DrunkardMob: Billions of Random Walks on Just a PC

Aapo Kyrola
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, USA
akyrola@cs.cmu.edu

ABSTRACT

Random walks on graphs are a staple of many ranking and recommendation algorithms. Simulating random walks on a graph which fits in memory is trivial, but massive graphs pose a problem: the latency of following walks across network in a cluster or loading nodes from disk on-demand renders basic random walk simulation unbearably inefficient. In this work we propose DrunkardMob[1], a new algorithm for simulating hundreds of millions, or even billions, of random walks on massive graphs, on just a single PC or laptop. Instead of simulating one walk a time it processes millions of them in parallel, in a batch. Based on DrunkardMob and GraphChi [19], we further propose a framework for easily expressing scalable algorithms based on graph walks.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval–*Information Filtering*

General Terms

Graph computation, Recommendations, Random walks

Keywords

recommender systems, random walks, graph computation

1. INTRODUCTION

Many popular algorithms for ranking and recommending nodes in graphs are based on random walk models. Most notably, the PageRank algorithm [26] and its personalized adaption the Personalized PageRank (PPR) compute ranking for webpages based on the probability that a "random web surfer" (we will use the term *random walker*) ends up on the page by following random links from web pages. The same model can be used for ranking users in social networks or graphs of other entities.

To compute the global PageRank, we can compute the stationary distribution of the random process by iterated multiplications of an initial ranking vector by the transition matrix (the *power method*). However, for computing all PPR vectors [26], the direct methods are too expensive, with complexity of $O(V^2)$ [16]. Fortunately, Fogaras et. al. [10] proved that by simulating a modest number of short random walk segments from each node, we can efficiently obtain a good approximation of the PPR vector for each user.

In this work, we show how to simulate hundreds of millions, or even billions, of number of short (random) walks on extremely large graphs, on just a single computer. We propose a new algorithm, DrunkardMob, which processes the graph from disk but maintains the current state of each random walk in memory. A typical modern PC or laptop has several gigabytes of memory and can scale up to several hundred million parallel walks. On high-end servers with large RAMs, we simulate billions of walks in parallel. Because the graph is processed efficiently from disk, we can process even the biggest social graphs on just a laptop. Our solution is generic, not limited to the basic PageRank algorithm, and is as easy to program as the basic in-memory random walk procedure. We implement DrunkardMob on GraphChi [19], a general graph computation system that allows us to naturally combine the walk simulator with other recommendation algorithms such as matrix factorization.

Outline: after reviewing related work and preliminaries in Section 2 and 3, Section 4 presents the DrunkardMob algorithm and discusses its implementation.We present a case study on implementing Twitter's Who-to-Follow algorithm using our system in Section 5. Evaluation of the system follows in Section 6, after which we conclude and discuss future research.

Summary of our contributions:

- DrunkardMob, an algorithm for simulating billions of walks on just a single computer.
- Generic architecture for representing graph walk -based computation in the *vertex-centric* computational model.
- Implementation of DrunkardMob on top of GraphChi and an evaluation of its performance and scalability.

DrunkardMob is available in the open-source at `https://github.com/GraphChi/`.

[1]Random walks are often called "drunkards' walks", thus our algorithm, which simulates many of such walks simultaneously is appropriately called DrunkardMob.

ACM RecSys'13 October 12 - 16 2013, Hong Kong, Hong Kong

http://dx.doi.org/10.1145/2507157.2507173 .

2. RELATED WORK

Random walks and recommender systems: Models based on random walks on a graph are popular in the recommender system research due to their scalability. This work was initially motivated by the problem of recommending friends or connections in a social network, called the *link prediction* problem [23]. Perhaps the most common approach is to return the top k ranked nodes of the Personalized Pagerank [26] vector for the user in question. Recently [12] describe an extensions to this technique used at the microblogging service Twitter (`http://www.twitter.com`). We implement their method in our case study (Section 5).

Random walk -based models are also used in other contexts: FolkRank [13] is an adapted version of PageRank for ranking in *folksonomies* (graphs of users, tags, and resources). TrustWalker [15] improves item-based recommendation by modeling trust between users in a social network, approximated by simulating random walks on the graph. Finally, [20] proposes a ranking of entities in an entity graph using random walks. In a closely related field, random walks have been successfully used for inference and learning in a large knowledge base [**?**]. Previous work has been evaluated on relatively small datasets, and we believe our toolkit will help researchers to tackle problems of much larger scale.

