Updating an Existing Social Network Snapshot via a Limited API

Norases Vesdapunt, Hector Garcia-Molina

Computer Science Department, Stanford University
{norases, hector}@cs.stanford.edu

January 2016

Abstract

We study the problem of graph tracking with limited information. In this paper, we focus on updating a social network snapshot. Say we have an existing partial snapshot of a network $G_1$. Over time $G_1$ becomes out of date. We want to update $G_1$ through a public API, restricted by the number of API calls allowed. Periodically recrawling every node in the snapshot is prohibitively expensive. We propose a scheme where we exploit indegrees and outdegrees to discover changes to the graph. We probe the graph and verify edges when there is ambiguity. We propose a novel strategy designed for limited information that can be adapted to different levels of staleness. We evaluate our strategy against recrawling on real datasets and show that it saves an order of magnitude of API calls while introducing minimal errors.

1 Introduction

A social system like Google+ or Twitter lets users interact and share resources such as photos and news articles. At the core of a social system is its social graph or network. A node in this graph represents a user (a human or sometimes an entity like a club or a corporation), while the links represent relationships among users (e.g., user $x$ is a friend of user $y$). Each node contains information about the user (sometimes called profile information), such as the name of the user and his or her interests.

A user $U$ may form or break relationships over time and in many cases it is useful to update the information on $U$. For example, say we have an existing partial snapshot of a network, call it $G_1$. Snapshot $G_1$ may contain the data crawled when doing entity resolution (ER) for a group of nodes. We would like to continue using $G_1$ for various tasks (e.g., more ER), but over time $G_1$ becomes out of date. So we want an updated version of $G_1$. One option for getting a new snapshot $G_2$ is to simply re-crawl the target network, but that is expensive
and time consuming due to API restrictions (See Section 2.1). Instead, can we simply “update” $G_1$ to make it up to date? For example, we can revisit the nodes in $G_1$ and see if the number of friends has changed. If it has not changed, then we assume its neighborhood has not changed. If the number is different, then we explore. We may also try to update the “more important nodes” in $G_1$ (maybe the ones with more friends?). The updated snapshot, call it $G_2$, will be less accurate than $G_2$ (done with a full re-crawl), but maybe it will be good enough? (Even $G_2$ is out of date by the time we finish the re-crawl.) We would have to define a metric to evaluate how much two snapshots differ, and then we could evaluate different ways to update $G_1$.

Clearly, there is huge interest in analyzing social graphs, to discover people’s preferences and trends (both influencers and content), predict social behavior, and make insightful recommendations (both products on the site and ad targeting). If the analysis is done by the organization that owns the data (e.g., Google, Facebook), then our approach is not needed. On the other hand, there are many cases where organizations need to study graphs generated by others. These include governments trying to protect their citizens, companies tracking customers and competitors, and data science researchers. Crawling the necessary graphs, and keeping them up to date, is a massive task and very expensive. Our techniques can provide very significant savings.

We believe that even if analysis of crawled graphs was only of interest to scientists, that would still be enough motivation for our work. But interest is much more widespread. We mention several commercial applications that need up-to-date social snapshots. In each application area, we list in parentheses some companies in the area that are benefiting from social graph crawling and analysis: 1) gathering a fuller understanding of a person of interest or candidate for journalists, recruiters, and people search/finders (Entelo, Spokeo, PeekYou), 2) identifying new interests of the customer for online marketing purposes (Walmart Labs), 3) monitoring online social activities for CRM applications (Nimble, Batchbook, and Salesforce), 4) real-time global analytics (social trends at Kensho), 5) computing accurate trust scores for peer-to-peer networks (Teapot Inc, Troo.ly), and 6) computing credit scores using social networks (Affirm, ZestFinance).

The summary of our contributions is as follows:
• We formulate the problem of updating a social network snapshot with limited resources (Section 2).
• We propose a new technique that exploits the fact that collecting indegrees and outdegrees is inexpensive. Furthermore, we propose multiple verification schemes with different accuracy-cost tradeoffs (Section 4).
• We propose a novel holistic algorithm that consists of 3 stages (S1-S3): S1 remove edges using a competing graph, S2 remove leftover edges using a list of remove actions, and S3 add edges using a list of add actions (Section 5).
• We propose evaluation metrics to compare our updated snapshot against the actual base graph at time 2 (Section 6).
• We evaluate our algorithm variations using various real-world datasets, and compare them to a full re-crawl approach (Section 7).
1. **names**: fullname + other names extracted from the profile
2. **id**: uniquely identifies a user and cannot be changed
3. **locations**: current place + places extracted from the profile
4. **descriptions**: {introduction, bragging_rights, tag-line, job_title, universities, organizations}
5. **urls**: urls from “other profiles” attribute + urls extracted from profile descriptions.
6. **screen_names**: possible screen names extracted from urls.
7. **posts**: up to 10 most recent posts posted by the user

### Graph

1. **in**: set of Google+ users who follow g (i.e., g’s inlinks). Google+ refers to this attribute as “have him/her in circles.”
2. **out**: set of Google+ users that g follows (i.e., g’s outlinks). Google+ refers to this attribute as “in his/her circles.”
3. **indegree**: \(|in|\). Takes value 0 when either a user has no followers in Google+ or the information is private.
4. **outdegree**: \(|out|\), similar to indegree

<table>
<thead>
<tr>
<th>Table 1: Google+ Attributes [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g)</td>
</tr>
<tr>
<td>(h)</td>
</tr>
</tbody>
</table>

**Figure 1**: Directed Relationship (g, h)

## 2 Preliminaries

Let us first define the “base” graph \(G^B_t\). \(G^B_t\) is the directed social graph at time \(t\) stored at the company who owns the social network such as Google+ or Twitter. Recall that a node in this graph represents a user (a human or sometimes an entity like a club or a corporation), while the links represent relationships among users (e.g., user x is a friend of user y).

We will use \(G^B\) to refer the base graph when we do not care about time. We will use lowercase letters (e.g., f, g, h) to refer to nodes in \(G^B\). Each node contains information about the user (sometimes called profile *attributes*), such as the name of the user and his or her interests. Note that in some systems (e.g., Twitter) there are two ways to identify a node: through a screen name (e.g., “sergeybrinn”) or through the actual user id (e.g., 50449264). As long as the API lets us reach the node in question, we will consider either type of identifier to be the node’s identity. We will use the notation \((g, h)\) to represent a link from g to h (as shown in Figure 1). We can view this link as g’s outlink or h’s inlink. The node attributes include indegree (outdegree): the number of inlinks (outlinks).

We will use dot notation to refer to node attributes. For instance \(g\.name\) is
the name of the user represented by \( g \). Some of the attributes contain a scalar text string or number, while some contain sets. For example, \( g.\text{out} \) is the set of \( GB \) identifiers for the users that \( g \) follows while \( g.\text{in} \) is the set of \( GB \) identifiers for the users that follow \( g \). Note that the \( \text{in} \) and \( \text{out} \) attributes encode the \( GB \) graph structure. Figure 1 shows the following relationship: \( g \) is following \( h \) (\( h \in g.\text{out} \)) and \( g \) is \( h \)'s follower (\( g \in h.\text{in} \)). In this case, \( g \) consumes the information feed produced by \( h \). In Google+, nodes have the profile attributes shown in Table 1.

Next we define a snapshot \( G_t \). A snapshot contains a subset of nodes and edges in \( GB \) at time \( t \). Snapshots also have attributes and we will use the same notation to refer to nodes and attributes as in the base graph. All of the nodes in the snapshot contain attributes \( \text{id}, \text{indegree}, \) and \( \text{outdegree} \). Some nodes in the snapshot also contain \( \text{in} \) and \( \text{out} \) (let us call these crawled nodes). Uncrawled nodes are nodes that we have yet crawled (only have \( \text{indegree} \) and \( \text{outdegree} \)). In other applications, one may collect other attributes and store them in the snapshot.

