Identifying Users in Social Networks with Limited Information

Norases Vesdapunt, Hector Garcia-Molina

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

October 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 two strategies that are designed for limited information and can be adapted to different limits. We evaluate our strategies against a naive one on a real dataset and show that our strategies can provide improved accuracy with significantly fewer API calls.

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 duplicate marketing efforts. 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, journalists (recruiters) often aggregate data from different social networks to gather a fuller understanding of a person of interest (candidate). Furthermore, 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 or graph alignment) and has been well studied [25]. 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. Real-world networks usually limit access to their social graphs. Access to full social network data is limited because of companies' privacy policies and the need to safeguard proprietary data. Twitter's current API limitation makes collecting its full network prohibitively expensive (time and computing resources). Other social networks like Google+, Facebook, and LinkedIn also do not allow a user to obtain the whole social graph.

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 (e.g., use `g`'s friends F_G to probe Twitter), 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. For instance, if one has the identity `t` of a Twitter node, one can request `t`'s profile information. One can ask for profiles for up to 100 nodes in the same request, but only 180 such requests can be made per time "window" (the time between operation quota resets). 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 [1, 23]. Furthermore, the limits and API usage rules keep changing over time.

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 in our discussion and experiments, we will focus on a scenario where we start with a Google+ user, and try to find the corresponding Twitter user. (We have also implemented and experimented with the inverse matching problem (from Twitter to Google+) but we do not cover it here due to space limitations.) We believe our algorithms are applicable to other scenarios and can be extended in a straightforward fashion.

The summary of our contributions is as follows:

- We formulate the problem of social network ER with limited resources (Section 3).
- We extend known techniques for matching graph nodes to our scenario where only partial neighborhood knowledge may be available (Section 4).
- We experimentally study the Twitter API to determine what calls are more effective for improving match accuracy. Based on these experiments we develop crawling heuristics, e.g., to determine the order in which to crawl nodes (Section 5).
- Based on our node matching techniques and on our heuristics, we propose two algorithms (TOP and GREEDY) for dynamically crawling a network and finding a matching node (Section 6).
- We evaluate our algorithms 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

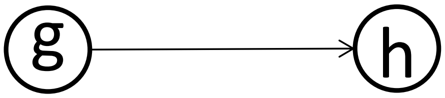


Figure 1: Following Relationship

Text

1. `names`: fullname + other names extracted from the profile
2. `id`: uniquely identifies a user and cannot be changed
3. `locations`: current_place + places extracted from the profile
4. `descriptions`: {introduction, bragging_rights, tag-line, job_title, universities, organizations}
5. `urls`: urls from “other profiles” attribute + urls extracted from profile descriptions. 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: http://www.youtube.com/user/sbrin1600/ and http://picasaweb.google.com/sergey.brin.
6. `twitter`: Twitter screen name extracted from urls. For example, we can extract ladygaga from https://twitter.com/ladygaga.
7. `screen_names`: possible screen names extracted from urls. For example, ladygaga is a potential screen name extracted from https://www.facebook.com/ladygaga.
8. `posts`: up to 10 most recent posts posted by the user

Graph

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

Table 1: G Attributes [2]

lowercase letters near `t` to refer to nodes in T. Note that in some systems (e.g., Twitter) there are two ways to identify a node: through a screen name (e.g., “sergeybrinn”) or through the actual user id (e.g., 50449264). As long as the API lets us reach the node in question, we will consider either type of identifier to be the node’s identity.

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

Text

1. `name`: fullname
2. `screen_name`: uniquely identifies a user, but a user may change his/her screen_name
3. `id`: uniquely identifies a user and cannot be changed
4. `description`: free text, 160 character limit
5. `location`: free text
6. `status`: “tweet”, 140 character limit per status
7. `url`: one link to website

Graph

1. `in`: set of T users who follow `t` (i.e., `t`'s inlinks). This is not part of T 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.
2. `out`: set of T users that `t` follows (i.e., `t`'s outlinks). Twitter refers to this attribute as `friends` but we will continue using `out` to be consistent.
3. `indegree`: `|in|`. Takes value 0 when a user has no followers in T. This information is available in the profile even for *protected* users: users whose text and graph information is only available to followers they approved [6].
4. `outdegree`: `|out|`, similar to `indegree`

Table 2: T Attributes [3]

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`.

3.2 API Operations

Next we discuss Google+ and Twitter APIs. (As we discuss in the next section, in our problem setting where we are given a single Google+ node to match, the critical factor is the number of Twitter API calls, not the Google+ calls.)

Before we define the 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 has 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).

Limit	Operations
15	1. `friendships/show`: returns relationships (following, followed by) between 2 given user_ids/screen_names [7]. 2. `followers/id`: 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. 3. `friends/id`: 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. 4. `followers/list`: 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. 5. `friends/list`: 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.
180	1. `users/search`: 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 [8]. 2. `users/lookup`: returns a user profile given a user_id/screen_name. Returns *nil* 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).

Table 3: Operation Limits (Number of Requests) per 15-Minute Window [10]

3.2.1 Google+ API

Currently, Google+ only provides a simple API for fetching user profiles and searching users [4]. 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.

3.2.2 Twitter API

Table 3 summarizes the twitter API. 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. Retrieved user profiles contain profile attributes specified in Table 2.

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) [4].

Twitter limits the number of API calls of a given type that can be made in each time window, currently 15 minutes. The limits are shown in the first column of Table 3. For example, in a 15 minute period, a program can at most make 15 calls of type `friends/id`.

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

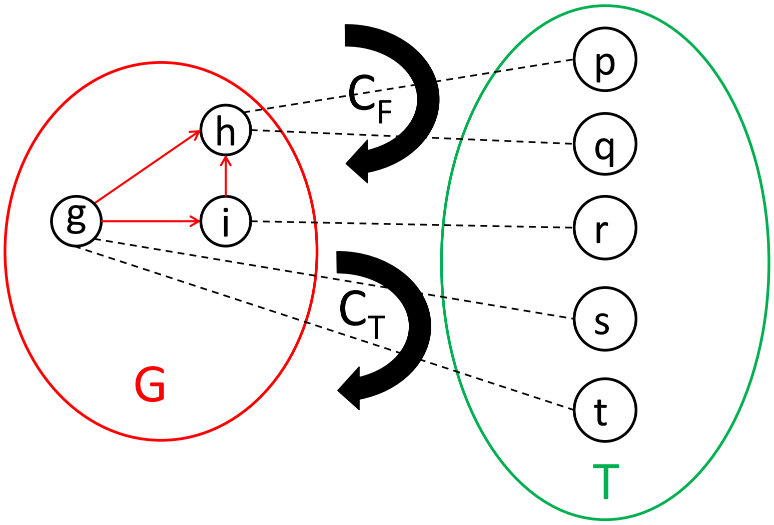


Figure 2: Example of a Match Task

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 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 **??**.

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.
  1. $StringSim(a, b) = \frac{2\{max(|a|,|b|)-EditDistance(a,b)\}}{|a|+|b|}$[5]
  2. $CosineSim(a, b)$ = 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:

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

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

3.6.2 Common Functions

`Candidates[g, T, L]`: Given a G node `g`, finds a set of potential T nodes that may match `g` (e.g., by doing a name search). Parameter `L` is the target number of candidates desired. (As we will see, `L` will be set to thresholds C_F or C_T illustrated in Figure 2.) `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. From our running example in Figure 2, `Candidates[h, T, L]` = {`p`, `q`} (assuming `L` is greater than 1).

`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. Table 4 describes graph similarity features we can use to compare `g` and `t`'s 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 three versions of `Match` each suited for different resource availability: (1) `MATCH_PRI`, (2) `MATCH_ADV`, and (3) `MATCH_GRDY`. We will define `MATCH_ADV` and `MATCH_GRDY` after we have defined prerequisite functions.

`MATCH_PRI[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. Let primitive features consist of text similarity features and a position feature (rank of the node `t` in `Candidates[g, T, L]`). Text similarity features are computed from similarities between `g` and `t`'s names, locations, descriptions, urls, and screen_names. These features are described in Table 4.

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

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

4 Strategy

In this section we explain the motivation behind the data analysis in the next section and provide a high-level intuition of our solution. Starting from `g`, we use `Candidates[g,T,L]` to get a set of T nodes that may represent the same

Primitive

Text

1. Name Similarity:
$$\max_{\texttt{x} \in \texttt{g.names}} \mathit{StringSim}(\texttt{x}, \texttt{t.name})$$
2. Location Similarity:
$$\max_{\texttt{x} \in \texttt{g.locations}} \mathit{StringSim}(\texttt{x}, \texttt{t.location})$$
3. Description Similarity:
$$\mathit{CosineSim}(\texttt{g.descriptions}, \texttt{t.description})$$
4. Url Similarity: $1\{\texttt{t.url} \in \texttt{g.urls}\}$
5. Screen Name Similarity:
$$\max_{\texttt{x} \in \texttt{g.screen\_names}} \mathit{StringSim}(\texttt{x}, \texttt{t.screen\_name})$$

Position
$$\frac{\mathit{position}}{|\texttt{Candidates}[\texttt{g}, \texttt{T}, \texttt{L}]|}$$
where *position* is the rank of $\texttt{t}$ in T user search results

Graph

1. Weighted Jaccard Similarity (WJ):
$$(a) \quad \frac{\texttt{Overlap}[\texttt{g}, \texttt{t}, \texttt{G}, \texttt{T}]}{|\texttt{TOut}[\texttt{g}, \texttt{G}, \texttt{T}].\texttt{scores}|}$$
$$(b) \quad \frac{\texttt{Overlap}[\texttt{g}, \texttt{t}, \texttt{G}, \texttt{T}]}{|\texttt{t.out}|}$$
2. Known Overlap Score (s_k):
$$\frac{\texttt{Overlap}[\texttt{g}, \texttt{t}, \texttt{G}, \texttt{T}] + 1}{|\texttt{TOut}[\texttt{g}, \texttt{G}, \texttt{T}].\texttt{scores}| + \texttt{NotOverlap}[\texttt{g}, \texttt{t}, \texttt{G}, \texttt{T}] + 1}$$
3. Unknown Overlap Score (s_u):
$$\frac{\texttt{Overlap}[\texttt{g}, \texttt{t}, \texttt{G}, \texttt{T}] + \texttt{NotOverlap}[\texttt{t}, \texttt{T}]| + 1}{|\texttt{TOut}[\texttt{g}, \texttt{G}, \texttt{T}].\texttt{scores}| + |\texttt{UnknownOut}[\texttt{t}, \texttt{T}]| + 1}$$

Table 4: Features for computing `Match[g, t]`

user. Our goal is to rank these candidates to see which is the best match. To effectively rank a candidate `t`, not only do we need its profile, but we also

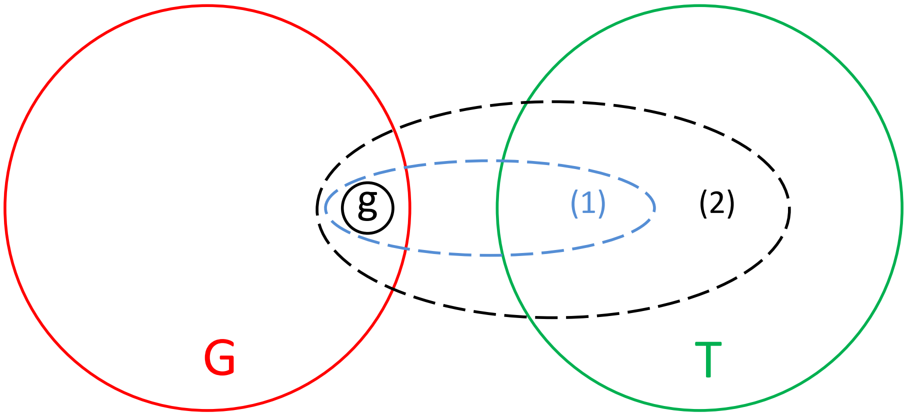


Figure 3: Exploration

need its neighborhood. Furthermore, we need to know if nodes in `t`'s neighborhood match nodes in `g`'s neighborhood, since neighborhood matches increase our confidence that `g` and `t` are good matches.