Scaling Personalized PageRank: As most work on link prediction and personalized web search has focused on Personalized PageRank (PPR), its scalability has also been object of intensive research, which we briefly review here. The seminal work by Jeh and Widom [16] proposed how PPR can be approximated by computing the rank vectors to a smaller set of hub nodes and using linear algebraic relations of the PPR vectors. However, to allow good accuracy for full personalization, the hub set would need to be very large.

Another important work by Fogaras et. al. [10] proposed using short random walk segments, *fingerprints*, to approximate the PPR vectors. They propose an external memory (disk-based) algorithm to efficiently simulate a very large number of walks from all vertices at once. Our work is closely related to their work but is more general and utilizes the memory of the computer to keep track of very large number of walks. To our knowledge, there is no available implementation of the algorithm of [10].

Related algorithm to compute PageRank on graphs streamed from disk was proposed by Das Sarma et. al. [7], but their approach is to sample the vertices and edges of the graph and simulate a small number of hops on the sampled graph to produce a large number of short walks. Their method requires a rather complicated scheme to stitch small walk segments and handle special cases, while our method and that of Fogaras can simulate the walks on the full graph and thus are not limited to PageRank. Bahmani et. al. [4] study how to efficiently update a database of random walk segments when new edges are inserted into the graph. Their method could be combined with DrunkardMob.

Finally, [3] proposes a method to compute PPR efficiently on the popular MapReduce [8] parallel data processing framework. Their method is a generalization of the method by Das Sarma et. al. [7]. While MapReduce provides impressive scalability with very large clusters, our method can process extremely large graphs on just a single computer.

3. PRELIMINARIES

Our objective is to simulate walks (not necessarily random) on a graph $G = (V, E)$, where V denotes the set of vertices (nodes) $v \in \mathbb{Z}$, and $E = \{(u, v) \,|\, u, v \in V\}$ the set of *directed* edges connecting the vertices. We call edge $e = (a, b)$ *out-edge* of vertex a and *in-edge* of vertex b. We generally assume the graph to be sparse, i.e that most vertices do not have an edge between them. In addition, each edge can be associated a value, typically a real valued *weight*. Let *out-degree* $D_o(v)$ be the number of out-edges of vertex v.

A **walk** $w_s^t = [s, v^1, v^2, ..., v^t]$, $s, v^j \in V$ on a graph is an ordered sequence of t visits to vertices (called *hops*), with **source vertex** (the origin of the walk) s. We denote $w^t(i)$ the i'th visit of the walk. Random variable W_s^t represents a walk where each visit $W_s^t(j+1)$ is chosen randomly according to some transition probability $\mathbf{P}(v \,|\, W^t(j) = v')$ based on the previous hop.

Example (global PageRank): For the global PageRank [26] model, the probability of moving from vertex v to v' is:

$$\mathbf{P}(v \,|\, W_s^t(j) = v') := \begin{cases} d/|V| & \text{if } (v', v) \notin E \\ d/|V| + \frac{1-d}{D_o(v')} & \text{if } (v', v) \in E \end{cases}$$

Above, d is the damping factor, which represents the probability of the random walk to *reset* and "teleport" to a random node in the graph.

Example (Personalized PageRank): The Personalized PageRank [26] model is used for estimating the ranking of nodes "personalized" to a source vertex s. The only difference to global PageRank is that when a walk resets, it will always return to the source:

$$\mathbf{P}(v \,|\, W_s^t(j) = v') := \begin{cases} 0 & \text{if } (v', v) \notin E \text{ and } v \neq s \\ d & \text{if } v = s \\ \frac{1-d}{D_o(v')} & \text{if } (v', v) \in E \end{cases}$$

The probabilities above are for unweighted PageRank, but are easily modified to use edge weights for bias.

In the context of this work, we do not require a walk to be random, or even follow a well-defined transition model. Computationally, walks are simulated by defining a **walk-update function**:

```
update := (G, vertex, walk-info) → (vertex, walk-info)
```

Above, `vertex` denotes the current visit of the walk, and type `walk-info` is an application dependent structure that stores information about the walk. For example to simulate Personalized PageRank, it is used to store the source vertex of the walk. In addition, it can also contain information about the current hop-number of the walk or an unique walk identifier. In general, we use as few bits as possible to represent `walk-info` in order to store as many walks in memory as possible. Walk-update function returns the next visit of the walk and the `walk-info` value, which it can modify. Walk-update functions are general and may utilize any information about the graph and it is straight-forward to define specialized walks that follow only, for example, certain types of edges.