We do not explain in detail here how snapshots are obtained. Just briefly, one starts with a set of seed nodes, and as those are visited, other nodes are discovered. For each node (seed or discovered) we get all of its data using \( \text{crawl\_node}(x,G_t) \) (Subroutine 1). A snapshot may end up being different from the original graph because there is a time delay in collecting the data. For simplicity in our examples we assume the snapshot is an exact copy of the base graph (but of course this assumption is not needed by our algorithm).

### 2.1 API Operations

Social networks usually offer common operations to collect information about the network. Relevant API operations to our problem are: lookup, inlinks, and outlinks. The notation for API calls is \text{operation(user\_id)}.

- **lookup(user\_id)**: returns a user profile given a user\_id. A user profile usually includes name, indegree, and outdegree of the user. User profiles can also be easily scraped for most networks (Google+, Twitter, and Instagram).
- **inlinks(user\_id)**: returns user\_ids of inlinks of a given user\_id.
- **outlinks(user\_id)**: returns user\_ids of outlinks of a given user\_id.
- **friendships(user\_id1, user\_id2)**: returns relationships (in, out) between 2 given user\_ids.

Social networks are often more strict on asking for edges than looking up user profiles. For example, on Twitter, one can look up 72,000 user profiles per hour (via API operation \text{lookup}) whereas one can only make 60 API calls per hour to collect edges (via API operation \text{inlinks} and \text{outlinks}) and make 720 API calls per hour to collect relationships (via API operation \text{friendships}) [2]. It is worth noting that asking for a relationship (edge between two users) is still 12 times cheaper than collecting all of the edges for a user. For more information on API operations and limits, see our technical report [13].
Subroutine 1 crawl_node(x, G_t)

Input: a node to crawl x, a snapshot graph to update in-place G_t

1: x.in ← inlinks(x.id)
2: x.out ← outlinks(x.id)
3: Add x, x.in, x.out to G_t (add nodes and edges)

2.2 Problem Definition

Say we have collected a snapshot G_1 of the base graph G_B^1. Let G_B^2 be the base graph at time 2 (time 2 > time 1). We would like to update G_1 to G_2 (may be less accurate than a full re-crawl of G_2) by efficiently invoking API calls. To obtain G_2, we are given the budget B = [b_lookup, b_in, b_out, b_friendships] where b_lookup, b_in, b_out, and b_friendships are the budget values (units are the number of API calls) on lookup, inlinks, outlinks, and friendships calls respectively.

3 Examples

![Figure 2: Single Edge Deletion](image)

In this section we provide a high-level intuition of our solution using examples. Let us walk through Figure 2 in detail. At time 1 we retrieved g, h, i’s inlinks and outlinks and saved them in the snapshot G_1. At time 2, we would like to update G_1. Let G_2[a] in Figure 2 be the actual G_2 (snapshot at time 2) if we were to re-crawl g, h, and i. Let G_2[e] be our estimate for G_2.

Next we will explain G_2[s] in Figure 2. Recall from Section 2.1 that social networks usually restrict API operation lookup more (i.e., using the terms introduced in Section 2.2, b_lookup is usually >> b_in and b_out). Therefore, we will exploit this fact and first collect a sketch of all nodes in our snapshot G_1. Let us define a sketch of a node g as \{g.indegree, g.outdegree\}. We construct the sketch G_2[s] by invoking inexpensive lookup(x.id) calls for all x ∈ G_1 (we use a in G_2[s] to denote this intermediate step).

Since we do not have the actual G_2[a] (shown in Figure 2), we try to estimate it based on the G_2[s] sketch. We obtain our initial estimate for G_2[e] (we will continue to use G_2 to denote G_2[e]) by copying G_1 (i.e., G_2 ← G_1). Let
x.a, denote the value of attribute a of node x in the snapshot G at time t. Let a conflict be a discrepancy between an indegree (outdegree) count in the sketch G_2[s] and the inlinks (outlinks) in our guess G_2. In Figure 2, there are two conflicts. In G_2[s], h.indegree = 1 whereas h has two inlinks in our initial guess G_2 (a copy of G_1). Another discrepancy is that in G_2[s], i.outdegree = 0 whereas i has one outlink in our initial guess G_2. Therefore, both inlinks of h and outlinks of i have changed since time 1. Since i.outdegree = 0, i has no outlink at time 2 (i.out_2 = {}). Therefore, we can infer that the edge (i, h) has been deleted. After removing the edge from G_2, we no longer have a conflict between the indegree count and the inlinks in G_2 (i.e., h.indegree = 1 and h.in_2 = {g}). Thus, in this case, we can infer G_2[a] using the sketch G_2[s] without making any expensive inlinks or outlinks API calls.

Note that this inference makes an assumption that a configuration that requires more edge creations and deletions are unlikely and we only want to find a solution with the smallest number of edge changes from G_1. For example, the rightmost configuration in Figure 2 is another configuration G'_2 that also matches the counts in G_2[s]. However, this configuration requires three more edge changes to the graph than the configuration G_2[a]. The edge changes in this configuration are the creations of the edges (g, j) and (j, h) and the deletion of the edge (g, h). We can argue that this configuration is less likely because it requires more edge changes.

Note that there are infinite configurations that match G_2 if we allow node and edge creations because we can keep adding more nodes and edges to the chain. For example, adding a node k and adding edges to create a chain g → j → k → h will create another valid configuration for G_2[s]. Therefore, as we look for our G_2[e] estimates, we restrict ourselves to the configurations with fewest number of edge changes. In some cases, the solution is unique and we can make all of the inferences. In other cases, there are multiple configurations that are valid (i.e., can explain our sketch G_2[s]) and minimal (i.e., have the smallest number of edge changes). To decide which of these configurations is more accurate, we have to crawl the actual network.

Next we will formally define valid and minimal.

**Definition 1** Given a snapshot G and a sketch G[s], we say that G is valid if it matches the counts in G[s]. For example, in Figure 2, G_2[a] is a valid snapshot for G_2[s] whereas G_1 is not. We will use the notation (G, G[s]) to denote the snapshot G and the sketch G[s] pair.

**Definition 2** Given two snapshots G_1 and G_2, dist(G_2, G_1) denotes the minimum number of edge changes to transform G_1 to G_2. For example, in Figure 2, dist(G_2[a], G_1) = 1 because removing the edge h → i in G_1 results in G_2[a].

**Definition 3** Given a previous snapshot G_1, a snapshot G_2, and a sketch G_2[s], we say that G_2 is the minimal configuration for G_1 and G_2[s] if and only if (G_2, G_2[s]) is valid and there is no other valid (G'_2, G_2[s]) such that dist(G_2, G_1) > dist(G'_2, G_1). In Figure 2, dist(G_2[a], G_1) = 1. On the
other hand, \( \text{dist}(G'_2, G_1) = 4 \). Therefore, \((G_2[a], G_1, G_2[s])\) is minimal whereas \((G'_2, G_1, G_2[s])\) is not.

![Figure 3: Multiple Edge Creations](image)

Next we will look at another example for when there are multiple minimal solutions satisfying the counts in \(G_2[s]\). Figure 3 shows that \(j\)’s indegree stays the same for both time 1 and time 2. Furthermore, \(g\)’s, \(h\)’s, and \(i\)’s indegrees and outdegrees have changed. Both \(G_2[a]\) and \(G'_2[a]\) in Figure 3 are minimal configurations that explain \(G_2[s]\). The edge changes for \(G_2[a]\) are the creations of the edges \((g, h), (h, i),\) and \((i, g)\). The edge changes for \(G'_2[a]\) are the creations of the edges \((g, i), (i, h),\) and \((h, g)\). Both configurations require three edge changes (three edge creations).

