

# Identifying Users in Social Networks with Limited Information

Norases Vesdapunt, Hector Garcia-Molina

March 2014

## 1 Abstract

We study the problem of Entity Resolution (ER) with limited information. ER is the problem of identifying and merging records that represent the same real-world entity. In this paper, we focus on the resolution of a single node  $g$  from one social graph (Google+ in our case) against a second social graph (Twitter in our case). We want to find the best match for  $g$  in Twitter, by dynamically probing the Twitter graph (using a public API), limited by the number of API calls that social systems allow. We propose a strategy that is designed for limited information and is robust to rule changing. We evaluate our strategy against random on a real dataset and show that our strategy is 46% faster than random and achieves 88.3% accuracy. We also propose a greedy strategy, which is a natural extension of bipartite matching subject to limited API calls, and it achieves 85% accuracy.

## 2 Introduction

A social system like Facebook 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 single user  $U$  may participate in more than one social system, and in many cases it is useful to match the user's node in one social graph to the same user's node in another graph. For example, say user  $U$  is represented by node  $g$  in Google+, and the same user is represented by node  $t$  in Twitter. Even though this is the same person, his/her profile information in each social network may differ. For instance, in  $g$  the user's name may be spelled one way, while in  $t$  it may be spelled differently. Similarly, the interests listed in  $g$  may not match exactly the interests listed in  $t$ . In some cases,  $U$ 's Google+ ( $g$ )

profile might include his Twitter handle, i.e., the identity of  $t$ . But in many cases the connection between  $g$  and  $t$  is not explicitly stated. (We will continue to use our Google+, Twitter example in this paper, since it will also be the focus of our experiments.)

There are many situations in which it is important to match a node like  $g$  in one system to its corresponding node like  $t$  in another system. For example, if user  $U$  were a customer at a store, the store could benefit from knowing that  $g$  and  $t$  are the same person so as to avoid sending duplicate promotional emails. As another example, an intelligence agency gathering information about terrorists may want to combine the data in  $g$  and  $t$  to obtain a better picture of a potential terrorist. Another possible application is identifying duplicate users (possibly spammers) in the same network. In addition, Wikipedia may want to merge near-duplicate articles in the same language by exploiting similarities in their links. Going beyond one language, identifying the same Wikipedia article across languages is useful for extending an article of one language by translating content from another language.

The problem of matching nodes (or records) from one system to nodes in another system is called *entity resolution* (ER) (or deduplication or record linkage) and has been well studied [22]. The traditional approach to entity resolution involves analyzing the *entire* graph (all records) from both systems, in order to find the best correspondence among all nodes. That is, to find the best matching Twitter record for our Google+ node of interest  $g$ , we examine many (or all) Twitter nodes, to find the one that maximizes some similarity metric. Furthermore, we also take into account the relationships among nodes. For example, say that  $g$  has a group of Google+ friends  $F_G$ , while a Twitter node  $t$  has Twitter friends (or followers)  $F_T$ . If the  $F_G$  nodes are “similar” to the  $F_T$  nodes, then we can be more confident that  $g$  and  $t$  refer to the same user.

While this type of global node and relationship analysis works well, in many situations one does not have access to the entire graphs of both networks. In this paper we focus on the ER problem when constrained by limited information. In particular, we focus on the resolution of a single node ( $g$  in our example) against a second social graph (Twitter in our example). (This type of single node/record resolution is sometimes called *dipping*.) We want to find the best match for  $g$  in Twitter, by *dynamically probing* the Twitter graph (using a public API). For example, one possible *strategy* is as follows: We take the user’s name recorded in  $g$ , and perform a search on Twitter to find users that have similar names. For each of these Twitter users, we fetch their nodes (profiles) and select the one that is most similar to  $g$ . Of course, there are many other possible strategies, and we will explore a number of them in this paper.

The key challenge for dynamic probing is that the API often limits the type and number of probes one can perform. Twitter recently changed its API limits, making it more difficult to probe its graph. Developers have to be more agile and diversify their requests to avoid hitting the new limits. Previously, a normal developer could perform at most 350 requests of any type per hour. Whitelisted developers could make at most 20,000 requests per hour. In the new API, Twitter no longer offers whitelisting and has changed from one universal limit

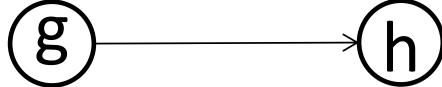


Figure 1: Following Relationship

to per-endpoint limits. Although Twitter has reduced the size of the resetting window (the time between operation quota resets) from one hour to 15 minutes, the limit on the number of requests for some operations such as getting followers of a node has been reduced from an average of 5.83 to 1 operation per minute. Developers have to spend time navigating and interpreting these new rules. For example, one can only perform at most 180 name searches (as suggested above) per window. If one has the identity  $t$  of a Twitter node, one can request  $t$ 's profile information. One can ask for profiles for 100 nodes in the same request, but only 180 such requests can be made per window. When one asks for the followers of node  $t$ , one can only get 5,000 followers at a time, and one can only submit 15 such requests per window [8, 20].

In summary, not only do we want to find a strategy that can accurately identify  $g$ 's corresponding node in Twitter, but we also want a strategy that consumes resources (i.e., probes) wisely. Furthermore, our strategy has to be robust and flexible to accommodate for changes to API limits at network owner's whim. In this paper we will explore this accuracy-cost tradeoff in detail. For concreteness, we will use Google+ and Twitter for our discussion and experiments but we believe our algorithms are applicable to other scenarios.

The summary of our contributions is as follows:

- We formulate the problem of social network ER with limited resources (Section 3).
- We propose an iteration workflow that repeats exploration and evaluation steps (Section 4).
- We propose two algorithms for dynamically crawling a network and finding a matching node. These algorithms exploit heuristics derived from our experiments and work with a limited API (Section 6).
- We evaluate our algorithms against random using a real-world dataset (Section 7).

### 3 Preliminaries

#### 3.1 Model

We consider two social systems/graphs,  $G$  (Google+) and  $T$  (Twitter). We will use lowercase letters near  $g$  (e.g.,  $f$ ,  $g$ ,  $h$ ) to refer to nodes in  $G$ , and lowercase letters near  $t$  to refer to nodes in  $T$ . Note that in some systems (e.g.,

<b>Text</b>
<ol style="list-style-type: none"> <li>1. <b>names</b>: fullname + other names extracted from the profile</li> <li>2. <b>id</b>: uniquely identifies a user and cannot be changed</li> <li>3. <b>locations</b>: current_place + places extracted from the profile</li> <li>4. <b>descriptions</b>: {introduction, bragging_rights, tag-line, job_title, universities, organizations}</li> <li>5. <b>urls</b>: urls from “other profiles” attribute + urls extracted from profile descriptions. Picasa, LinkedIn, Facebook, Pinterest, and YouTube are common urls besides Twitter. For example, Sergey Brin’s profile contains links to his YouTube and Picasa accounts: <a href="http://www.youtube.com/user/sbrin1600/">http://www.youtube.com/user/sbrin1600/</a> and <a href="http://picasaweb.google.com/sergey.brin">http://picasaweb.google.com/sergey.brin</a>.</li> <li>6. <b>twitter</b>: Twitter screen name extracted from urls. For example, we can extract ladygaga from <a href="https://twitter.com/ladygaga">https://twitter.com/ladygaga</a>.</li> <li>7. <b>screen_names</b>: possible screen names extracted from urls. For example, ladygaga is a potential screen name extracted from <a href="https://www.facebook.com/ladygaga">https://www.facebook.com/ladygaga</a>.</li> <li>8. <b>posts</b>: up to 10 most recent posts posted by the user</li> </ol>
<b>Graph</b>
<ol style="list-style-type: none"> <li>1. <b>in</b>: set of <math>G</math> users who follow <math>g</math> (i.e., <math>g</math>’s inlinks). Google+ refers to this attribute as “have him/her in circles.”</li> <li>2. <b>out</b>: set of <math>G</math> users that <math>g</math> follows (i.e., <math>g</math>’s outlinks). Google+ refers to this attribute as “in his/her circles.”</li> <li>3. <b>indegree</b>: <math> in </math>. Takes value 0 when either a user has no followers in <math>G</math> or the information is private.</li> <li>4. <b>outdegree</b>: <math> out </math>, similar to <b>indegree</b></li> </ol>

Table 1:  $G$  Attributes [1]

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.

In  $G$  and  $T$ , nodes have the profile *attributes* shown in Tables 1 and 2. We will use dot notation to refer to 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,  $g.out$  is the set of  $G$  identifiers for the users that  $g$  follows while  $g.in$  is the set of  $G$  identifiers for the users that follow  $g$ . Note that the **in** and **out** attributes encode the  $G$  graph structure. Figure 1 shows the following relationship:  $g$  is following  $h$  ( $h \in g.out$ ) and  $g$  is  $h$ ’s follower ( $g \in h.in$ ). In this case,  $g$  consumes the information feed produced by  $h$ .

<b>Text</b>
<ol style="list-style-type: none"> <li>1. <b>name</b>: fullname</li> <li>2. <b>screen_name</b>: uniquely identifies a user, but a user may change his/her screen_name</li> <li>3. <b>id</b>: uniquely identifies a user and cannot be changed</li> <li>4. <b>description</b>: free text, 160 character limit</li> <li>5. <b>location</b>: free text</li> <li>6. <b>status</b>: “tweet”, 140 character limit per status</li> <li>7. <b>url</b>: one link to website</li> </ol>
<b>Graph</b>
<ol style="list-style-type: none"> <li>1. <b>in</b>: set of <math>T</math> users who follow <math>t</math> (i.e., <math>t</math>’s inlinks). This is not part of <math>T</math> attributes available in profiles but we will use this notation. To obtain this information, we have to invoke API operations discussed later in Table 3.</li> <li>2. <b>out</b>: set of <math>T</math> users that <math>t</math> follows (i.e., <math>t</math>’s outlinks). Twitter refers to this attribute as <b>friends</b> but we will continue using <b>out</b> to be consistent.</li> <li>3. <b>indegree</b>: <math> in </math>. Takes value 0 when a user has no followers in <math>T</math>. This information is available in the profile even for <i>protected</i> users: users whose text and graph information is only available to followers they approved [3].</li> <li>4. <b>outdegree</b>: <math> out </math>, similar to <b>indegree</b></li> </ol>

Table 2:  $T$  Attributes [2]

### 3.2 API Operations

Before we define API operations, we define pagination:

- **Pagination:** An API provider splits the results of a query into pages and an application can issue a request to fetch one result page. On each result page, the next page token is included. Developers can issue a request with a next page token to retrieve the next result page. Note that an API provider gets full control over the order of the result pages returned by the API (e.g., an application cannot skip result pages or request the last result page first).

Currently,  $G$  only provides a simple API for fetching user profiles and searching users [6]. For our application, two operations are relevant:

1. **people/get**: returns the user profile given his/her `user_id`. The profile returned contains only text attributes. All graph attributes discussed in Table 1 are not returned.
2. **people/search**: returns `user_ids` and full names of users that contain attributes matching the given text. For example, given the text “Stanford University,” **people/search** returns profiles with full names “Stanford University,” “Stamford University Bangladesh,” and “Julie Stanford.” This example demonstrates that “matching” means partial/approximate

match since inexact matches to our query like “Stamford University Bangladesh” are also returned. Marissa Mayer’s profile is also in the result because she has indicated in her profile that she attended Stanford University. Each request returns up to 20 user\_ids. If there are more than 20 relevant user\_ids matching the given text, all user\_ids are accessible through pagination.

The problem with the current Google+ API is that the 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.

$T$  provides a limited API for social connections discussed in Table 3. The notation for API calls is `operation(parameter1 , parameter2, ...)` where parameters are either `user_id`, `screen_name`, or `text` depending on the operation. For example, `users/search(name)` submits a name search to Twitter and returns user nodes matching the given name.

### 3.3 Operation Costs/Limits

The Google+ API limits each application to 10,000 requests per day. This is a universal limit for any type of API call under HTTP API (People, Activities, and Comments) [6].

For Twitter, limits and details of each operation are described in Table 3. Retrieved user profiles contain profile attributes specified in Table 2.

### 3.4 Problem Definition

Given a  $G$  node  $g$ , find a node in  $T$  that represents the same real-world user. Assume that we have full access to  $G$  (e.g., operations in  $G$  are free) but we can only access  $T$  through a limited API. The procedure for finding the matching node must respect API constraints, i.e., can only make a limited number of calls to explore parts of  $T$ .

### 3.5 Running Example

We introduce a simple example for a match task in Figure 2. Nodes in  $G$  are  $g$ ,  $h$ , and  $i$ . Nodes in  $T$  are  $p$ ,  $q$ ,  $r$ ,  $s$ , and  $t$ .  $g$  follows  $h$  and  $i$ , i.e.,  $g.out = \{h, i\}$ . Node  $h$  is followed by  $g$  and  $i$ , i.e.,  $h.in = \{g, i\}$ . Potential candidates in  $T$  that match  $g$  are  $s$  and  $t$ . Potential candidates in  $T$  that match  $h$  are  $p$  and  $q$ . A potential candidate in  $T$  that matches  $i$  is  $r$ . We will discuss how to find these candidates in Section 3.6.2. Let  $F$  be the fraction of  $g.out$  that we crawl. Let  $C_F$  be the number of candidates we consider a potential match

Limit	Operations
15	<ol style="list-style-type: none"> <li>1. <b>friendships/show</b>: returns relationships (following, followed by) between 2 given user_ids/screen_names [4].</li> <li>2. <b>followers/id</b>: returns up to 5,000 user_ids per request of most recent followers of a given user_id/screen_name. For users who have more than 5,000 followers, pagination can be used to retrieve all of them. For example, if a user has 12,000 followers, an application needs to issue 3 requests to retrieve all of them.</li> <li>3. <b>friends/id</b>: returns up to 5,000 user_ids per request of users that a given user_id/screen_name follows (most recent). For users who follow more than 5,000 other users, pagination can be used to retrieve all of them.</li> <li>4. <b>followers/list</b>: returns up to 20 user profiles per request of most recent followers of a given user_id/screen_name. For users who have more than 20 followers, pagination can be used to retrieve all of them.</li> <li>5. <b>friends/list</b>: returns up to 20 user profiles per request of users that a given user_id/screen_name follows (most recent). For users who follow more than 20 other users, pagination can be used to retrieve all of them.</li> </ol>
180	<ol style="list-style-type: none"> <li>1. <b>users/search</b>: returns up to 20 user profiles per request matching query based on relevance given “topical interest, full name, company name, location, or other criteria”; does not support exact match searches. If there are more than 20 user profiles relevant to the query, pagination can be used to retrieve up to 1,000 profiles [5].</li> <li>2. <b>users/lookup</b>: returns a user profile given a user_id/screen_name. Returns <i>nil</i> if no matching profile is found. An application may provide up to 100 user_ids/screen_names per request (returns a list of user profiles).</li> </ol>

Table 3: Operation Limits (Number of Requests) per 15-Minute Window [8]

for each of  $g.out$ . Lastly, let  $C_T$  be the number of candidates we consider a potential match for  $g$ . We will study these thresholds in Section 5.5.2.

### 3.6 Basic Functions

Many of our strategies will use common procedures (e.g., computing the similarity between two nodes). Here we describe these functions.

#### 3.6.1 Similarity Functions

- **String Similarity:** We first define basic similarity functions for comparing two strings:  $StringSim(a, b)$  and  $CosineSim(a, b)$ . These functions are used to compute features in Table 4.

