HAL
open science

Scaling Out Link Prediction with SNAPLE

Anne-Marie Kermarrec, François Taïani, Juan Manuel Tirado Martin

To cite this version:

Anne-Marie Kermarrec, François Taïani, Juan Manuel Tirado Martin. Scaling Out Link Prediction with SNAPLE. 16th Annual ACM/IFIP/USENIX Middleware Conference, Dec 2015, Vancouver, Canada. pp.12, 10.1145/2814576.2814810 . hal-01244663

HAL Id: hal-01244663

https://inria.hal.science/hal-01244663

Submitted on 16 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scaling Out Link Prediction with SNAPLE: 1 Billion Edges and Beyond

Anne-Marie Kermarrec
INRIA Rennes, France
anne-marie.kermarrec@inria.fr

Francois Taiani
U. of Rennes 1 - IRISA - ESIR
Rennes, France
ftaiani@irisa.fr

Juan M. Tirado*
University of Cambridge
Cambridge UK
jmt78@cl.cam.ac.uk

ABSTRACT

A growing number of organizations are seeking to analyze extra large graphs in a timely and resource-efficient manner. With some graphs containing well over a billion elements, these organizations are turning to distributed graph-computing platforms that can scale out easily in existing data-centers and clouds. Unfortunately such platforms usually impose programming models that can be ill suited to typical graph computations, fundamentally undermining their potential benefits.

In this paper, we consider how the emblematic problem of link-prediction can be implemented efficiently in gather-apply-scatter (GAS) platforms, a popular distributed graph-computation model. Our proposal, called SNAPLE, exploits a novel highly-localized vertex scoring technique, and minimizes the cost of data flow while maintaining prediction quality. When used within GraphLab, SNAPLE can scale to very large graphs that a standard implementation of link prediction on GraphLab cannot handle. More precisely, we show that SNAPLE can process a graph containing 1.4 billions edges on a 256 cores cluster in less than three minutes, with no penalty in the quality of predictions. This result corresponds to an over-linear speedup of 30 against a 20-core standalone machine running a non-distributed state-of-the-art solution.

Keywords

link prediction, graph processing, parallelism, distribution, scalability

1. INTRODUCTION

Graph computing is today emerging as a critical service for many large-scale on-line applications. Companies such as Twitter, Facebook, and Linked-In are capturing, storing, and analyzing increasingly large amounts of connected data stored as graphs. As the size of these graphs increases, these companies are moving away from standalone one-machine deployments [12] and are instead looking for distributed solutions [27, 25] that can harvest the resources of multiple machines to process these graphs in parallel. Distribution unfortunately comes with an extra complexity, which can considerably hamper a solution's scalability if not properly managed. To work around this challenge, distributed graph processing engines[1] offer optimized programming models (gather-apply-scatter, bulk synchronous processing [43]) that limit the propagation of data to well-defined points of the execution and the graph. Fitting an existing graph algorithm to these models, while controlling the networking costs this creates, is unfortunately a difficult task that remains today more a craft than a science.

In this paper, we focus on the particular problem of link-prediction [22] in large graphs, an emblematic graph analysis task that appears in numerous applications (content recommendations, advertising, social mining, forensics). Implementing link prediction on a distributed graph engine raises two critical challenges: (1) traditional link prediction approaches are ill-fitted to the programming models of graph processing engines; (2) because of this bad fit communication costs can be difficult to keep under control, reducing the benefits of distribution.

More precisely, the link prediction problem considers a graph G in which some edges are missing, and tries to predict these missing edges. To be able to scale, most practical solutions search for missing edges in the vicinity of individual vertices, using bounded graph traversal techniques such as bounded random walks, or d-hops neighborhood traversal.

Unfortunately, these graph traversal techniques requires large amounts of information to be propagated between vertices, and do not lend themselves to the highly localized models offered by distributed graph engines, such as the Bulk Synchronous Processing (BSP) model of Pregel [27], or the Gather Apply Scatter (GAS) model of GraphLab [25]. In both cases, a naive application of traversal techniques requires vertex information to be replicated to maintain locality, and can lead to high communication and memory costs. This is particularly true of graphs in which the likelihood of two vertices being neighbors can be computed easily, but which require large amounts of information to be shared between vertices, such as in social graphs with large user profiles.

In this paper, we propose a highly scalable approach to link-prediction that can be implemented efficiently within

*The work presented was performed while Juan was with Inria.

[1] *distributed graph engines* for short

the Gather Apply Scatter (GAS) model. The resulting system, called SNAPLE, relies on a scoring framework of potential edges that eschews large data flows along graph edges. Instead, SNAPLE combines and aggregates similarity scores along the paths of the original graph, and thus avoids explicit and costly graph traversal operations. We demonstrate the benefits of SNAPLE with a prototype based on Graphlab [25], which we evaluate using real datasets deployed on top of a testbed with 32 nodes and 256 cores. Our experiments show that SNAPLE's performance goes well beyond that of a standard GAS implementation, and is able to process a graph containing 1.4 billions edges in 2min57s on our testbed, when a naive GraphLab version fails due to resource exhaustion. We obtain these results with no penalty in prediction quality compared to a traditional approach. SNAPLE further demonstrates linear or over-linear speedups in computation time against a single machine deployment running a state-of-the-art non-distributed solution while improving predictions.

In the following we first describe link-prediction in more detail, and introduce the Gather Scatter Apply (GAS) programming model (Section 2). In Section 3, we present the principles of SNAPLE, our link prediction framework. We then detail how SNAPLE can be implemented efficiently on a GAS engine in Section 4. Section 5 presents an exhaustive evaluation of our approach. Finally, Section 6 discusses related work, and Section 7 concludes.

2. BACKGROUND AND PROBLEM

2.1 Link-prediction.

Link prediction [22] seeks to predict missing edges from a graph G. Edges might be missing because the graph is evolving over time (users create new social links), or because G only captures a part of a wider ground truth. Predicted edges can be used to recommend new users (social graphs), new items (bipartite graph), or uncover missing information (social mining).

Formally, link prediction considers two graphs[2] $G = (V, E)$ and $G' = (V, E')$ so that G contains less information than G' in the form of missing edges: $E \subsetneq E'$. For instance G and G' might represent the same social graph captured at different points in time, or G might represent an incomplete snapshot of a larger graph represented by G' (an interaction network, a set of related topics, etc.). The goal of link prediction is then to predict which are the edges of G' that G lacks, i.e. to determine $E' \setminus E$.

Link prediction strategies fall into unsupervised and supervised approaches. The typical approach for unsupervised link prediction is sketched in Algorithm 1. The algorithm executes a parallelizable loop that iterates through all vertices of G (lines 1-3). Each iteration hinges on the scoring function $\text{score}(u, z)$ (line 2) which reflects how likely the edge (u, z) is to appear in G'. In this basic version, the algorithm scores all vertices z that are not already in u's neighborhood (noted $\Gamma_G(u) = \{v \in V | (u, v) \in E\}$), and returns the k vertices with the highest scores as predicted neighbors for u (operator $\mathbf{argtop}^k$).

The function $\text{score}(u, z)$ may only use topological properties, such as the connectivity-based metrics proposed in the seminal work of Liben-Nowell and Kleinberg [22] (e.g. the number of common neighbors between u and z, $|\Gamma_G(u) \cap \Gamma_G(z)|$). This score may also exploit vertex content, i.e. the additional application-dependent knowledge attached to vertices [7, 16, 31, 32, 39], such as user profiles, tags, or documents. In many domains, pure topological metrics tend to be the main drivers of link generation, and are therefore almost always present in the prediction process. Scoring functions using only topological metrics are also more generic as they do not rely on information that is external to the graph (tags, content, user profiles, etc.) and might not be available in all data sets.

Supervised approaches build upon unsupervised strategies and leverage machine-learning algorithms to produce optimized scoring functions [23, 38]. Supervised approaches tend to perform better, but at the cost of an important learning effort, as they must often scan the whole graph to build an accurate classification model. In this paper, we therefore focus on unsupervised approaches, but the key ideas we present can be extended to supervised schemes.

Algorithm 1 Unsupervised top k link-prediction

Require: k,score
1: **for** $u \in V$ **do**
2: $\quad prediction_u \leftarrow \underset{z \in V \setminus \Gamma_G(u)}{\mathbf{argtop}^k} \big(\text{score}(u, z)\big)$
3: **end for**

2.2 Scaling link prediction

While research on link prediction originally sought to maximize the quality of predicted edges, with little consideration for computation costs, its practical relevance for social networks and recommendation services has put it at the forefront of current system research. One critical challenge faced by current implementations is the fast growing size of the graphs they must process [12, 21].