This network exploration process can be described as follows. At any given time we have a STATE representing what we have discovered about G and T. The STATE includes the id of the discovered nodes, but some of their information (profile, outlinks) may be missing. At each step we must decide what new information we want to get (through the API), e.g., a node's profile, its outlinks, or its potential candidates. The challenge is in deciding what new information to fetch, since some information may be more helpful than other in improving our ranking of `g`'s candidates. The heuristics we develop in Section 4 will help us make these decisions.

After we exhaust our budget we need to compute our final ranking of the candidates, using the final state we have obtained. In Section 4.1 we discuss how we do this ranking, taking into account that our state information is not complete.

In summary, our process is:

- Get the potential candidates for `g`.
- **Exploration:** Explore G and T, at each step requesting the information that we think has the best chance of improving the ranking of `g`'s candidates. In Figure 3, we expand our knowledge of T from (1) to (2) after an exploration step.
- **Evaluation:** When our budget has been exhausted (for our K allowed windows), we rank the candidates and select the one that seems the most likely to represent the same user as `g`.

4.1 Scoring Candidates

In this section we explain how to compare each candidate node x in T to our target node g (i.e., how to score each candidate x). Our scoring function depends on the stage we are in (exploration vs. evaluation stages). When we are exploring, the best candidate is the one that we should crawl to improve our final evaluation. On the other hand, when we are evaluating, the best candidate is the one that most likely matches g.

Recall from Section 3.6 that we have three match functions: MATCH_PRI, MATCH_ADV, and MATCH_GRDY. First, we define MATCH_ADV and MATCH_GRDY, then we define the auxiliary functions that construct MATCH_ADV.

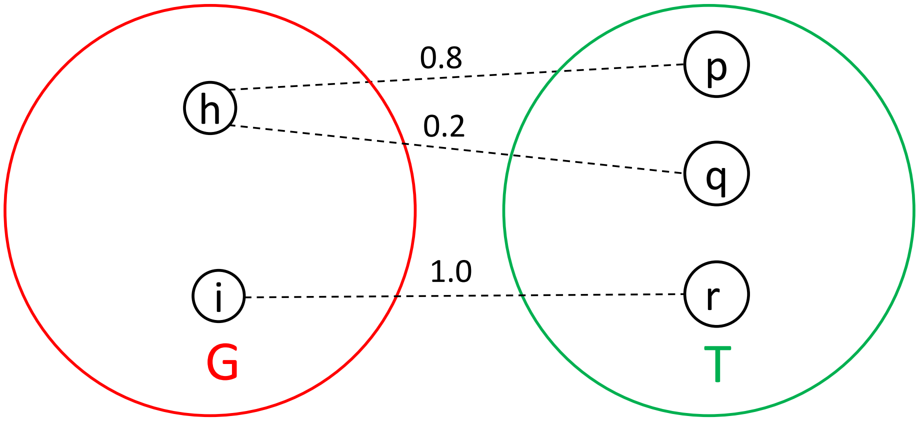


Figure 4: Illustration for TOut

- MATCH_ADV[g, t, explore]: a Match function that computes features from both local profile attributes and graph features. In addition to primitive features used in MATCH_PRI[g, t], we use graph features described in Table 4. The parameter explore is a boolean flag. When explore is false, we set the weights of s_k and s_u to zero. MATCH_ADV is a linear combination of these scores.

- MATCH_GRDY[g, t, explore]: Similar to MATCH_ADV but rather than using graph features, MATCH_GRDY uses greedy features computed from the greedy weighted bipartite matching from g.out to x.out for each candidate x (replacing Overlap in Table 4 with GreedyOverlap). This is a natural extension of the state-of-the-art bipartite graph matching technique [12, 24] to apply to our scenario where we have limited API calls. The details for MATCH_GRDY and GreedyOverlap are in Section 6 where we describe Algorithm GRDY. A very simple but effective GreedyOverlap[g, x] estimates the the intersection of g.out and x.out using NameMatch (similarity between g and x's names).

Next we define the auxiliary functions that construct `MATCH_ADV`.

- `TOut[g, G, T]`: Given a G node `g`, returns a hash table of (node, score) pairs {`BEST_MATCH_PRI[x, T]` 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 `BEST_MATCH_PRI` function.

  Recall from our running example in Figure 2 that `g.out` = {`h`, `i`}. From Figure 4, `MATCH_PRI[h, p]` = 0.8 and `MATCH_PRI[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. `TOut[g, G, T]` = {(`p`, 0.8), (`r`, 1.0)}.

  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.nodes` be the set of all keys in `H` and `H.scores` be the list of all values in `H`. In our example, `TOut[g, G, T][p]` = 0.8, `TOut[g, G, T][r]` = 1.0, `TOut[g, G, T].nodes` = {`p`, `r`}, and `TOut[g, G, T].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 `TOut[g, G, T].nodes` and `t.out`. Then Overlap returns the sum of `TOut` scores associated with the nodes in the intersection:

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

  We want to compute the similarity between the neighborhood sets (`g.out` and `t.out`). Using the simple Jaccard Similarity ignores the weights for one of the sets (namely `TOut[g, G, T]`). We solve this issue with `Overlap`. We use `Overlap` to define Weighted Jaccard Similarity (WJ) described in Table 4. Let F^T be `TOut[g, G, T].nodes`. Intuitively, $WJ(a)$ captures $F^T \subseteq$ `t.out` while $WJ(b)$ captures `t.out` $\subseteq F^T$. This is similar to Fuzzy Jaccard Similarity [24] but we split the feature into two ($WJ(a)$ and $WJ(b)$) to let the logistic regression classifier learn the weights.

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

- `UnknownOut[t, T]`: When we have an incomplete T, there may exist some outlinks of t that we do not know. Let us call this set `UnknownOut[t, T]`. Although we do not know `UnknownOut[t, T]`, we can compute `|UnknownOut[t, T]|` as follows:

  $$|\texttt{UnknownOut}[\texttt{t},\texttt{T}]| = \texttt{t.outdegree} - \texttt{t.out}$$

- NotOverlap[g, t, G, T]: Given a G node g and a T node t, returns the number of nodes that are in t.out but not in TOut[g, G, T].nodes (i.e., $|\texttt{t.out} - \texttt{TOut}[\texttt{g}, \texttt{G}, \texttt{T}].\texttt{nodes}|$).

Now that we have all of the necessary components, we can construct s_k and s_u as shown in Table 4.

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 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 MATCH_PRI[g, s] $>$ MATCH_PRI[g, t], we will choose to crawl s to retrieve its outlinks. If the first outlink returned overlaps with F^T, 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 MATCH_ADV[g, s, true] will always be greater than MATCH_ADV[g, t, true]. Based on MATCH_ADV, 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 (no more overlap).

Let us summarize the desired behaviors of MATCH_ADV. At the beginning of our match task when t has the most unknown links, we want to boost MATCH_ADV[g, t, true]. As we explore t's outlinks and find that all of them overlap with F^T, we want to keep MATCH_ADV[g, t, true] 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 MATCH_ADV[g, s, true] 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 5(a), let us first explain notations used in Figure 5. We have three types of nodes: check, cross, and question mark. Check nodes represent nodes in F^T (i.e., TOut[g, G, T].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 5(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.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 5, 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).

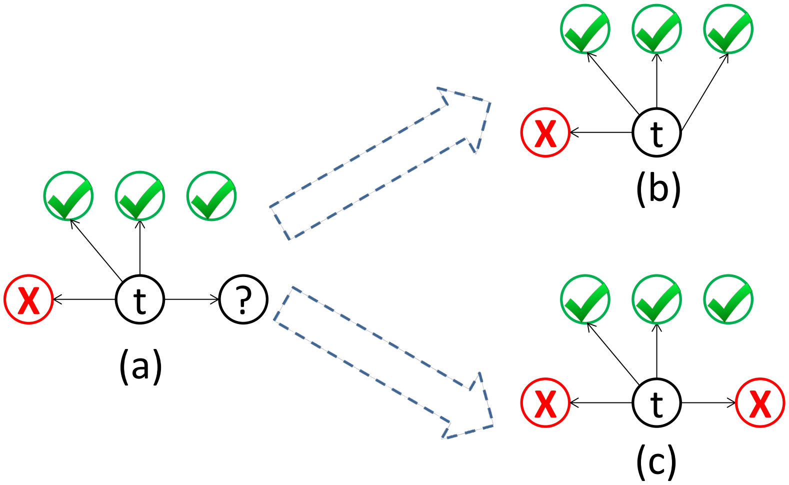


Figure 5: Diagram Explaining s_k and s_u

Case (a): There are three check nodes, translating to `|TOut[g, G, T]|` $= 3$. `t` links to two check nodes, one cross node , and one question mark node. This translates to `Overlap[g, t, G, T]`$= 2$, `NotOverlap[g, t, G, T]` $= 1$ (the number of cross nodes in discovered `t.out`), and `|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 `|TOut[g, G, T]|` $= 3$. `t` links to three check nodes, one cross node , and no question mark nodes. This translates to `Overlap[g, t, G, T]`$= 3$, `NotOverlap[g, t, G, T]` $= 1$, and `|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 `Overlap[g, t, G, T]`$= 2$, `NotOverlap[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., `Overlap[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 $[b_1, b_2, b_3, ..., b_n]$ 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 Data Analysis

We explained why we conduct our data analysis in the previous section. Here we study the characteristics of our dataset and then provide a collection of heuristics for our exploration algorithm. We will present our strategies in Section 6 and then evaluation experiments in Section 7.

5.1 Dataset

5.1.1 Training Set GTR

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+ [11]. 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+.