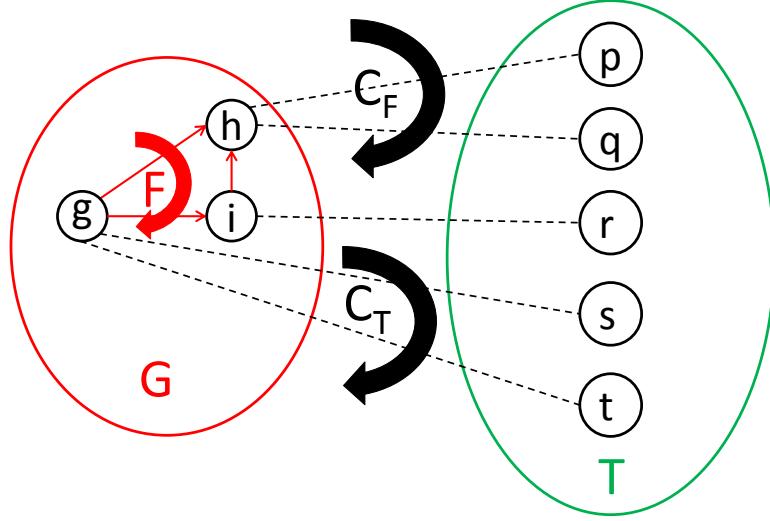


Figure 2: Example of a Match Task

1.  $\text{StringSim}(a, b) = \frac{2\{\max(|a|, |b|) - \text{EditDistance}(a, b)\}}{|a| + |b|}$  [9]
2.  $\text{CosineSim}(a, b)$  = cosine similarity of uniwords and biwords extracted from  $a$  and  $b$  weighted using term frequency-inverse document frequency (tf-idf) with stemming and stop words removed

- **Set Similarity:** For comparing two sets  $A$  and  $B$ , we introduce the commonly used Jaccard Similarity:

$$\text{JaccardSim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

We will describe later how we adapt Jaccard Similarity to our application when we define Overlap.

### 3.6.2 Common Functions

**Candidates**[ $g$ ,  $T$ ]: Given a  $G$  node  $g$ , finds a set of  $T$  nodes that may possibly match  $g$  (e.g., by doing a name search). **Candidates** is a general concept. On  $T$ , we can use `users/search` but may need multiple API calls to collect the full set of candidates. For now, let's assume that we get the full set of candidates. Later in our experiments, we will discuss the limits (i.e.,  $C_F$  and  $C_T$  in our running example) and tradeoffs on the number of candidates we retrieve. From our running example in Figure 2,  $\text{Candidates}[h, T] = \{p, q\}$ . Similarly,  $\text{Candidates}[i, T] = \{r\}$  and  $\text{Candidates}[g, T] = \{s, t\}$ .

**Match**[ $g$ ,  $t$ ]: Given a  $G$  node  $g$  and a  $T$  node  $t$ , returns an estimated similarity score. The score ranges from 0 ( $g$  and  $t$  are completely different) to 1 ( $g$  and  $t$  are identical). Various features that can be used when implementing

the match function including text and graph similarities are defined in Table 4. Table 4 describes similarity features we can use to compare two profiles  $g$  and  $t$  and their outlink neighborhoods (the users that  $g$  and  $t$  follow). We use logistic regression to train the weights of these similarity features. Our match score is defined as logistic function of a linear combination of features using the learned weights. In other words, our match score is the probability that  $g$  and  $t$  are the same person predicted by the logistic regression classifier.

We define two versions of `Match` each suited for different resource availability: (1) `PrimitiveMatch` and (2) `AdvancedMatch`. We will define `AdvancedMatch` after we have defined prerequisite functions.

`PrimitiveMatch[g, t]`: a `Match` function that computes features from only local profile attributes. Local means these attributes are available as part of a user profile and no additional API calls are needed to fetch these attributes. These features are the primitive features described in Table 4. They consist of text similarity features and a position feature (rank of the node  $t$  in `Candidates[g, T]`). Text similarity features are computed from similarities between  $g$  and  $t$ 's names, locations, descriptions, urls, and screen\_names.

`BestPrimitiveMatch[g, T]`: Given a  $G$  node  $g$ , returns the best match node from `Candidates[g, T]` ranked using the `PrimitiveMatch` function. Also returns the associated match score. For instance, say we have two candidates for  $g$ :  $s$  and  $t$ . If  $\text{PrimitiveMatch}[g, s] = 0.1$  and  $\text{PrimitiveMatch}[g, t] = 0.4$ ,  $\text{BestPrimitiveMatch}[g, T] = (t, 0.4)$ .

We will explain advanced graph features for comparing neighborhood sets in Section 4.1.

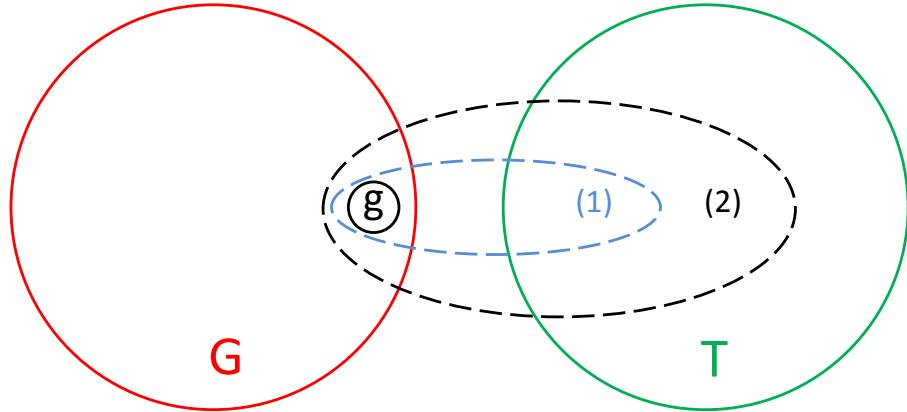


Figure 3: Exploration

<b>Primitive</b>
<b>Text</b>
<ol style="list-style-type: none"> <li>1. Name Similarity:  <math display="block">\max_{x \in g.names} StringSim(x, t.name)</math> </li> <li>2. Location Similarity:  <math display="block">\max_{x \in g.locations} StringSim(x, t.location)</math> </li> <li>3. Description Similarity:  <math display="block">CosineSim(g.descriptions, t.description)</math> </li> <li>4. Url Similarity: <math>1\{t.url \in g.urls\}</math></li> <li>5. Screen Name Similarity:  <math display="block">\max_{x \in g.screen_names} StringSim(x, t.screen_name)</math> </li> </ol>
<b>Position</b>
$\frac{\text{position}}{ \text{Candidates}[g, T] }$ <p>where <i>position</i> is the rank of <i>t</i> in <i>T</i> user search results</p>
<b>Graph</b>
<ol style="list-style-type: none"> <li>1. Weighted Jaccard Similarity:  <math display="block">(a) \frac{\text{Overlap}[g, t, G, T]}{ \text{TOut}[g, G, T].scores }</math> <math display="block">(b) \frac{\text{Overlap}[g, t, G, T]}{ t.out }</math> </li> <li>2. Known Overlap Score (<math>s_k</math>):  <math display="block">\frac{\text{KnownOverlap}[g, t, G, T] + 1}{ \text{TOut}[g, G, T].scores  + \text{KnownNotOverlap}[g, t, G, T] + 1}</math> </li> <li>3. Unknown Overlap Score (<math>s_u</math>):  <math display="block">\frac{\text{KnownOverlap}[g, t, G, T] +  \text{UnknownOut}[t, T]  + 1}{ \text{TOut}[g, G, T].scores  +  \text{UnknownOut}[t, T]  + 1}</math> </li> </ol>

Table 4: Features for computing  $\text{Match}[g, t]$

## 4 Strategy

In this section we explain the motivation behind the experiments in the next section and provide a high-level intuition of our solution. First we need to define

*discovered nodes*. Discovered nodes are nodes we found in our target network  $T$  through `Candidates[g, T]`. These nodes may refer to nodes fully crawled, partially crawled, or not crawled at all. There are two aspects of our strategy: exploration and evaluation.

- **Exploration:** Given budget  $B$  of additional API calls we can make, how do we expand the the universe of discovered nodes? In Figure 3, we expand our knowledge of  $T$  from (1) to (2) after an exploration step.
- **Evaluation:** Given a set of discovered nodes, what is our best guess for the corresponding node for  $g$  in  $T$ ?

Our strategy repeats these two steps until our budget is exhausted as shown in Figure 4.

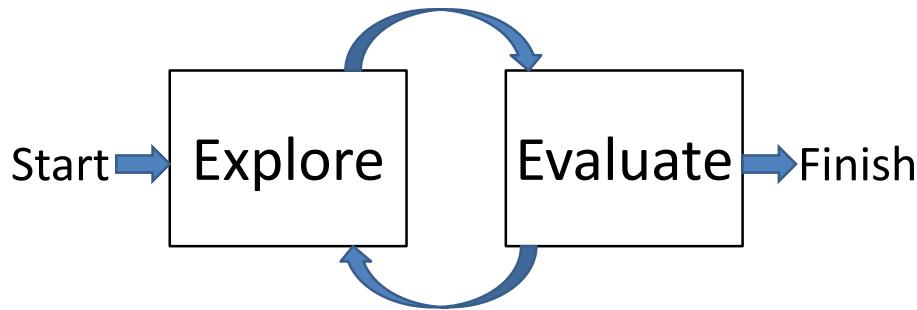


Figure 4: Iteration Workflow

#### 4.1 Evaluation

The evaluation step computes the scores of discovered nodes in our current state (our knowledge of  $T$ ). For each discovered node  $t$ , we compute a score  $\text{Match}[g, t]$ . The best candidate is the one with the highest score.

Recall from Section 3.6 that we have two match functions: `PrimitiveMatch` and `AdvancedMatch`. First, we define the auxiliary functions that construct `AdvancedMatch`, then we define `AdvancedMatch`.

- $\text{TOut}[g, G, T]$ : Given a  $G$  node  $g$ , returns a hash table of (node, score) pairs  $\{\text{BestPrimitiveMatch}[x, T] \text{ for } x \in g.out\}$ . In other words, it returns the best guess and the associated match score for nodes in  $T$  corresponding to nodes in  $g.out$  predicted by the `BestPrimitiveMatch` function.

Recall from our running example in Figure 2 that  $g.out = \{h, i\}$ . From Figure 5,  $\text{PrimitiveMatch}[h, p] = 0.8$  and  $\text{PrimitiveMatch}[h, q] = 0.2$ . Therefore, our best guess for nodes in  $T$  corresponding to  $g.out$  are  $p$  and  $r$  with match scores 0.8 and 1.0 respectively.  $\text{TOut}[g, G, T] = \{(p, 0.8), (r, 1.0)\}$ .

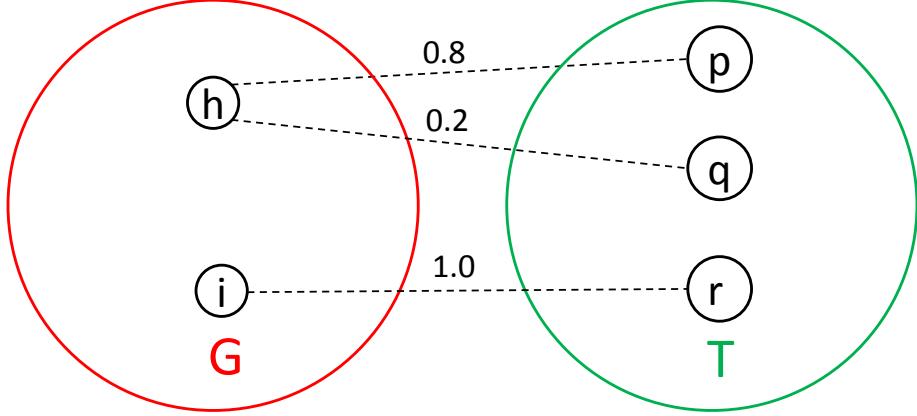


Figure 5: Illustration for  $\text{TOut}$

Let us use the following notation for accessing elements in a hash table. Given a hash table  $H$  of (key, value) pairs,  $H[k]$  returns a value for the given key  $k$ . Let  $H.\text{nodes}$  be the set of all keys in  $H$  and  $H.\text{scores}$  be the list of all values in  $H$ . In our example,  $\text{TOut}[g, G, T][p] = 0.8$ ,  $\text{TOut}[g, G, T][r] = 1.0$ ,  $\text{TOut}[g, G, T].\text{nodes} = \{p, r\}$ , and  $\text{TOut}[g, G, T].\text{scores} = [0.8, 1.0]$ .

- **Overlap**[ $g, t, G, T$ ]: Given a  $G$  node  $g$  and a  $T$  node  $t$ , **Overlap** first computes the intersection of  $\text{TOut}[g, G, T].\text{nodes}$  and  $t.\text{out}$ . Then **Overlap** returns the sum of  $\text{TOut}$  scores associated with the nodes in the intersection:

$$\text{Overlap}[g, t, G, T] = \sum_{i \in \text{TOut}[g, G, T].\text{nodes} \cap t.\text{out}} \text{TOut}[g, G, T][i]$$

We want to compute the similarity between the neighborhood sets ( $g.\text{out}$  and  $t.\text{out}$ ). Using the simple Jaccard Similarity ignores the weights for one of the sets (namely  $\text{TOut}[g, G, T]$ ). Therefore, our solution is **Overlap**. We use **Overlap** to define Weighted Jaccard Similarity described in the Graph Features section of Table 4. Intuitively,  $WJ(a)$  captures  $F^T \subseteq t.\text{out}$  while  $WJ(b)$  captures  $t.\text{out} \subseteq F^T$ . This is similar to Fuzzy Jaccard Similarity [21] but we split the feature into two to let the logistic regression classifier learn the weights.

Since fully crawling  $T$  is expensive, we may not have the full set of  $t.\text{out}$ . In the next section, we will develop overlap scores that take into account the fact that we have only partially crawled  $t.\text{out}$ .

- **KnownOut[t, T]**: Given a  $T$  node  $t$ , returns ids of nodes in  $T$  that  $t$  follows (i.e.,  $t$ 's outlinks), based on an incomplete/partially-crawled  $T$ .
- **UnknownOut[t, T]**: When we have an incomplete  $T$ , there may exist some outlinks of  $t$   $t.out$  that we do not know. Let us call this set  $\text{UnknownOut}[t, T]$ . Although we do not know  $\text{UnknownOut}[t, T]$ , we can compute  $|\text{UnknownOut}[t, T]|$  as follows:  

$$|\text{UnknownOut}[t, T]| = t.outdegree - |\text{KnownOut}[t, T]|.$$
- **KnownOverlap[g, t, G, T]**: Given a  $G$  node  $g$  and a  $T$  node  $t$ , returns the weighted intersection of  $\text{TOut}[g, G, T].nodes$  and  $\text{KnownOut}[t, T]$ . Note that this is similar to  $\text{Overlap}[g, t, G, T]$  but replaces  $t.out$  with  $\text{KnownOut}[t, T]$  because we do not have the complete outlinks of  $t$ .
- **KnownNotOverlap[g, t, G, T]**: Given a  $G$  node  $g$  and a  $T$  node  $t$ , returns the number of nodes in  $\text{KnownOut}[t, T]$  but not in  $\text{TOut}[g, G, T].nodes$  (i.e.,  $|\text{KnownOut}[t, T] - \text{TOut}[g, G, T].nodes|$ ).
- **AdvancedMatch[g, t]**: a Match function that computes features from both local profile attributes and graph features. In addition to primitive features  $\text{PrimitiveMatch}[g, t]$ , we use Weighted Jaccard Similarity ( $WJ$ ) as well as Known and Unknown Overlap Scores ( $s_k$  and  $s_u$  respectively) described in Table 4.  $\text{AdvancedMatch}$  is a linear combination of these scores.