A first strategy to scale Algorithm 1 is to improve the performance of individual iterations. A frequent optimization limits the search for missing edges to the vicinity of individual vertices. If one notes $\Gamma_G^K(u)$ the K-hop neighborhood of u in G, defined recursively as

$$\begin{aligned} &\Gamma_G^1(u) = \Gamma_G(u) \\ &\Gamma_G^K(u) = \Gamma_G^{K-1}(u) \cup \{z \in V | \exists v \in \Gamma_G^{K-1}(u) : (v, z) \in E\} \end{aligned} \tag{1}$$

this optimization will only consider the vertices of $\Gamma_G^K(u) \setminus \Gamma_G(u)$ at line 2 as potential new neighbors in G' (Equation 2 below), instead of the much larger set $V \setminus \Gamma_G(u)$. K is generally small (2 or 3). We use $K = 2$ in this work.

$$prediction_u \leftarrow \underset{z \in \Gamma_G^K(u) \setminus \Gamma_G(u)}{\mathbf{argtop}^k} \big(\text{score}(u, z)\big) \tag{2}$$

This optimization works well because social graphs, and field graphs in general, tend to present high clustering coefficients. As a result, most of the edges to be predicted in G' will connect vertices only separated by a few hops in G [44]. Other optimizations on standalone machines leverage specialized data structures and memory layout to exploits data locality and minimize computing costs [20, 33].

A second strategy seeks to scale Algorithm 1 horizontally by deploying it onto a distributed infrastructure. That is the case of Twitter for instance, who recently moved their

[2]For brevity's sake, we consider directed graphs in our explanations, but the same principles carry over to undirected graphs.

Who-to-Follow service to a distributed solution, from an initial single machine deployment [12]. This transition can leverage a growing number of graph processing engines [5, 6, 12, 15, 17, 26, 27, 34], which aim to facilitate the realization of scalable graph processing tasks. These engines do so by implementing highly parallelizable programming models such as *Bulk Synchronous Processing* (**BSP**) [17, 27, 34, 43], or *Gather, Apply, Scatter* (**GAS**) [25] .

2.3 The GAS model

In this work, we focus particularly on the GAS model, which can be seen as a refinement of map-reduce and BSP for graphs. Its reference implementation, GraphLab [11, 25], is particularly scalable, and was found to perform best in a recent comparison of modern graph engines across a number of typical graph computing tasks [13].

More precisely, a GAS program assumes every vertex $u \in V$ and every edge $(u, v) \in E$ of a graph $G = (V, E)$ is associated with some mutable data, noted D_u and $D_{(u,v)}$. A GAS program consists of a sequence of GAS super-steps (or *steps*). Each step comprises three conceptual *phases* that execute in parallel at each vertex and manipulate these data. (Our notation follows closely that of [11].)

1. The *gather* phase is similar to a map-and-reduce step. This phase collects the data associated with a vertex u's neighbors $\Gamma_G(u)$ and with u's out-going[3] edges $\{u\} \times \Gamma_G(u)$. It maps this data through a user-provided **gather**() function; and aggregates the result using a general **sum**() function defined by the user.

$$\Sigma \leftarrow \underset{v \in \Gamma_G(u)}{\mathbf{sum}} \big\{ \mathbf{gather}(D_u, D_{(u,v)}, D_v) \big\} \tag{3}$$

2. In the *apply* phase, the result of the gather phase, Σ, is used to update the current data of node u, D_u, using a user-defined **apply** function.

$$D'_u \leftarrow \mathbf{apply}\big(\Sigma, D_u\big) \tag{4}$$

3. Finally the *scatter* phase uses Σ and the new value of D_u to propagate information within u's neighborhood using a **scatter** function provided by the user.

$$\forall v \in \Gamma_G^{-1}(u) : D'_{(v,u)} \leftarrow \mathbf{scatter}\big(\Sigma, D'_u, D_{(v,u)}\big) \tag{5}$$

2.4 Link prediction in the GAS model

The GAS model facilitates the scheduling and parallelization of vertex operations while increasing content locality. GAS engines, however, can require some substantial effort to adapt existing graph algorithms, for two reasons. First, graph traversals, a primitive strategy of many graph algorithms, are difficult to express in the GAS paradigm without adding substantial complexity and overhead. This is because the accesses and updates of a GAS step are limited to adjacent vertices and edges.

The second difficulty pertains to the limited access to topological information offered by the GAS paradigm. In the GAS model, vertices drive and organize the computation (principally in the *gather* and *scatter* phases), but are not expected to be an object of computation *per se*, in order to deal with densely connected vertices in power-law distributions. Vertices and neighborhoods are therefore not exposed directly as accessible data.

[3]We present here the case in which the *gather* phase works on out-going edges and the *scatter* phase on incoming edges, but the GAS model can also use edges in the reverse direction, or even treat all edges symmetrically.

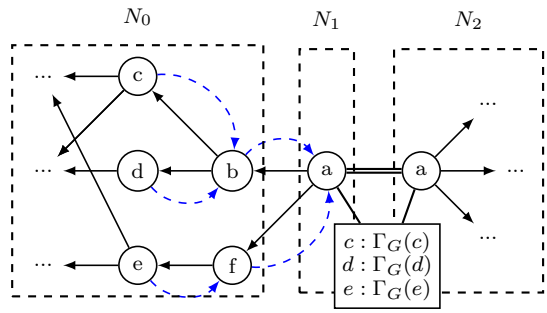


Figure 1: Example of data propagation when computing $score(a, c)$, $score(a, d)$ and $score(a, e)$ with a naive GAS approach deployed on three computing nodes ($N_{0,1,2}$). We have to propagate $\Gamma_G(c)$, $\Gamma_G(d)$ and $\Gamma_G(e)$ through b and f (dashed blue arrows). Then, this data has to be transferred to vertex a, on a different machine (N_1), and replicated onto the replicas of a (here on N_2) for synchronization.

These two limitations are particularly problematic for the link prediction algorithm captured in equation (2). This algorithm traverses all vertices in the K-hop neighborhood $\Gamma_G^K(u)$ of individual vertices, and requires access to topological information in its 'score()' function (see Figure 1). Choosing a naive approach to work around these limitations, by propagating state, and replicating data across vertices, can be extremely counter-productive, and tend to self-defeat the design rationale of the GAS paradigm.

3. THE SNAPLE FRAMEWORK

To address the above challenges, we have developed a novel and lightweight framework that computes a similarity score ($score(u, z)$) between two vertices u and z without the costs associated with traditional similarity metrics. Our framework avoids a direct and costly computation of similarity by propagating intermediate results along the paths of the graph G. This propagation involves two low-overhead steps: in a first *path-combination step* we combine raw similarity values along 2-hop paths in G, resulting in a *path-similarity* for each 2-hop paths connecting u to z. In a second *path-aggregation step*, we then aggregate these path-similarity values to compute $score(u, z)$. These two steps are configurable with user-provided functions for combination and aggregation, and provide in effect a configurable scoring framework tailored to the GAS paradigm.

3.1 Path-combination

Our approach starts from a "raw" similarity metric, that we use as a basic building block to construct our scoring framework. We consider topological metrics that can be computed from the neighborhoods of the two nodes u and z one wishes to compare:

$$sim(u, z) = f\big(\Gamma(u), \Gamma(z)\big) \tag{6}$$

where f is a similarity metric on sets, for instance Jaccard's coefficient [35]. This approach can be extended to content-based metrics [14] by including content data in f.

Table 1: Examples of combinators $\otimes$

name	$a \otimes b$	$sim_v^\star(u,z)$
linear	$\alpha a + (1-\alpha)b$	$\alpha sim(u,v) + (1-\alpha) sim(v,z)$
eucl	$\sqrt{a^2+b^2}$	$\sqrt{sim(u,v)^2 + sim(v,z)^2}$
geom	$\sqrt{a \times b}$	$\sqrt{sim(u,v) \times sim(v,z)}$
sum	a+b	$sim(u,v) + sim(v,z)$
count	1	1

In the following, we limit ourselves to the 2-hop neighborhood of u when searching for candidate nodes, i.e. we use $K = 2$ in (2), a typical value for this parameter. As explained earlier, a first challenge when directly using the similarity shown above in the GAS model to implement the score(u,z) function of equation (2), is the inherent difficulty to access data attached to the nodes $z \in \Gamma^2(u) \setminus \Gamma(u)$, which are not direct neighbors of u. One naive solution consists in using an initial GAS step to propagate a node's information to its neighbors, and make this data accessible to neighbors of neighbors in subsequent steps.

$$D'_u.neighborhood \leftarrow \{(v, D_v) \mid v \in \Gamma(u)\} \tag{7}$$

Unfortunately, and as we will show in our evaluation, the redundant data transfer and additional storage this approach causes make it highly inefficient, yielding counterproductive results in particular on very large graphs.