To discover the findable users, we examined all X users and grouped them into 8 buckets, as shown in Figure 6. 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

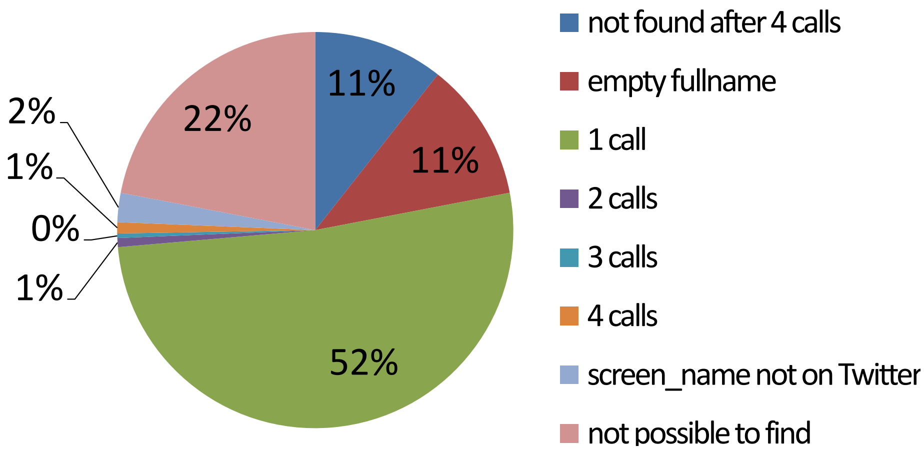


Figure 6: Number of `users/search` calls vs. candidates discovered for GTR

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 6 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 6, 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 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

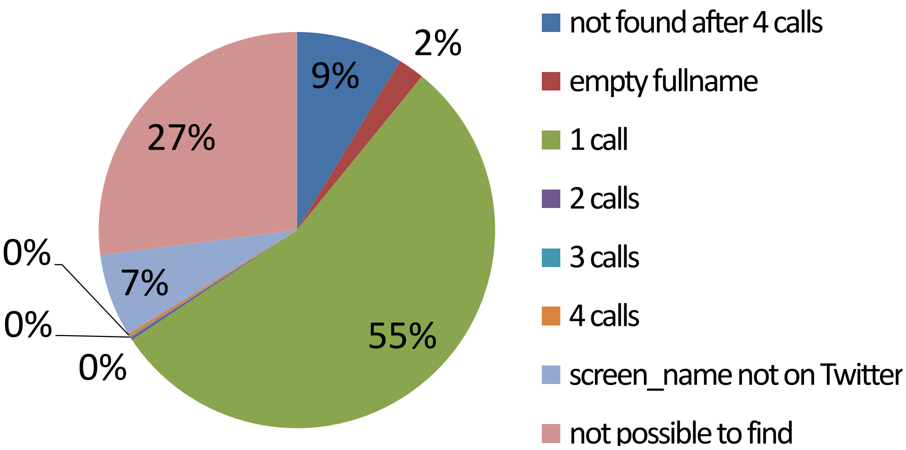


Figure 7: Number of `users/search` calls vs. candidates discovered for GTE

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+" [16]. 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 4,381 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 7 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.

5.1.3 Random Testing Set GRND

Note that there is a bias in GTE in that all the nodes have a nonempty `twitter` attribute. Therefore, we develop a second testing set where Google+ users are selected at random whether they have a `twitter` attribute or not. Ideally, we should select random Google+ user ids but this is not feasible because these ids are 21-digit integers, most of which have not been taken. Instead, we randomly sample from the 5.5 million user profiles we have collected in the process of collecting GTR and GTE. We call this set GRND.

For each node in GRND, we manually look for a matching Twitter user. The process involves a variety of probes to Twitter and the general Web, and a visual comparison of various fields and images. For example, if the name is "Fred George Smith II," we may perform searches on Twitter for "Fred George Smith," "Fred Smith," and "Fred George." To verify a match, we would compare the profile attributes such as location, education, photos, links, posts, and friends. If there is not enough information, we may further search the web for more attributes to verify whether two profiles represent the same person.

Because manual matching is so time consuming, we use a Google+ random set of size 100. However, as we will see, the results for GRND are consistent with those for GTE, so we are confident that GRND is adequate. We note that we were able to manually find matching Twitter nodes for 52% of GRND nodes. Of the remaining 48%, about one tenth were unmatchable (by us) because they use non-Latin characters in their names.

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 8. 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.

5.3.2 Twitter

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

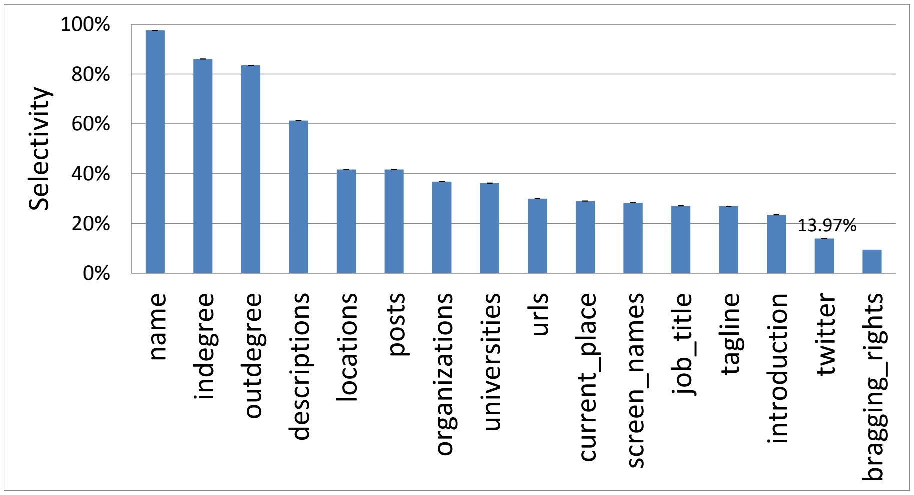


Figure 8: Selectivity of Google+ Attributes

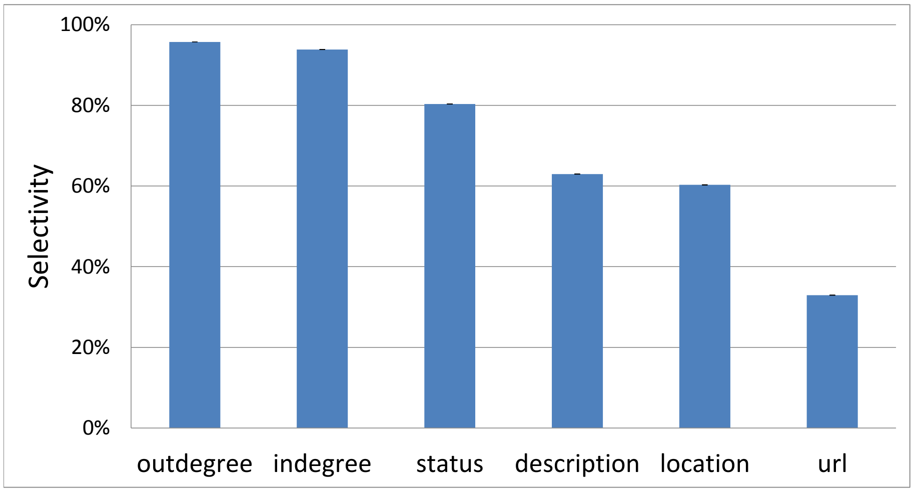


Figure 9: Selectivity of Twitter Attributes

`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 [26].

We display the selectivity of Twitter attributes in Figure 9. We can see again that the link attributes are almost always available (more than 90% of the time). Other texual 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 ($0 <$ `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 ($0 <$ `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 10, 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 [9]. 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.

5.4.2 Twitter

In Figure 11, 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

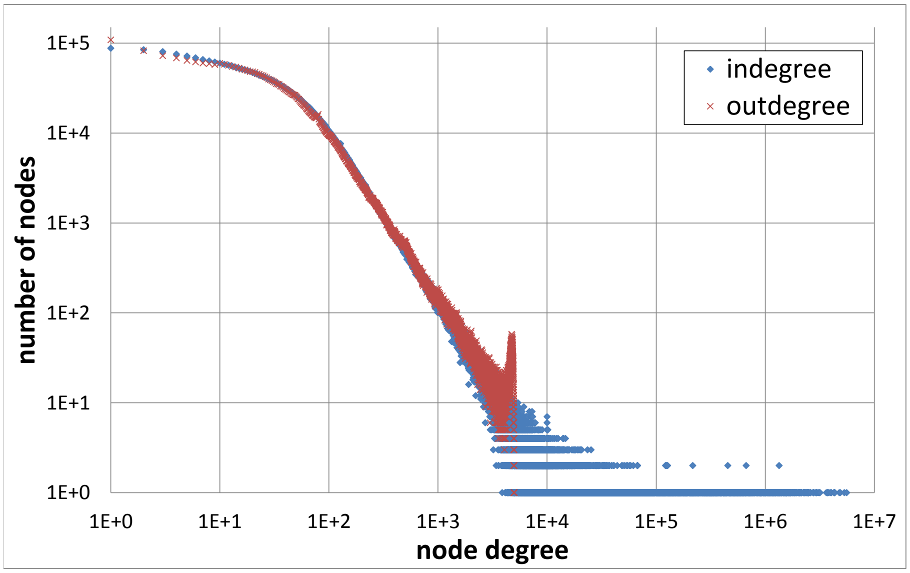


Figure 10: Google+ Degree Distribution (log-log scale)

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 [14]. 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 Heuristics

In all of our heuristic studies, 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).

5.5.1 MATCH_PRI

Our very first study is to evaluate the performance of using the primitive features (explained in Table 4) alone. We will use precision as a metric to evaluate MATCH_PRI. 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 the top k candidates returned by a strategy contain the matching

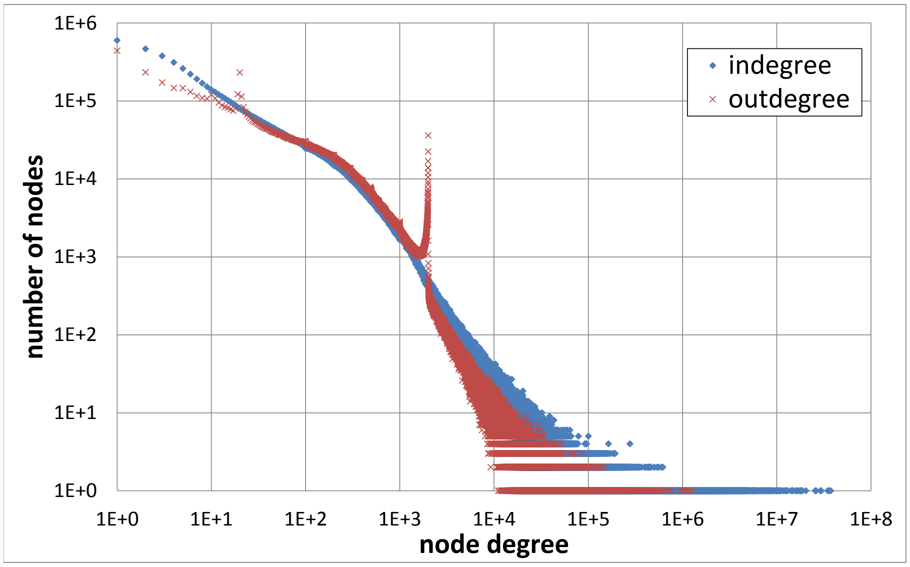


Figure 11: Twitter Degree Distribution (log-log scale)

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 12 shows how precision increases as k increases. The vertical error bars represent 95% confidence intervals [15]. Let us consider n trials with s successes. For a 95% confidence interval, we have an error percentile α 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 $k = \Phi^{-1}\{1 - \alpha/2\}$ (Φ^{-1} is a probit function) [27]. 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 `MATCH_PRI` 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, `MATCH_PRI` 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.

