewer compressions since the spare capacity created by each compression—half the store size—is larger an thus fills up less often for a given data stream. On the other hand, the task of compression itself is more resource intensive for larger neighborhood stores. Recall that for a neighborhood store of size n, the greedy compression method requires $\Theta(n^2)$ space and $\Theta(n^2 log n)$ time. The reduction in the number of times compression is invoked is, by contrast, only linear. This effect explains the initial reduction in the throughput. Beyond a certain value of neighborhood store size, however, another effect dominates: Larger neighborhood stores can store more discriminating information about neighborhoods of vertices. As a result, our method's search for a match is better guided

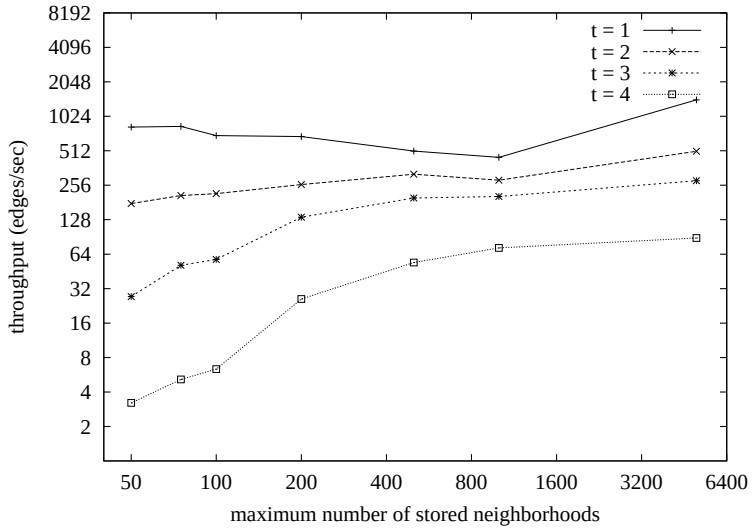

Figure 4: Effect of maximum transition size (t)

(specifically, pruning based on the checks on neighborhoods is more effective). Since the cost of the search dominates other costs in our methods, this effect is ultimately the dominant one as neighborhood store size is increased. This reason is also why the curves level off: Once the neighborhood store is large enough to hold the entire working set, further increases in size yield no benefit. Finally, this effect is also the reason why the shape of the curve is more pronounced at higher values of the parameter t (maximum mapping size): The cost of search (as a fraction of the total cost of the method) rises rapidly with rising t, making less important all factors other than those directly affecting the cost of the search.

Figure 4 presents results very similar to those in Figure 3. Both experiments used the same datasets and methods except that in the latter experiment, neighborhoods were merged using the intersection method instead of the union method used in the former. We note that the general trend with regards to increasing t (maximum mapping size) values similar to that in Figure 3: As t increases, throughput falls rapidly. The effect of increasing neighborhood store sizes on throughput is, however, different in an interesting way: We note that the early dips in throughput with increasing neighborhood store size are much less pronounced or absent in Figure 4. This difference is explained by the fact that the search pruning effect discussed earlier is much more pronounced when the merged neighborhoods are intersections of the originals. When neighborhoods are merged, they often end up with far fewer vertices than the originals. As a result, when the nMatch function (Section 3) searches for potential matches for neighborhoods in the fraud clutches, the search terminates quickly. (Recall that fraud clutches are not compressed, so there is no reduction in the vertices that must be matched in the fraud clutches.) As an aside we note that, although not quantified in the figure, using the intersection method of merging neighborhoods has another advantage: since intersection shrinks the size of neighborhoods, the total storage used, measured in bytes, is likely to be lower than that used by a neighborhood store that uses the union method of merging.