In order to overcome this limitation, SNAPLE uses a *path-combination* step that returns a similarity value (termed *path-similarity*) for each 2-hop path $u \to v \to z$ connecting a source vertex u with a candidate vertex z:

$$sim_v^\star(u,z) = sim(u,v) \otimes sim(v,z) \tag{8}$$

where $\otimes$ is a binary operator that is monotonically increasing on its two parameters, such as a linear combination, or generalized means. We call the operator $\otimes$ a *combinator*[4]. Intuitively, equation (8) seeks to capture the homophily often observed in field graphs: if u is similar to v and v to z, then u is likely to be similar to z.

Table 1 lists five examples of combinators that we consider in this work: a linear combination, the Euclidean distance, the geometric mean, a plain sum (a special case of linear combination), and a basic counter (a degenerated case where all similarity values are stuck to 1).

3.2 Path-aggregation

The *path-combination* step we have just described provides a similarity value $sim_v^\star(u,z)$ for each 2-hop path $u \to v \to z$ connecting u to z. Multiple such paths are however likely to exist. For instance, in Figure 2, z can be reached from u over two 2-hop paths: $u \to v \to z$ and $u \to f \to z$, delivering two different similarity values, $sim_v^\star(u,z)$ and $sim_f^\star(u,z)$. SNAPLE aggregates these different values to obtain the final score of a node $z \in \Gamma^2(u) \setminus \Gamma(u)$:

$$score(u,z) = \bigoplus_{v \in \Gamma(u) \cap \Gamma^{-1}(z)} sim_v^\star(u,z) \tag{9}$$

where $\Gamma^{-1}(z)$ is the inverse neighborhood of z, and $\oplus$ is a multiary operator that can be decomposed in a generalized sum $\oplus^{\text{pre}}$ (the incremental application of a commutative and associative binary operator), and a normalization function $\oplus^{\text{post}}$ that takes the results of $\oplus^{\text{pre}}$ and its number of arguments as input to produce its result:

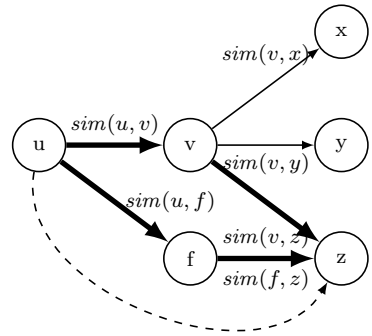
$$\bigoplus_{x \in X} x = \oplus^{\text{post}}\left(\left(\bigoplus_{x \in X}^{\text{pre}} x\right), |X|\right) \tag{10}$$

We call the operator $\oplus$ an *aggregator*. This second step, termed *path aggregation*, is akin to a reduce step, and particularly well adapted to the GAS model. Several $\oplus$ operators can be used such as addition, multiplication, selecting the largest similarity, etc. We use three in this paper, *Sum*, arithmetic means (*Mean*), and geometric means (*Geom*), which we list in Table 2.

Path-aggregation is a form of path-ensemble measure that combines a classic similarity metrics with information regarding the number of paths connecting a source and sink vertices. The combination of a similarity metric $sim()$, a combinator $\otimes$, and an aggregator $\oplus$ creates the design space of SNAPLE's similarity framework. In Table 3, we show an excerpt of the possible combinations we investigate in this work by combining elements of Tables 1 and 2.

Using Jaccard's coefficient on neighborhoods as our similarity metric, we systematically combine the first three combinators ($\oplus$) of Table 1 (*linear*, *eucl*, and *geom*) with the three aggregators of Table 2 (*Sum*, *Mean*, *Geom*) to obtain a total of nine scoring mechanisms. By adapting the similarity and combinator, we are also able to define two extra scoring methods (gray rows): a score function similar to the personalized page rank [4] (*PPR*), and a basic scoring approach (*counter*) that counts the number of 2-hop paths existing between u and z. This large set shows both the expressiveness and flexibility of the SNAPLE framework.

Individual aggregators can greatly affect the final behavior of the resulting scoring function. Figure 3 illustrates this



$$score(u,z) = \bigoplus_{w \in \{v,f\}} sim_w^\star(u,z)$$

Figure 2: Two paths connect u to z. Both paths yield different similarity values, $sim_v^\star(u,z)$ and $sim_f^\star(u,z)$, which Snaple summarizes with an *aggregator* ($\oplus$).

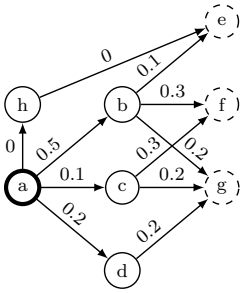
Table 2: Examples of aggregators $\oplus$

name	$a \oplus^{\text{pre}} b$	$\oplus^{\text{post}}(\sigma, n)$	$\oplus_{x \in X} x$
Sum	$a+b$	σ	$\sum_{x \in X} x$
Mean	$a+b$	$\frac{1}{n}\sigma$	$\frac{1}{\lvert X \rvert}\sum_{x \in X} x$
Geom	$a \times b$	$\sigma^{\frac{1}{n}}$	$(\prod_{x \in X} x)^{\frac{1}{\lvert X \rvert}}$

[4]We limit ourselves to 2-hop paths, but this approach can be extended to longer paths by recursively applying $\otimes$ to the raw similarities of individual edges (in functional terms, essentially executing a fold operation on the raw similarity values along the path connecting u to z).

Table 3: Examples of $score(u, z)$ combinations in Snaple's scoring framework

$sim(u, v)$	**combinator** ($\otimes$)	**aggregator** ($\oplus$)	**score name**
Jaccard	linear		linearSum
Jaccard	eucl		euclSum
Jaccard	geom	Sum	geomSum
$1/\lvert\Gamma_v\rvert$	*sum*		*PPR*
–	*count*		*counter*
Jaccard	linear		linearMean
Jaccard	eucl	Mean	euclMean
Jaccard	geom		geomMean
Jaccard	linear		linearGeom
Jaccard	eucl	Geom	euclGeom
Jaccard	geom		geomGeom



	$score(a, e)$	$score(a, f)$	$score(a, g)$
linearSum	0.3	0.6	**0.75**
linearMean	0.15	**0.3**	0.25
linearGeom	0	**0.28**	0.24

Figure 3: Example of scores obtained using different aggregators and the *linear* combinator (α=0.5). Edge labels show the similarity between vertices.

impact when using the *linear* combinator ($\alpha = 0.5$) with the three aggregators of Table 2. In this example, a is connected to e and f through two 2-hop paths, and to g through three 2-hop paths. The highest score obtained with each aggregator is shown in bold in table.

The *Mean* and *Geom* aggregators (two bottom lines in the table), average out the number of paths connecting a to each candidate vertices, and as a result, consider f the vertex most similar to a. By contrast, the *Sum* aggregator takes into account the connectivity of candidate vertices in its final score, and rates g over f (first line). This means a vertex with lower path-similarities (such as g here) can obtain a high final score if enough paths connect it to the source vertex (a). Finally, let us note how the *Geom* aggregator penalizes vertices such as e which are connected through paths with very low path-similarity (here $a \to h \to e$).

4. IMPLEMENTING SNAPLE IN GAS

The approach we have just presented addresses the challenges inherent to the GAS model that we discussed in Section 2.4. Even using Snaple's vertex score however, the candidate space $\Gamma^2(u) \setminus \Gamma(u)$ of equation (2) remains generally too large to be fully explored. Indeed, if we note n the average out-degree of vertices in G, a blind application of (2) can require up to $O(n^2)$ scores to be computed per vertex on average, leading to $O(|V|n^2)$ scoring operations for the whole graph $G = (V, E)$.