In the next two sections, we assume that we have complete information about links in network T. We will discuss a situation with an incomplete network T in Section 5.5.4.

5.5.2 Threshold C_F

Figure 2 shows the threshold C_F that we would like to study. C_F is the number of candidates we consider a potential match for each of `g.out`. We would like to understand how varying C_F affects the precision of our match task. This study

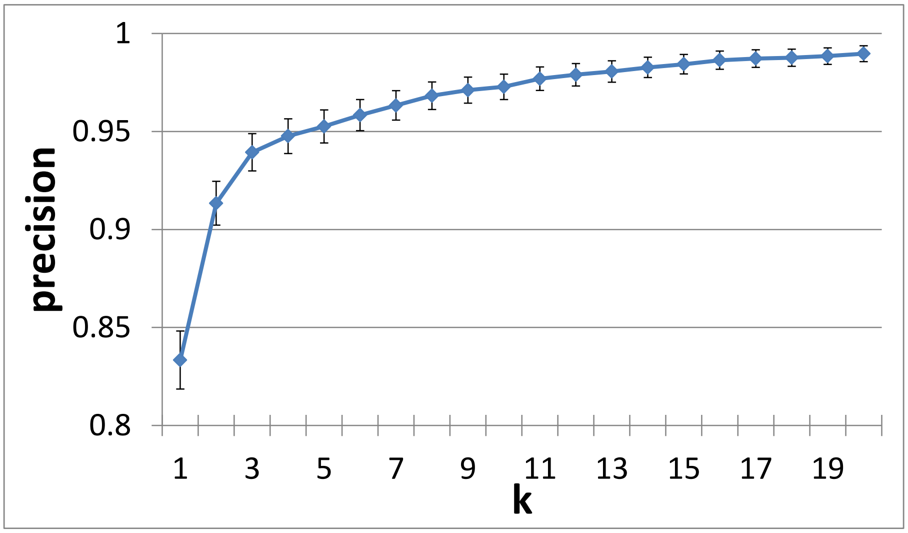


Figure 12: Precision@k, `MATCH_PRI`, 95% Confidence

will help us shape our heuristics for making API calls (Section 6) since we have limited resources and cannot set C_F to maximum.

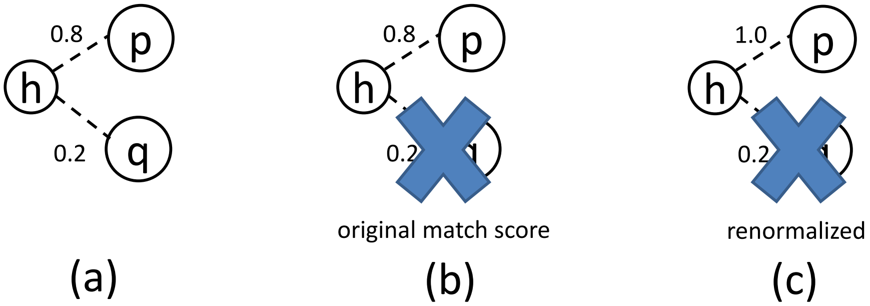


Figure 13: Renormalizing Match Scores

Let us have complete `g.out` and then vary C_F. We will study when we have incomplete `g.out` in the next section. Recall that the 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 `MATCH_PRI`. Prune here means discard candidates with lower match scores. The candidates we kept are used to compute graph features used in `MATCH_ADV`. 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 `MATCH_PRI[h, p]` $= 0.8$ and `MATCH_PRI[h, q]` $= 0.2$ (see Figure 13(a)). When $C_F = 2$, we keep both candidates `p` and `q`, the two candidates with the

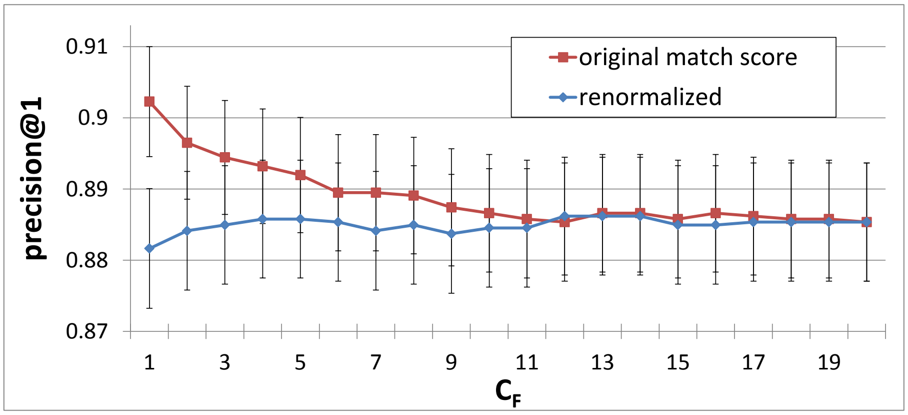


Figure 14: C_F vs. *precision*@1, 80% Confidence

highest scores. When $C_F = 1$, we keep only `p`, the single candidate with the highest score. In Figure 13, When we prune candidate `q`, we have two options for the match score of `p`: **1)** keep the original match score as in Figure 13(b) or **2)** renormalize the score so that the sum of the match scores of the candidates for `h` is 1 as in Figure 13(c).

To compare these options, we study them and present the results in Figure 14. Figure 14 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` $\in$ `Candidates[g, T, L]`) we rank them using `MATCH_ADV` 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 14, 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 Figure12 that `MATCH_PRI` achieves 83% *precision*@1 and the improvement in precision for higher k values is marginal. Therefore, keeping more candidates actually confuses `MATCH_ADV` 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 studies.

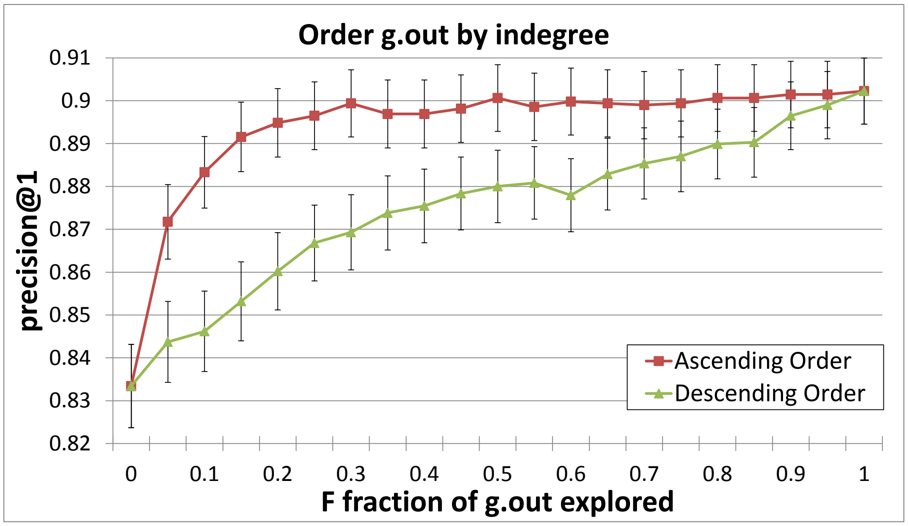


Figure 15: Fraction of `g.out` crawled vs. *precision*@1, Ascending vs. Descending Order, 80% Confidence

5.5.3 Outlink crawling order

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

In this section, we will discuss how to rank outlinks `g.out` to maximimize precision. Figure 15 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 `MATCH_ADV`) helps boost precision over `MATCH_PRI`: 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 16, we compare different schemes for ordering outlinks: by indegree, 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 16 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% pre-

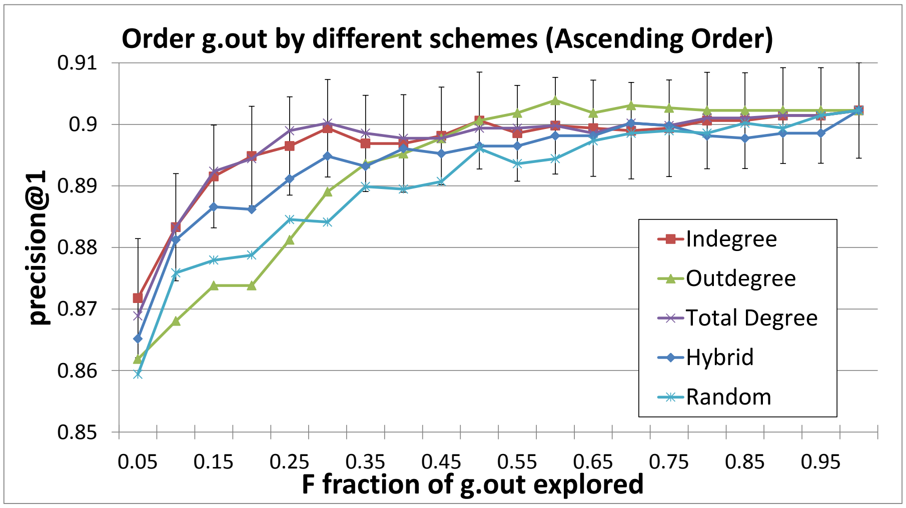


Figure 16: Comparing Different Schemes, 80% Confidence

cision, when $F < 50\%$, ordering `g.out` by indegree or total degree maximizes *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 `l`) and a popular user like Lady Gaga, we can infer that `g`'s relationship is stronger with `l` than with Lady Gaga. Two random users are more likely to both follow Lady Gaga than to both follow `l`. Thus, users with low indegrees are strong signals for distinguishing `g`'s true identity.

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 *precision*@1 (from 83% to 90% for $X = 0.3$).