The intuition behind  $s_k$  and  $s_u$  is as follows. Say we are considering two candidates to match  $g$ :  $s$  and  $t$  as in Figure 2 with outdegrees of 5 and 10 respectively. Let  $F^T$  be  $\text{TOut}[g, G, T].nodes$ . Suppose the number of  $s$  and  $t$ 's outlinks that overlap with  $F^T$  are 1 and 10 respectively. If  $\text{PrimitiveMatch}[g, s] > \text{PrimitiveMatch}[g, t]$ , we will choose to crawl  $s$  to retrieve its outlinks. If the first outlink returned overlaps with  $F^T$ ,  $\text{Overlap}[g, s, G, T]$  will be greater than 0 and therefore Weighted Jaccard Similarity ( $WJ$ ) will be positive.  $WJ$  for  $t$  is 0 because we have not crawled  $t$ . Thus  $\text{AdvancedMatch}[g, s]$  will always be greater than  $\text{AdvancedMatch}[g, t]$ . Based on  $\text{AdvancedMatch}$ , we will continue to expend API calls to explore outlinks of  $s$  even if all subsequent outlinks do not overlap. These calls would be better utilized to explore  $t$  if  $s$  starts to look unpromising.

Let us summarize the desired behaviors of  $\text{AdvancedMatch}$ . At the beginning of our match task when  $t$  has the most unknown links, we want to boost  $\text{AdvancedMatch}[g, t]$ . As we explore  $t$ 's outlinks and find that all of them overlap with  $F^T$ , we want to keep  $\text{AdvancedMatch}[g, t]$  high so that we will continue to favor crawling  $t$  over  $s$ . On the other hand, as we explore  $s$ 's outlinks and find that only one of them overlaps with  $F^T$ , we want to decrease  $\text{AdvancedMatch}[g, s]$  and switch to crawling  $t$  instead. Next, we demonstrate with an example that the scores  $s_k$  and  $s_u$  achieve all of these desired behaviors.

Before we provide an example of Figure 6(a), let us first explain notations used in Figure 6. We have three types of nodes: check, cross, and question

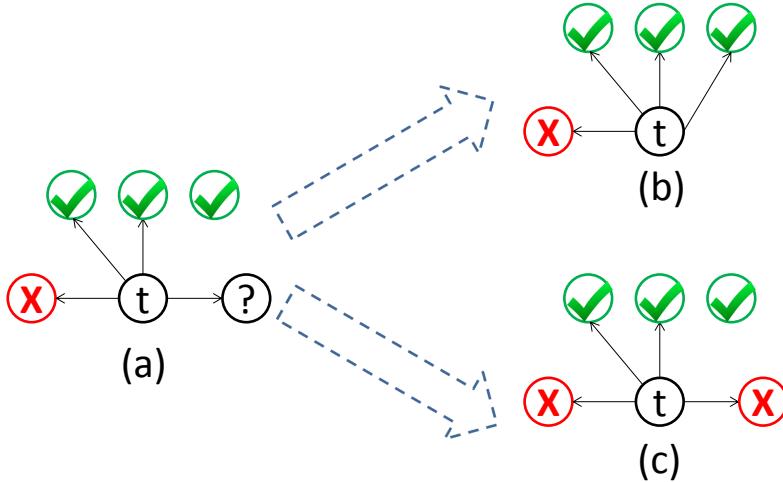


Figure 6: Diagram Explaining  $s_k$  and  $s_u$

mark. Check nodes represent nodes in  $F^T$  (i.e.,  $\text{TOut}[g, G, T].\text{nodes}$ ). Cross nodes represent nodes not in  $F^T$ . Question mark nodes are unknown (can be either check or cross but we have not yet crawled it). In Figure 6(a), we have crawled three outlinks of  $t$  and there is one unknown outlink. Two out of three outlinks overlap with  $F^T$ . The unknown outlink may or may not overlap with  $F^T$ .

Let's work out a simple example to calculate  $s_k$  and  $s_u$  before and after retrieving a link. Let's assume we know the exact mapping of each node in  $g.\text{out}$  onto  $T$ . In reality, we usually are not certain about the mapping and we have to include match scores in our calculation. In Figure 6, recall that we have three types of nodes: check, cross, and question mark. Check nodes represent nodes in  $F^T$ . Cross nodes represent nodes not in  $F^T$ . Question mark nodes are unknown. We start from (a) and by crawling the question mark node, we either go to (b) or (c). Case (b) is when the new link points to a check node. Case (c) is when the new link points to a cross node. We will walk through the calculation for all cases (a), (b), and (c).

Case (a): There are three check nodes, translating to  $|\text{TOut}[g, G, T]| = 3$ .  $t$  links to two check nodes, one cross node, and one question mark node. This translates to  $\text{KnownOverlap}[g, t, G, T] = 2$ ,  $\text{KnownNotOverlap}[g, t, G, T] = 1$  (the number of cross nodes in discovered  $t.\text{out}$ ), and  $|\text{UnknownOut}[t, T]| = 1$ . We have defined all of the components to compute  $s_k$  and  $s_u$ .  $s_k = \frac{2+1}{3+1+1} = \frac{3}{5}$  and  $s_u = \frac{2+1+1}{3+1+1} = \frac{4}{5}$ .

Case (b): There are three check nodes, translating to  $|\text{TOut}[g, G, T]| = 3$ .  $t$  links to three check nodes, one cross node, and no question mark nodes. This translates to  $\text{KnownOverlap}[g, t, G, T] = 3$ ,  $\text{KnownNotOverlap}[g, t, G, T] = 1$ , and  $|\text{UnknownOut}[t, T]| = 0$ . Therefore,  $s_k = \frac{3+1}{3+1+1} = \frac{4}{5}$  and

$$s_u = \frac{3+0+1}{3+0+1} = 1.$$

Case (c): There are three check nodes, translating to  $|TOut[g, G, T]| = 3$ .  $t$  links to two check nodes, two cross nodes, and no question mark nodes. This translates to  $KnownOverlap[g, t, G, T] = 2$ ,  $KnownNotOverlap[g, t, G, T] = 2$ , and  $|UnknownOut[t, T]| = 0$ . Therefore,  $s_k = \frac{2+1}{3+2+1} = \frac{1}{2}$  and  $s_u = \frac{2+0+1}{3+0+1} = \frac{3}{4}$ .

Let  $s_k(a)$  be  $s_k$  for (a) (and the same notation for  $s_u$ ). We can see that  $s_k(b) > s_k(a)$  and  $s_u(b) > s_u(a)$  because we have discovered an additional check node. On the other hand,  $s_k(a) > s_k(c)$  and  $s_u(a) > s_u(c)$  because we have discovered an additional cross node.

Note that we add 1 to both the numerator and the denominator of  $s_k$  and  $s_u$  to prevent the scores from being zero when there is no overlap or being infinity when the denominator is zero. It is useful for the score to incorporate how many links we have retrieved for each candidate even when there is no overlap so that we can prioritize among which candidate to crawl next. Say we have two candidates and we have found no overlap in either of them yet, i.e.,  $KnownOverlap[g, t, G, T] = 0$ . We would prefer to explore the candidate that we have explored less (lower  $|KnownOut|$ ) and that has more unknown links (higher  $|UnknownOut|$ ).

## 4.2 Exploration

We can view the budget  $B$  as a single value or a vector  $[b_1, b_2, b_3, \dots, b_n]$  with a budget value for each type of API call (e.g.,  $b_1$  for `users/search`,  $b_2$  for `friends/list`). For a single value budget, we must decide which type of calls to make. For example, we have a budget of 10 calls, we have to split these 10 calls among all types of calls.

In our case, we focus on a vector budget because it represents the Twitter API. For each type of API call, we must decide on which nodes we want to spend our calls. For example from Figure 2, if we have one additional `users/search` call, do we choose to find more candidates for  $h$  or for  $i$ ? As another example, if we have one additional `friends/list` call, do we choose to find more outlinks for  $s$  or for  $t$ ?

Our solution will be a collection of heuristics because it is too difficult for exact answers. Based on a set of experiments, we will provide insights into how to order nodes to make API calls. For example, through our experiments we will learn that we should sort  $h$  and  $i$  in ascending order based on their indegree and then make `users/search` calls in this order.

## 5 Preliminary Experiments

We explained why we conduct our experiments in the previous section. We will split our experiments into two parts: the preliminary experiments and the evaluation experiments (see Section 7). Here we present preliminary experiments

to understand the characteristics of our dataset and then provide a collection of heuristics for our exploration algorithm.

## 5.1 Dataset

### 5.1.1 Training Set

For our training set GTR, we will use a set of Google+ users with known matching Twitter accounts. To obtain GTR, we started by crawling Sergey Brin’s inlinks (followers) on Google+. Sergey Brin is a co-founder of Google who has 3.9 million inlinks on Google+ [10]. Only 9,727 inlinks are visible to a Web browser on his profile page. We crawled these profiles and discovered 532 users who put their Twitter screen\_name on their Google+ profile page (nonempty `twitter` attribute). We call this initial set X.

We would like to restrict our attention to “findable” Twitter users. We define “findable” as users for which the Twitter search interface is able to locate the screen\_name based on the user’s `fullname` on Google+. Recall that screen\_name is an identifier of a user. We use `fullname` rather than `screen_name` because when we evaluate our strategies on our test set, we will not have `screen_name`. We would like to train our strategies using only available information in our test set. In some cases, a Twitter screen\_name may not actually exist on Twitter because it has been changed, suspended, or deleted. Another issue is when the Twitter search interface is unable to locate the screen\_name based on user’s `fullname` on Google+. Thus, we removed users that are not findable.

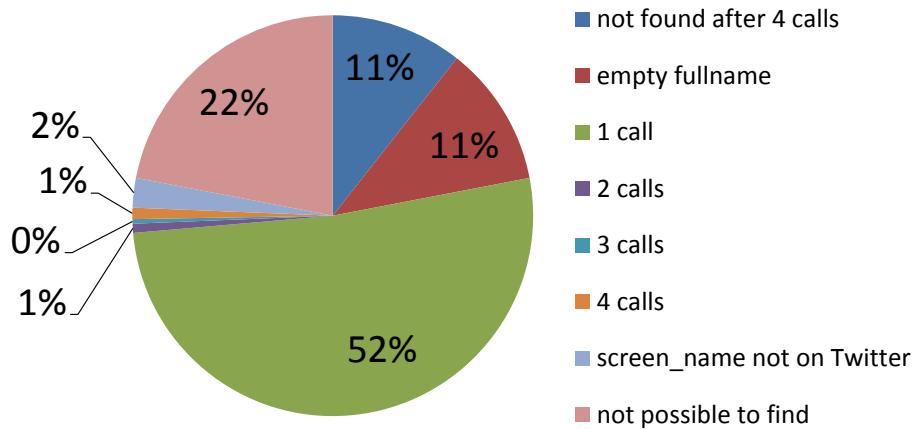


Figure 7: Number of `users/search` calls vs. candidates discovered for GTR

To discover the findable users, we examined all X users and grouped them into 8 buckets, as shown in Figure 7. The bucket “`screen_name not on Twitter`” contains users with changed, suspended, deleted screen\_names (discovered using the lookup interface, `users/lookup`). For the remaining users, we searched in Twitter (using `users/search` calls) using only their `fullname` (if there is no

`fullname`, it falls in the bucket “empty fullname”). The bucket “1 call” contains the users whose screen\_name was found in the results of the first `users/search` call (recall that each call could return up to 20 results). The bucket “2 calls” contains the users whose screen\_name appeared in the results of the second call. The bucket “3 calls” and “4 calls” are analogous. The bucket “not possible to find” contains users such that (a) we exhausted the results from the Twitter search interface after up to 4 calls, and (b) the screen\_name did not appear. The final bucket “not found after 4 calls” contains the remaining X users. These users might be findable with more than 4 calls or not findable at all.

Figure 7 shows the percentage of users in each bucket. We are not going to include in our GTR the users in the buckets “not found after 4 calls,” “empty fullname,” “screen\_name not on Twitter,” and “not possible to find”, representing  $11+11+2+22=46\%$  of the X users. Note that users in the bucket “not found after 4 calls” might actually be findable with more than 4 calls but we believe the number of such users is insignificant. Thus, the remaining 4 buckets (1-4 calls) will make up our GTR and constitute a total of 293 users. The vast majority of our GTR users (96.25%) can be found within one call. This fact will be exploited by one of our heuristics.

Incidentally, although not shown in Figure 7, out of 120 users in the bucket “not possible to find”, we know that 87.5% of them are not findable after just one call. That is the first call returned fewer than 20 results and the matching screen\_name did not appear.

### 5.1.2 Testing Set

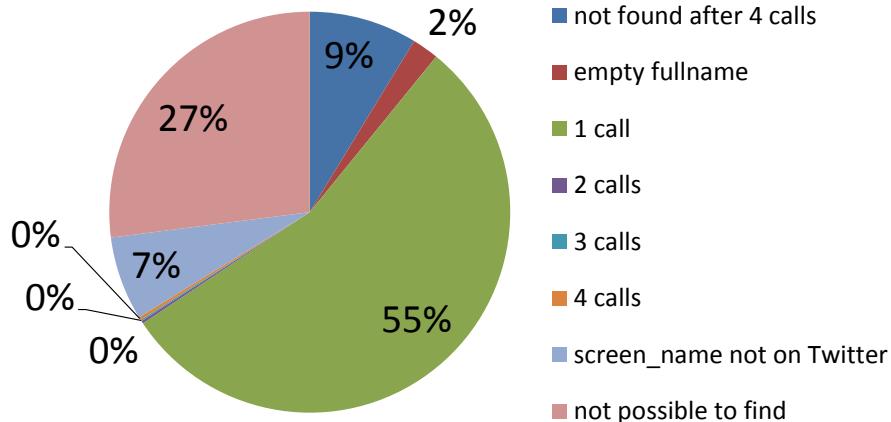


Figure 8: Number of `users/search` calls vs. candidates discovered for GTE

For our testing set GTE, we would like a set of Google+ users with known matching Twitter users. To obtain GTE, we started by crawling Google+ users listed on [recommendedusers.com](#), an unofficial website (not affiliated with Google) that recommends Google+ users to follow. The site provides various

categories based on different topics of interest. We chose to crawl the “People’s Choice” category because the users listed are chosen by the Google+ community based on “[sharing] the most interesting and valuable content on Google+” [14]. Furthermore, these users are very diverse (photographers, writers, web celebrities, entrepreneurs, and artists) and we would like to crawl an unbiased sample of Google+ users. We crawled users who follow at least one “People’s Choice” user and discovered 4381 users who put their Twitter screen\_name on their Google+ profile page (nonempty `twitter` attribute). We call this initial set  $X$ .

Similar to constructing GTR, we removed users that are not findable from  $X$ . Figure 8 shows the percentage of users in each bucket. Our GTE comes from 4 buckets (1-4 calls) and constitutes a total of 2425 users. The vast majority of our GTE users (98.97%) can be found within one call.