This challenge is not inherent to the GAS model, but

Algorithm 2 Snaple's link-prediction as a GAS program

Require: k, sim, $\otimes$, $\oplus^{\text{pre}}$, $\oplus^{\text{post}}$, k_{local}, thr_Γ

▷ *Step 1: Obtain a sample of u's neighbors* $D_u.\hat{\Gamma}$
1: $\mathbf{gather}_1(D_u, D_{(u,v)}, D_v)$:
2: $\quad \gamma \leftarrow \{v\}$
3: $\quad$ **if** $rand() > thr_\Gamma/|\Gamma(u)|$ **then** $\gamma \leftarrow \emptyset$;
4: $\quad$ **return** γ
5: $\mathbf{sum}_1(\gamma_1,\gamma_2)$: **return** $\gamma_1 \cup \gamma_2$ ▷ *Computing* Σ_1
6: $\mathbf{apply}_1(D_u, \Sigma_1)$: $D_u.\hat{\Gamma} \leftarrow \Sigma_1$

▷ *Step 2: Estimate similarities*
7: $\mathbf{gather}_2(D_u, D_{(u,v)}, D_v)$:
8: $\quad$ **return** $\{(v, sim(u, v) \equiv f(D_u.\hat{\Gamma}, D_v.\hat{\Gamma}))\}$
9: $\mathbf{sum}_2(\gamma_1,\gamma_2)$: **return** $\gamma_1 \cup \gamma_2$ ▷ *Computing* Σ_2
10: $\mathbf{apply}_2(D_u, \Sigma_2)$:
11: $\quad D_u.\text{sims} \leftarrow \underset{(v,s_v)\in\Sigma_2}{\mathbf{argtop}}^{k_{local}}(s_v)$

▷ *Step 3: Compute recommendations*
12: $\mathbf{gather}_3(D_u, D_{(u,v)}, D_v)$:
13: $\quad$ **if** $v \notin D_u.\text{sims}.keys$ **then return** $\emptyset$;
14: $\quad \hat{\Gamma}_u \leftarrow D_u.\hat{\Gamma}_u$; $\Gamma_v^{max} \leftarrow D_v.\text{sims}.keys$
15: $\quad$ **return** $\{(z, D_u.\text{sims}[v] \otimes D_v.\text{sims}[z], 1) \mid z \in \Gamma_v^{max} \setminus \hat{\Gamma}_u\}$
16: $\mathbf{sum}_3(\gamma_1,\gamma_2)$: **return** $\text{merge}(\oplus^{\text{pre}}, \gamma_1, \gamma_2)$ ▷ *Comp.* Σ_3
17: $\mathbf{apply}_3(D_u, \Sigma_3)$:
18: $\quad score \leftarrow \emptyset$
19: $\quad$ **for** $(z, s_z^{\text{pre}}, n_z) \in \Sigma_3$ **do** $score[z] \leftarrow \oplus^{\text{post}}(s_z^{\text{pre}}, n_z)$
20: $\quad D_u.\text{predicted} \leftarrow \underset{z\in score.keys}{\mathbf{argtop}}^{k}(score[z])$

must be addressed in our implementation to obtain a tractable solution. We attack it using a two-pronged approach: first, we truncate neighborhoods that are larger than a *truncation threshold* thr_Γ (with thr_Γ reasonably large, e.g. 200), to limit the memory overhead induced by very large neighborhoods, and minimize the cost of computing raw similarity values. We note $\hat{\Gamma}(u)$ this truncated neighborhood, which we use to compute the raw similarity values of equation (6).

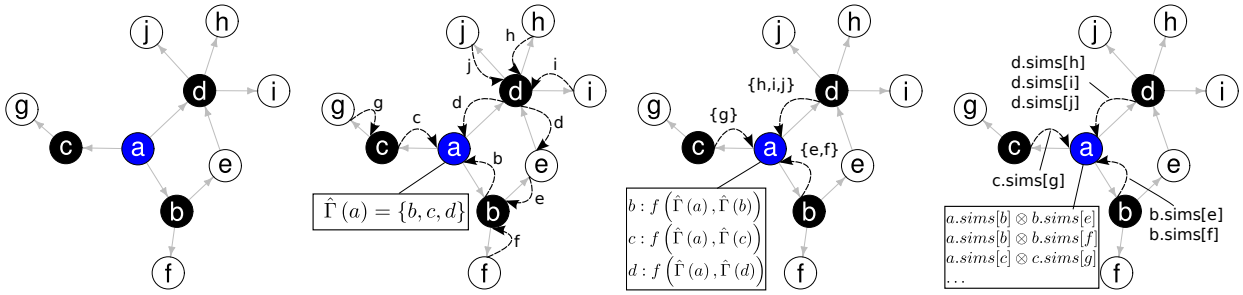
Second, we do not consider all the paths $u \to v \to z$ over $\Gamma^2(u)$, but only those paths going through the k_{local} edges with the highest similarity values at individual vertices. Said differently, we sample down $\Gamma^2(u)$ to $\left(\Gamma_{k_{local}}^{max}\right)^2(u)$ in equation (2), where

$$\Gamma_{k_{local}}^{max}(u) = \underset{v\in\Gamma(u)}{\mathbf{argtop}}^{k_{local}} f\big(\hat{\Gamma}(u), \hat{\Gamma}(z)\big) \qquad (11)$$

and $(\Gamma_{k_{local}}^{max})^2$ is defined in relation to $\Gamma_{k_{local}}$ as in (1).

The resulting GAS program is shown in Algorithm 2, and illustrated on a small example in Figure 4. The program comprises three GAS steps, each made of a *gather* (**gather** and **sum** functions), and an *apply* phase (**apply** function). We do no use any *scatter* phase. We employ the notation used in [25] where D_u, D_v and $D_{(u,v)}$ are the program state and meta-data for vertices u, v and edge (u, v) respectively.

In the first step (lines 1-6, Fig. 4b), we construct a list of neighbors by collecting the ids of adjacent vertices. To limit memory overheads, we limit the size of the neighborhood set $D_u.\hat{\Gamma}_u$ to the truncation threshold thr_Γ. This uses a uniform



(a) Initial graph (b) Step 1 (c) Step2 (d) Step 3

Figure 4: Illustrating Algorithm 2. In Step 1, each vertex constructs its list of neighbors $\hat{\Gamma}$. This list of neighbors is propagated to adjacent vertices (Step 2) to compute raw similarities sims$[\cdot]$ (for a: with c, d, and b). Only the k_{local} most similar vertices are kept. Finally (Step 3), raw similarities are used to compute path similarities, which are then combined into final scores.

random variable $rand()$ over $[0, 1]$ (line 3) to approximate the truncation under GAS constraints.

In the second step (lines 7-11, Fig. 4c), we use the truncated neighborhoods $D_x.\hat{\Gamma}_x$ to compute raw similarities between u and each of its neighbors v (line 8). At the end of this step, we only keep the top k_{local} neighbors of u (**argtop**$^{k_{local}}$ operator, line 11), and store them into a dictionary D_u.sims with their corresponding raw similarity s_v. When this second step terminates, the keys of D_u.sims are the vertices of $\Gamma^{max}_{k_{local}}(u)$ as defined in (11).

In the final step (lines 12-20, Fig. 4d), we first compute *path similarities* (Sec. 3.1) using the combinator $\otimes$ (line 15). These path similarities are limited to the k_{local} 2-hop neighborhood of u, i.e. to paths $u \rightarrow v \rightarrow z$ such that $v \in D_u$.sims.*keys* (line 13) and $z \in D_v$.sims.*keys* (line 15). We then aggregate the path-similarities leading to the same candidate vertex z as explained in Section 3.2 using the $\oplus^{\text{pre}}$ (line 16) and $\oplus^{\text{post}}$ operators (line 19). More precisely, γ_1, γ_2 and Σ_3 are sets of triplets $(z, s_z, n_z) \in V \times \mathbb{R} \times \mathbb{N}$ that associate each candidate vertex z to a similarity s_z and a counter n_z. n_z counts the number of paths over which s_z has been accumulated. The function *merge* at line 16 performs a double fold (or reduce) operation on the vertices of γ_1 and γ_2 to compute the generalized sum and the count of equation (10), which are then used at line 19:

$$merge(\oplus^{\text{pre}}, \gamma_1, \gamma_2) = \left\{ \left(z, \bigoplus^{\text{pre}}_{\substack{(z,s_z,-)\\ \in\gamma_1\cup\gamma_2}} s_z, \sum_{\substack{(z,-,n_z)\\ \in\gamma_1\cup\gamma_2}} n_z\right) \;\middle|\; z \in \gamma_{1\downarrow V} \cup \gamma_{2\downarrow V} \right\}$$

where $\gamma_{i\downarrow V}$ is the projection of γ_i on its first component V. The program finally returns the top k vertices with the best scores as predictions (line 20).