5.5.4 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 $\mathtt{x} \in \mathtt{Candidates}[\mathtt{g}, \mathtt{T}, \mathtt{C_T}]$, we have to decide the order in which to 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 X to 1 and 1.0 respectively, i.e., fully crawl

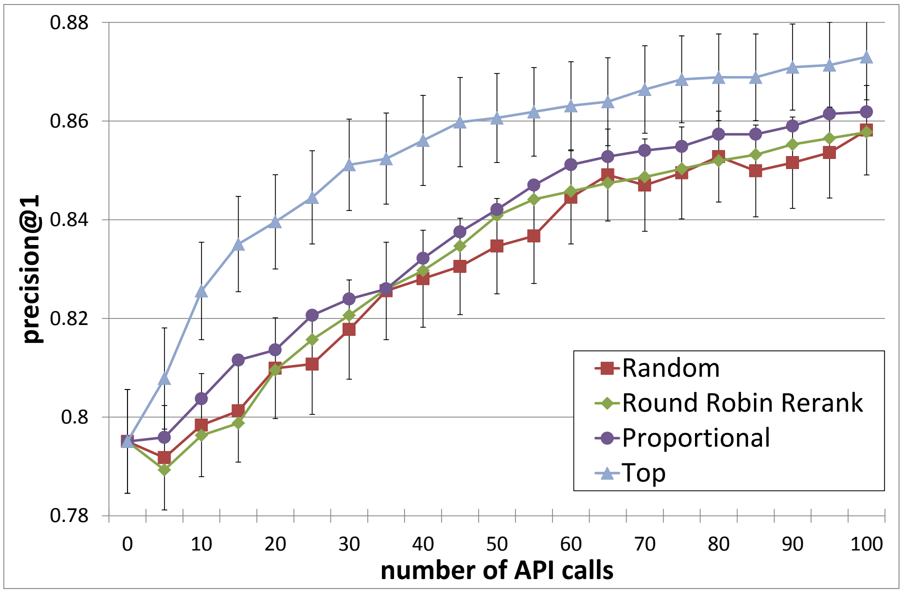


Figure 17: Comparing Outlink Crawling Strategies, 80% Confidence

`g.out` and keep the top candidate for each based on `MATCH_PRI`.

We have explored several strategies to make API calls to obtain outlinks of $\texttt{Candidates}[\texttt{g}, \texttt{T}, \texttt{C}_\texttt{T}]$:

1. **Random:** choose a candidate at random.
2. **Round Robin Rerank:** first sort candidates based on `MATCH_PRI`, 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 `MATCH_ADV`. 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 `MATCH_PRI`.
4. **Proportional:** split API calls proportionally to candidates based on `MATCH_PRI`.
5. **Top:** first sort candidates based on `MATCH_PRI` (Step 1), then make one API call for the top candidate (Step 2), and finally rerank based on the new information using `MATCH_ADV` (Step 3). Repeat steps 2-3 until it has exhausted all API calls.

Figure 17 shows how *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.

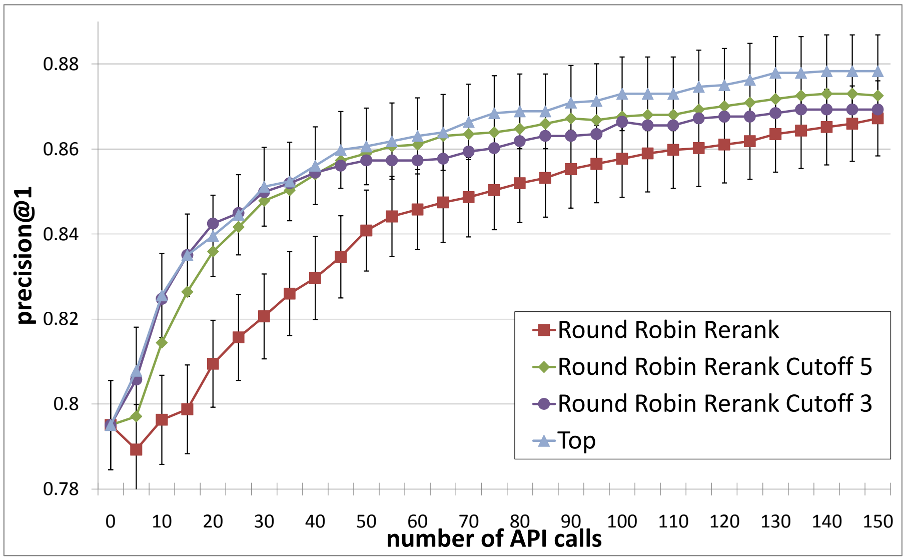


Figure 18: Comparing Round Robin with Different Cutoffs, 80% Confidence

Figure 18 shows how *precision*@1 increases as we increase the number of API calls for the Top and different Round Robin Rerank strategies. Recall from Figure 12 that the top 3 and 5 candidates returned by `MATCH_PRI` 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 18 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. 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 $\mathtt{Candidates}[\mathtt{g}, \mathtt{T}, \mathtt{C_T}]$.

6 Algorithms

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

Algorithm 1 GENERAL ALGORITHM(g, G, T, B, N)

Input: Target node g, known network G, limited knowledge network T, vector of budget per round values $B = [b_{lookup}, b_{search}, b_{out}]$, number of rounds N
Output: t, best guess for matching node of g in T.

```
 1: $C \leftarrow \texttt{Candidates}[\texttt{g}, \texttt{T}, \texttt{C}_\texttt{T}]$
 2: $F^T \leftarrow$ empty hash table
 3: $F^G \leftarrow \texttt{g.out}$
 4: STATE $= \{C, F^T, F^G, \texttt{g}, T, TC\}$
 5: for $i = 1 \rightarrow N$ do
 6:     CRAWL_X[$B$, STATE]
 7: end for
 8: $t \leftarrow \arg\max_{x \in C} \texttt{MATCH\_X}[\texttt{g}, \texttt{x}, \texttt{false}]$
 9: if $\texttt{MATCH\_X}[\texttt{g}, \texttt{x}, \texttt{false}] \geq \theta$ then
10:     return t
11: end if
12: return “Node Not Found”
```

Algorithm 2 CRAWL_FULL(B, STATE)

Input: B not used, current state STATE
Output: STATE, update the current state in place.

```
 1: for $f \in F^G$ do
 2:     if $f.twitter \neq nil$ then
 3:         $u$ = users/lookup[f.twitter, T]
 4:         if $u \neq nil$ then
 5:             Insert a key-value pair ($u$, 1.0) into $F^T$
 6:             Continue
 7:         end if
 8:     end if
 9:     U = users/search[f.fullname, T]
10:     # Implies using $C_F = 1$ below
11:     (u, score) = BEST_MATCH_PRI[f, U, T]
12:     Insert a key-value pair (u, score) into $F^T$
13: end for
14: for $c \in$ C do
15:     # Store c.out in $TC$
16:     Obtain c.out using friends/list[c.id] calls
17: end for
```

except for PRI require crawling the networks. We will first describe a general algorithm and then detail different crawling strategies.

GENERAL ALGORITHM: All of our algorithms have the structure shown in Algorithm 1. First we find candidates using users/search calls (Line 1). In some algorithms, we crawl the networks (Line 5-6). Then we compute the

Algorithm 3 CRAWL_TOP(B, STATE)

Input: budget values per round B, current state STATE
Output: `STATE`, update the current state in place.

```
1: Each $f \in F^G$ has indegree f.indegree
2: Sort $F^G$ by indegree in ascending order
3: $F^G_{lookup} \leftarrow F^G$
4: $[r_{lookup}, r_{search}, r_{out}] \leftarrow [0, 0, 0]$
5: while $F^G_{lookup} \neq nil$ AND $r_{lookup} < b_{lookup}$ do
6:     $f \leftarrow$ Pop($F^G_{lookup}$)
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:     # Implies using $C_F = 1$ below
21:     (u, score) = BEST_MATCH_PRI[f, U, T]
22:     Insert a key-value pair (u, score) into $F^T$
23: end while
24: $C' \leftarrow IncompleteCandidates(C)$
25: while $C' \neq nil$ AND $r_{out} < b_{out}$ do
26:     $c \leftarrow \arg\max_{x \in C'}$ MATCH_ADV[g, x, true]
27:     $r_{out} + = 1$
28:     # Store c.out in $TC$
29:     Add to c.out friends/list[c.id] (make 1 call)
30:     $C' \leftarrow IncompleteCandidates(C')$
31: end while
```

match scores for the retrieved candidates (Line 8), where the third parameter of `MATCH_X` (boolean flag) is the `explore` parameter defined in Section 4.1. `MATCH_X` is a variable function that differs by strategy. Then we select the candidate `t` with the maximum match score s_t. If $s_t \geq \theta$, we return `t`. Otherwise there is not enough evidence that `t` is the matching node, so we return "Node Not Found" (Line 9-12). As we can see in Section 7.3, we can vary θ to achieve different levels of precision and recall.

Next we describe STATE in detail. F^T (Line 2) is a hash table storing mappings from `g.out` to T (see `TOut[g, G, T]` for more details). Each entry

Algorithm 4 CRAWL_RAND(B, STATE)

Input: budget values per round B, current state STATE
Output: `STATE`, update the current state in place.

```
$F^G_{lookup} \leftarrow random\ permutation\ of\ F^G$
$[r_{lookup}, r_{search}, r_{out}] \leftarrow [0, 0, 0]$
while $F^G_{lookup} \neq nil$ AND $r_{lookup} < b_{lookup}$ do
    $f \leftarrow$ Pop($F^G_{lookup}$)
    if $f.twitter \neq nil$ then
        $r_{lookup} += 1$
        $u$ = users/lookup[f.twitter, T]
        if $u \neq nil$ then
            Insert a key-value pair ($u$, 1.0) into $F^T$
            Remove $f$ from $F^G$
        end if
    end if
end while
while $F^G \neq nil$ AND $r_{search} < b_{search}$ do
    $f \leftarrow$ Pop($F^G$)
    $r_{search} += 1$
    U = users/search[f.fullname, T]
    # Implies using $C_F = 1$ below
    (u, score) = BEST_MATCH_PRI[f, U, T]
    Insert a key-value pair (u, score) into $F^T$
end while
$C' \leftarrow IncompleteCandidates(C)$
while $C' \neq nil$ AND $r_{out} < b_{out}$ do
    $c \leftarrow$ random member of $C'$
    $r_{out} += 1$
    # Store c.out in $TC$
    Add to c.out friends/list[c.id] (make 1 call)
    $C' \leftarrow IncompleteCandidates(C')$
end while
```

in F^T is a (`u`, score) pair where `u` is the best guess for each $x \in$ `g.out` in T and score is `MATCH_PRI[x, u]`. For example, recall Figure 4 where `g.out`={`h`, `i`}, we have $F^T = \{($`p`, 0.8), (`r`, 1.0)$\}$. F^T is used in CRAWL_FULL, CRAWL_TOP, and CRAWL_RAND. TC is a cached copy of crawled nodes in T.

PRI: We first describe our baseline algorithm. PRI uses CRAWL_X = null and `MATCH_X` = `MATCH_PRI`. This strategy 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: Next we will explain an algorithm when we have full knowledge of all profiles in T and outlinks of each of candidate C (defined in Line 1 of

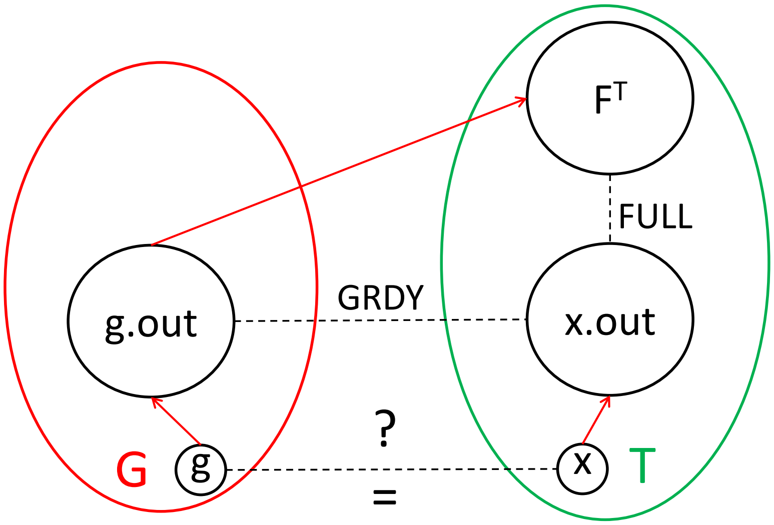


Figure 19: Illustration of Our Strategies

Algorithm 1), i.e., we know `x.out` for each `x` $\in C$. FULL uses CRAWL_X = CRAWL_FULL (Algorithm 2) and `MATCH_X` = `MATCH_ADV`. The intuition for CRAWL_FULL is that matching nodes should not only have similar node properties but should also have similar 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 "friend" nodes.