## 5.2 Caching

Since we will be frequently accessing GTR and GTE users, we decided to do a full crawl of their Twitter network. That is for each user, we first made `users/search` calls, and then we fully crawled the outlinks of each candidate returned by `users/search`. We use this cached data to simulate API calls for our experiments so we do not have to invoke API calls on Twitter and wait each time we hit their rate limit.

## 5.3 Selectivity

Selectivity of an attribute  $a$  of graph  $G$  is defined as the fraction of nodes for which  $a$  is not missing (and privacy settings set to public for  $a$ ) over the number of nodes in  $G$ .

### 5.3.1 Google+

We constructed our sample of Google+ profiles from GTE. We collected the profiles of “People’s Choice” users and their inlinks. For each of the inlinks, we also collected the profiles of their outlinks. As a result, we collected 5 million Google+ user profiles.

We display the selectivity of attributes in Figure 9. We can see that the network information is usually available (public): 84% of the sampled users are following someone (`outdegree > 0`) and almost 86% of the sampled users have at least one follower (`indegree > 0`). Other textual attributes are significantly more selective: 61% for `descriptions` and below 40% for other attributes. Thus, our algorithm should exploit link attributes because they are usually available even in the absence of textual attributes. We also note on the plot that the selectivity for `twitter` is only 14% which means that people typically do not put their Twitter screen\_name on their Google+ profile and that our match task is useful.

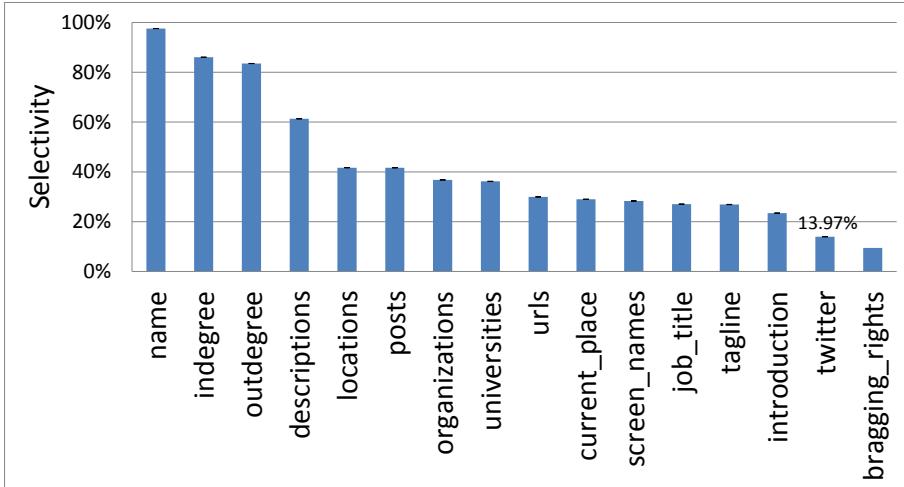


Figure 9: Selectivity of Google+ Attributes

### 5.3.2 Twitter

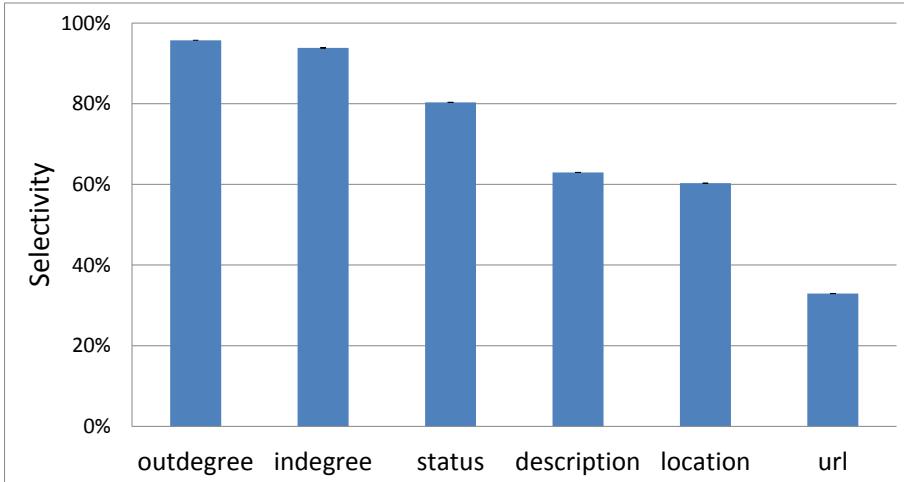


Figure 10: Selectivity of Twitter Attributes

First we explain how we sampled Twitter users. For each user  $u$  in GTE, we made a `users/search` call to get a set of candidates for  $u$ . For each candidate  $t$ , we crawled  $t.out$  and collected their profiles. As a result, we collected 16 million Twitter user profiles. We believe that our sample size is significant because it is 8% of 200 million, which is the number of monthly active Twitter

users [23].

We display the selectivity of Twitter attributes in Figure 10. We can see again that the link attributes are almost always available (more than 90% of the time). Other textual attributes like location and description are only available for roughly 60% of the users, and url selectivity is even lower. The difference in selectivity between link and textual attributes suggests that we should exploit the link information in our algorithm.

Interesting statistics about the Twitter following graph are:

- 96% of Twitter users are following someone (`outdegree > 0`).
- 17% of Twitter users are following from 1 to 20 people ( $0 < \text{outdegree} < 21$ ). The maximum number of profiles returned by a `friends/list` call is 20.
- 36% of Twitter users are following between 1 and 100 people ( $0 < \text{outdegree} < 101$ , five `friends/list` calls required).
- 1% of Twitter users are following more than 5,000 people (maximum number of ids returned by a `friends/id` call).
- 7% of Twitter users are protected (we cannot crawl inlinks and outlinks).

Therefore, we will exploit following connections (outlinks) on both networks to help us resolve a match task. We will focus on crawling outlinks rather than inlinks because people usually have lower outdegrees than indegrees. Crawling inlinks will exhaust the Twitter limit fast. Furthermore, each node explicitly generates the outlink connection whereas inlinks are generated from other users.

## 5.4 Degree Distributions

We use the same sample as in Section 5.3 to generate our degree distributions.

### 5.4.1 Google+

From Figure 11, both degree distributions (indegree and outdegree) follow power law distributions with a slope of 1.76. The tail of the outdegree distribution is cut off at around 5,000 which is the limit imposed by Google [7]. In our sample of 5 million Google+ profiles, there are seven users with an outdegree greater than 5,000. There is only one user with an outdegree above 10,000 and his outdegree was 10,477 at the time of our crawl. Recall that we can collect only up to 10,000 social connections visible to a Web browser on a profile page. This outdegree distribution verifies that our assumption that we have the complete following graph is mostly accurate. On the other hand, the indegree distribution has a much longer tail. The user with the largest indegree at the time of our crawl was Lady Gaga who had 5.6 million followers.

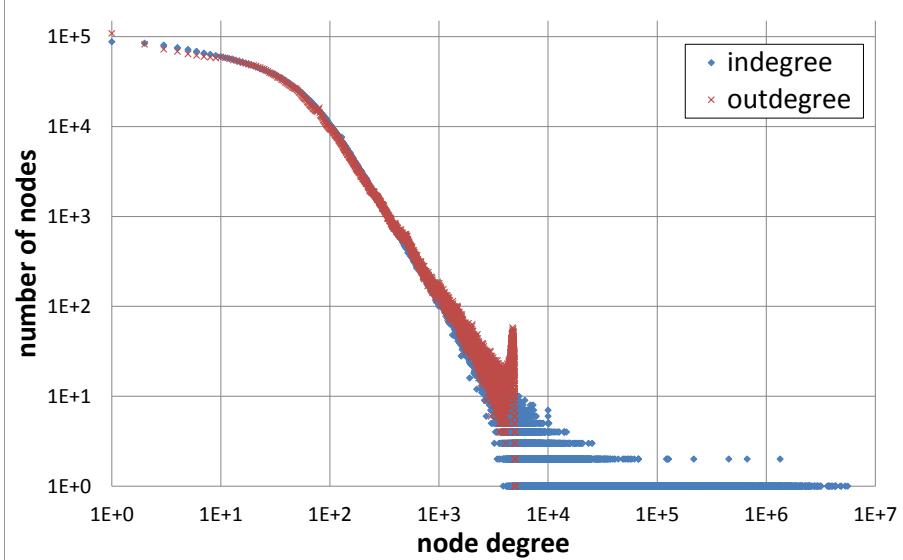


Figure 11: Google+ Degree Distribution (log-log scale)

#### 5.4.2 Twitter

In Figure 12, we can see that indegree and outdegree distributions follow power law distributions with slopes of 1.63 and 1.725 respectively. The outdegree distribution shows a huge spike at degree 2,000. This is the following limit Twitter imposes to prevent follow spam. A user can follow more people if he/she has more followers. If a user has outdegree more than 2,000, for every 10 followers he/she has, he/she can follow 11 people. For example, if you have 2,000 followers, you can follow 2,200 people [13]. We can see that the outdegree distribution has a steeper slope than the indegree distribution. Furthermore, the indegree distribution has a much longer tail. The user with the largest indegree at the time of our crawl was Lady Gaga who had 37.4 million followers. Therefore, we should focus on crawling outlinks rather than inlinks to prevent us from quickly hitting the limit.

### 5.5 Experiments

In all of our experiments, we first train our strategy using GTR and then use GTE for evaluation. Recall from Figure 2 that  $C_T$  represents the number of candidates we obtain for matching a given target. We will set  $C_T$  to 20 because almost 99% of our GTE users can be found within one `users/search` call (each call returns up to 20 users).

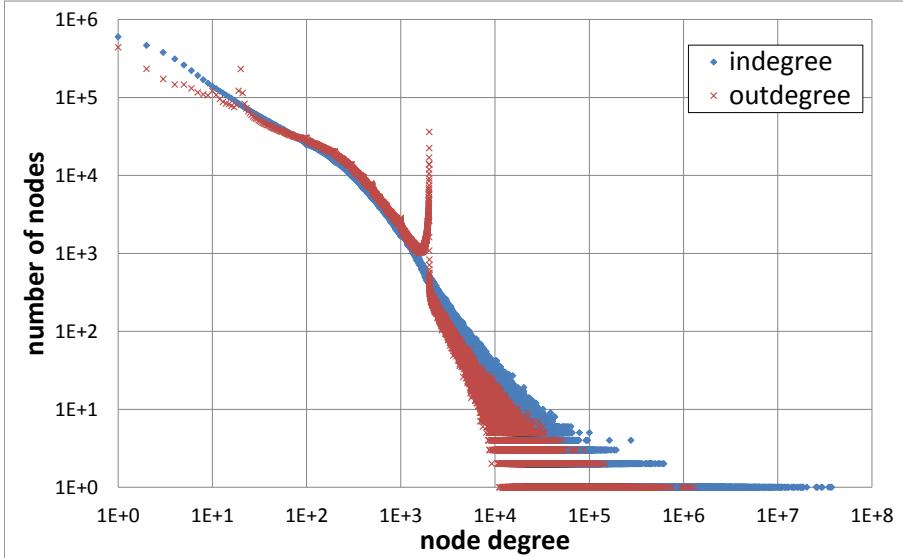


Figure 12: Twitter Degree Distribution (log-log scale)

### 5.5.1 PrimitiveMatch

Our very first experiment is to evaluate the performance of using the primitive features (explained in Table 4) alone. We will use precision as a metric to evaluate **PrimitiveMatch**. We define  $precision@k$  as the precision of a strategy at the top  $k$  candidates. Precision in our context is defined as the fraction of the match tasks where the top  $k$  candidates returned by a strategy contain the matching node. The matching node for a given target node  $g$  in  $G$ , is the node that represents the same real-world entity as  $g$  in network  $T$ .

Figure 13 shows how precision increases as  $k$  increases. The vertical error bars represent 95% confidence intervals. Let us consider  $n$  trials with  $s$  successes. For a 95% confidence interval, we have an error percentile  $\alpha$  of 5%. The confidence intervals are computed using Wald interval for Bernoulli trials: the formula is  $p \pm k\sqrt{p(1-p)/n}$  where  $p = s/n$  and  $k = \Phi^{-1}\{1 - \alpha/2\}$  ( $\Phi^{-1}$  is a probit function) [24]. This technique approximates the error distribution using a normal distribution. We can see that it is statistically significant that precision increases when  $k$  increases. For example,  $precision@3$  is about 94% for the **PrimitiveMatch** strategy. This means that 94% of the time we ran our matching task, the matching node was in the top 3 ranked by match score. We learn that 83% of the time, **PrimitiveMatch** returns the matching node at the top. Therefore, we will consider this value our baseline. Without using any graph feature, we can achieve 83%  $precision@1$ .

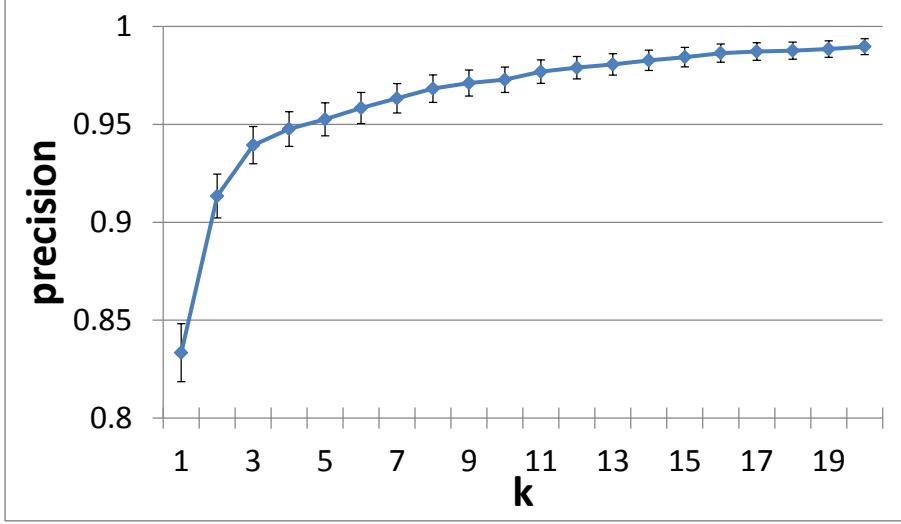


Figure 13: Precision@k, PrimitiveMatch, 95% Confidence

### 5.5.2 Thresholds

Figure 2 shows two thresholds that we would like to study: **1)**  $C_F$  the number of candidates we consider a potential match for each of  $g.out$ , and **2)**  $F$  the fraction of outlinks  $g.out$  that we crawl. We would like to understand how varying thresholds  $C_F$  and  $F$  affects the precision of our match task. This study will help us shape our heuristics for making API calls (Section 6) since we have limited resources and cannot set these thresholds to maximum.

We assume that we have complete information about links in network  $T$ . We will discuss a situation with an incomplete network  $T$  in Section 5.5.3.