Suppose we initialize \(G_2\) by copying \(G_1\) as before. Then after we deduce using the \(G_2[s]\) sketch, we have two minimal solutions. One of the options we have is to verify. Verify means to invoke an API call to check whether an edge exists. We will explain later how to pick an edge to verify. As an illustration of verifying, let’s retrieve \(g\)’s outlinks (i.e., invoking `outlinks(g.id)` call). Say `outlinks(g.id)` returns \(\{h\}\) (i.e., \(g\)\textunderscore out\(_2\) = \(\{h\}\)). We can deduce that \(G'_2[a]\) is invalid because in that case, \(g\)\textunderscore out\(_2\) = \(\{i\}\). Therefore, we can prune that solution and that leaves us with \(G_2[a]\) in Figure 3 as our estimate \(G_2[e]\). We arrive at our final \(G_2[e]\) by inferring the creations of the edges \((h, i)\) and \((i, g)\) from the \(G_2[s]\) sketch.

From the two examples above, we learned that we may be able to infer some edges and avoid asking for inlinks and outlinks of all of the nodes. Furthermore, in some cases, there is more than one configuration that satisfies the counts in our sketch. As we crawl and update our snapshot, we can prune invalid configurations.

In general, we find discrepancies between the current guess \(G_2\) and \(G_2[s]\), make inferences, and decide to make API calls to crawl nodes in \(G_2\) efficiently (e.g., verify edges that help us prune invalid solutions). After each API call, we update \(G_2\) and repeat the previous step again until we have reached the API budget \(B\).

Next we discuss the number of possible configurations for a graph \(G\) with \(n\) nodes. Figure 4 shows that for each node pair \((g, h)\), there are four possible configurations. And for \(n\) nodes, there are \(\binom{n}{2} = \frac{n(n-1)}{2}\) node pairs. Therefore,
there are \(4^{\frac{n(n-1)}{2}} = 2^{n(n-1)}\) possible configurations for a graph \(G\) with \(n\) nodes. Therefore, the number of possible configurations grow exponentially as \(n\) grows.

However, most of these configurations are invalid (i.e., do not match with the counts in the sketch). If each node has outdegree \(\leq d\), we can compute a tighter bound for the number of valid configurations as follows. For each node \(x\), there are \(\binom{n-1}{d} = \frac{(n-1)(n-2)\ldots(n-d)}{d!}\) possible ways of selecting \(d\) outlinks from \(x\). Therefore, there are \(\left[\frac{(n-1)(n-2)\ldots(n-d)}{d!}\right]^n = O\left(\left[\frac{n^d}{d!}\right]^n\right) = O(n^{dn})\) valid configurations for a graph \(G\) with \(n\) nodes with outdegree at most \(d\).

This means enumerating all valid configurations is prohibitively expensive. Therefore, we will develop algorithms that exploit some heuristics and refrain from evaluating all configurations.

4 Strategy

Our strategy relies on the assumption that fewer changes are more likely. That is, our initial guess \(G_2\) is that the snapshot remains unchanged and then we gradually fix the snapshot until our configuration is valid. As mentioned in Section 3, as we refine \(G_2\) we may encounter an ambiguous situation, in which case we can either make an educated guess or we can verify by probing the base graph. The verification option is discussed in Section 4.2.1. Our strategy consists of three steps: 1) list actions, 2) remove edges, and 3) add edges.

4.1 List Actions

The first step is listing actions required to resolve the conflicts between our initial guess \(G_2\) and the sketch \(G_2[s]\). Let us use Figure 5, as our running example. We first explain our notation for actions. The actions are in the form of “operation \(A \rightarrow B\)”. Operations are add and remove. \(A\) and \(B\) are sets of nodes. The action “operation \(A \rightarrow B\)” means we need to perform operation for the edge \(a \rightarrow b\) where \(a \in A\) and \(b \in B\). For example, “remove \(\{g, h\} \rightarrow i\)” means we have to remove the edge \(g \rightarrow i\) or \(h \rightarrow i\). We also use * to denote a wildcard node which can be any node in \(G_1\) or a new node added after time 1. As shown in Figure 6(a), the actions are “remove \(\{g, h\} \rightarrow i\)”, “remove \(g \rightarrow \{i, j\}\)”, “remove \(h \rightarrow \{i, j\}\)”, and “add \(i \rightarrow *\)”.

Figure 4: Possible Configurations for a node pair \(\{g, h\}\)
Figure 5: Running Example

- remove $\{g, h\} \rightarrow i$
- remove $g \rightarrow j$
- add $i \rightarrow *$
- add $i \rightarrow j$
- add $i \rightarrow *$

Figure 6: Steps for Our Strategy

We generate the remove action “remove $\{g, h\} \rightarrow i$” from comparing the indegree discrepancy of node $i$ between $G_1$ and $G_2[s]$ in Figure 5. $i$ has indegree of 2 in $G_1$ but decreases to 1 in $G_2[s]$. Therefore, either $h$ or $g$ must have stopped following $i$ at time 2.

Similarly, we generate the add action “add $i \rightarrow *$” from comparing the outdegree discrepancy of node $i$ between $G_2$ and $G_2[s]$ in Figure 5. $i$ has outdegree of 0 in $G_2$ but increases to 1 in $G_2[s]$. Therefore, $i$ must have started following some node at time 2. Note that add actions always contain a * on either the left or the right hand side. This is because we do not know which node a particular node is connected to (can be any existing node or even a new node) unlike the remove actions where we know the set of nodes connecting to a particular node from the previous snapshot.

The 4 actions listed in Figure 6(a) are the ones we can observe from the degree discrepancies between $G_2$ and $G_2[s]$. Note that we may uncover more
actions later (will discuss in Section 4.2.1).

In our example (Figure 6), each action is distinct (no duplicate action). However, we may have copies of the same action more than once. When two actions are identical, instead of having two actions, we will add a count field to denote the number of copies. Say or example that \( i \) has outdegree of 3 (i.e., \( i.\text{out} = \{g, h, j\} \) ) in \( G_1 \) but decreases to 1 in \( G_2 \). Therefore, \( i \) must have stopped following two out of the three nodes \( \{g, h, j\} \) at time 2. We generate the remove action “remove \( i \rightarrow \{g, h, j\} \) [2]” where [2] indicates the count. When the count is 1 in our examples, we omit the [1]. For simplicity, we will continue with examples with the count of 1. We will explain how to deal with counts in Section 4.4.

Later when we discuss our holistic strategy in Section 5, we will generate a list for each type of action (one for remove, one for add). Our holistic strategy consists of multiple stages and each stage will focus on one type of action.

### 4.2 Remove Edges

The second step is removing particular edges from \( G_2 \) based on the remove actions listed on the previous step. We first delete the matching remove action pairs. Let us define matching.

**Definition 4** Two actions “remove \( A \rightarrow B \)” and “remove \( C \rightarrow D \)” (where \( A, B, C, D \) are sets of nodes) are matching when there exists \( x \rightarrow y \) such that \( x \in A \cap C \text{ and } y \in B \cap D \). For example, in Figure 6(a), “remove \( \{g, h\} \rightarrow i \)” and “remove \( g \rightarrow \{i, j\} \)” are matching because there is a common edge \( g \rightarrow i \) that satisfies both actions. Therefore, by removing this common edge, we can complete two actions on the list.