Next we look at FULL visually (see Figure 19). 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` $\in C$.

Finally, we describe CRAWL_FULL in detail. The for loop (Line 1-12) fully constructs F^T as described in the previous paragraph. Using what we have learned in Section 5.5.2, CRAWL_FULL keeps only the top candidate for each of `g.out`, i.e., C_F=1. Note that Line 15 collects `c.out` which `MATCH_ADV` needs for computing the component
`Overlap[g, c, G, T]`.

Next we describe algorithms for when we have limited 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. TOP uses CRAWL_X = CRAWL_TOP (Algorithm 3) and `MATCH_X` = `MATCH_ADV`. 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 data analyses, 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., 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 CRAWL_TOP (Algorithm 3). We first define subroutines we use in Algorithm 3: $Pop(X)$ and $IncompleteCandidates(X)$. Given an ordered list X, $Pop(X)$ returns the first member of X and removes it from X. Given a set of candidates X, $IncompleteCandidates(X)$ 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 CRAWL_TOP in detail. CRAWL_TOP consists of three while loops. The first two loops (Line 5-15 and Line 16-22) construct F^T. The first loop (Line 5-15) looks up screen_names for nodes in `g.out` that have their T screen_name on their G profile page. The second loop (Line 16-22) 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 `MATCH_PRI`. The last loop (Line 25-30) crawls outlinks on T for candidates for eventually matching `g`. This loop repeats the following two steps until we exhaust the budget: **1)** ranking the best candidate to crawl next, and **2)** crawling the best candidate (Line 26 and Line 28 respectively). Recall that `MATCH_ADV` uses graph features computed from the weighted intersection of F^T and `x.out` (`Overlap`).

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 uses CRAWL_X = CRAWL_RAND (Algorithm 4) and `MATCH_X` = `MATCH_ADV`. CRAWL_RAND is very similar to CRAWL_TOP except for two differences. The first difference is in Lines 1-2 where instead of sorting F^G by indegree in ascending order, CRAWL_RAND places in F^G a random permutation of the outlinks (`g.out`). The second difference is in Line 26 where we choose the top candidate to crawl its outlinks: here CRAWL_RAND selects a random candidate.

GRDY: Finally, we developed a greedy algorithm that exploits only one type of API call: `friends/list`. GRDY uses CRAWL_X = CRAWL_GRDY (Algorithm 5) and `MATCH_X` = `MATCH_GRDY`. Line 2-7 of CRAWL_GRDY is similar to the last loop of CRAWL_TOP (Line 25-30) because it also crawls outlinks on T for candidates for `g`.

We also look at GRDY visually (see Figure 19). For a target node `g` and a candidate `x`, CRAWL_GRDY first crawls outlinks `g.out` on G and `x.out` on T. Then, `MATCH_GRDY` computes a match score based on how much `g.out` and `x.out` overlap.

Next we explain in detail `GreedyOverlap` and `MATCH_GRDY`.

Algorithm 5 CRAWL_GRDY(B, STATE)

Input: budget values per round B, current state STATE

Output: `STATE`, update the current state in place.

```
$C' \leftarrow IncompleteCandidates(C)$
while $C' \neq nil$ AND $r_{out} < b_{out}$ do
    $c \leftarrow \arg\max_{x \in C'}$ MATCH_GRDY[g, x, true]
    $r_{out} += 1$
    # Store c.out in $TC$
    Add to c.out friends/list[c.id] (make 1 call)
    $C' \leftarrow IncompleteCandidates(C')$
end while
```