5. EVALUATION

We present a detailed experimental evaluation of SNAPLE implemented on top of the Graphlab distributed graph engine [11]. Graphlab implements the GAS paradigm over an asynchronous distributed shared memory abstraction, and is specifically designed for processing large graphs in distributed deployments.

We explore the space of possible link predictors that can be designed using SNAPLE's scoring framework and the implementation presented in Algorithm 2. In particular, we evaluate how different configurations modify the quality of link predictions, and the total computing time.

Table 4: The datasets used in the evaluation

dataset	$\lvert V\rvert$	$\lvert E\rvert$	domain
gowalla [8]	196,591	0.95M	social network
pokec [40]	1.6M	30.6M	social network
orkut [28]	3M	223M	social network
livejournal [2]	4.8M	68.9M	co-authorship
twitter-rv [18]	41M	1.4B	microblogging

5.1 Experimental setup

We have implemented SNAPLE on top of GraphLab version 2.2 compiled using GCC 4.8 with C++11 compatibility enabled. We use the Warp engine offered by GraphLab with its default configuration. We run our experiments on the Grid5000 testbed (`https://www.grid5000.fr`) using two kind of computing nodes: *type-I* and *type-II* machines. *Type-I* nodes are equipped with 2 Intel Xeon L5420 (2.5 Ghz) processors, 4 cores per processor (8 cores per node), 32 GBytes of memory, and Gigabit Ethernet. *Type-II* nodes are more powerful machines designed for big-data workloads with 2 Intel Xeon E5-2660v2 (2.2 Ghz) processors per node, possessing each 10 cores (20 cores per node), 128 GBytes of memory and 10-Gigabit ethernet connection. We deploy our experiments on up to 32 *type-I* nodes (256 cores) and up to 8 *type-II* nodes (160 cores). The experiments are executed in *type-I* machines if not indicated otherwise. All the nodes have access to a shared network file system (NFS) where we store the graphs to be loaded.

5.2 Evaluation protocol

For our experiments, we use the set of publicly available datasets described in Table 4. They consist of a set of static directed graphs, except *gowalla* and *orkut* which are undirected. We transform these last two datasets into directed graphs by duplicating edges in both directions. These datasets provide a representative set of graphs from different domains and cover a broad range of sizes, from small instances with less than 1 million edges (*gowalla*) to large graphs with over 1 billion edges (*twitter-rv*).

In order to emulate the prediction process we follow a

Table 5: Snaple clearly outperforms a direct implementation of similarity-based link-prediction on GraphLab, both in terms of recall and execution time. The results were obtained on 4 *type-II* nodes (80 cores). Gains (for recall), and speedups (for time) are shown in brackets.

	score(u,z)	*dataset* **gowalla** *recall*	*time (s)*	**pokec** *recall*	*time (s)*	**livejournal** *recall*	*time (s)*
	Baseline	0.12	119.9	0.05	213.8	0.12	1010.7
Snaple		$thr_\Gamma = \infty$, $k_{local} = \infty$					
	linearSum	0.28 (×2.3)	72.8 (×1.6)	**0.14** (×**2.8**)	60.2 (×3.5)	**0.31** (×**2.5**)	224.5 (×4.5)
	counter	**0.33** (×**2.7**)	63.1 (×1.9)	0.12 (×2.4)	59.7 (×3.5)	0.28 (×2.3)	218.0 (×4.6)
	PPR	0.26 (×2.1)	68.0 (×1.7)	0.12 (×2.4)	56.6 (×3.7)	0.30 (×2.5)	222.5 (×4.5)
		$thr_\Gamma = 20$, $k_{local} = \infty$					
	linearSum	**0.26** (×**2.1**)	66.1 (×1.8)	**0.12** (×**2.4**)	58.3 (×3.6)	0.27 (×2.2)	213.8 (×4.7)
	counter	0.23 (×1.9)	69.8 (×1.7)	0.11 (×2.2)	60.0 (×3.5)	0.26 (×2.1)	211.8 (×4.7)
	PPR	0.24 (×2.0)	68.5 (×1.7)	0.11 (×2.2)	60.1 (×3.5)	**0.28** (×**2.3**)	211.5 (×4.7)
		$thr_\Gamma = \infty$, $k_{local} = 20$					
	linearSum	**0.28** (×**2.3**)	1.1 (×109.0)	**0.13** (×**2.6**)	12.8 (×16.7)	**0.30** (×**2.5**)	32.5 (×31.0)
	counter	0.24 (×2.1)	1.0 (×119.0)	0.12 (×2.4)	13.0 (×16.4)	0.27 (×2.2)	29.6 (×34.1)
	PPR	0.26 (×2.1)	1.1 (×109.0)	0.11 (×2.2)	13.5 (×15.8)	0.29 (×2.4)	38.3 (×26.3)
		$thr_\Gamma = 20$, $k_{local} = 20$					
	linearSum	**0.26** (×**2.1**)	1.1 (×109.0)	**0.11** (×**2.2**)	9.4 (×22.7)	**0.26** (×**2.1**)	28.7 (×35.2)
	counter	0.22 (×1.8)	1.1 (×109.0)	0.10 (×2.0)	11.0 (×19.4)	0.24 (×2.0)	26.4 (×38.2)
	PPR	0.25 (×2.0)	1.2 (×99.9)	0.10 (×2.0)	11.2 (×19.0)	0.26 (×2.1)	25.7 (×39.3)

similar approach to the one taken in [36]. We randomly remove one outgoing edge from each vertex with $|\Gamma(u)| > 3$. After the execution, we obtain k (with $k = 5$ fixed) predictions for each vertex. Experiments with Snaple are parametrized by a *score combination* (taken from Table 3), a truncation threshold thr_Γ, and a sampling parameter k_{local}. Unless indicated otherwise $thr_\Gamma = 200$. We use Jaccard's coefficient as raw similarity (function f in equation (6)). The *linear combinator* is configured with $\alpha = 0.9$, which was found to return the best predictions.

We evaluate approaches against two metrics: *recall* and *execution time*. Recall is the proportion of removed edges that are successfully returned by the algorithm. Execution time is measured from when the graph has been successfully loaded (as reported by GraphLab) until after all predictions have been computed. We ignore the time spent to load the graph from primary storage, as this time depends on the GraphLab implementation and the network file system, two aspects outside the scope of this work.

Let us note that recommendation systems are usually evaluated using a second metric, *precision*, which is the proportion of correct recommendations within the returned answers. Because both the number of edges we remove from each vertex and the number of predictions we return are fixed (one edge, and $k = 5$ predictions), precision is in fact proportional to recall in our set-up, and we therefore do not report it.

5.3 Comparison with a direct implementation

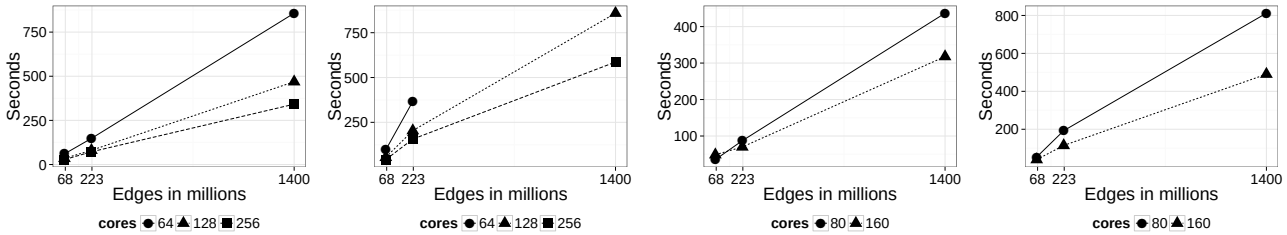
We first compare Snaple against a direct implementation of Algorithm 1 on GraphLab using Jaccard's coefficient. For a fair comparison, we limit the search of candidates to 2-hop neighborhoods, as in Snaple. We call this implementation Baseline. As discussed in Section 2.4, Baseline needs to directly compute similarity between every pair of vertices that lie 2 hops away of each other. Because the GAS model only provides access to direct neighbors, Baseline must propagate and store neighborhood information along every 2-hop paths, resulting in substantial overheads.