**Definition 5** Two matching action pairs \( P \) (actions \( a_1, a_2 \)) and \( Q \) (actions \( a_3, a_4 \)) are competing when \( \{a_1, a_2\} \cap \{a_3, a_4\} \neq \emptyset \). We will discuss an example of competing matching pairs next.

Continuing our example in Figure 5 and our list of actions in Figure 6(a), we have two competing matching pairs: 1) “remove \( \{g, h\} \rightarrow i \)” and “remove \( g \rightarrow \{i, j\} \)” (common edge is \( g \rightarrow i \)), and 2) “remove \( \{g, h\} \rightarrow i \)” and “remove \( h \rightarrow \{i, j\} \)” (common edge is \( h \rightarrow i \)). In this case, there is an ambiguity where an inlink of \( i \) was deleted between time 1 and 2 (either \( g \rightarrow i \) or \( h \rightarrow i \)).

1: remove \( \{g, h\} \rightarrow i \)
2: remove \( g \rightarrow \{i, j\} \)
3: remove \( h \rightarrow \{i, j\} \)

**Figure 7:** Competing Graph for remove actions from Figure 6(a)
Next we will discuss how to come up with a reasonable guess for $G_2$ when there are ambiguities. We can build an undirected graph of competing matching pairs (let us call this the \textit{competing graph}) as follows: \textbf{1)} list all possible matching pairs \textbf{2)} for each action pair $(a_x, a_y)$, draw edges to all other action pairs containing action $a_x$ or $a_y$. From our example (Figure 7), we have two matching pairs $(1, 2)$ and $(1, 3)$. And because action 1 is common in both pairs, we create an edge between these pairs. For this simple case, choosing either pair will allow us to match one pair.

Next we will formally define \textit{competing graph}.

\begin{definition}
Given a list of matching pairs $V = \{v_1, v_2, ..., v_n\}$, the competing graph is $C(V, E)$, an undirected graph on $V$ where an edge $(v_i, v_j)$ exists iff $v_i$ and $v_j$ are competing.
\end{definition}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure8}
\caption{Bipartite Competing Graph}
\end{figure}

Let us look at another example where we have 3 matching pairs and the competing graph as shown in Figure 8. If we choose, the pair $(1, 2)$, we cannot choose $(1, 3)$ or $(2, 4)$. Therefore, it is easy to see that to maximize the number of pairs we match, we should choose $(1, 3)$ and $(2, 4)$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure9}
\caption{Competing Graph with an Odd Cycle}
\end{figure}

From the example above, we can formally define our current problem: given a competing graph, find a maximum node set such that no two nodes in the set are adjacent. Recall that for each matching action pair (node), we only need to perform one action to satisfy the pair. Thus, for each node we select, we save an action. The more nodes we select, the more actions we save.

Our problem is equivalent to the maximum independent set problem in graph theory and it is NP-hard. If a graph is bipartite, maximum independent sets
can be found in polynomial time (using K鰊ig’s theorem). A graph is bipartite if an only if it does not contain an odd cycle. Next we will give an example of a competing graph with odd cycle. Figure 9 shows a competing graph that has a cycle of size 3. Note that this is a simple example whereas in practice, a competing graph can have multiple overlapping cycles. Therefore, competing graphs may not be partite and our problem is NP-hard.

Next we will discuss our algorithm that exploits verification (Algorithm 2). We will first define lemmas that our algorithm will use.

**Lemma 1** Nodes with degree 0 are independent of other nodes so they should all be added to our solution and can be removed from our graph.

**Lemma 2** It is optimal to choose nodes with degree 1 to be in our solution first [8].

**Proof** Say we choose a node $x$ with degree 1 and let $x$ connect to some node $y$ (i.e., an edge $(x, y)$ is in our graph). Let $y$’s degree be $d$. Let the cost be -1 when we select a node to be in our solution (i.e., select the node to perform the action pair) and +1 when we remove the node from the graph. By choosing $x$, the total cost is 0 because we select $x$ (-1) and remove $y$ (+1). On the other hand, if we choose $y$, the total cost is greater than or equal to 0 because we select $y$ (-1) and remove $x$ (+1). We can easily see that $y$’s degree is at least 1 because $y$ is connected to $x$. If $y$’s degree is $d$ (greater than 1), we have to remove $d - 1$ nodes besides node $x$ and the total cost will be $d - 1 \geq 0$. Therefore, choosing $x$ (cost 0), is at least as good at choosing $y$ (cost $\geq 0$). □

**4.2.1 Verification**

As discussed in Section 2, we can optionally eliminate ambiguities by probing the base graph (if our API budget permits it). To illustrate, let us return to our example in Figure 5, our list of actions in Figure 6(a), and the competing graph in Figure 7. We have two matching pairs (1, 2) (common edge is $g \rightarrow i$) and (1, 3) (common edge is $h \rightarrow i$). There is an ambiguity where an inlink of $i$ was deleted between time 1 and 2 (either $g \rightarrow i$ or $h \rightarrow i$). Therefore, we can probe the base graph (i.e., invoke an `inlinks(i.id)` call) to verify which pair is correct. Let the actual snapshot at time 2 be $G_2[a]$ in Figure 5. Then `inlinks(i.id)` will return {$g$}. In this case, we will remove $h \rightarrow i$ and delete the second action pair. Furthermore, as shown in Figure 6(b), we can reduce the action “remove $g \rightarrow \{i, j\}$” to “remove $g \rightarrow j$” after learning that $g \rightarrow i$ exists at time 2.

For the remaining unmatched edge remove actions, we need to insert an edge add action so that our solution is consistent with the sketch. In this case, we have one remove action left: “remove $g \rightarrow j$” (shown in Figure 6(b)). After removing “$g \rightarrow j$”, we delete the action from the list and insert the action “add $* \rightarrow j$” (Figure 6(c)). We need to insert this action because the indegree of node $j$ in $G_2$ in Figure 6(c) is different from in $G_2[s]$ in Figure 5. $j$ has indegree of 1 in $G_2$ but increases to 2 in $G_2[s]$. 

12
In another example when we have a cycle in Figure 9, we can use verification as a heuristic to break cycles in a way that also reduced the divergence with the real graph. In this case, if we verify that $g \rightarrow j$ has been removed from the real graph, we can delete the competing pairs $(1, 3)$ and $(1, 4)$.

### 4.2.2 Algorithm for Removing Edges

Next we explain Algorithm 2 for removing edges after we have a competing graph in detail. (For now we assume counts are 1. Larger counts are discussed in Section 4.4) A similar greedy algorithm for finding non-adjacent nodes in a generic graph was proposed in [8]. Their algorithm iteratively chooses a node with minimum degree until the graph is empty. At each iteration, it removes a minimum-degree node and its adjacent nodes (neighbors) from the graph. The difference is that we can use verification to verify whether the action in question actually was performed to the social network.

Our algorithm takes as an input a competing graph $C$ and output a list of non-adjacent nodes $L$. Each node represents a remove action (resulting action from a matching pair of actions) that we need to perform on $G_2$. Our algorithm iteratively chooses a node with the minimum degree (Line 4) and verify if the node’s action is true or false (Line 5). Verify returns true when the node’s action actually happened to the social network (returns false otherwise). If $verify(x)$ returns true, we add the node $x$ to our solution, remove $x$ and all of its neighbors from the graph (Line 5-9). If $verify(x)$ returns false, we remove only the node $x$ from the graph Line 9). We continue with the smaller graph until the graph is empty (Line 3-10) and return the solution $L$ (Line 11).