Algorithm 6 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 $total\_score$
```

- `GreedyOverlap[g, t]` (Algorithm 6): Given a G node `g` and a T node `t`, `GreedyOverlap` estimates the 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

Greedy

1. Weighted Jaccard Similarity:

$$(a)\quad \frac{\texttt{GreedyOverlap}[\texttt{g},\texttt{t},\texttt{G},\texttt{T}]}{|\texttt{g.out}|}$$

$$(b)\quad \frac{\texttt{GreedyOverlap}[\texttt{g},\texttt{t},\texttt{G},\texttt{T}]}{|\texttt{t.out}|}$$

2. Known Greedy Overlap Score (s_{kg}):

$$\frac{\texttt{GreedyOverlap}[\texttt{g},\texttt{t},\texttt{G},\texttt{T}]+1}{|\texttt{g.out}|+|\texttt{t.out}|+1}$$

3. Unknown Greedy Overlap Score (s_{ug}):

$$\frac{\texttt{GreedyOverlap}[\texttt{g},\texttt{t},\texttt{G},\texttt{T}]+|\texttt{UnknownOut}[\texttt{t},\texttt{T}]|+1}{|\texttt{g.out}|+|\texttt{UnknownOut}[\texttt{t},\texttt{T}]|+1}$$

Table 5: Greedy Features for computing `MATCH_GRDY`

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 [12, 24] 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 maximal strategy.

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

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

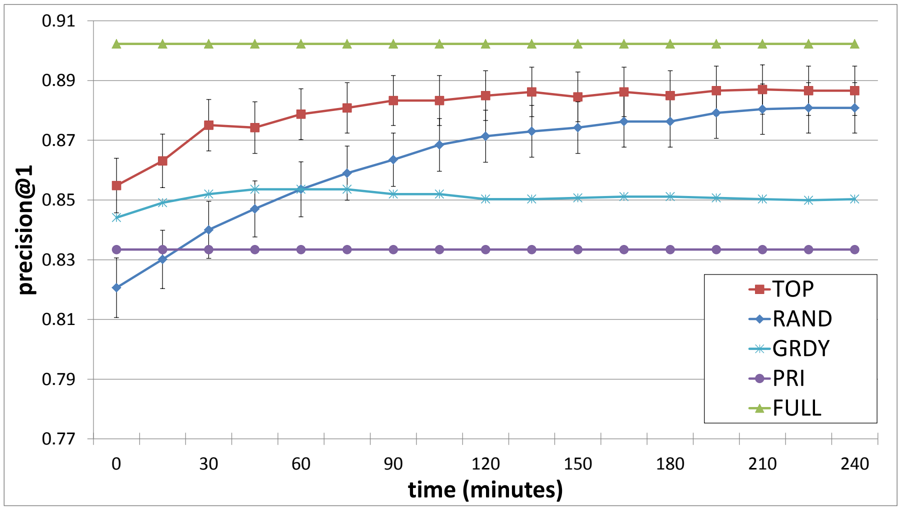


Figure 20: Comparing All Strategies, 15-min windows, 80% Confidence

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.

7.1 Experimental Setup

We first use dataset GTR to train `MATCH_PRI`, `MATCH_ADV`, and `MATCH_GRDY` and then use datasets GTE and GRND for evaluation. We have three budget values per round $B = [b_{lookup}, b_{search}, b_{out}]$. For each node in GTE (same for GRND) `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, 20]`.

7.2 Evaluation of Crawling Strategies

In this section, we evaluate different crawling strategies (CRAWL_X). We will drop the prefix CRAWL in this section to save space. Different crawlers collect different versions of the network cache TC. We want to evaluate the output of the crawlers. To do this, we take the output TC, score the candidates based on the appropriate `MATCH_X`, and select the top candidate. Our metric is *precision*@1: looking at the top candidate `t`, what fraction of the cases, `t` indeed matches `g`. Given this type of evaluation, we use GTE because a match always exist in the candidate set of each node.

We first present results for the actual Twitter API settings [10] as of August 1st, 2014 where the resetting window size is 15 minutes and $B = [18000, 180, 15]$.

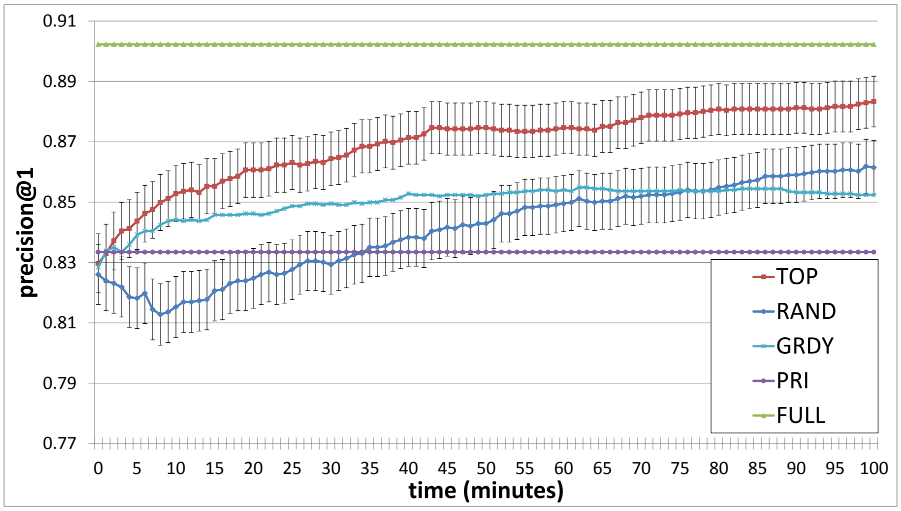


Figure 21: Comparing All Strategies, 1-min windows, 80% Confidence

A resetting window is the time between the operation quota resets. The budget is restored for each new window. This maps nicely to our strategies where we have B budget values per round and N number of rounds (i.e., one round is one window). 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 20 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 different crawling strategies 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 the appropriate `MATCH_X` and current knowledge of T (TC), 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 (`MATCH_PRI`) and does not need to make API calls to crawl T, thus the curve for PRI is flat. Our upper bound, FULL achieves 90% precision (see Figure 20). Recall from Section 6 and Algorithm 2 that FULL fully constructs F^T and `x.out` for each candidate $\mathtt{x} \in C$ 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 C, **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 `MATCH_ADV` 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 2% higher precision than the baseline PRI while RAND achieves

82% precision (1% below PRI). At 30 minutes (third window), TOP reaches 87.5% precision while RAND only starts to beat PRI. As time progresses, TOP continues to achieve higher precision and at 135 minutes stabilizes at 88.6%. This result shows that spending more than two hours (120 minutes) for a match task will increase precision only 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 1% lower precision than PRI because crawling the wrong friend (`g.out`) and candidate actually confuses `MATCH_ADV`. For example, if RAND randomly chooses Lady Gaga from `g.out` to find a match on Twitter (and successfully finds her using `BEST_MATCH_PRI`) and randomly crawls a candidate who happens to follow Lady Gaga. RAND will score this candidate higher because `Overlap` will be positive.

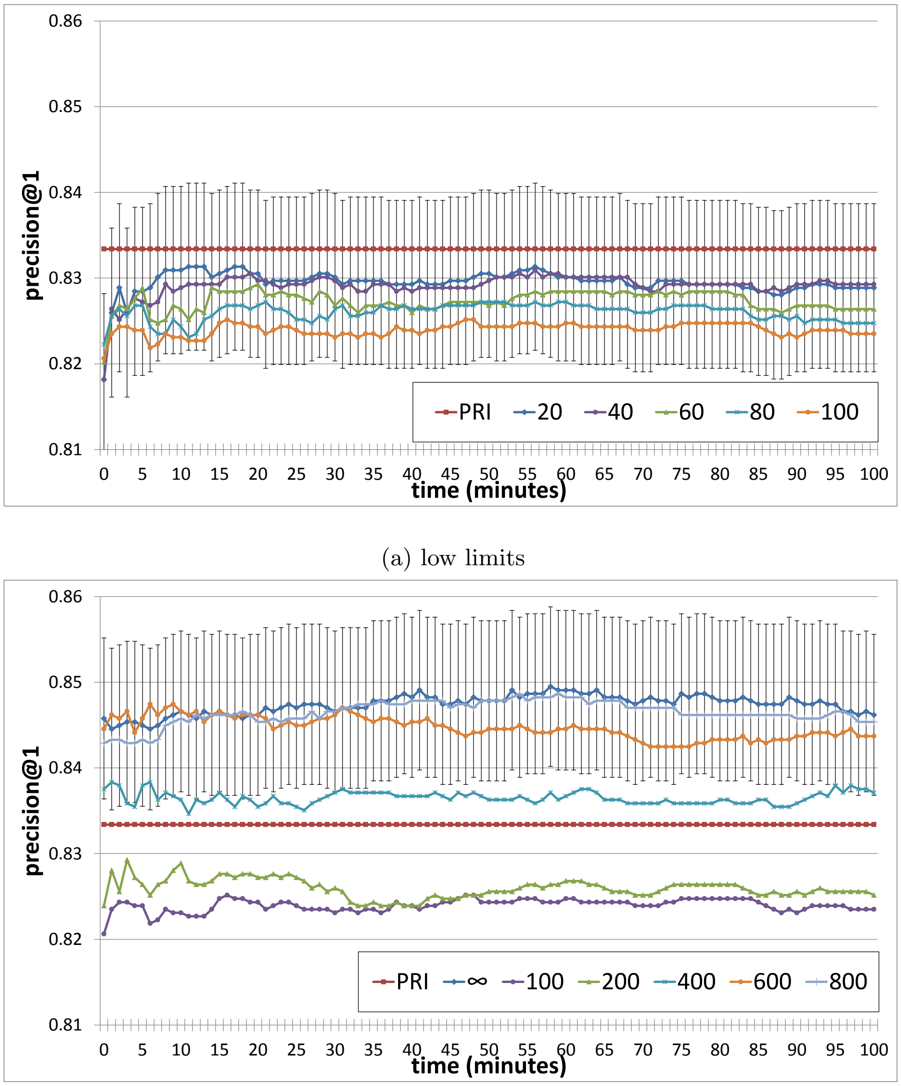
GRDY: Next we describe the performance of GRDY as compared to other strategies. At time 0 (first window), GRDY achieves about 1.1% higher precision than PRI. GRDY reaches its maximum precision at 45 minutes at about 85.4% and declines slightly thereafter (statistically insignificant). Surprisingly, as we crawl more of outlinks of candidates (`x.out`), it is harder to match `g.out` correctly using our greedy strategy (recall Figure 19). TOP beat GRDY at all times while RAND only starts to beat GRDY at 60 minutes.

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 21 shows that RAND takes 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 2 minutes while RAND starts to beat PRI only at 35 minutes (RAND requires 94% more time than TOP to reach the same 83.3% precision). At time 0, TOP, RAND, and GRDY achieve slightly lower precision than PRI because of the same reason given in the 15-min case. TOP beats GRDY at all times while RAND only starts to beat GRDY at 75 minutes. Therefore, the best strategy is TOP.

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 405 minutes over RAND to reach the highest precision value of 88.6%. And this precision value is only 1.6% below the precision of FULL. For the 1-min window case, however, TOP is still the most effective.

We also studied varying the maximum number of `g.out` to compute `GreedyOverlap`. Let us call this threshold α. We first sort each `f` in `g.out` by



(a) low limits

(b) high limits

Figure 22: GRDY, *precision*@1, 80% Confidence

f.indegree in ascending order based on results in Section ??. 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.out, we only choose the 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 22(a) shows that when we have low limits $\alpha <= 100$, choosing the smallest α (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 22(b) shows that higher α achieves higher precision. GRDY starts to beat PRI at $\alpha = 400$. Threshold $\alpha >= 600$ achieve similar precision. Threshold $\alpha >= \infty$ achieves the highest precision at 84.95%.

We have also studied *precision*@k (for $k > 1$) and the results are analogous (see Figures 23 - 24). For *precision*@2 (Figure 23), TOP achieves 95% precision while PRI and FULL achieve 91% and 95.5% respectively. For *precision*@3 (Figure 24), TOP and RAND achieves 96.5% precision while PRI and FULL achieve 94% and 96.7% respectively. Figures 20 - 24 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.

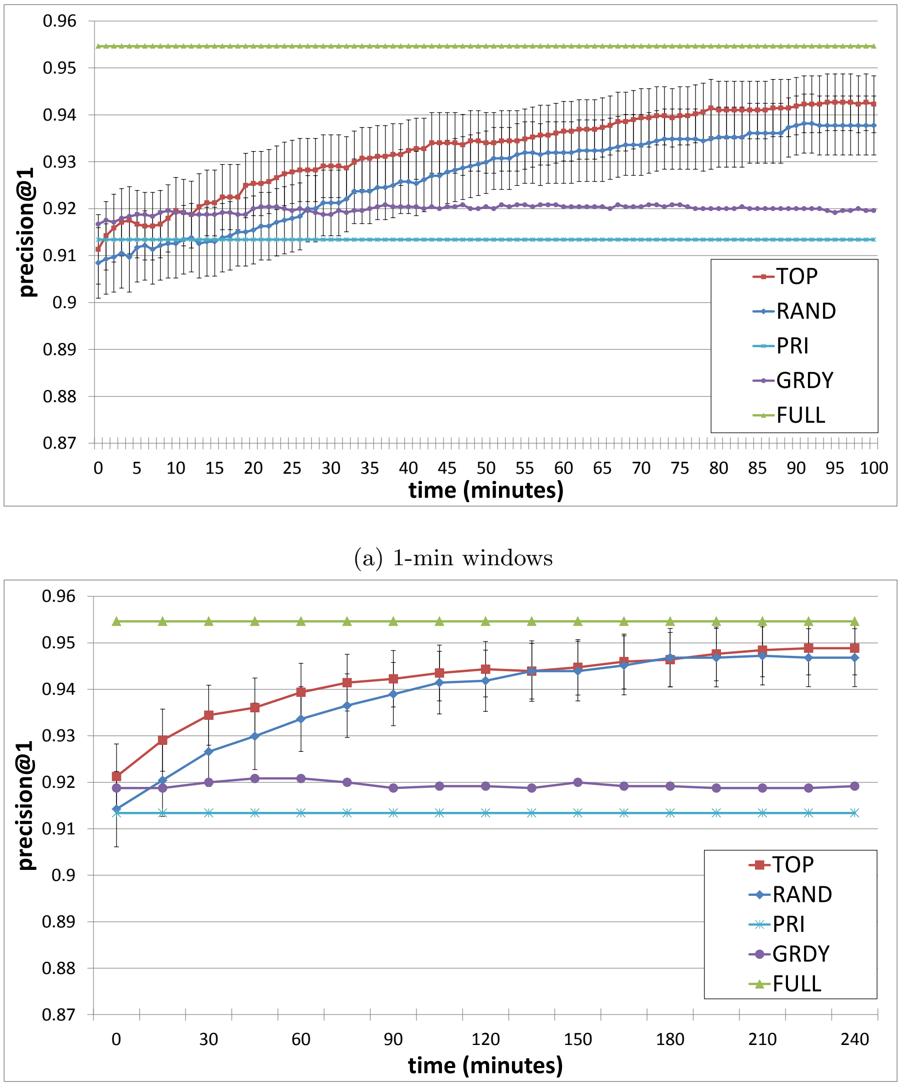
7.3 Evaluation of Whole Strategies

Next we evaluate our full algorithms (Algorithm 1 with particular CRAWL_X and `MATCH_X` selected). We use GRND and use the metrics *precision* and *recall* (defined below) because it is appropriate for studying whole strategies. Recall that for GRND, there may not be a match in the candidate set of each node. We use the same Twitter API settings as in the beginning of Section 7.2 (15-min windows). For algorithms whose performance varies over time, we use the time limit one hour because one hour is enough for our algorithms to deliver high-accuracy results.