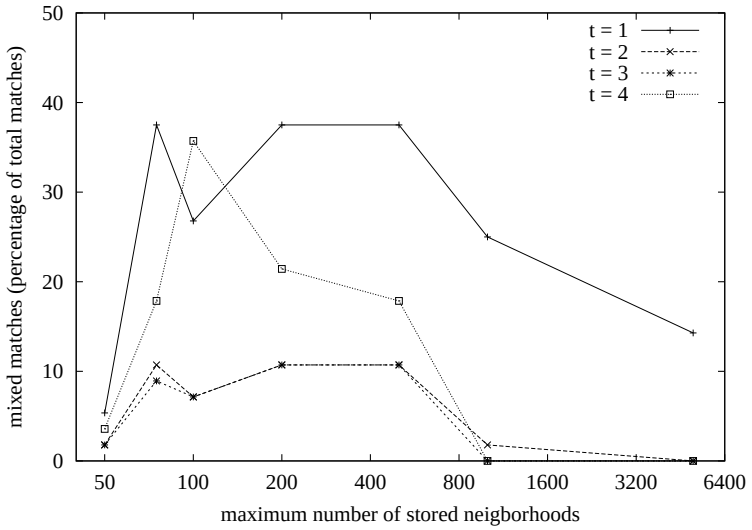

Figure 5: Missed matches due to intersection compression loss

The benefits of the intersection method of merging neighborhoods comes at the cost of potentially missing matches. (In contrast, the union method does not miss any matches, but may generate spurious matches.) Figure 5 illustrates these misses for the same set of experiments described in Figure 4. The horizontal axis is identical to that in Figure 4 and the vertical axis plots the percentage of the generated moves that went undetected by our method, for each of four values of the parameter t (maximum mapping size, which corresponds to the maximum number of simultaneously moving vertices that are detectable). Recall that the dataset includes moved clutches in which as many as four vertices move at a time. Therefore, it is not surprising that there is a significant percentage of misses for $t < 4$. What is perhaps surprising is that there is a high number of misses for $t = 4$, especially at high compression levels (small neighborhood store sizes). This phenomenon is caused by the neighborhoods of the moved vertices being compressed and losing critical edges. Recall that we do not compress the (known) neighborhoods that participate in fraud clutches. However, this courtesy cannot be extended to the moved vertices, since we don't know the identity of those vertices (it being part of the problem we seek to solve).

Figure 6 summarizes the results of some experiments studying the two heuristics used for compressing the neighborhood store. (Both axes have a logarithmic scale.) We observe that the threshold heuristic does not perform very well, overall, compared with the greedy heuristic. It's performance depends strongly on how often the threshold for similarity of neighborhoods is met (thereby speeding up the matching process), which in turn depends on the threshold (in addition to the data). For these experiments, the threshold was set to -0.5, which is aggressive (since it means we cut short the search for a similar neighborhood when we find one that differs by no more than 50%. (The dataset for this experiment is the same one used for those in Figures 3 and 4.) If the threshold is not met in a large number of cases, the threshold heuristic is less efficient than the greedy because it makes a greater number of neighborhood lookups.