We compare Baseline against 12 configurations of Snaple, by using three scoring configurations from Table 3 (*linearSum*, *counter*, and *PPR*), and by varying the truncation threshold thr_Γ and the sampling parameter k_{local} between ∞ (in effect no truncation, resp. no sampling) and 20 (a low value to exacerbate the effect of each parameter). We execute Baseline and the 12 Snaple configurations on 4 *type-II* nodes (80 cores), and apply them to the datasets *gowalla*, *pokec* and *livejournal*. (*orkut* and *twitter-rv* cause Baseline to fail by exhausting the available memory.) Table 5 reports the recall values and execution times we obtain, with the recall gains (resp. speedups) shown in brackets for Snaple configurations, computed against Baseline.

The table shows that even without truncation or sampling ($thr_\Gamma = k_{local} = \infty$), Snaple clearly outperforms Baseline both in terms of recall (which more than doubles on all datasets) and execution times, with speedups ranging from 1.6 to 4.6. Truncation ($thr_\Gamma = 20$) brings a small improvement in speed-up, while causing recall to go down slightly, be it with or without k_{local}. The sampling parameter k_{local} has the largest impact on the execution time by far, yielding speedups ranging from 15.8 (*pokec*) up to 109 on *gowalla* when applied alone ($thr_\Gamma = \infty$), while having a minimal impact on recall. Adding truncation to k_{local} further improves execution times for the *pokec* (speed-ups ranging from 19 to 22.7) and *livejournal* (35.2 to 39.3) (last 3 lines of the table).

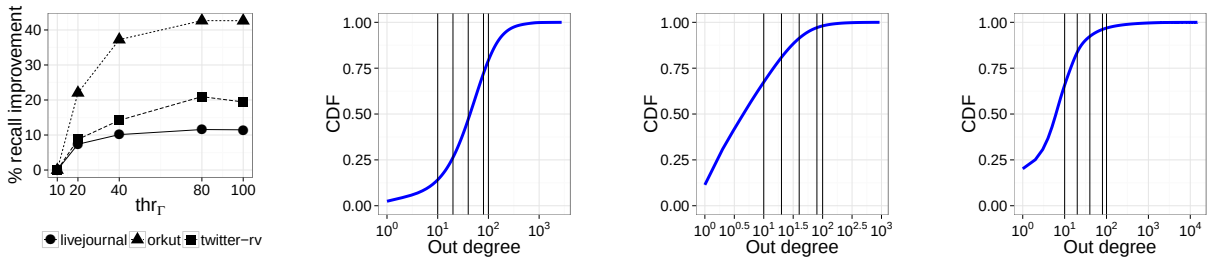
In terms of score configuration, *linearSum*, *counter*, and *PPR* tend to produce similar recall values, with a slight advantage for *linearSum* (best recalls are shown in bold). The differences in execution times between the three score configuration are within the experimental noise, and not significant.

This first experiment demonstrates the flexibility of Snaple, which offers a large number of configurations. These results also illustrate the benefits of Snaple in terms of recall and execution time, and highlight the importance of sampling to get high speedup while maintaining a good



(a) $k_{local} = 40$ and *type-I* (b) $k_{local} = 80$ and *type-I* (c) $k_{local} = 40$ and *type-II* (d) $k_{local} = 80$ and *type-II*

Figure 5: Snaple scales linearly graph sizes (measured in edges). Execution times are reported for both *type-I* and *type-II* nodes, and for $k_{local} \in \{40, 80\}$. Missing points indicate configurations not fitting into memory.



(a) Relative recall gain (b) *orkut* CDF (c) *livejournal CDF* (d) *twitter-rv* CDF

Figure 6: (a) Relative recall improvement for different thr_Γ values against $thr_\Gamma = 10$, with *linearSum*. (b,c,d) The CDF of out degrees in each dataset shows how the recall stops improving when thr_Γ goes beyond the out degree of 80% of the vertices in a dataset. Vertical bars indicate threshold values of 10, 20, 40, 80 and 100.

recall. Although the effect of thr_Γ is less pronounced than that of k_{local}, truncation remains an important mechanism to deal with highly connected vertices. In Sections 5.5 and 5.7 we return to the impact of both parameters, with a more fine-grained sensitivity analysis of their effects. We otherwise fix $thr_\Gamma = 200$ for our other experiments.

5.4 Scalability

We assess the scalability of Snaple by applying the *linearSum* score to *livejournal*, *orkut* and *twitter-rv* on varying numbers of *type-I* and *type-II* nodes. Figure 5 shows the results we obtain for two values of k_{local} (40 and 80). Other scoring configurations return near-identical results.

Snaple scales linearly with graph sizes (measured in edges), for both values of k_{local}, and on both *type-I* and *type-II* nodes. Doubling k_{local} increases the number of paths to be considered, and increases the execution time by 70%. The required memory also increases, which makes it impossible to run the experiment on *twitter-rv* with $k_{local} = 80$ and only 8 *type-I* machines (64 cores). The most exhaustive experiment we run processes the *twitter-rv* dataset (1.4 billion edges) with $k_{local} = 80$ in less than 10 minutes (585s) using 256 *type-I* cores (32 machines) or in a similar time with 160 *type-II* cores (8 machines). These executions yield a recall of 0.093. (A value we return to when we analyze the effect of k_{local} on recall and execution time in Sec. 5.7.)

5.5 Impact of the truncation threshold thr_Γ

The truncation of neighborhoods using thr_Γ (Sec. 4) serves two purposes: to limit memory overheads, in particular in the case of densely connected components, and to improve execution times (particularly in conjunction with k_{local}).

By truncating neighborhoods, we might however loose relevant information, disturb the computation of raw similarities, and finally decrease the quality of link predictions. This effect is however limited to vertices whose neighborhood is larger than thr_Γ. To investigate this phenomenon, Figures 6b, 6c and 6d show the CDF of vertex degrees in *orkut*, *livejournal* and *twitter-rv*, and superimpose five different values of thr_Γ (10, 20, 40, 80 and 100, shown as vertical lines). These figures show that already with $thr_\Gamma = 100$, only a minority of vertices will be truncated across all three datasets, including very small minorities (around 1%) in the case of *livejournal* and *twitter-rv*.

This analysis is confirmed by Figure 6a, which shows how recall improves using *linearSum* and $k_{local} = 80$, while varying thr_Γ from 10 to 100 (recall values are normalized by the recall obtained with $thr_\Gamma = 10$, and the relative improvement is shown). The impact of thr_Γ is strongest on *orkut*, whose degree distribution varies strongly in the interval of values taken by thr_Γ. In all three datasets, recall stabilizes when thr_Γ reaches 80, which is when thr_Γ covers at least 80% of all vertices in all three graphs. This shows that the impact of thr_Γ on recall can be minimized by selecting it appropriately, while limiting the memory impact of the most densely connected vertices.

5.6 Impact of the vertex selection mechanism

We limit the number of paths to be explored by sampling the neighborhood of each node to a set with the k_{local}

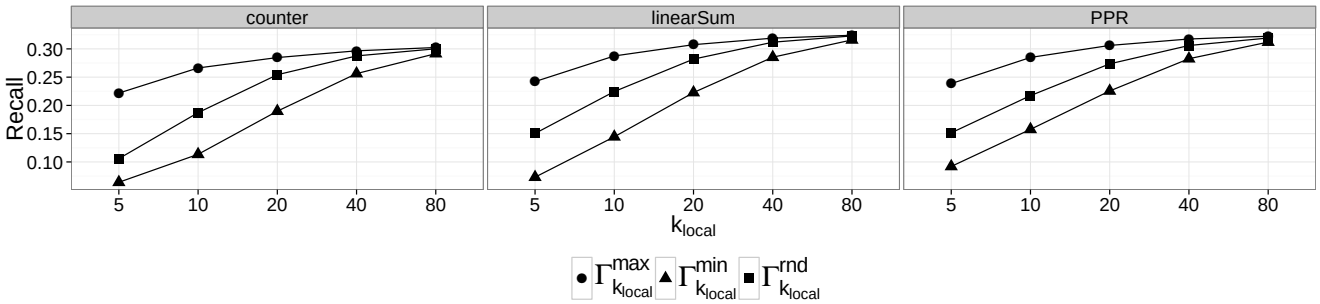


Figure 7: Recall using 3 different neighbor sampling policies for $k_{local} = 5, 10, 20, 40, 80$ with the *livejournal* dataset. Selecting the most similar vertices ($\Gamma^{max}_{k_{local}}$) improves the recall, in particular for small values of k_{local}.

most similar neighbors using the $\Gamma^{max}_{k_{local}}$ function (Step 2 of Algorithm 2). Discarding vertices reduces the computing time with the inconvenient of reducing the set of explored candidates. This has a direct impact on recall. Using $\Gamma^{max}_{k_{local}}$, we consider the most similar vertices to be good candidates for the sampling. In order to demonstrate that sampling using similarity improves the search of potential vertex candidates, and thus recall, we compare our proposed neighbors selection $\Gamma^{max}_{k_{local}}$ with $\Gamma^{min}_{k_{local}}$ and $\Gamma^{rnd}_{k_{local}}$. As mentioned $\Gamma^{max}_{k_{local}}$ selects the k_{local} neighbors with the largest similarity, the second those with the smallest similarity, and the last one selects a random neighbor.