Next we explain possible outcomes for each node in GRND. All 5 possible outcomes are shown in Figure 25. First there are two cases for the output of the algorithm: node found or not found (either no candidate or all candidates score below the threshold θ). Second there are two cases for the manual finding: match exists or no match. Finally, if both the algorithm and the manual finding report a match, there are two possible outcomes: they agree (correct) or disagree (incorrect).

Based on the outcomes, we can define precision and recall in various ways. Here we define precision and recall as follows: $precision = \frac{a}{a+b+c}$ and $recall = \frac{a}{a+b+d}$. Precision is defined as the fraction of cases where the selection was indeed the correct matching T node for $\mathtt{g}$. Increasing the match score threshold θ increases precision but decreases recall.

Figure 26 summarizes our results. To obtain Figure 26, first we select a desired recall (DR). Then we select θ that maximizes the achieved precision while keeping recall at least DR. Finally, we plot precision (x-axis) vs. DR (y-axis).

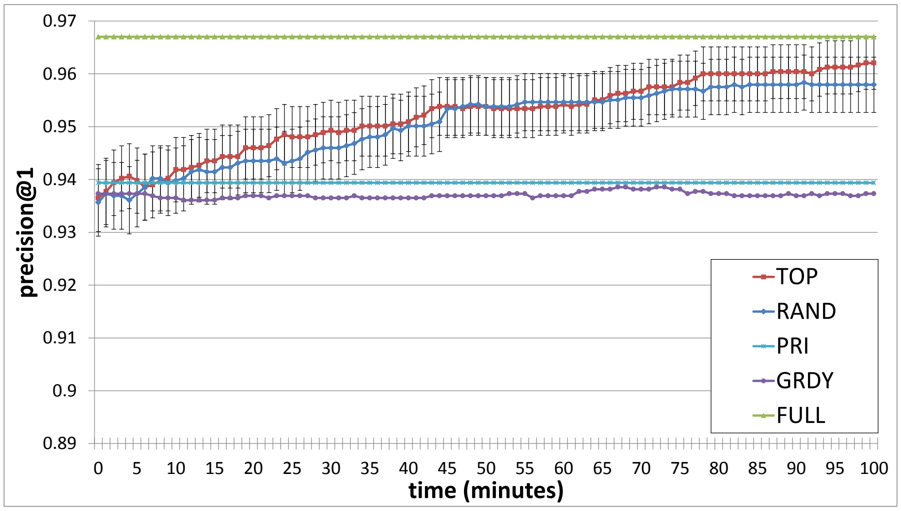


(a) 1-min windows

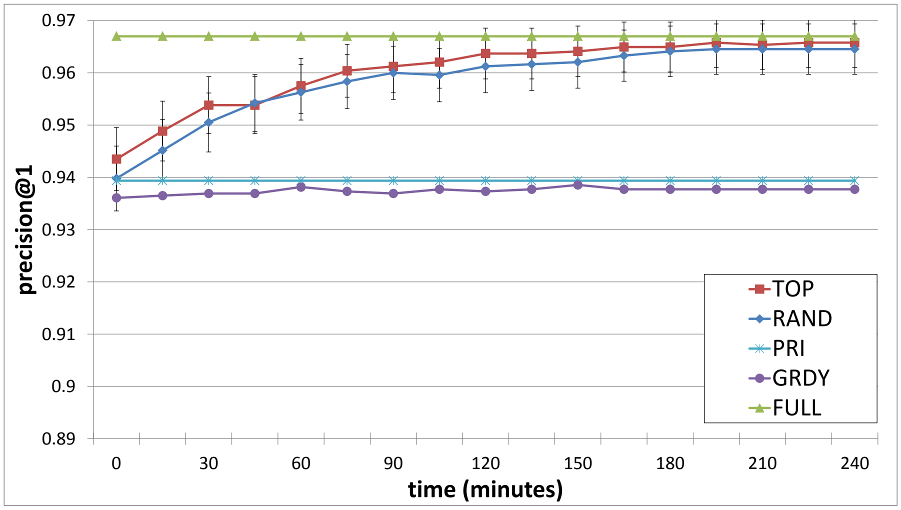
(b) 15-min windows

Figure 23: Comparing All Strategies, *precision*@2, 80% Confidence

Figure 26 shows that TOP is the best strategy overall. For example, when DR is 40%, TOP, RAND, and FULL achieve over 71% precision whereas GRDY and PRI achieve below 47% precision. TOP, RAND, and FULL achieve almost the same precision for most DR levels. Therefore, one hour is enough for TOP and RAND to catch up with FULL. The precision improvement of TOP over PRI is significant (up to 30% increase). GRDY performs about the same as



(a) 1-min windows



(b) 15-min windows

Figure 24: Comparing All Strategies, *precision*@3, 80% Confidence

PRI because it greedily matches outlinks of the target node to outlinks of the candidate. Thus GRDY is biased in favor of candidates with high outdegrees.

It is worth noting that the highest possible recall we can achieve is 75%. This is because for some nodes in GRND, a match exists (findable by a human with a lot of additional work and data) but the matching node is not in our limited candidate set. TOP, RAND, and FULL are able to achieve 63.5% recall

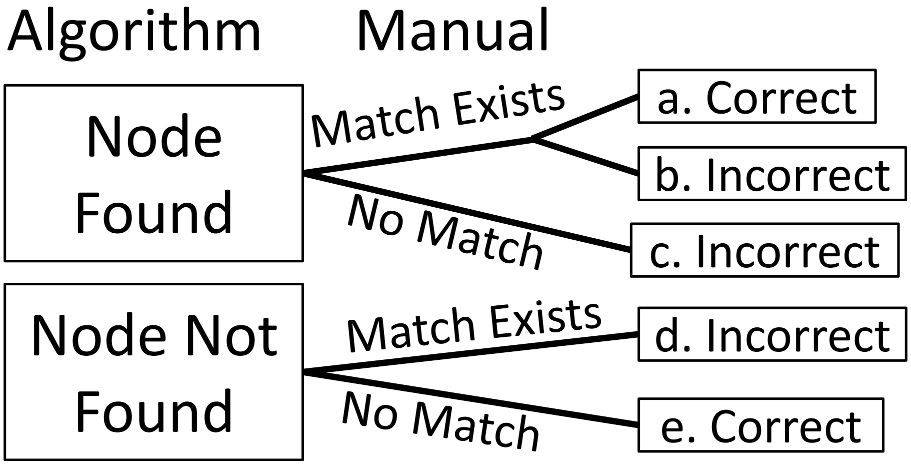


Figure 25: Possible Outcomes for each Node in GRND

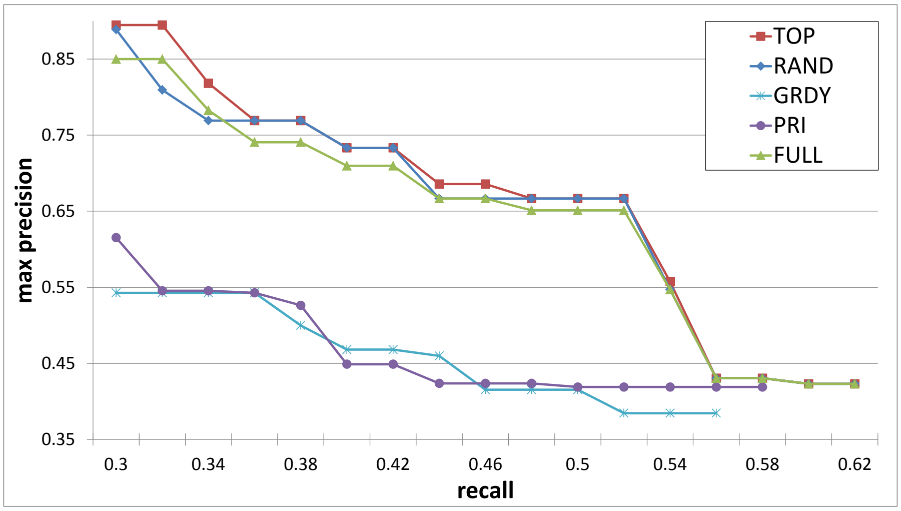


Figure 26: Comparing All Strategies, 1 hour, desired recall vs. max precision

(at least 4% higher than other algorithms). However, at this level of recall, the best precision is only about 42%.

8 Related Work

Entity Resolution (ER) has been well studied (see [25] 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 [21] for a recent survey), social networks, and de-anonymization. Recently, there has been a vast amount of work done on analyzing social graphs (see [13] for a recent book). For example, Schiöberg et al.[22] crawled over 95% of Google+ profiles at the birth of Google+ (Sep-Oct 2011) and presented the characteristics of Google+. Magno et al.[19] conducted a similar study in Dec 2011 by crawling 27M Google+ profiles and 580M links. Kwak et al.[18] 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). Korula et al. and Narayanan et al. summarized the recent literature in [17, 20]. 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. [12], 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.

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. Algorithms in preexisting literature (referenced in our paper) were designed assuming the accessibility of the full social graph. We modified and improved several existing algorithms to make them applicable to the real-world constraints of limited API calls. We also developed new ideas like known and unknown overlap scores.

9 Conclusion

We have studied the problem of ER with limited information. We have proposed two heuristic exploration algorithms: TOP and GRDY. Algorithm GRDY slightly beats our baseline Algorithm PRI for the GTE dataset but performs slightly worse for the GRND dataset because it does not exploit other API calls. Algorithm TOP is the best strategy for both GTE and GRND datasets because it efficiently explores the network and achieves high precision quickly. In some cases, the gains in precision that Algorithm TOP achieves are signficant: we can go from 45% precision (Algorithm PRI) to about 75% precision (Algorithm TOP) in one hour while keeping the desired recall at 40% for GRND. In other cases, the gains in precision are relatively modest. For example, we can go from 83% precision (Algorithm PRI) to about 88% precision (Algorithm TOP) in a couple of hours for GTE.

For some applications, the 5% increase in precision (for GTE) may or may not be worth the effort of crawling Twitter. 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.

Although the objective of our experiments is to match Google+ nodes to Twitter nodes, we believe our strategies can be applied directly (using the same algorithms and tuning the weights) to other social networks with limited API access (e.g., Flickr and Instagram). One reason we can generalize is because Twitter set the standard interface for social network APIs (i.e., other APIs have the same operations such as searching for a specific user and collecting user relationships). Another reason is that our concepts (e.g., exploration and evaluation, approximate neighborhood matching) are applicable to other scenarios. Our preliminary study and implementation of the inverse matching problem (from Twitter to Google+) uses very similar code and also shows analogous results.

References

[1] https://dev.twitter.com/docs/api/.

[2] https://plus.google.com.

[3] https://twitter.com.

[4] https://developers.google.com/+/api/.

[5] http://docs.python.org/2/library/difflib.html.

[6] About public and protected Tweets. https://support.twitter.com/articles/14016-about-public-and-protected-tweets, 2014.

[7] GET friendships/show. https://dev.twitter.com/docs/api/1.1/get/friendships/show, 2014.

[8] GET users/search. https://dev.twitter.com/docs/api/1.1/get/users/search, 2014.

[9] Limits on circles. https://support.google.com/plus/answer/1733011?hl=en, 2014.

[10] REST API v1.1 limits per window by resource. https://dev.twitter.com/docs/rate-limiting/1.1/limits, 2014.

[11] Sergey Brin Google+ page. https://plus.google.com/+SergeyBrin/about, 2014.

[12] 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.

[13] E. David and K. Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, New York, NY, USA, 2010.

[14] I. A. Gray. DO YOU KNOW THE TWITTER LIMITS? `http://iag.me/socialmedia/guides/do-you-know-the-twitter-limits/`, 2012.

[15] C. Grinstead and J. Snell. *Introduction to Probability.* 2nd ed. American Mathematical Society, 1997.

[16] +Interactive. Unofficial Google+'s Recommended users. `http://www.recommendedusers.com`, 2014.

[17] N. Korula and S. Lattanzi. An efficient reconciliation algorithm for social networks. *PVLDB*, 7(5):377–388, 2014.

[18] 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.

[19] 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.

[20] 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.

[21] C. Olston and M. Najork. Web crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, Mar. 2010.

[22] 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.

[23] M. Sippey. Changes coming in version 1.1 of the Twitter API. `https://dev.twitter.com/blog/changes-coming-to-twitter-api`, 2012.

[24] 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.

[25] S. E. Whang and H. Garcia-Molina. Developments in generic entity resolution. *IEEE Data Eng. Bull.*, 34(3):51–59, 2011.

[26] K. Wickre. Celebrating #Twitter7. `https://blog.twitter.com/2013/celebrating-twitter7`, 2014.

[27] A. M. Winkler. Confidence intervals for Bernoulli trials. `http://brainder.org/2012/04/21/confidence-intervals-for-bernoulli-trials/`, 2012.