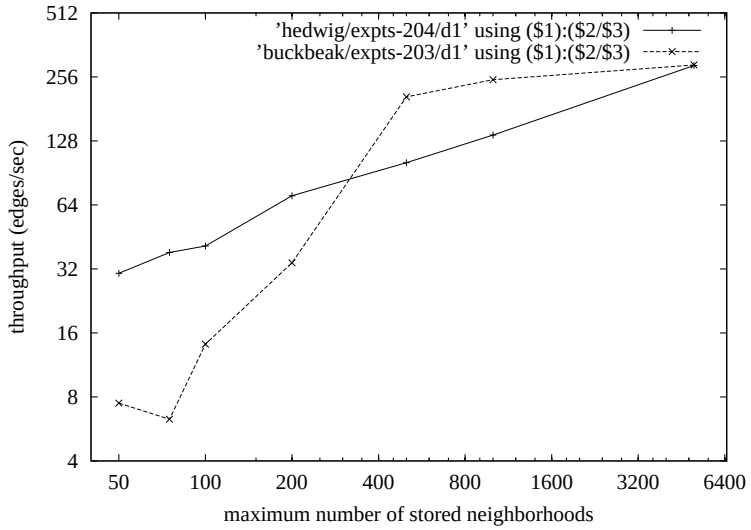

Figure 6: Effect of neighborhood store compression heuristics

5 Related Work

The problem studied in this paper is influenced by the work of Cortes, Pregibon, and Volinsky on *Communities of Interest*, based on experiences with call-graph data at AT&T [CPV01, CP01]. Those paper describe in detail the characteristics of the call graph data and applications, data structures, methods for it. In particular, the idea of "guilt by association" is similar to our definition of fraud clutches. The focus of that work is efficiently detecting fraudulent patterns over very large datasets. In contrast, our focus in this paper has been on keeping track of groups of fraudulent users after they have been detected. The two problems are thus complementary. These papers describe a method for maintaining a (relatively) small history of the call graph by adding a new graph to the weighted-down version of the current graph. Thus the effect of an edge decreases exponentially with time. This property is our method for compressing the neighborhood store, although the particulars are very different. The method of [CPV01] is useful for compressing the sequence of changing weights on edges of a known graph structure. Our neighborhood compression method is based on compressing an unknown graph as it streams in. Their method has been tested with datasets that are several orders of magnitude larger than those in our experiments. On the other hand, our method has the ability to work with the given amount of storage by using a lossy representation scheme.

As described in Section 3, maximum weight matching is a subproblem of our method. Related work on the problem includes classic algorithms [Law76] and more recent work on efficient approximation algorithms [Pre99]. A fast (but $O(n^3)$) implementation of nonbipartite weighted matching is the *wmatch* program by Rothberg [Rot], based on Gabow's thesis [Gab73].

Our problem formulation is closely related to the graph isomorphism problem, on which there is large amount of work. The *nauty* program is a fast program for computing auto-

morphism groups, based on the methods in [McK81].

The streaming data model has received much attention recently in the context of sensor data [MF02], continuous queries [BW01, CDTW00], and query processing [DGGR02]. In [GMMO00], the authors present a streaming algorithm for clustering datapoints with guarantees on the quality of the clustering of points seen so far.

The general idea of using a succinct summary of a graph for various purposes has a large body of work associated with it. For example, this idea is developed in semistructured databases as graph schemas, representative objects, and data guides, which are used for constraint enforcement, query optimization, and query-by-example interfaces [BDFS97, NUWC97, GW97]. There is related work in mathematics on the topic of graph minors, which are obtained from a graph by contracting and deleting some edges [Bol98]. It should be interesting to investigate if some of these ideas can be adapted to this problem.

6 Conclusion

We described the problem of tracking groups (clutches) of agents as they move in a network of hosts. We presented our motivation for this problem based on applications to detecting fraud in telephone networks, detecting distributed denial-of-service attacks in computer networks, and illegal banking activities. We formalized this problem using the idea of a restricted graph isomorphism with the requirement of a common vertex between matched graphs. We presented a method that uses these restrictions to yield a method for detecting moving clutches in streaming graph data. Our method can cope with a small amount of working storage, and can dynamically adapt to changes in the available storage. It is based on using a lossy compression method to store a large graph in a given amount of space. We presented an experimental evaluation of our methods. As continuing work, we are exploring alternate definitions of the problem. In particular, we are working on generalizing our definition of a moving clutch of agents. We also plan to explore links with work in other fields on the general problem of lossy compression of graphs.

References

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, 1997.

[Bel89] S.M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[Bol98] Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, New York, 1998.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, September 2001.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[CP01] Corinna Cortes and Daryl Pregibon. Signature-based methods for data streams. *Data Mining and Knowledge Discovery*, 5:167–182, 2001.

[CPV01] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. Communities of interest. In *Fourth International Symposium on Intelligent Data Analysis (IDA 2001)*, Lisbon, Portugal, 2001.

[DGGR02] A. Dobra, M. Garofalakis, J. E. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 2002. To appear.

[Gab73] Hal Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University, 1973.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Company, November 1990.

[GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, November 2000.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, Athens, Greece, 1997.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[McK81] Brendan McKay. Practical graph isomorphism. *Congressus Numerantim*, 30:45–87, 1981.

[MF02] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the International Conference on Data Engineering*, 2002.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of the International Conference on Data Engineering*, pages 79–90, 1997.

[Pre99] Robert Preiss. Linear time $\frac{1}{2}$-approximation algorithm for maximum weighted matching in general graphs. In *STACS*, number 1563 in LNCS, page 248, March 1999.

[Rot] E. Rothberg. The *wmatch* program for finding a maximum-weight matching for undirected graphs. Live OR collection. Available at `http://www.orsoc.org.uk/home.html`.