However, $verify(x)$ needs to make an API call `friendships(user_id1, user_id2)` to the social network and the number of such calls is limited. Thus, we propose 3 variations of $VERIFY_X(x)$: $VERIFY_{ALL}$, $VERIFY_{LIMITX}$, and $VERIFY_{NONE}$. $VERIFY_{ALL}$ (Algorithm 3) always make an API call. $VERIFY_{LIMITX}$ (Algorithm 4) only makes an API call the degree of the given node is greater than $X$. For example, $VERIFY_{LIMIT0}$ will return true for nodes with degree 0 and will make an API call for nodes with degree greater than 0. $VERIFY_{NONE}$ (Algorithm 5) never makes any API call and always returns true. We evaluate these variations in Section 7.

### 4.3 Add Edges

The third step is adding edges to $G_2$ based on the actions from the first step as well as the actions added in the second step. Here we may have to add new nodes after we match all of the edge pairs to add. We also have to make sure we avoid adding self loops (i.e., $i \rightarrow i$) because these are not allowed in social networks. Let us define matching for add actions.

**Definition 7** Two actions add $x \rightarrow *$ and add $* \rightarrow y$ (where $x$ and $y$ are nodes) are matching when $x \neq y$. For example, in Figure 6(c), add $i \rightarrow *$ and $* \rightarrow j$ are matching because there is a common edge $i \rightarrow j$ that satisfies both actions. Therefore, by adding this common edge, we complete two actions on the list.
Algorithm 2 FIND_NONADJACENT_NODES($C$)

**Input:** competing graph $C$

**Output:** list of non-adjacent nodes $L$

1: Each $i \in C.n$odes has degree $i.d$ and adjacent nodes $i.\text{adj}$
2: $L \leftarrow$ empty list
3: while $C$ is not empty do
4: \hspace{1em} $x = \text{arg min } i.d$ // Select a node with the minimum degree $i \in C.n$odes
5: \hspace{1em} if $\text{VERIFY}_X(x) == \text{true}$ then
6: \hspace{2em} $L.append(x)$
7: \hspace{2em} $C.remove\_nodes(x.\text{adj})$
8: \hspace{1em} end if
9: \hspace{1em} $C.remove\_node(x)$
10: end while
11: return $L$

Algorithm 3 VERIFY_ALL($x$)

**Input:** node $x$ to verify

**Output:** true if $x$ was performed to the network, false otherwise

1: return $\text{verify}(x)$

Algorithm 4 VERIFY_LIMITX($x$)

**Input:** node $x$ to verify, $x$ has degree $x.d$

**Output:** true if $x$ was performed to the network, false otherwise

1: if $x.d > X$ then
2: \hspace{1em} return $\text{verify}(x)$
3: end if
4: return true

Algorithm 5 VERIFY_NONE($x$)

**Input:** node $x$ to verify

**Output:** true if $x$ was performed to the network, false otherwise

1: return true

Continuing our example in (Figure 6(c)), we have one matching pair: $* \rightarrow j$ and add $i \rightarrow *$ (common edge is $i \rightarrow j$). After adding $i \rightarrow j$ and removing the matching pair from the list, we have no action left and we have completed constructing $G_2$ (Figure 6(d)).

The main issue with add edges is that $*$ may be any node (including new nodes) and we do not have enough information to make a good guess. Therefore, we will not compute the competing graph for add edges. Instead, we will simply list add actions and make API calls (inlinks and outlinks) to collect new edges.
4.4 Dealing with Counts $> 1$

Next we will explain how we can modify Algorithm 2 FIND_NONADJACENT_NODES($C$) to also handle counts greater than 1. A simple solution is to make copies of actions when the count is greater than 1. This solution is not efficient because the size of the competing graph $C$ may grow significantly (both the number of nodes and edges). We propose a better strategy by using counts.

Let us explain how to deal with counts via an example in Figures 10, 11. We start with the competing graph $C$ as shown in Figure 10. We will walk through part of Algorithm 2. Let $x = (1, 2)$ be the node we select (Line 4). Let \text{VERIFIY}_X(x) return true (Line 5). We add $x$ to our solution list $L$ (Line 6).

We modify Line 7 as follows. Here we don’t always delete all of the neighbors (nodes adjacent to $x$) as before. We first decrement the counts of actions 1 and 2. Action 1 now has a count of 1 and Action 2 now has a count of 0. We delete Action 2 from the list of remove actions and delete neighbors of $x$ that involves Action 2. Therefore, we delete node(2, 5). Finally, we delete $x$ from $C$ (Line 9) and arrive at Figure 11. In Figure 11, the count of Action 1 has been decremented to 1 and Action 2 has been deleted.

Next we will discuss a simple example for when we mistakenly add a non-competing edge when the counts are greater than 1. In Figure 12, the competing action is Action 1 but the count of Action 1 is 2. Since we have 2 copies of Action 1, we can use them for both node (1, 2) and (2, 3). Hence, the edge between the two nodes is not necessary and we should delete it. Therefore, when the counts can be greater than 1, we need to go through all of the nodes and delete
1: remove \{g, h\} \rightarrow i \ [2]
2: remove g \rightarrow \{i, j\} \ [1]
3: remove h \rightarrow \{i, j\} \ [1]

**Figure 12:** Incorrect Competing Graph

non-competing edges after generating a competing graph.

5 Putting it All Together

Now that we have all of the pieces, we can define our holistic strategy in stages as follows.

5.1 Stage S0

We start with an initial guess $G_2$ equals to $G_1$, i.e., assuming that the graph has not changed (let us call this stage S0). Then we collect the sketch $G_2[s]$ as defined in Section 3.

5.2 Stage S1

In this stage we start by listing remove actions $A_{rmv}$ (defined in Section 4.1) based on $G_2$ and $G_2[s]$ from the previous stage. Next we compute the competing graph $C$ based on $A_{rmv}$ as defined in Section 4.2. After that, we call Algorithm 2 FIND_NONADJACENT_NODES($C$). Note that we have variations for the verification step in that algorithm (ALL, LIMITX, and NONE). Here we make API calls through verify(x), which calls friendships(user_id1, user_id2)). Recall from Section 2.1 that friendships is an order of magnitude cheaper than inlinks(user_id) and outlinks(user_id) calls. This algorithm returns a list $L$ of remove actions we need to perform on $G_2$. Lastly, we go through $L$ and remove the edges from $G_2$ as well as delete actions from $A_{rmv}$ (see Section 4.4 for counts > 1).

5.3 Stage S2

In this stage we remove leftover edges (not removed in S1) from $G_2$. We skip this stage if the list of leftover remove actions $A_{rmv}$ is empty. Next we pop the action $a$ from $A_{rmv}$ with the highest count (See Section 7.5 for why we use the highest count). Recall that each action will have a singleton (one node) either on the left hand side (LHS) or RHS. If RHS of the action is a singleton i (e.g., the action is “remove \{g, h\} \rightarrow i”), we call inlinks(i). We update $G_2$ to be consistent with the list of inlinks i.in returned by the API. Otherwise (LHS is a singleton), we call outlinks(i) and update $G_2$ in the same fashion. Next we
recompute $A_{rmv}$ because $G_2$ is updated (some actions may be deleted or have their count decremented). We repeat all of the steps in this stage until $A_{rmv}$ is empty.