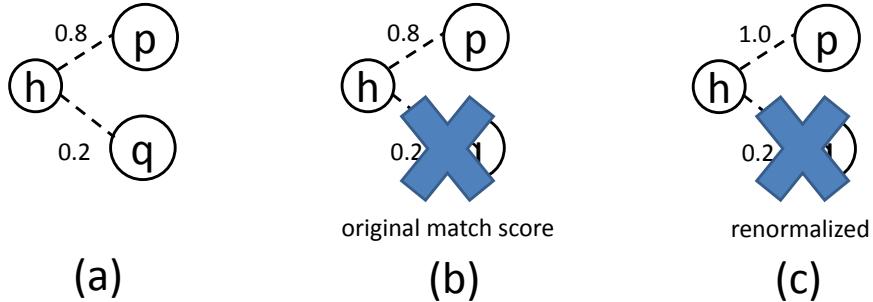


Figure 14: Renormalizing Match Scores

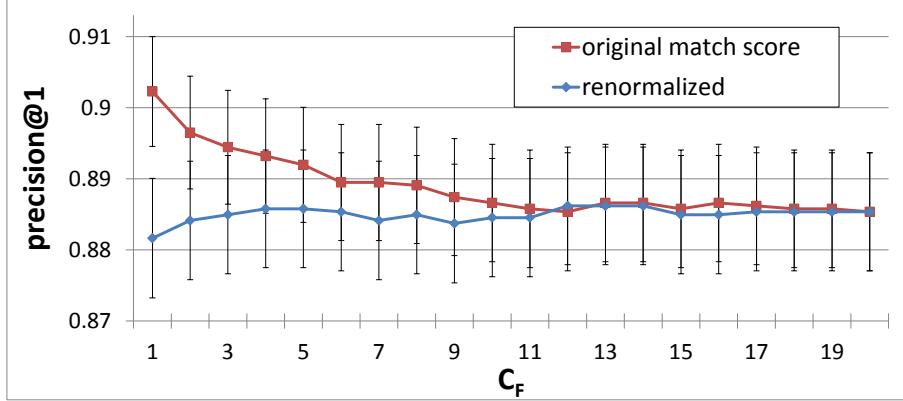


Figure 15:  $C_F$  vs.  $precision@1$ , 80% Confidence

$C_F$

Let us set the threshold  $F$  to the maximum value of 1.0 (fully crawl  $g.out$ ) and then vary  $C_F$ . Threshold  $C_F$  is the number of candidates kept for each of the outlinks of the target node. We prune candidates based on their match scores from `PrimitiveMatch`. The candidates we kept are used to compute graph features used in `AdvancedMatch`. The more candidates we prune, the fewer API calls we need.

To illustrate, say  $h \in g.out$ , and we have two candidates for  $h$ :  $p$  and  $q$  where  $\text{PrimitiveMatch}[h, p] = 0.8$  and  $\text{PrimitiveMatch}[h, q] = 0.2$  (See Figure 5). When  $C_F = 2$ , we keep both candidates  $p$  and  $q$ , the two candidates with the highest scores. When  $C_F = 1$ , we keep only  $p$ , the single candidate with the highest score. When we prune candidate  $q$ , we have two options for the match score of  $p$ : **1)** keep the original match score as in Figure 14(b) or **2)** renormalize the score so that the sum of the match scores of the candidates for  $h$  is 1 as in Figure 14(c).

To compare these options, we study them and present the results in Figure 15. Figure 15 shows how  $precision@1$  of both strategies (original match score vs. renormalized) varies as we increase  $C_F$ . We run each strategy on all nodes in GTE and  $C_F \in [1, 20]$ . For each  $g$  node, we first make  $C_F$  `users/search` calls. After we have retrieved all outlinks of each candidate ( $x.out$  for  $x \in \text{Candidates}[g, T]$ ) we rank them using `AdvancedMatch` and current knowledge of  $T$ , and select the top node. The y-axis shows  $precision@1$ : the fraction of cases where the selection was indeed the correct matching  $T$  node for  $g$ . The vertical error bars represent 80% confidence intervals.

In Figure 15, if we keep the original match scores,  $precision@1$  starts at 90% for  $C_F = 1$  and decreases as we increase  $C_F$ . The drop in precision as  $C_F$  increases is counterintuitive since we expect the rise when we keep more candidates. Recall from Figure 13 that `PrimitiveMatch` achieves 83%  $precision@1$ .

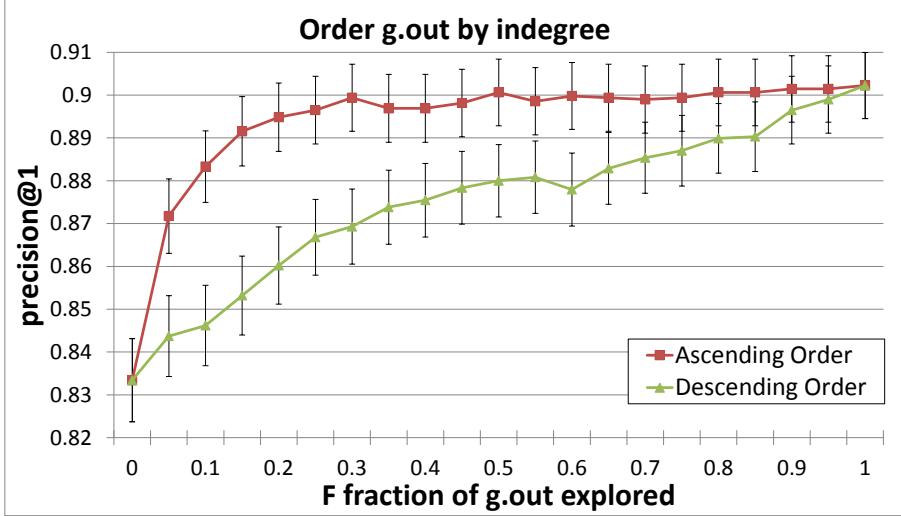


Figure 16:  $F$  vs.  $precision@1$ , Ascending vs. Descending Order, 80% Confidence

and the improvement in precision for higher  $k$  values is marginal. Therefore, keeping more candidates actually confuses `AdvancedMatch` and hurts precision. On the other hand, if we renormalize the match scores,  $precision@1$  increases slightly as we increase  $C_F$  and then stabilizes at 88.5%. It is easy to see that keeping the original match scores consistently outperforms renormalizing them and the highest precision is when we keep only the top candidate (i.e.,  $C_F = 1$ ). We will therefore continue using  $C_F = 1$  in all subsequent experiments.

#### $F$ and outlink crawling order

Let us set  $C_F$  to 1 and then vary  $F$ . Recall from Figure 2 that  $F$  is the fraction of  $g.out$  to crawl (i.e., invoke a `users/search` call to find a matching node on  $T$ ). For example, in Figure 2, if  $F = 0.5$ , we only crawl 50% of  $g.out$ , i.e., we crawl either  $h$  or  $i$ . As  $F$  increases, we require more `users/search` calls and in most cases we do not have enough resources to set  $F = 1.0$ .

In this section, we will discuss how to rank outlinks  $g.out$  to maximize precision. Figure 16 shows how  $precision@1$  increases as  $F$  increases when we order  $g.out$  by indegree. When we compare descending and ascending orders, we see that ascending order consistently outperforms descending order with 80% confidence. Furthermore, adding graph features (using `AdvancedMatch`) helps boost precision over `PrimitiveMatch`: crawling only 5% of  $g.out$  increases  $precision@1$  from 83% to 87%. Lastly, precision reaches 90% and stabilizes around  $F = 0.3$ . Thus crawling only 30% of  $g.out$  is sufficient to reach the maximum precision.

In Figure 17, we compare different schemes for ordering outlinks: by inde-

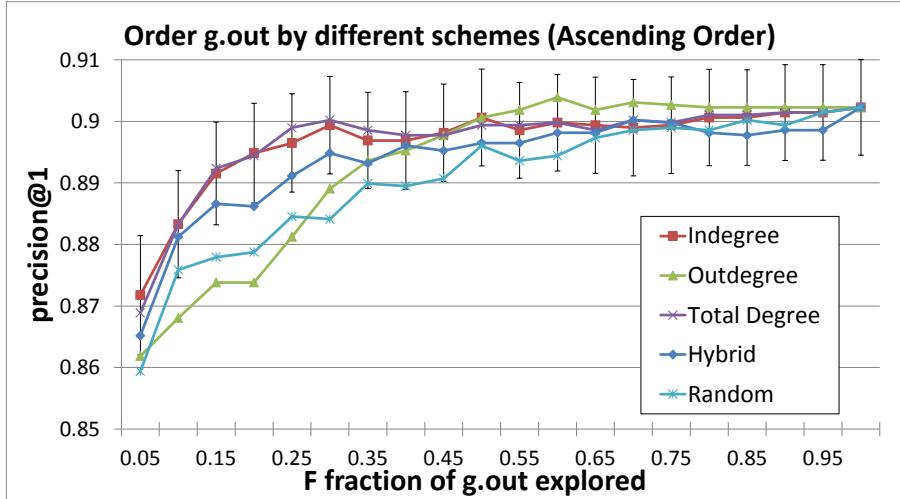


Figure 17: Comparing Different Schemes, 80% Confidence

gree, outdegree, total degree, hybrid, and random, all in ascending order. Total degree is the sum of indegree and outdegree. In the hybrid scheme, we take half of the outlinks to crawl from the indegree-ascending outlinks and the other half from the outdegree-ascending outlinks. Note that when we merge the two lists of outlinks, they may overlap and we have to continue taking the next lowest degree from each list until we obtain the desired  $F$ . Figure 17 shows that most ordering schemes (except for ordering by outdegree) beat random for lower values of  $F$ . Although, for  $F > 50\%$ , all schemes converge to 90% precision, when  $F < 50\%$ , ordering  $g.\text{out}$  by indegree or total degree maximizes  $\text{precision}@1$ . These two orderings reach 90% precision when  $F$  is only 0.3 while random ordering requires  $F = 0.7$  to reach 90% precision.

Let us explain the intuition behind indegree and total degree schemes performing better than the outdegree scheme. We can consider indegree (number of followers) a good proxy for popularity. If  $g$  follows a user with low indegree (let us call this user 1) and a popular user like Lady Gaga, we can infer that  $g$ 's relationship is stronger with 1 than with Lady Gaga. Two random users are more likely to both follow Lady Gaga than to both follow 1. Thus, users with low indegrees are strong signals for distinguishing  $g$ 's true identity from other candidates.

Since the outdegree scheme performs poorly and indegree is a factor of the total degree scheme, we can conclude that the total degree performs well because of indegree. In summary, we learn that **1)** we should order outlinks by indegree in ascending order, and **2)** graph features help increase  $\text{precision}@1$  (from 83% to 90% for  $F = 0.3$ ).

### 5.5.3 Incomplete Network Information for $T$

In this section we study how to crawl outlinks in  $T$  (e.g., outlinks of  $s$  and  $t$  in Figure 2) when we have limited API calls during the exploration phase. For candidates  $x \in \text{Candidates}[g, T]$ , we have to decide the order in which retrieve candidate's outlinks. Recall from Table 3 that there are two operations for obtaining outlinks of a node:

`friends/list` and `friends/id`. In order to analyze the incremental improvement in precision, we will focus on the more restrictive operation `friends/list` where the maximum number of outlinks per API call is 20. Let us set  $C_F$  and  $F$  to 1 and 1.0 respectively, i.e., fully crawl `g.out` and keep the top candidate for each based on `PrimitiveMatch`.

We have explored several strategies to make API calls to obtain outlinks of `Candidates[g, T]`:

1. **Random:** choose a candidate at random.
2. **Round Robin Rerank:** first sort candidates based on `PrimitiveMatch`, then do a round of round robin (go through all sorted candidates, give one API call to each) and then rerank candidates based on the new information using `AdvancedMatch`. Repeat until exhaust all API calls.
3. **Round Robin Rerank Cutoff  $i$ :** same as Round Robin Rerank but the first step only keeps the top  $i$  candidates ranked based on `PrimitiveMatch`.
4. **Proportional:** split API calls proportionally to candidates based on `PrimitiveMatch`.
5. **Top:** first sort candidates based on `PrimitiveMatch`, then make one API call for the top candidate, rerank based on the new information using `AdvancedMatch`. Repeat until exhaust all API calls.

Figure 18 shows how  $\text{precision}@1$  increases as the number of API calls increases for different crawling strategies. Top is clearly the best strategy because it shows a statistically significant improvement in precision over all other strategies. For example, Top reaches 86% precision in 45 API calls while other strategies require at least 95 API calls to reach the same precision. Proportional and Round Robin Rerank perform roughly the same as Random.

Next we will evaluate round robin strategies with different cutoff values. Figure 19 shows how  $\text{precision}@1$  increases as we increase the number of API calls for the Top and different Round Robin Rerank strategies. Recall from Figure 13 that the top 3 and 5 candidates returned by `PrimitiveMatch` contain the matching node 94% and 96% respectively. Therefore, we will use 3 and 5 as the cutoff values when we evaluate round robin strategies. Figure 19 shows that smaller cutoff values ( $C_T$ ) improve precision when there are limited API calls. For example, cutoff value 3 achieves the same precision as Top when we have fewer than 40 API calls. As the limit on API calls grows, cutoff value 3 and no cutoff converge to the same 87% precision. Cutoff value 5 achieves lower precision than cutoff value 3 for limited API calls (up to 40 calls). As the limit on API calls increases, cutoff value 5 achieves higher precision than cutoff value 3. Thus we should use small cutoff values when we have limited API calls and increase the cutoff values as the limit on API calls increases.

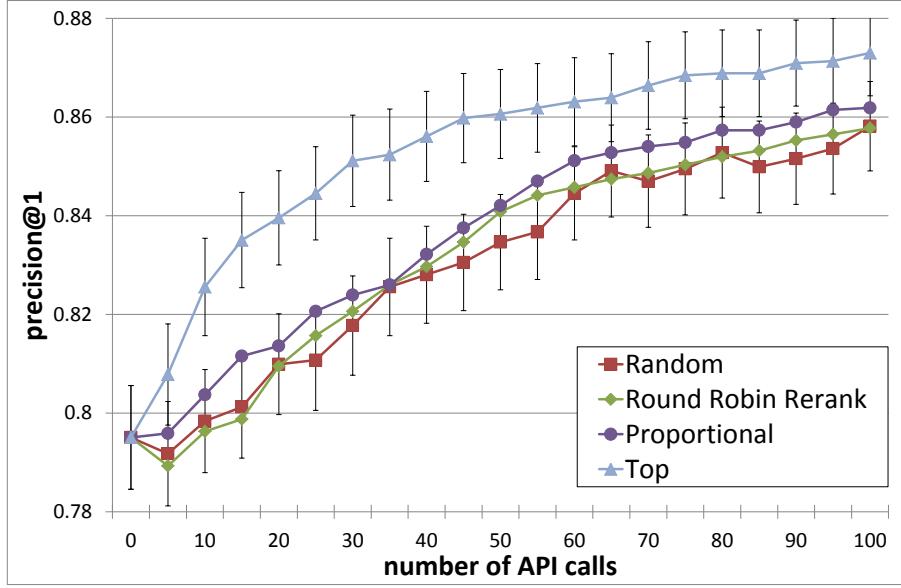


Figure 18: Comparing Outlink Crawling Strategies, 80% Confidence

Top outperforms Round Robin Rerank for all cutoff values. For example, Top reaches 87% precision in 90 API calls while Round Robin with Cutoffs require at least 120 API calls to reach the same precision. Thus we should use strategy Top to explore outlinks of  $\text{Candidates}[g, T]$ .

## 6 Algorithms

We will describe 5 exploration algorithms, 3 of them are baselines for comparison, and 2 are our contribution. The first 2 baselines are not new; the third baseline is a variation on one of our new algorithms.