Figure 7 shows the recall for various *score* methods in the *livejournal* dataset for different k_{local} values. As k_{local} grows the obtained recall converges for the three policies because we expand the search until we explore the same candidates in all cases. However, for small values of k_{local} $\Gamma^{max}_{k_{local}}$ always gets larger recall than $\Gamma^{min}_{k_{local}}$ and $\Gamma^{rnd}_{k_{local}}$. For $k_{local} = 5$, $\Gamma^{max}_{k_{local}}$ doubles $\Gamma^{min}_{k_{local}}$ recall and increases 50% compared to $\Gamma^{rnd}_{k_{local}}$. This result indicates that using the similarity as a criterion to limit the amount of vertices to explore is particularly effective for small values of k_{local}. Reducing the number of vertices to explore reduces both the computation time and the amount of memory to use.

5.7 Impact of the sampling parameter k_{local}

The definition of k_{local} puts an upper bound limit of k^2_{local} candidate vertices to be scored. Reducing the search space reduces time and storage costs, but penalizes recall. However, different scores may get different recall values for the same k_{local} value. We now analyze the impact of varying k_{local} on the computing time and recall for the scoring methods proposed in Table 3.

Figure 8 shows the recall and computing time for *livejournal* and *twitter-rv* datasets. We reduce the presented figures to these datasets due to the lack of space. The *Sum* aggregator is exhaustive in the sense that it accumulates all pair-wise similarities, thus taking into account the popularity of a vertex z (i.e. the number of paths connecting u to z) in its final score. This behavior explains why the recall increases together with k_{local}. By contrast, *Mean* and *Geom* work differently. *Mean* improves the recall obtained with *Sum* for small values of k_{local} (see *linearMean*). When k_{local} increases the recall obtained with *Mean* goes down. We believe this may occur because *Mean* averages an increasing number of path-similarities for larger k_{local} values, many of which might have low scores. Additionally, averaging eliminates information about the popularity of a vertex, contrarily to *Sum*. The *Geom* aggregator show the same pattern in a stronger form, which is probably due to its sensitivity to non-similar vertices ($sim^{\star}(u, z) = 0$).

Thoroughly understanding the properties of each scoring configuration would require a longer and deeper analysis outside the scope of this work. On the basis of our current results, we can however propose some guidelines. For scenarios requiring the best predictions, but not necessarily the shortest times, the *linearSum* score seems to be the best solution. By increasing k_{local} it produces the best recall values, at the cost of higher computation times. For scenarios demanding the best results under a tight time budget, the *Mean* aggregator with small values of k_{local} appears competitive, and in some cases a better solution that *linearSum*.

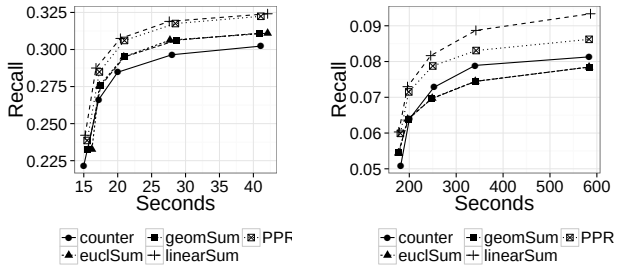
5.8 Sensitivity to k and to removed edges

For completeness, we analyze how recall evolves when we vary the number of answers Snaple returns. This is shown in Figure 9 for *livejournal* and *pokec*, and five score configurations, when k takes the values $5, 10, 15, 20$. In this range, the recall increases substantially with k. (The other scores based on the *Mean* and *Geom* aggregators follow a similar pattern.)

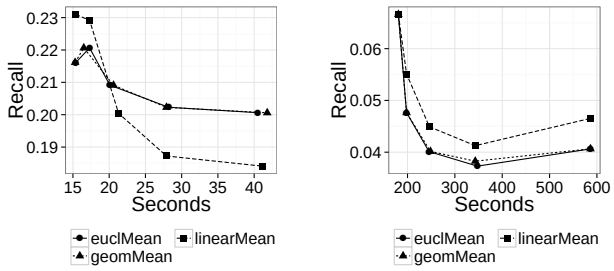
Similarly, we investigate the sensitivity of recall when we increase the number of edges that are removed per vertex. Removing more edges, we remove paths between vertices, which makes it more difficult for Snaple to find relevant vertices. As a result recall decreases (Figure 10). (If a vertex has fewer edges than the number to be removed, we removed all the edges except one.) The recall decreases proportionally to the number of removed edges. We observe similar patterns when using *Mean* and *Geom* aggregators.

5.9 Comparison to Cassovary

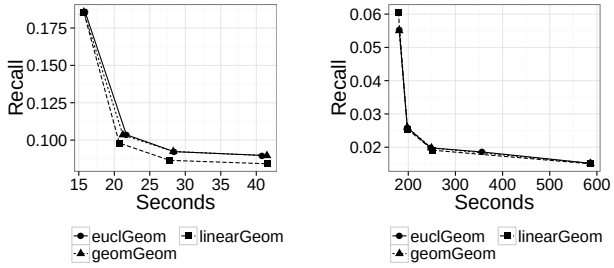
For completeness, we conclude our evaluation by comparing Snaple with a single-machine solution. This comparison serves two objectives: to assess the impact of Snaple when networking costs disappear, and to provide a reference point to gauge the benefits of distribution when processing very large graphs. We use Cassovary [42] for this comparison, a multithreaded in-memory graph library from Twitter. Cassovary is able to load relative large graphs and can traverse a graph fully allocated into main memory. It has been shown to be a efficient solution for computing random



(a) *Sum* aggregator for *livejournal* (left) and *twitter-rv* (right)

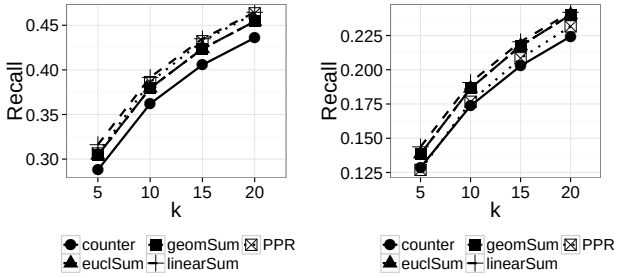


(b) *Mean* aggregator for *livejournal* (left) and *twitter-rv* (right)



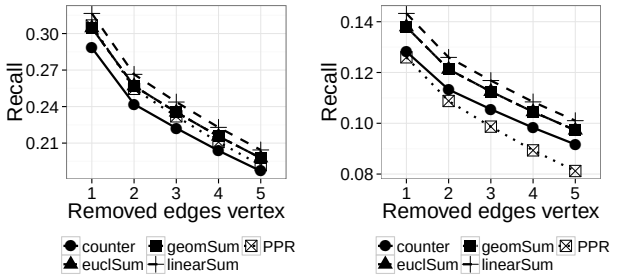
(c) *Geom* aggregator for *livejournal* (left) and *twitter-rv* (right)

Figure 8: Comparison of computing time against recall for different scoring configurations on 256 cores. Each point corresponds to a different value of $k_{local} = 5, 10, 20, 40, 80$. The sum aggregator gets the highest recall, improving as k_{local} grows.



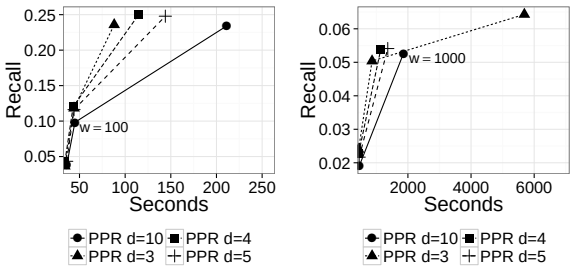
(a) *livejournal* (b) *pokec*

Figure 9: Evolution of recall when increasing the number of recommended links k with $k_{local} = 80$.



(a) *livejournal* (b) *pokec*

Figure 10: Evolution of recall when increasing the number of removed edges per vertex with $k_{local} = 80$.