5.4 Stage S3

In this stage we add edges to $G_2$. Same as in S2, we will learn about these edges using `inlinks(user_id)` and `outlinks(user_id)` calls. This step is similar to the previous stage S2 but replace “remove” with “add.” We start by listing add actions $A_{add}$ based on $G_2$ and $G_2[s]$. Next we pop the action $a$ from $A_{add}$ with the highest count. Recall that each action will have a * (wildcard node) either on LHS or RHS. If LHS of the action is a * (e.g., the action is “$\text{add } * \rightarrow i$”), we call `inlinks(i)`. We update $G_2$ by adding new the inlinks $i.in$ returned by the API. Otherwise (RHS is a *), we call `outlinks(i)` and update $G_2$ in the same fashion. Next we recompute $A_{add}$ because $G_2$ is updated. We repeat all of the steps in this stage until $A_{add}$ is empty.

6 Evaluation Metrics

Next we explain evaluation metrics to evaluate our updated snapshot $G_2$ against the actual base graph $G_B^2$ at time 2.

6.1 Error Rates

Let $E(X)$ denote edges in $X$. We can define edge error $s$-rate as follows:

$$error\ s\text{-rate} = \frac{|E(G_2) \cup E(G_B^2) - E(G_2) \cap E(G_B^2)|}{|E(G_B^1) \cup E(G_B^2)|}$$

The numerator is the number of mistakes that we made (difference between our snapshot $G_2$ and the actual base graph $G_B^2$) and the denominator is the number of possible mistakes (the size of the graph). We denote this the s-rate because the rate is w.r.t. the Size of the graph.

Let $fp_{rmv}$ denote false positive removal: number of edges removed from $G_2$ but not supposed to (edges present in $G_B^2$). Similarly, let $fn_{rmv}$ denote false negative removal: number of edges not removed in $G_2$ but supposed to (edges not present in $G_B^2$). We can define $fp_{add}$ and $fn_{add}$ in the same fashion. Thus we can also define edge error $s$-rate as follows:

$$error\ s\text{-rate} = \frac{fp_{rmv} + fn_{rmv} + fp_{add} + fn_{add}}{|E(G_B^1) \cup E(G_B^2)|}$$

Alternatively, we can change the denominator to the number of changes to the $G_B^2$ graph. We can define an alternative edge error $c$-rate as follows:

$$error\ c\text{-rate} = \frac{fp_{rmv} + fn_{rmv} + fp_{add} + fn_{add}}{\#\text{edges removed} + \#\text{edges added}}$$
The denominator consists of two terms. The first term the number of edges removed is $|E(G^B_1) - E(G^B_2)|$. Similarly, the second term the number of edges added is $|E(G^B_2) - E(G^B_1)|$. We denote this the c-rate because the rate is w.r.t. the Changes to the graph.

6.2 API Rates

Another important factor is the number of API calls made. We could count and report API calls by their type, but we believe this introduces unnecessary complexity. Instead, we simply add up the number of calls, weighting each type equally. (Note that counting by type makes the baseline RECRAWL strategy look worse since it makes the more expensive type of call (fewer calls allowed per time unit). Thus, our single API count hurts the strategy we propose.)

Let $N(X)$ denote nodes in $X$. We can define api s-rate as follows:

$$\text{api s-rate} = \frac{\#\text{API calls}}{|N(G^B_1) \cup N(G^B_2)|}$$

The numerator is the number API calls that we made and the denominator is the size of the graph (the number of nodes in the union of the base graphs).

Alternatively, we can change the denominator to the number of changes to the graph. We can define an alternative api c-rate as follows:

$$\text{api c-rate} = \frac{\#\text{API calls}}{\#\text{edges removed} + \#\text{edges added}}$$

7 Experiments

The goal of these experiments is to compare our strategy to the baseline strategy RECRAWL (will define in Section 7.2) applied to a real dataset. We want to see how our strategy save the API calls over the baseline RECRAWL. We also compare verification strategies NONE, LIMITX, and ALL. Furthermore, we would like to verify that ordering actions by their count in decreasing order for stages S2 and S3 reduces the number of API calls. Moreover, we would like to see the tradeoff between the amount of verification and errors. Lastly, we would like to compare the two datasets $DS_{stale}$ and $DS_{fresh}$ (big vs. small updates to the graph we are updating).

We have conducted experiments on two graphs, Slashdot and Google+, and with differing time intervals between snapshots. The graphs vary in size, density, and rate of change, giving us insight into different scenarios. Unless specified, our plots show the results after the final stage S3.

7.1 Datasets

For our real-world datasets, we will use snapshots from Slashdot and Google+ social networks. In this paper, we focus mainly on the Slashdot snapshots, but in Section 7.8 we describe our Google+ data and results.
Table 2: Datasets

<table>
<thead>
<tr>
<th></th>
<th>DS_{stale}</th>
<th>DS_{fresh}</th>
<th>DG_{stale}</th>
<th>DG_{fresh}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G^B_1$</td>
<td>nodes</td>
<td>70917</td>
<td>74606</td>
<td>9825</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>35543</td>
<td>40289</td>
<td>960778</td>
</tr>
<tr>
<td></td>
<td>nodes removed</td>
<td>2006</td>
<td>17</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>nodes added</td>
<td>6094</td>
<td>241</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>edges removed</td>
<td>7462</td>
<td>91</td>
<td>9844</td>
</tr>
<tr>
<td>$G^B_2$</td>
<td>edges added</td>
<td>7338</td>
<td>554</td>
<td>2245</td>
</tr>
<tr>
<td></td>
<td>old-old</td>
<td>47171</td>
<td>256</td>
<td>2265</td>
</tr>
<tr>
<td></td>
<td>old-new</td>
<td>12</td>
<td>283</td>
<td>1491</td>
</tr>
<tr>
<td></td>
<td>new-old</td>
<td>405</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>new-new</td>
<td>60403</td>
<td>1113</td>
<td>6030</td>
</tr>
</tbody>
</table>

Slashdot is a technology news website that allows users to mark other users as friends or foes. We will use three snapshots ($t_1$, $t_2$, $t_3$) available at [3] in our experiments. We will present our results for the friends relationships only. The results for foes relationships are analogous.

7.1.1 Dataset DS_{stale} ($t_1 - t_2$)

Our first dataset DS_{stale} is the Slashdot friends snapshots collected on 11/6/2008 and 2/16/2009 (3 months apart). The details of the base snapshot ($G^B_1$) and the new snapshot we would like to collect ($G^B_2$) are shown in Table 2. Under edges added, we categorize types of edges based on source and target nodes. For example, old-new means the edges originate from an old node (exists in $G^B_1$) and link to a new node (not in $G^B_1$). We can see that most of the new edges involve at least one old node (source or target, or both). Therefore, our strategy that relies on the changes in the degrees of the old nodes will be effective.

$G^B_2$ is not directly available to our algorithm. When our algorithm makes calls to access the base graph, we do not call Slashdot but instead emulate the call by using our copy of $G^B_2$.

7.1.2 Dataset DS_{fresh} ($t_2 - t_3$)

Our second dataset DS_{fresh} is the Slashdot friends snapshots collected on 2/16/2009 and 2/21/2009 (5 days apart). The details of the base snapshot ($G^B_1$) and the new snapshot we would like to collect ($G^B_2$) are shown in Table 2. We will use this dataset to show that our strategy is even more effective when the graph has not changed much.

7.2 Baseline Strategy RECRAWL

Our baseline strategy RECRAWL first scans the nodes in $G^B_1$ ($N(G^B_1)$) and recrawls each node to discover any changes in edges (edge addition and removal) or node deletion. This step may discover new nodes ($N(G^B_2) - N(G^B_1)$). We have to crawl the new nodes as well. Each node requires 2 API calls (one to collect
inlinks and another to collect outlinks). Therefore, RECRAWL makes about 150K API calls for both datasets $DS_{stale}$ and $DS_{fresh}$.