**PRI:** We first describe our baseline algorithm for the exploration phase. Our baseline is **PrimitiveMatch** i.e., using primitive features only. This strategy only requires one `users/search` call to retrieve candidates. The algorithm returns **BestPrimitiveMatch** described in Section 3.6.2. Recall that **BestPrimitiveMatch** ranks candidates based on local information, namely the textual attributes and the position of the candidate ranked by Twitter. Recall from Section 5.5.1 that this baseline achieves about 83% precision. Let us call this strategy PRI.

**FULL:** We will start with an algorithm when we have full knowledge of all profiles in  $T$  and outlinks of each of candidate  $\in \text{Candidates}[g, T]$ , i.e., we know  $x.\text{out}$  for each  $x \in \text{Candidates}[g, T]$ . We present the strategy for this scenario in Algorithm 1 (let us call it FULL). The intuition for FULL is that matching nodes should not only have similar node properties but should also have similar

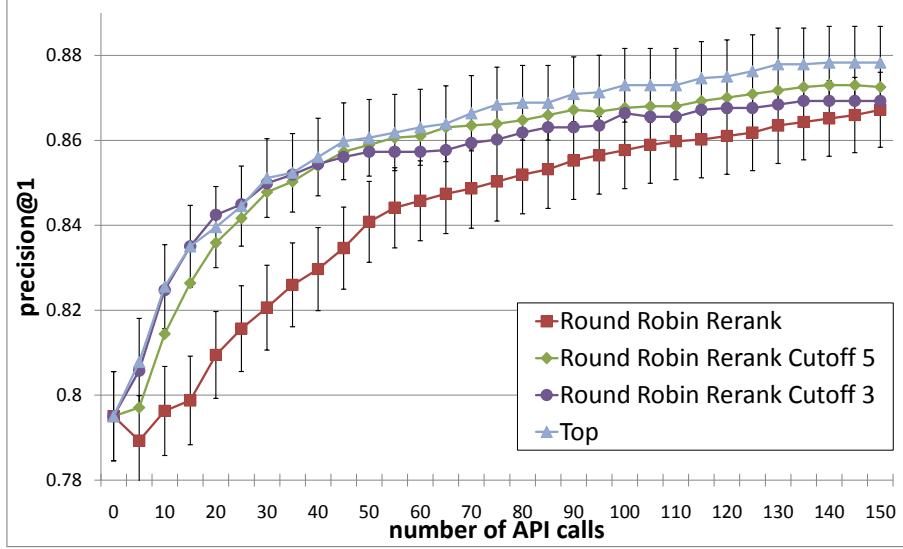


Figure 19: Comparing Round Robin with Different Cutoffs, 80% Confidence

---

**Algorithm 1** FULL( $g, G, T$ )

**Input:** Target node  $g$ , known network  $G$ , limited knowledge network  $T$

**Output:**  $t$ , best guess for matching node of  $g$  in  $T$ .

```

1:  $F^T \leftarrow$  empty hash table
2:  $F^G \leftarrow g.out$ 
3: for  $f \in F^G$  do
4:   if  $f.twitter \neq nil$  then
5:      $u = users/lookup[f.twitter, T]$ 
6:     if  $u \neq nil$  then
7:       Insert a key-value pair  $(u, 1.0)$  into  $F^T$ 
8:       Continue
9:     end if
10:   end if
11:    $U = users/search[f.fullname, T]$ 
12:    $(u, score) = BestPrimitiveMatch[f, U, T]$ 
13:   Insert a key-value pair  $(u, score)$  into  $F^T$ 
14: end for
15: for  $x \in Candidates[g, T]$  do
16:   Obtain  $x.out$  using  $friends/list[x.id]$ 
17: end for
18:  $t \leftarrow \arg \max_{x \in Candidates[g, T]} AdvancedMatch[g, x]$ 
19: return  $t$ 

```

---

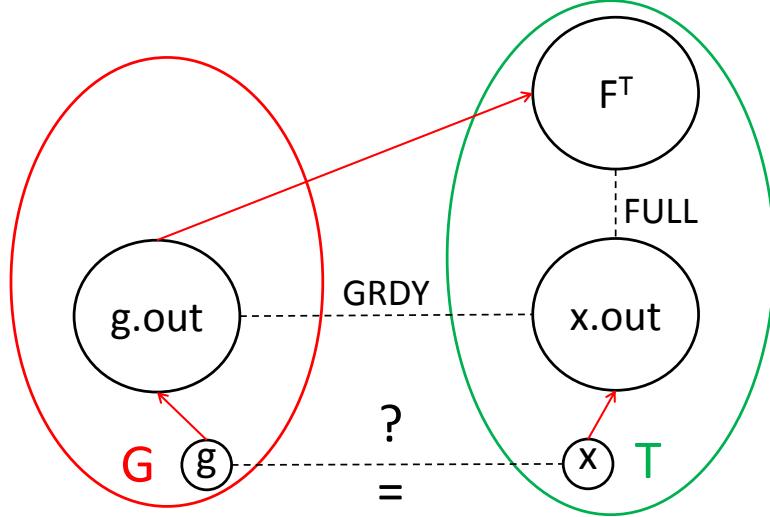


Figure 20: Illustration of Our Strategies

outlinks. Of course, the outlinks on one side point to  $G$  nodes while the others point to  $T$  nodes, so we must try to match these “friends” nodes.

Next we look at FULL visually (see Figure 20). For a target node  $g$  and a candidate  $x$ , FULL first crawls outlinks  $g.out$  on  $G$  and  $x.out$  on  $T$ . Then, it maps each  $g.out$  to a corresponding node in  $T$ . Let  $F^T$  be the set of corresponding nodes. FULL computes a match score based on how much  $F^T$  and  $x.out$  overlap. This is repeated for all possible candidates  $x \in \text{Candidates}[g, T]$ .

Finally, we describe FULL in detail.  $F^T$  (Line 1) is a hash table storing mappings from  $g.out$  to  $T$ , where each key is the best guess  $c$  for each  $x \in g.out$  in  $T$ , and each value is the corresponding match score  $\text{PrimitiveMatch}[c, x]$ . For example, recall Figure 5 where  $g.out = \{h, i\}$ , we have  $F^T = \{(p, 0.8), (r, 1.0)\}$ . Note that after Line 14,  $F^T = \text{TOut}[g, G, T]$ . Using what we have learned in Section 5.5.2, FULL keeps only the top candidate for each of  $g.out$ , i.e.,  $C_F=1$ . Note that Line 16 collects  $x.out$  to compute  $\text{Overlap}[g, x, G, T]$ . Then we use  $\text{Overlap}$  to compute  $\text{AdvancedMatch}$ .

Next we describe algorithms for when we have limits in budget values for operations to explore  $T$ . Let us have budget  $B = [b_{lookup}, b_{search}, b_{out}]$  where  $b_{lookup}$ ,  $b_{search}$ , and  $b_{out}$  are the budget values for operations `users/lookup`, `users/search`, and `friends/list` respectively.

**TOP:** We can now extend FULL to take into account the limit of each type of API call. The challenge is to **(a)** decide what to incrementally crawl to improve our accuracy, and **(b)** effectively matching nodes when we only have partial information. Combining all that we have learned from our experiments, we developed an algorithm that exploits diversified API calls to Twitter with the following strategies: **1)** keep only top 1 candidate for each of  $g.out$ , i.e.,

---

**Algorithm 2** TOP( $g, G, T, B$ )

---

**Input:** Target node  $g$ , known network  $G$ , limited knowledge network  $T$ , vector of budget values  $B = [b_{lookup}, b_{search}, b_{out}]$

**Output:**  $t$ , best guess for matching node of  $g$  in  $T$ .

```
1:  $F^T \leftarrow$  empty hash table
2:  $F^G \leftarrow g.out$ 
3: Each  $f \in F^G$  has indegree  $f.indegree$ 
4: Sort  $F^G$  by indegree in ascending order
5:  $F_{lookup}^G \leftarrow F^G$ 
6:  $[r_{lookup}, r_{search}, r_{out}] \leftarrow [0, 0, 0]$ 
7: while  $F_{lookup}^G \neq nil$  AND  $r_{lookup} < b_{lookup}$  do
8:    $f \leftarrow \text{Pop}(F_{lookup}^G)$ 
9:   if  $f.twitter \neq nil$  then
10:     $r_{lookup} += 1$ 
11:     $u = \text{users}/lookup[f.twitter, T]$ 
12:    if  $u \neq nil$  then
13:      Insert a key-value pair  $(u, 1.0)$  into  $F^T$ 
14:      Remove  $f$  from  $F^G$ 
15:    end if
16:   end if
17: end while
18: while  $F^G \neq nil$  AND  $r_{search} < b_{search}$  do
19:    $f \leftarrow \text{Pop}(F^G)$ 
20:    $r_{search} += 1$ 
21:    $U = \text{users}/search[f.fullname, T]$ 
22:    $(u, score) = \text{BestPrimitiveMatch}[f, U, T]$ 
23:   Insert a key-value pair  $(u, score)$  into  $F^T$ 
24: end while
25:  $C \leftarrow \text{IncompleteCandidates}(\text{Candidates}[g, T])$ 
26: while  $C \neq nil$  AND  $r_{out} < b_{out}$  do
27:    $c \leftarrow \arg \max_{x \in C} \text{AdvancedMatch}[g, x]$ 
28:    $r_{out} += 1$ 
29:   Add to  $c.out$  friends/list[c.id]
30:    $C \leftarrow \text{IncompleteCandidates}(C)$ 
31: end while
32:  $t \leftarrow \arg \max_{x \in \text{Candidates}[g, T]} \text{AdvancedMatch}[g, x]$ 
33: return  $t$ 
```

---

$C_F=1$ , **2**) order  $g.out$  by indegree in ascending order, and **3**) iteratively choose a candidate and crawl its outlinks using our Top strategy. Let us call this strategy TOP (Algorithm 2). We first define subroutines we use in Algorithm 2:  $\text{Pop}(X)$  and  $\text{IncompleteCandidates}(X)$ . Given an ordered list  $X$ ,  $\text{Pop}(X)$  returns the first member of  $X$  and removes it from  $X$ . Given a set of candidates  $X$ ,  $\text{IncompleteCandidates}(X)$  returns a set of partially crawled candidates whom

their outlinks are not fully obtained (i.e.,  $|x.out| < x.outdegree$ ). In Line 27, we assume that the correct pagination parameter is provided to `friends/list`. Recall from Section 3.2 that the first `friends/list` request for each node returns a next page token required for retrieving the next result page.

Next we describe TOP in detail. TOP consists of three while loops. The first two loops (Line 7-17 and Line 18-24) construct `TOut[g, G, T]` ( $F^T$ ). The first loop (Line 7-17) looks up screen.names for nodes in  $g.out$  that have their  $T$  screen\_name on their  $G$  profile page. The second loop (Line 18-24) invokes `fullname` searches for each  $g.out$  that has not been identified in the previous loop. After the `users/search` call,  $F^T$  collects the top candidate for matching each  $g.out$  based on `PrimitiveMatch`. The last loop (Line 26-31) crawls outlinks on  $T$  for candidates for eventually matching  $g$ . This loop is the iteration workflow from Figure 4 where we repeat evaluation and exploration steps (Line 27 and Line 29 respectively). Recall that `AdvancedMatch` uses graph features computed from the weighted intersection of  $F^T$  and  $x.out$  (`Overlap` and `KnownOverlap`).

**RAND:** In order to understand the value of our TOP link ordering strategy, we also consider an algorithm RAND that randomizes the crawl order for  $F^G$  and  $C$ . RAND is very similar to TOP except for two differences. The first difference is in Lines 3-4 where instead of sorting  $F^G$  by indegree in ascending order, RAND places in  $F^G$  a random permutation of the outlinks ( $g.out$ ). The second difference is in Line 27 where we choose the top candidate to crawl its outlinks: here RAND selects a random candidate. The pseudo code for RAND is shown in Algorithm 3. Given an ordered list  $X$ ,  $Shuffle(X)$  in Line 2 of Algorithm 3 returns a randomly permuted order of elements in  $X$ .

**GRDY:** Finally, we developed a greedy algorithm that exploits only one type of API call: `friends/list`. Let us call this algorithm GRDY (Algorithm 4). Before we describe GRDY, we will explain `GreedyOverlap` and `GreedyMatch`.

- **GreedyOverlap**[ $g, t$ ] (Algorithm 5): Given a  $G$  node  $g$  and a  $T$  node  $t$ , `GreedyOverlap` estimates the intersection of  $g.out$  and  $t.out$  using `NameMatch` (similarity between  $g$  and  $t$ 's names). Line 1-6 generates a list  $L$  of triples (node1, node2, score) where the score is the name similarity between node1 and node2. After sorting  $L$  by score in descending order in Line 8, the final loop (Line 11-17) goes through all triples in  $L$  and greedily matches each node pair if neither of them has been matched with any other node. This process is essentially doing weighted bipartite matching greedily rather than maximally. For a given bipartite graph (bigraph)  $BG = ((X, Y), E)$  where  $X$  and  $Y$  are two disjoint sets of nodes ( $V = X \cup Y$ ), and  $E$  is set of weighted edges connecting nodes in  $X$  to  $Y$ . Maximum weighted bipartite matching [11, 21] finds the mapping from  $X$  to  $Y$  that yields the highest sum of the weights out of all possible matchings and it requires  $O(|V|^2|E|)$ . In our case,  $X$  is  $g.out$ ,  $Y$  is  $t.out$ , and  $E$  is name similarity scores between all node pairs ( $|E| = |X||Y|$ ). The cost of sorting list  $L$  ( $O(|E|lg|E|)$ ) in Line 8 dominates `GreedyOverlap`. Therefore, our greedy strategy saves  $O(\frac{|V|^2|E|}{|E|lg|E|}) = O(\frac{|V|^2}{lg(|X||Y|)}) = O(\frac{(|X|+|Y|)^2}{lg|X|+lg|Y|})$  over the

---

**Algorithm 3** *RAND( $g, G, T, B$ )*


---

**Input:** Target node  $g$ , known network  $G$ , limited knowledge network  $T$ , vector of budget values  $B = [b_{lookup}, b_{search}, b_{out}]$

**Output:**  $t$ , best guess for matching node of  $g$  in  $T$ .

```

1:  $F^T \leftarrow$  empty hash table
2:  $F^G \leftarrow Shuffle(g.out)$ 
3:  $F_{lookup}^G \leftarrow F^G$ 
4:  $[r_{lookup}, r_{search}, r_{out}] \leftarrow [0, 0, 0]$ 
5: while  $F_{lookup}^G \neq nil$  AND  $r_{lookup} < b_{lookup}$  do
6:    $f \leftarrow Pop(F_{lookup}^G)$ 
7:   if  $f.twitter \neq nil$  then
8:      $r_{lookup} += 1$ 
9:      $u = users/lookup[f.twitter, T]$ 
10:    if  $u \neq nil$  then
11:      Insert a key-value pair  $(u, 1.0)$  into  $F^T$ 
12:      Remove  $f$  from  $F^G$ 
13:    end if
14:   end if
15: end while
16: while  $F^G \neq nil$  AND  $r_{search} < b_{search}$  do
17:    $f \leftarrow Pop(F^G)$ 
18:    $r_{search} += 1$ 
19:    $U = users/search[f.fullname, T]$ 
20:    $(u, score) = BestPrimitiveMatch[f, U, T]$ 
21:   Insert a key-value pair  $(u, score)$  into  $F^T$ 
22: end while
23:  $C \leftarrow IncompleteCandidates(Candidates[g, T])$ 
24: while  $C \neq nil$  AND  $r_{out} < b_{out}$  do
25:    $c \leftarrow$  random member of  $C$ 
26:    $r_{out} += 1$ 
27:   Add to  $c.out friends/list[c.id]$ 
28:    $C \leftarrow IncompleteCandidates(C)$ 
29: end while
30:  $t \leftarrow \arg \max_{x \in Candidates[g, T]} AdvancedMatch[g, x]$ 
31: return  $t$ 

```

---

maximal strategy.