(a) *livejournal* (b) *twitter-rv*

Figure 11: Recall and computing time using a standalone link-prediction solution on top of Cassovary using random walks to emulate PPR. Both solutions are run on a *type-II* machine with $w = 10, 100, 1000$.

walks [19, 24]. It is also used in production by Twitter [12].

In a first attempt, we implemented the solution described by Algorithm 1 (with the 2-hop optimization). However, neither the recall nor computing time were competitive. We therefore moved on to a multithreaded version of the personalized page rank (PPR) [30] approximation based on random walks [37] to improve on these results. For each vertex we run w random walks of depth d. $d = 2$ reaches the neighbors of a vertex ; $d = 3$ its neighbors of neighbors and so on. Once the random walks terminates, the k most visited vertices not included into $\Gamma(v)$ are returned as predictions. Increasing w and d, we force the algorithm to explore a larger number of vertices in a similar way we do when varying k_{local}. We modify w and d configurations in order to find the largest recall in the shortest time.

Figure 11 compares the recall and computing time for Cassovary running on a *type-II* machine when varying w and d on the *livejournal* and *twitter-rv* datasets. We observe that increasing d does not necessarily improve recall, with $d = 3$ yielding recall values very close to that of larger values of d. By contrast, larger values of w tend to yield better recall value, but they also significantly increase the computing time. In the case of *twitter-rv*, we run an extra configuration with $d = 3$ and $w = 10000$ getting 0.06 recall in 90 minutes. Unfortunately, we had to stop other configurations with larger d values and $w = 10000$ as they took too long to complete.

Snaple is designed to run on a distributed environment

Table 6: Snaple also outperforms a state-of-the-art single-machine solution (results obtained on one *type-II* node)

dataset	Cassovary		Snaple		
	recall	time(s)	recall	time(s)	speedup
livejournal	0.24	93	0.30	45.8	2.03
twitter-rv	0.06	5420	0.08	600.7	9.02

taking advantage of the speed of multiple simultaneous computing nodes. A comparison between Snaple on multiple machines and Cassovary on a single machine would therefore not be fair to Cassovary. For that reason, we compare the best results obtained in our previous analysis of Cassovary (best recall in the shortest time) with the results obtained running Snaple on a single machine. We use $k_{local} = 20$, which produces recall values close to that of Cassovary (slightly higher in fact). The results we obtain (Table 6) show that Snaple is faster than Cassovary (with speedups of 2.03 and 9.02) while increasing recall. This demonstrates that even on single machine deployments Snaple provides a competitive solution in terms of both prediction quality and execution time.

We can also use the results obtained with Cassovary to assess the benefits of distribution when processing very large graphs: the recall obtained by Cassovary on *twitter-rv* (0.06, Table 6) is obtained by Snaple in 177s (2min57s) when using *linearSum* with $k_{local} = 5$ on 256 *type-I* cores (32 *type-I* machines), as reported by Figure 8a (right-hand chart). This corresponds to a speedup of 30.62 against Cassovary, while only using 12.8 more cores.

6. RELATED WORK

The processing of large graphs on a single machine has been the subject of a number of works, ranging from cases in which the entire graph can fit into main memory [12], to more recent approaches exploiting secondary storage using specialized data structures, such as GraphChi [20] and X-Stream [33]. Although competitive and resource efficient, these approaches were not designed for distribution. As a result, they cannot scale beyond the limit of a single machine, and are unable to tap into the increased resources offered by a cluster, or a cloud infrastructure, which we can.

If we turn to distributed solutions, many works have proposed abstraction models to facilitate the distributed processing of large amounts of data, including graphs. The GAS engines we presented in Section 2 form one strand of this line of work. Related to the GAS paradigm, the *Bulk Synchronous Parallel* (or BSP) model [43] organizes a parallel computation into asynchronous supersteps that are then synchronized using barriers. Pregel [27] adapts the BSP model to graphs, and organizes computation around functions executing on vertices that exchange messages between each supersteps.

The complexity of programming models such as BSP has led researchers to propose distributed shared memory solutions [6, 15]. However, the cost of remote memory accesses tends to make these solutions impracticable when results must be returned rapidly. An improvement has been proposed through PGAS (Parallel Global Address Space) [5] by reducing the number of remote memory accesses. The popularity of MapReduce solutions has also prompted the development of high-level languages such as Hive [41] or Pig [29] to encode distributed data computations. To unify these various models, GraphX [10] offers abstraction layers that combine different graph engines when performing a graph analysis, in an effort to shield developers from the details of each engine. In spite of their merits, none of these solutions considers link-prediction, and none addresses the problem of non-locality when traversing graphs [26].

7. CONCLUSION

In this paper we have presented the design, implementation and evaluation of Snaple, a highly-scalable approach to the link-prediction problem optimized for the gather-apply-scatter (GAS) model of distributed graph engines. We have provided an exhaustive evaluation of our prototype in a cluster using a representative array of large publicly available datasets. Snaple is able to compute the predictions of a graph containing 1.4 billion edges in less than 3 minutes when other naive GraphLab solutions fail due to resource exhaustion. Additionally, we demonstrate that Snaple has an over-linear speedup of 30 when compared with a state-of-the-art non-distributed solution, while improving prediction quality.

Our work opens several exciting paths for future research. One such path involves the extension of Snaple to supervised link-prediction strategies, which may improve recall while taking advantage of distributed computing. We also would like to port Snaple to other distributed graph processing platforms such as Giraph [1], Bagel [3] or Stinger [9] to provide more comparison points between these platforms. Finally, we plan to perform a more in-depth analysis of the data structures used in our prototype to understand how individual operations can be further improved, and help reduce latency.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

8. REFERENCES

[1] Apache. Apache giraph. `http://giraph.apache.org/`, 2014.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, 2006.

[3] Bagel. Bagel. `https://github.com/mesos/spark/wiki/Bagel-Programming-Guide`, 2014.

[4] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, 2011.

[5] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *IPDPS*, 2007.

[6] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, 1993.

[7] M. J. Brzozowski and D. M. Romero. Who should i follow? recommending people in directed social networks. In *ICWSM*, 2011.

[8] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, 2011.

[9] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *HPEC*, 2012.

[10] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[12] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: the who to follow service at twitter. In *WWW*, pages 505–514, 2013.

[13] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[14] M. Hasan and M. Zaki. A survey of link prediction in social networks. In C. C. Aggarwal, editor, *Social Network Data Analytics*, pages 243–275. Springer US, 2011.

[15] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. *SIGOPS Oper. Syst. Rev.*, 1995.

[16] A.-M. Kermarrec, F. Taïani, and J. M. Tirado. Cheap and cheerful: Trading speed and quality for scalable social-recommenders. In *DAIS*, 2015.

[17] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Eurosys*, 2013.

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[19] A. Kyrola. Drunkardmob: Billions of random walks on just a pc. In *RecSys*, 2013.

[20] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[21] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (salsa) and the tkc effect. *Comput. Netw.*, 33(1-6):387–401, June 2000.

[22] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[23] R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla. New perspectives and methods in link prediction. In *KDD*, 2010.

[24] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*, 2014.

[25] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 2012.

[26] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[28] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, 2007.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, 2008.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[31] R. Parimi and D. Caragea. Predicting friendship links in social networks using a topic modeling approach. In *PAKDD*, 2011.

[32] M. Rowe, M. Stankovic, and H. Alani. Who will follow whom? exploiting semantics for link prediction in attention-information networks. In *ISWC*, 2012.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

[34] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, page 22. ACM, 2013.

[35] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[36] P. Sarkar and A. W. Moore. Fast nearest-neighbor search in disk-resident graphs. In *KDD*, 2010.

[37] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz. To randomize or not to randomize: Space optimal summaries for hyperlink analysis. In *WWW*, 2006.

[38] S. Scellato, A. Noulas, and C. Mascolo. Exploiting place features in link prediction on location-based social networks. In *KDD*, 2011.

[39] R. Schifanella, A. Barrat, C. Cattuto, B. Markines, and F. Menczer. Folks in folksonomies: social link prediction from shared metadata. In *WSDM*, 2010.

[40] L. Takac and M. Zabovsky. Data analysis in public social networks. In *Int. Science Conf. and Int. Workshop Present Day Trends of Innovation*, 2012.

[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[42] Twitter. Cassovary. `https://github.com/twitter/cassovary`, 2014.

[43] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 1990.

[44] D. Yin, L. Hong, and B. D. Davison. Structural link analysis and prediction in microblogs. In *CIKM*, 2011.