### 7.3 Overall Results

Figure 13 shows that variations of our VERIFY strategy (ALL, LIMIT0, LIMIT1, NONE defined in Section 4.2.2) significantly reduce the number of API calls over the baseline strategy RECRAWL (defined in Section 7.2) for dataset $DS_{stale}$. The left and right y-axes show the API and error s-rates (w.r.t. the size of the graph). Our most expensive strategy ALL (always verify) saves 10x API calls over RECRAWL while introducing only 0.05% error rate. Our cheapest strategy NONE (never verify) saves 30% API calls over ALL but triples the error rate. In summary, our strategy is 10-15 times cheaper than RECRAWL while introducing minimal errors.

Figure 14 shows the errors in more detail. The left and right y-axes are now the absolute number of API calls and errors respectively. Recall that $fn_{rmv}$ errors are the edges removed from $G^B_2$ we failed to remove. This error is the most common among our strategies because our strategies rely on the degree discrepancies between $G_1$ and the sketch $G^B_2$. When a node removes an existing outlink and adds a new outlink (inlink), the outdegree (indegree) stays the same. LIMIT0 saves 1.4K API calls over ALL but only introduces slightly more $fp_{rmv}$ and $fn_{rmv}$ errors than ALL. LIMIT1 suffers from 20% more errors while only saving 400 more API calls over LIMIT0. Recall that LIMIT0 skips verification when the node in the competing graph has zero degree, i.e., has no competing matching pair. This is our best strategy and we will continue presenting results for LIMIT0.
7.4 Results by Stage

Figure 15 shows how the number of API calls grow and the number of errors shrink after each stage (S0-S3). The left and right y-axes are the absolute number of API calls and errors respectively. S3 (add edges) is the most expensive stage because in our datasets there are a lot more edges to add than to remove (almost 8x). Moreover, our strategy focuses on saving the API calls for removing edges (S1-S2). At S0, we make our initial guess that the snapshot stays the same. At this stage we have 7.5K fn_rmv and 55K fn_add errors. From stage S0-S2, we only remove edges and therefore the add edge errors (fn_add) stay the same. S1 and S2 effectively remove almost all of the remove edge errors fn_rmv. Similarly, S3 effectively remove almost all of the add edge errors fn_add. We did not introduce any fp_add. Therefore, our strategy LIMIT0 effectively updates the snapshot to G_2 with minimal errors.

7.5 Ordering Actions to Make API Calls

Next we study the effect on the number of API calls when we order actions by their count in decreasing order vs increasing order for stages S2 and S3. We omit the figure for this study. We learned that the increasing order (query the actions with the lowest count first) requires 150% more API calls than the decreasing order for strategy LIMIT0 (similar results for other strategies). This is because when we choose the action with the highest count first (say the count
is 10), we decrement up to 10 actions in the list. It may not be 10 actions because our sketch might have missed some of the graph change when the degree remains unchanged from time 1 to 2. And for some actions decremented, the count is 0 and the action is deleted from the list. Thus, we should use the decreasing order.

7.6 Verification vs. Errors

Figure 16 shows the tradeoff between the amount of verification and the number of errors. The left and right y-axes are the absolute number of API calls and errors respectively. The x-axis shows the values of limit X (variations for VERIFY_LIMITX). Recall that limit 0 means we always verify and limit infinity means we never verify (same as VERIFY_NONE). As we increase the limit, we make more fn_rmv and fp_rmv errors. This is because these two errors are correlated. Recall that fn_rmv errors are when we fail to remove an edge whereas fp_rmv errors are when we wrongly remove an edge. As we increase the limit, we make an API call to verify less often and wrongly remove edges more often. Each time we remove the wrong edge, we also ignore removing the correct edge because we will incorrectly think that we have fixed the inconsistency between G_2 and the sketch G_2[s], and as a result, skip removing this correct edge. The number of fn_rmv errors grows from 204 to 476. The number of fp_rmv errors grows from 6 to 72. On the other hand, as we increase the limit, we also save API calls. However, increasing limit 0 to 4, we only save 1K API calls (9%) while introducing more than 100% more errors. Therefore, we believe LIMIT0 is our best strategy overall.
To compare the two datasets, we first compare the absolute number of API calls and errors, then we compare the rates. We first compare the API and error rates w.r.t. the size of the graph (s-rate) and then compare the rates w.r.t. the changes to the graph (c-rate). The most interesting comparison is the c-rates. One would expect that if we normalize by the number of changes to the graph, both datasets would have the same API and error rates. For example, if we make 10 API calls and 10 errors when there are 10 changes to the graph, we would expect to make 20 API calls and 20 errors when there are 20 changes to the graph (i.e., double the API calls and errors when double the changes to the graph). However, we find that the number of API calls and errors do not grow linearly in proportion to the number of changes to the graph (See Figure 19).

Figure 17 shows that variations of our strategy significantly save the number of API calls over the baseline strategy RECRAWL for dataset \( DS_{fresh} \). Dataset \( DS_{fresh} \) has far fewer updates to the graph than dataset \( DS_{stale} \) (5 days apart vs. 3 months apart between 2 snapshots). The left and right y-axes are the absolute number of API calls and errors respectively. Note that the baseline strategy RECRAWL still needs to make 150K API calls for dataset \( DS_{fresh} \).
On the other hand, our strategies only need fewer than 400 API calls. Our strategies avoid making API calls when the degrees do not change (i.e., node degree stays the same when we collect the new sketch $G_2[s]$). This result illustrates that our strategy thrives even more when there are a few updates to the graph ($G_2^B$ is very similar to $G_1^B$). Strategy LIMIT0 saves 13% API calls over strategy ALL while not introducing any more error. It is worth noting that for $DS_{fresh}$, NONE introduces very few errors (3 more $fn_rmv$ errors) while saving 20% API calls over strategy ALL. Therefore, NONE is the best strategy overall for dataset $DS_{fresh}$.

To compare the two datasets $DS_{stale}$ and $DS_{fresh}$, we will study the API and error rates rather than the absolute numbers. We will focus on the strategy LIMIT0 for studying these rates. (Results are analogous for the other strategies.) Figure 18 compares the two datasets using the API and error s-rates. The left and right y-axes show the API and error s-rates (w.r.t. the Size of the graph). Our strategies are orders of magnitude better in $DS_{fresh}$ than $DS_{stale}$ but this is expected.

Alternatively, we can compare $DS_{stale}$ and $DS_{fresh}$ using the rates with respect to the number of changes to the graph. Figure 19 compares the two...
datasets using the API and error c-rates. When we update the snapshot more frequently (dataset $DS_{fresh}$), we make far fewer errors per graph change (more than 50% lower error rate than dataset $DS_{stale}$). This is because the confusion compounds more than proportionally to the number of changes to the graph (i.e., our competing graph grows faster than linearly in terms of the number of nodes and edges). However, when we update the snapshot less frequently (dataset $DS_{stale}$), we make fewer API calls per graph change. This is because each API call (outlinks and inlinks) returns more information (new edges or removed edges) when there are more changes to the graph. Therefore, frequently updating the snapshot yields lower error rate but is more costly (higher API calls per graph change rate).

In summary, our strategies beats RECRAWL by saving orders of magnitude of API calls while introducing minimal errors. The savings are even more significant in the case of dataset $DS_{fresh}$ where the number of changes to the graph is small. LIMIT0 is the best strategy overall for dataset $DS_{stale}$ while NONE is the best for dataset $DS_{fresh}$.