- **GreedyMatch**[ $g, t$ ]: a **Match** function that computes features from both local profile attributes and greedy features described in Table 5. It is almost identical to **AdvancedMatch** but **GreedyMatch** replaces **Overlap** with **GreedyOverlap** and changes the denominators of the scores appropriately. **GreedyMatch** is a linear combination of these scores.

Next, we describe GRDY in detail. GRDY is similar to the last loop (Line

---

**Algorithm 4** GRDY( $g, G, T, B$ )

---

**Input:** Target node  $g$ , known network  $G$ , limited knowledge network  $T$ , vector of budget values  $B = [b_{lookup}, b_{search}, b_{out}]$

**Output:**  $t$ , best guess for matching node of  $g$  in  $T$ .

```
1:  $C \leftarrow IncompleteCandidates(Candidates[g, T])$ 
2: while  $C \neq nil$  AND  $r_{out} < b_{out}$  do
3:    $c \leftarrow \arg \max_{x \in C} GreedyMatch[g, x]$ 
4:    $r_{out} += 1$ 
5:   Add to  $c.out$  friends/list [ $c.id$ ]
6:    $C \leftarrow IncompleteCandidates(C)$ 
7: end while
8:  $t \leftarrow \arg \max_{x \in Candidates[g, T]} GreedyMatch[g, x]$ 
9: return  $t$ 
```

---

**Algorithm 5** GreedyOverlap( $g, t, G, T$ )

---

**Input:** Target node  $g$ , candidate node  $t$  for matching  $g$ , known network  $G$ , limited knowledge network  $T$

**Output:**  $score$ , the total score that estimates the intersection of  $g.out$  and  $t.out$ .

```
1:  $L \leftarrow$  empty list
2: for  $i \in g.out$  do
3:   for  $j \in t.out$  do
4:     Insert triple  $(i, j, NameMatch[i, j])$  into  $L$ 
5:   end for
6: end for
7: Each  $l \in L$  is a triple (node1, node2, score)
8: Sort  $L$  by score in descending order
9:  $M \leftarrow$  empty set
10:  $total\_score \leftarrow 0$ 
11: for  $l \in L$  do
12:    $(i, j, score) = l$ 
13:   if  $i \notin M$  AND  $j \notin M$  then
14:      $M = M \cup \{i, j\}$ 
15:      $total\_score += score$ 
16:   end if
17: end for
18: return  $total\_score$ 
```

---

26-31) of TOP. GRDY (Line 2-7) is similar to the last loop of TOP (Line 26-31) because it also crawls outlinks on  $T$  for candidates for  $g$ . This loop is the iteration workflow from Figure 4 where we repeat evaluation and exploration steps (Line 3 and Line 6 respectively).  $GreedyMatch$  uses greedy features computed from the greedy weighted bipartite matching from  $g.out$  to  $x.out$  for each candidate  $x$  ( $GreedyOverlap$ ). This is a natural extension of the state-of-the-art

<b>Greedy</b>
1. Weighted Jaccard Similarity:
(a) $\frac{\text{GreedyOverlap}[g, t, G, T]}{ g.out }$
(b) $\frac{\text{GreedyOverlap}[g, t, G, T]}{ t.out }$
2. Known Greedy Overlap Score ( $s_{kg}$ ):
$\frac{\text{GreedyOverlap}[g, t, G, T] + 1}{ g.out  +  \text{KnownOut}[t, T]  + 1}$
3. Unknown Greedy Overlap Score ( $s_{ug}$ ):
$\frac{\text{GreedyOverlap}[g, t, G, T] +  \text{UnknownOut}[t, T]  + 1}{ g.out  +  \text{UnknownOut}[t, T]  + 1}$

Table 5: Greedy Features for computing  $\text{GreedyMatch}[g, t]$

bipartite graph matching technique [11, 21] to apply to our scenario where we have limited API calls.

We also look at GRDY visually (see Figure 20). For a target node  $g$  and a candidate  $x$ , GRDY first crawls outlinks  $g.out$  on  $G$  and  $t.out$  on  $T$ . Then, it computes a match score based on how much  $g.out$  and  $x.out$  overlap.

Recall that a resetting window is the time between the operation quota resets. The budget is restored for each new window. Here we have described all strategies for only one resetting window. For multiple windows, we run our strategy again each time we reach a new window.

## 7 Evaluation Experiments

The goal of these experiments is to compare exploration strategies PRI, FULL, TOP, RAND, and GRDY applied to a real dataset while subject to real-world constraints. We want to see how TOP and RAND improve precision over the baseline PRI. We also want to see how close they are to the upper bound precision that FULL achieves. Furthermore, we would like to verify that the ordering in TOP improves precision over RAND given the same time constraint. Lastly, we would like to verify that TOP improves precision over GRDY that only exploits one type of API call.

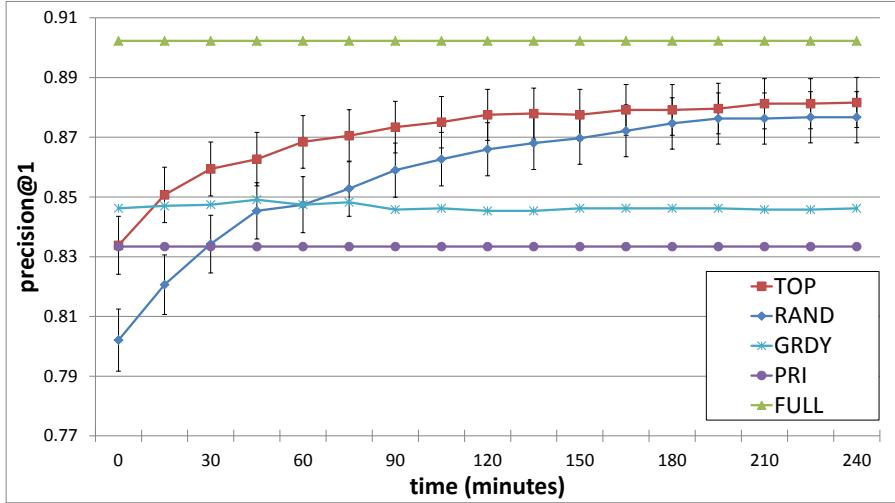


Figure 21: Comparing All Strategies, 15-min windows, 80% Confidence

## 7.1 Experimental Setup

We first use dataset GTR to train `PrimitiveMatch`, `AdvancedMatch`, and `GreedyMatch` and then use dataset GTE for evaluation. We have three budget values  $B = [b_{lookup}, b_{search}, b_{out}]$ . For each node in GTE  $g$  in  $G$ , given  $g$ 's profile and  $g.out$ , our task is to find the same user  $g$  in  $T$  by exploring  $T$  using limited budget  $B$ . Recall from Section 5.1.2 that most GTE users (99%) can be found within one `users/search` call. Therefore, we only make one `users/search` call to retrieve `Candidates[g, T]` and  $|\text{Candidates}[g, T]|$  is at most 20.

## 7.2 Results

We first present results for the actual Twitter API settings [8] as of February 20th, 2014 where the resetting window size is 15 minutes and  $B = [18000, 180, 15]$ . Recall from Table 3 that  $b_{lookup}=180$  and each request can batch up to 100 screen\_names, implying that the total  $b_{lookup}$  is 18000. Figure 21 shows how  $precision@1$  increases with time (in 15 minute increments) for different strategies. Note that at time 0 (first window) we receive the first set of budget values. We run the algorithm on each of the 2,425 nodes in GTE. For each  $g$  node, after we exhaust the budget of a window we rank Twitter nodes using `AdvancedMatch` and current knowledge of  $T$ , and select the top node. Recall that the y-axis shows  $precision@1$ : the fraction of cases where the selection was indeed the correct matching  $T$  node for  $g$ .

Our baseline, `PRI` achieves 83% precision. `PRI` uses only local information (`PrimitiveMatch`) and does not need to make API calls to crawl  $T$ , thus the

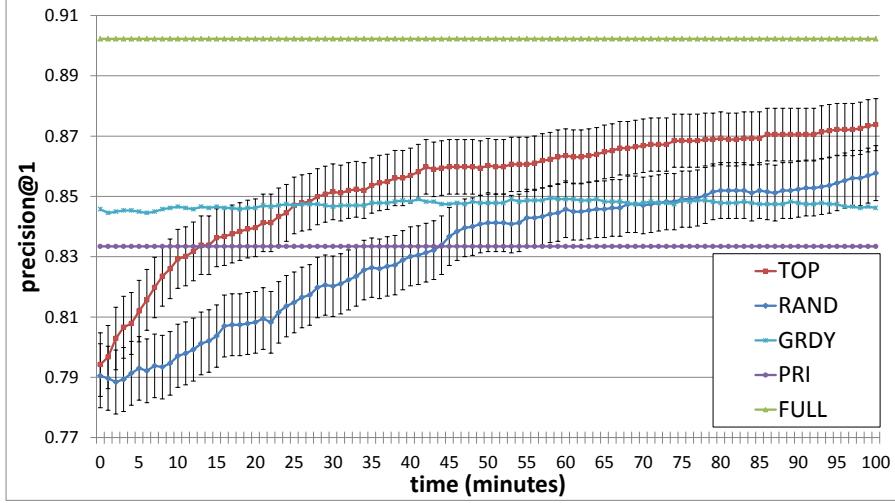


Figure 22: Comparing All Strategies, 1-min windows, 80% Confidence

curve for PRI is flat. Our upper bound, FULL achieves 90% precision (see Figure 21). Recall from Section 6 and Algorithm 1 that FULL fully constructs  $F^T$  and  $\mathbf{x.out}$  for each candidate  $\mathbf{x} \in \text{Candidates}[g, T]$  at time 0. Thus the precision for FULL is constant at all time values. FULL does not reach 100% precision for three reasons: **1)** we only make one `users/search` call to obtain `Candidates[g, T]`, **2)** some users have multiple Twitter accounts but we only accept our ground truth (the account on Google+ profile) as the correct answer, and **3)** FULL uses `AdvancedMatch` which is based on a logistic regression and may make mistakes.

**TOP:** Let us first describe the performance of TOP as compared to the baselines. At time 0 (first window) when we receive the first set of budget values, TOP achieves the same precision as the baseline PRI while RAND achieves 80% precision (3% below PRI). RAND achieves the same precision as PRI at 30 minutes (third window). As time progresses, TOP continues to achieve higher precision and at 120 minutes reaches 87.75%. After that, the marginal improvement in precision is very small. Although not shown in the figure, TOP finally stabilizes at 300 minutes (21st window) at 88.3%. This result shows that spending more than two hours (120 minutes) for a match task will increase precision very marginally. The difference in precision between TOP and RAND is significant from 0 to 90 minutes (where error bars do not overlap).

**RAND:** Next we describe the performance of RAND as compared to TOP and the baselines. At time 0, RAND achieves about 3% lower precision than PRI because the Unknown Overlap Score ( $s_u$ ) is biased in favor of nodes with high outdegrees. Recall from Table 4 that  $s_u$  is one of the graph features used in `AdvancedMatch`. The value of  $s_u$  will be close to 1.0 for nodes with high

outdegrees. This is because at the beginning (before we have started crawling outlinks)  $\text{outdegree} = |\text{UnknownOut}[t, T]|$ , which is in both the numerator and the denominator of the formula for  $s_u$ . Therefore, when a node has high outdegree,  $|\text{UnknownOut}[t, T]|$  dominates other terms and the value of  $s_u$  is close to 1.0. RAND stabilizes at 540 minutes (37th window) at 88.3%, which is 46% more time than TOP. Therefore, both TOP and RAND can eventually achieve precision@1 that is 1.7% lower than FULL.

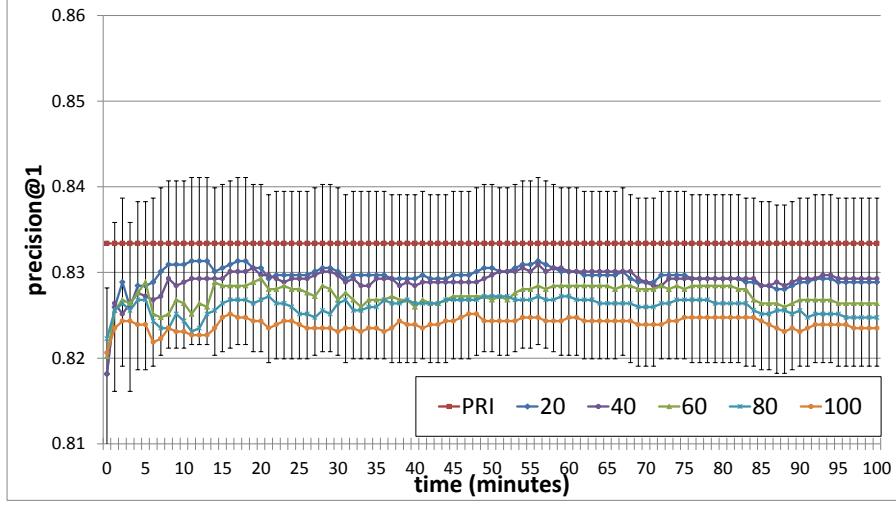
**GRDY:** Next we describe the performance of GRDY as compared to other strategies. GRDY is the most effective algorithm for time limit  $< 15$  minutes. At time 0 (first window), GRDY achieves about 1.3% higher precision than PRI. GRDY reaches its maximum precision at 45 minutes at about 85% and declines slightly thereafter (statistically insignificant). Surprisingly, as we crawl more of outlinks of candidates ( $x.\text{out}$ ), it is harder to match  $g.\text{out}$  correctly using our greedy strategy (recall Figure 20). TOP starts to beat GRDY at 15 minutes while RAND only starts to beat GRDY at 60 minutes (300% more time).

Next we present results for the more fine-grained settings where the resetting window size is 1 minute and  $B = [1200, 12, 1]$  (divide the original budget values for the 15-minute window size by 15). We are studying this hypothetical setting to see what would change if Twitter changes a key parameter of its API.