7.8 Google+

7.8.1 Datasets

Google+ is an interest-based social network that allows users to follow other users. It is a directed network (a user you follow may not follow you back). The
The current Google+ API does not support asking for social connections (in and out). To work around this problem, we manually crawled social connections by parsing in and out attributes of profile pages, using a Python program. Instead of using the Google+ API, we wrote a custom crawler that poses as a Web browser and sends requests to Google+ with user ids we would like to ask for social connections. The profile pages returned meant for a Web browser include social connections. Only up to around 10,000 connections of each type (in and out) are visible on each profile page.

We collected our partial snapshots using the procedure shown in Algorithm 6. Starting from our seed node s, we collect its outlinks k hops/levels away (Line 4-12). For each node we collect, we crawl its outlinks and update $G_e$ in-place (Line 7-8).

Our Google+ datasets are collected from June to September 2015 using COLLECT_SNAPSHOT(\(u, 2\)) where \(u\) is a user randomly selected from Sergey Brinn followers on Google+. We then prune the peripheral nodes (nodes that are distance 3 from \(u\)) and effectively end up with a 2-level extended ego network of \(u\).

### Dataset DG\_stale

Our dataset $DG_{\text{stale}}$ is the Google+ snapshots collected on 6/11/2015 and 9/3/2015 (12 weeks apart). The details of the base snapshot ($G^B_1$) and the new snapshot we would like to collect ($G^B_2$) are shown in Table 2. Note that Google+ snapshots are a lot more dense (20 times higher...
Algorithm 6 COLLECT_SNAPSHOT(s, k)

Input: seed node s, the number of levels k to crawl s
Output: snapshot $G_t$ at time $t$

1: $t \leftarrow$ current time
2: $G_t \leftarrow$ empty graph
3: $S \leftarrow \{s\}$
4: for $i = 0 \rightarrow k$ do
5:  $S' \leftarrow S$
6:  $S \leftarrow$ empty set
7:  for each $x \in S'$ do
8:     $x$.out $\leftarrow$ outlinks($x$.id)
9:  Add $x$, $x$.out to $G_t$ (add nodes and edges)
10:  $S = S \cup x$.out
11: end for
12: end for
13: return $G_t$

Figure 20: Comparing All Strategies: RECRAWL Baseline vs. Variations of our Strategy, Dataset $DG_{stale}$

average degree) than Slashdot ones.

7.8.1.2 Dataset $DG_{fresh}$ Our dataset $DG_{fresh}$ is the Google+ snapshots collected on 9/3/2015 and 9/10/2015 (1 week apart). The details of the base snapshot ($G_B^1$) and the new snapshot we would like to collect ($G_B^2$) are shown in Table 2.
7.8.2 Results

Figure 20 (analogous to Figure 13) summarizes the results for our $DG_{stale}$ dataset. The first thing to notice is that our solution continues to provide significant savings over the baseline strategy RECRRAWL with very few errors. There are however some minor differences. Compared to $DS_{stale}$ (Figure 13), our strategies require more API calls for $DG_{stale}$. Another interesting difference is that strategy NONE saves even more API calls over ALL (over 90% API calls over ALL) but quadruples the error rate. Like our Slashdot datasets, Figure 21 shows that most of the errors are $fn_{rmv}$. As expected, Figure 22 shows that when the dataset is fresh, our strategies make very few mistakes (only 3 errors in most of our strategies).

To compare the two datasets $DG_{stale}$ vs. $DG_{fresh}$, we first compare the API and error rates w.r.t. the size of the graph (s-rate) and then compare the rates w.r.t. the changes to the graph (c-rate). Figure 23 shows that the s-rates agree with earlier Slashdot results. However, Figure 24 is different from the Slashdot results in Figure 19. Recall from Figure 19 that when we update the snapshot more frequently (dataset $DS_{fresh}$), we make far fewer errors per graph change at the cost of more API calls per graph change. On the other hand, for dataset $DG_{fresh}$, we make both fewer errors and fewer API calls per graph change. Therefore, for Google+, we should run our strategy to update snapshots frequently (weekly works well based on our experiments).
### 8 Related Work

The problem we address here is a type of graph snapshot update where we have limited information and we can only discover edges (nodes) at a limited rate. As far as we know, this type of snapshot update has not been studied earlier.

There are several other areas that are related to our work: updating a web crawl, social network analyses with a limited API, and social network crawling techniques. Cho et al.\[6\] proposed effective page refresh policies for web crawlers to improve freshness and age of web pages. Vesdapunt et al.\[14\] studied entity resolution across social networks with a limited API.

Recently, there have been numerous works on social network crawling. Schiöberg et al.\[12\] crawled over 95% of Google+ profiles between Sep and Oct 2011 and presented its characteristics. Magno et al.\[10\] crawled 27M Google+ profiles and 580M links in Dec 2011. Gjoka et al.\[7\] proposed Metropolis-Hasting and reweighted random walks for unbiased sampling Facebook. Catanese et al.\[4\] sampled Facebook using breadth-first-search and uniform crawlers. Kwak et al.\[9\] crawled the whole Twitter snapshot in July 2009: 42M profiles and 1.5B links. Chau et al.\[5\] proposed a framework of parallel crawlers with a central-
ized queue. Ye et al.[16] proposed greedy and lottery strategies for crawling static social network snapshots and studied the strategies on Flickr, LiveJournal, Orkut, and Youtube. Our strategy could be used to refresh their snapshots with low cost.

Recently, there have been several works on defending against social network crawling. Wilson et al.[15] proposed SpikeStrip, a link-encryption server add-on for rate-limiting crawlers. Their encryption prevents crawlers from aggregating and correlating information from different browsing sessions. They experimented on anonymized Facebook data from 1.2M London users and showed that SpikeStrip effectively prevents distributed crawlers while imposing minimal server throughput overhead. In another work, Mondal et al.[11] proposed Genie, a system for limiting large-scale crawlers. Genie distinguishes between honest users and crawlers using their access pattern (e.g., crawlers visit user profiles that are further away in the social network). Our strategy minimizes the number of accesses to the social network (API calls).

9 Conclusion

We have studied the problem of graph tracking with limited information. We have proposed an algorithm with variations in the verification step that works well in practice. In particular, we found two variations to be the best: LIMIT0 and NONE. Variation NONE works well when the previous snapshot is still fresh.
Figure 24: LIMIT0 API and Error C-Rates, Dataset $DG_{stale}$ vs. $DG_{fresh}$

Variation LIMIT0 works better when the snapshot is stale. Both algorithms significantly reduce the number of API calls over the baseline Algorithm RECRAWL while making minimal errors. We inexpensively get the sketch to help verify select nodes and edges based on observing discrepancies between the degrees in the sketch and the previous snapshot. We also learned that when there are more changes to the graph, we end up making fewer API calls per change because each API call yields more information (more new edges or removed edges). However, we also introduce more errors per change due to the compounding confusion.

In practice, when there are a large number of changes to the graph (e.g., addition of new nodes), our strategy may not be as effective. However, in the worst case we only do marginally worse that the RECRAWL strategy. To see this, consider the two components to our cost: X, the cost of acquiring the sketch, and Y, the verification cost. In the worst case, we will end up verifying all of the nodes and Y will be comparable to the cost of RECRAWL. The worst case penalty is thus X, which is almost negligible compared to Y (because building the sketch uses much less expensive API calls).

Although our experiments focus on Slashdot and Google+ snapshots, we believe our strategy can be applied directly to other social systems with a limited API such as Twitter, Instagram, Tumblr, and Flickr. The API calls we use are the same interface shared by these popular social systems. These systems share the same operations such as looking up information (indegree, outdegree) for a
specific user, and collecting user relationships. They also allow users to collect all of the inlinks and outlinks through pagination. The cost of each API call may vary from system to system, but that can be modeled by weights. Therefore, our solution can either be used directly or can be easily adapted.

References