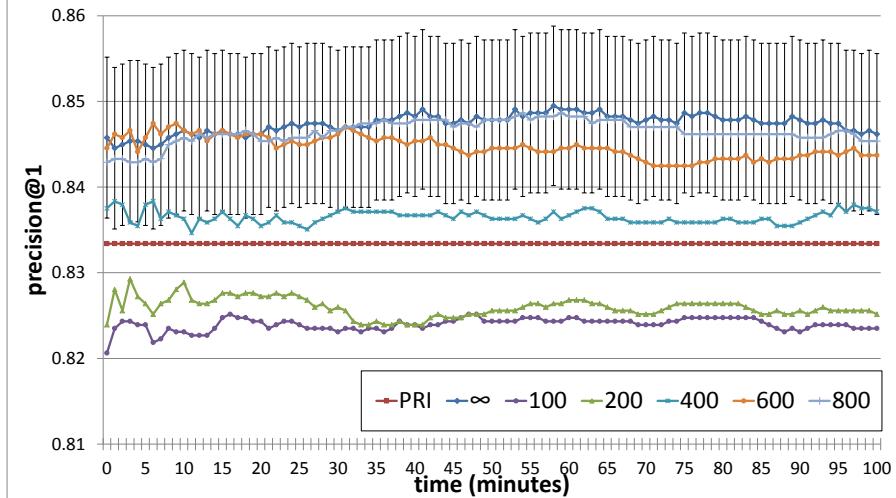
Figure 22 shows that TOP and RAND take longer to beat PRI and GRDY. This is because the budget is delayed, i.e., rather than receiving all of the budget every 15 minutes in the 15-min window case, the budget is divided equally and is given to us every minute in the 1-min window case. For example, in the 15-min window case, we receive all  $B = [18000, 180, 15]$  at time 0 (first window); to accumulate the same budget values in the 1-min window case, we have to wait until time 14 (15th window). TOP starts to beat PRI at 13 minutes while RAND starts to beat PRI only at 44 minutes (RAND requires 200% more time than TOP to reach the same 83.3% precision). At time 0, TOP and RAND achieve about 4% lower precision than PRI because of the same reason given in the 15-min case. TOP starts to beat GRDY at 25 minutes while RAND only starts to beat GRDY at 70 minutes (180% more time). Therefore, the best strategy is a hybrid solution where we use GRDY for time limit  $< 25$  minutes and switch to TOP for when we have 25 minutes or more.

From both experiments, we conclude that TOP is generally the most effective strategy and the improvement over RAND and GRDY is statistically significant. To reach baseline precision of PRI, TOP saves 30 and 33 minutes over RAND for the 15-min and 1-min window experiments respectively. For the 15-min window case, TOP saves 240 minutes over RAND to reach the highest precision value of 88.3%. And this precision value is only 1.7% below the precision of FULL. For the 1-min window case, however, GRDY is the most effective for time limit  $< 25$  minutes.

We also studied varying the maximum number of  $g.\text{out}$  to compute **GreedyOverlap**. Let us call this threshold  $\alpha$ . We first sort each  $f$  in  $g.\text{out}$  by  $f.\text{indegree}$  in ascending order based on results in Section 5.5.2. For threshold  $\alpha = 20$ , we select the top 20 outlinks of  $g$  to be used to compute **GreedyOverlap**. In other words, rather than using all outlinks from  $g.\text{out}$ , we only choose the



(a) low limits



(b) high limits

Figure 23: GRDY,  $precision@1$ , 80% Confidence

top 20 least popular outlinks as a proxy of  $g.out$  to compute `GreedyOverlap`. Setting  $\alpha = \infty$  is equivalent to using all  $g.out$ . Figure 23(a) shows that when we have low limits  $\alpha \leq 100$ , choosing the smallest  $\alpha$  (i.e.,  $\alpha = 20$ ) achieves the highest precision but the difference is not statistically significant. However, the precision is lower than PRI. For higher limits  $\alpha > 100$ , Figure 23(b) shows that higher  $\alpha$  achieves higher precision. GRDY starts to beat PRI at  $\alpha = 400$ .

Threshold  $\alpha \geq 600$  achieve similar precision. Threshold  $\alpha \geq \infty$  achieves the highest precision at 84.95%.

We have also studied *precision@k* (for  $k > 1$ ) and the results are analogous (see Figures 24 - 25). For *precision@2* (Figure 24), TOP achieves 94% precision while PRI and FULL achieve 91% and 95.5% respectively. For *precision@3* (Figure 25), TOP and RAND achieves 95.5% precision while PRI and FULL achieve 94% and 96.7% respectively. Figures 21 - 25 show that as  $k$  increases, the precision improvement of TOP over RAND decreases. Furthermore, as  $k$  increases, TOP and RAND require more time to reach the precision that PRI achieves.

Although the objective of our experiments is to match Google+ nodes to Twitter nodes, we believe our strategies can be applied directly to other social networks with limited API access (e.g., Flickr and Instagram) because Twitter set the standard for social network APIs. The inverse matching problem (from Twitter to Google+) is work in progress.

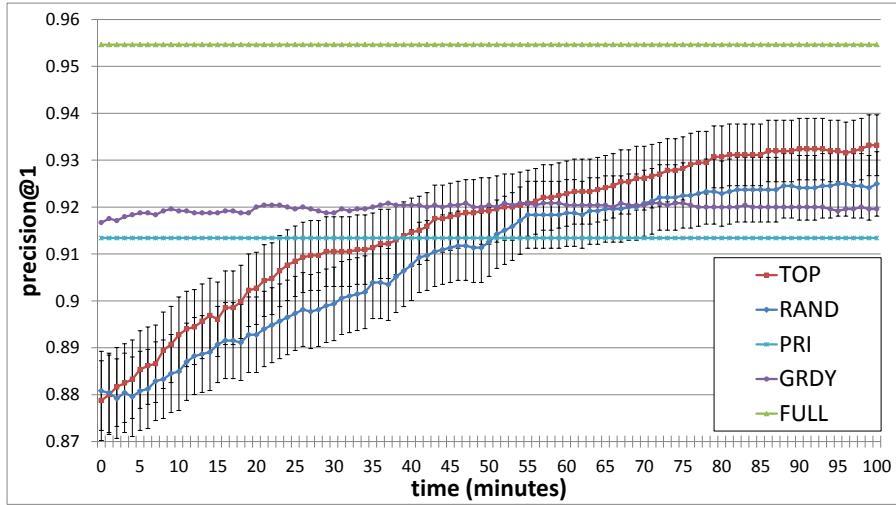
## 8 Related Work

Entity Resolution (ER) has been well studied (see [22] for a recent survey). The problem we address here is a type of ER where users are the entities and the graph nodes are the records to be resolved. However, in our setting we have limited information and we can only discover records (nodes) at a limited rate. As far as we know, this type of ER has not been studied earlier.

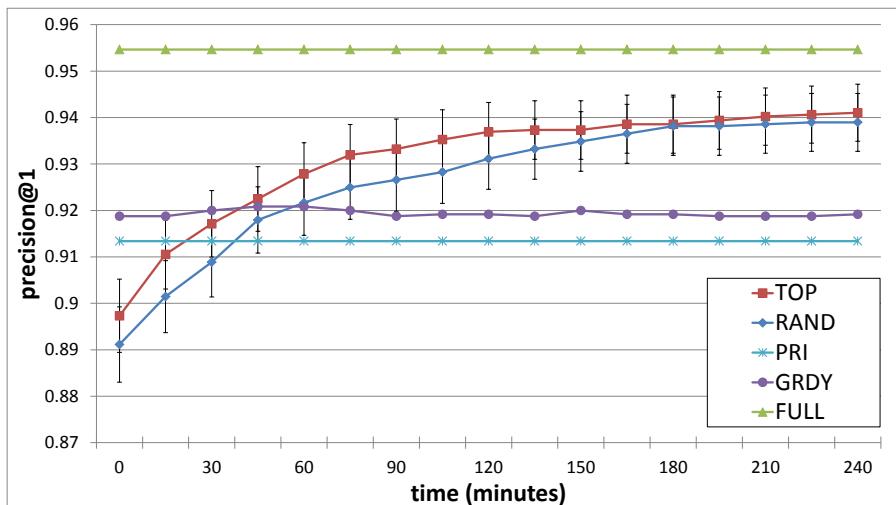
There are a number of other areas that are also related to our work: web crawling (see [18] for a recent survey), social networks, and de-anonymization. Recently, there has been a vast amount of work done on analyzing social graphs (see [12] for a recent book). For example, Schiöberg et al.[19] crawled over 95% of Google+ profiles at the birth of Google+ (Sep-Oct 2011) and presented the characteristics of Google+. Magno et al.[16] conducted a similar study in Dec 2011 by crawling 27M Google+ profiles and 580M links. Kwak et al.[15] crawled the whole Twitter graph in July 2009: 42M profiles and 1.5B links. We have not seen any work on matching Google+ users to their Twitter accounts.

Recently, there have been several works on de-anonymization (or network alignment). Narayanan et al. summarized the recent literature in [17]. In addition, they proposed a greedy seed propagation algorithm for large-scale de-anonymization in social networks. They used a network overlap technique similar to ours but they used only the network structure and ignored profile attributes. Also, they assumed that all links in the target graph are available.

The closest related work to ours is Cui et. al. [11], which finds email correspondents in Facebook using profile attributes and graph matching together. Their strategy is to iteratively update the similarity matrix using maximum weighted bipartite matching based on the assumption that two nodes are similar if their friends are similar. They also assumed that all links in both the email and Facebook graphs are available.



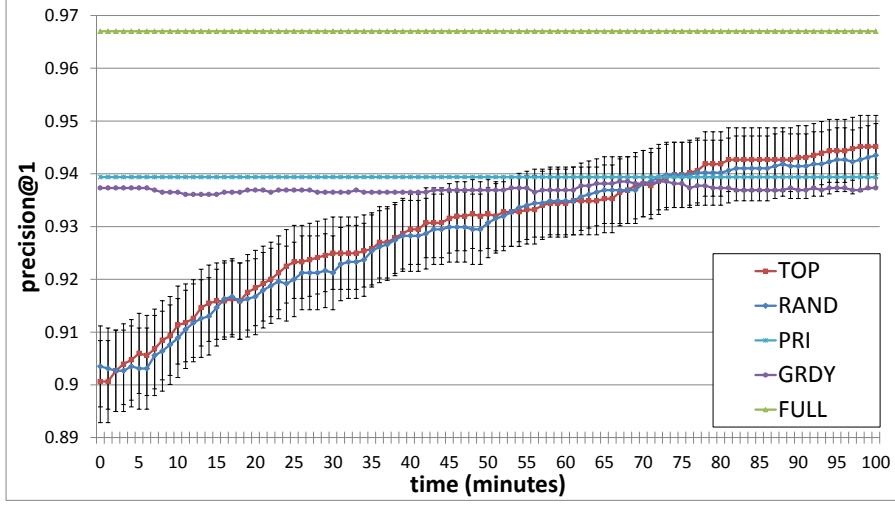
(a) 1-min windows



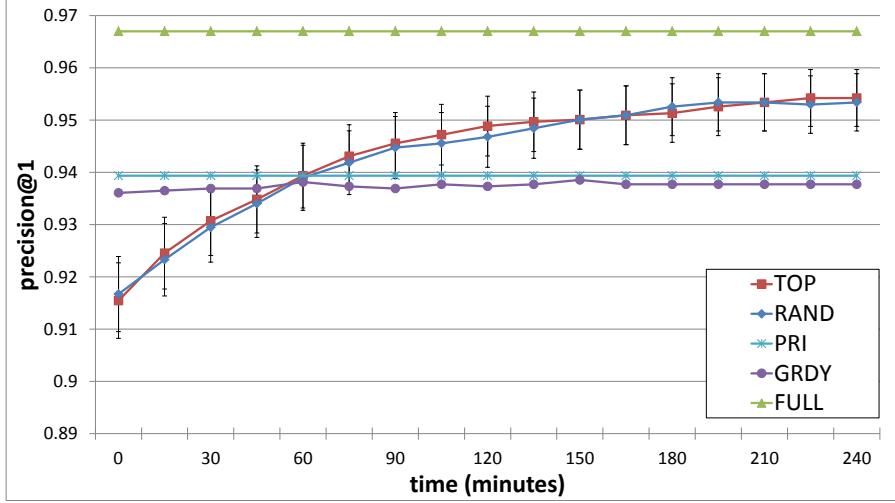
(b) 15-min windows

Figure 24: Comparing All Strategies,  $precision@2$ , 80% Confidence

All work that we know of assumes we have the full graph. To the best of our knowledge, we are the first to consider the scenario of limited API calls.



(a) 1-min windows



(b) 15-min windows

Figure 25: Comparing All Strategies,  $precision@3$ , 80% Confidence

## 9 Conclusion

We have studied the problem of ER with limited information. We have proposed two heuristic exploration algorithms: TOP and GRDY. Algorithm GRDY beats Algorithm PRI after the first window but the precision does not grow much as time progresses (the gains in precision are at less than 2%) because it does not

exploit other API calls. Algorithm TOP efficiently explores the network and achieves high precision quickly. The gains in precision that Algorithm TOP achieves are relatively modest, e.g., we can go from 83% precision (Algorithm PRI) to about 88% precision in a couple of hours. Clearly there are many applications where this effort is not worthwhile. For instance, if we want to send advertisements to Twitter users, then the 83% precision is probably enough. However, we believe that are also important applications where the delays are tolerable and the improved precision is significant. For instance, in an intelligence application we may be identifying “dangerous individuals” so the extra precision is critical.

## References

- [1] <https://plus.google.com>.
- [2] <https://twitter.com>.
- [3] About public and protected Tweets. <https://support.twitter.com/articles/14016-about-public-and-protected-tweets>, 2013.
- [4] GET friendships/show. <https://dev.twitter.com/docs/api/1.1/get/friendships/show>, 2013.
- [5] GET users/search. <https://dev.twitter.com/docs/api/1.1/get/users/search>, 2013.
- [6] Google+ API. <https://developers.google.com/+api/>, 2013.
- [7] Limits on circles. <https://support.google.com/plus/answer/1733011?hl=en>, 2013.
- [8] REST API v1.1 limits per window by resource. <https://dev.twitter.com/docs/rate-limiting/1.1/limits>, 2013.
- [9] SequenceMatcher. <http://docs.python.org/2/library/difflib.html>, 2013.
- [10] Sergey Brin Google+ page. <https://plus.google.com/+SergeyBrin/about>, 2013.
- [11] Y. Cui, J. Pei, G. Tang, W.-S. Luk, D. Jiang, and M. Hua. Finding email correspondents in online social networks. *World Wide Web*, 16(2):195–218, Mar. 2013.
- [12] E. David and K. Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, New York, NY, USA, 2010.
- [13] I. A. Gray. DO YOU KNOW THE TWITTER LIMITS? <http://iag.me/socialmedia/guides/do-you-know-the-twitter-limits/>, 2012.

- [14] +Interactive. Unofficial Google+'s Recommended users. <http://www.recommendedusers.com>, 2013.
- [15] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.
- [16] G. Magno, G. Comarela, D. Saez-Trumper, M. Cha, and V. Almeida. New kid on the block: Exploring the google+ social graph. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 159–170, New York, NY, USA, 2012. ACM.
- [17] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 173–187, Washington, DC, USA, 2009. IEEE Computer Society.
- [18] C. Olston and M. Najork. Web crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, Mar. 2010.
- [19] D. Schiöberg, F. Schneider, H. Schiöberg, S. Schmid, S. Uhlig, and A. Feldmann. Tracing the birth of an osn: Social graph and profile analysis in google+. In *Proceedings of the 3rd Annual ACM Web Science Conference*, WebSci '12, pages 265–274, New York, NY, USA, 2012. ACM.
- [20] M. Sippey. Changes coming in version 1.1 of the Twitter API. <https://dev.twitter.com/blog/changes-coming-to-twitter-api>, 2012.
- [21] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 458–469. IEEE Computer Society, 2011.
- [22] S. E. Whang and H. Garcia-Molina. Developments in generic entity resolution. *IEEE Data Eng. Bull.*, 34(3):51–59, 2011.
- [23] K. Wickre. Celebrating #Twitter7. <https://blog.twitter.com/2013/celebrating-twitter7>, 2013.
- [24] A. M. Winkler. Confidence intervals for Bernoulli trials. <http://brainder.org/2012/04/21/confidence-intervals-for-bernoulli-trials/>, 2012.