3.1 Large Graphs

If the graph fits into main memory (RAM), simulating walks on a graph is trivial, using the Algorithm 1. On line 6 we have added a call to `recordHop()` function, which is used

to keep track of the visits made by the walk (not described). The algorithm can be easily made parallel by executing several walks simultaneously. This simple algorithm is able to execute millions of hops in second on a typical PC.

Algorithm 1: In-memory walk on a graph

```
1 SimulateWalk(G, src, upFun) begin
2     walkInfo ← initializeWalk(src)
3     curvertex ← src
4     for i = 1...t do
5         (curvertex, walkinfo) ← upFun(G, src, walkInfo)
6         recordHop(src, walkInfo, curvertex)
7     end
8 end
```

If the graph does not fit into RAM, it needs to be accessed from disk (parts of the graph may reside in RAM) or the graph must be partitioned and distributed across a cluster of computers. Unfortunately, in both cases the simple algorithm 1 is extremely inefficient.

If the graph resides on disk, each hop requires loading the next vertex from disk. Even if the graph is partially in memory, on typical real-world graphs [22, 19] a large portion of the hops would require accessing vertices stored on disk. On a typical rotational hard drive, random seek costs a few milliseconds, limiting us to some hundreds of hops per second. Flash-based Solid-State Drives (SSD) have much better performance, but even they provide several orders of magnitude slower random access performance than DRAM.

In the distributed setting, the graph is partitioned across a cluster. Now each hop with the simple algorithm moves the walk to a partition owned either by the node owning the current vertex, or to another partition. If the hop requires changing partition, this will incur a cost equivalent to the network latency. With 1Gb Ethernet the typical latency is in the order of 0.1 milliseconds, allowing maximum of tens of thousands of walks per second.

In both cases, the inefficiency arises from increased latency between hops. In this work, we propose to solve this problem by simulating a very large number walks in parallel. Instead of executing one walk a time, we instead consider each vertex in turn and add one hop to each walk currently at the given vertex. Similar idea was first proposed in the external memory setting by [10], but we instead use the available RAM to keep track of walks while efficiently processing the graph from disk.

4. DRUNKARDMOB

We now describe the main contribution of this paper, the DrunkardMob algorithm.

4.1 High-level description

Instead of simulating one walk a time, DrunkardMob reverses the execution and instead simulates a massive number of walks in parallel, possibly from a large number of source vertices, and processes one *vertex* a time: at each vertex, all walks currently visiting that vertex are processed and moved forward. The graph is streamed from disk, thus all memory can be utilized for storing the walk states. The performance relies on compact representation of walks in memory, which we discuss in the next section.

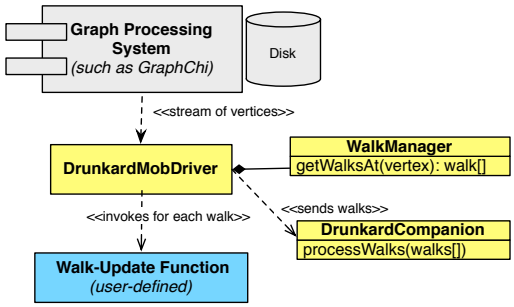


Figure 1: DrunkardMob: Class diagram.

The high-level object-oriented class diagram of the algorithm is shown in Figure 1. We assume that the graph is loaded efficiently from disk by some graph processing system, which provides a stream of vertices with their incident edges to the `DrunkardMobDriver`. `WalkManager` is a container that keeps track of the current vertex each walk is visiting (its interface is similar to an associated map from vertex ids to lists of walk-items). For each vertex it receives from the graph processor, `DrunkardMobDriver` queries `WalkManager` for the set of walks currently at that vertex. For each walk in the list, it invokes the user-defined walk-update function that returns new destination for the walk and possibly updates the meta-data associated with the walk. In addition, `WalkManager` provides the list of walks at the vertex to `DrunkardCompanion`, which is a component that keeps track of the visit frequencies (discussed in Sec. 4.3). Finally, `DrunkardMobDriver` provides the updated walks to the `WalkManager`. This cycle is repeated over all vertices in the graph for a predefined number of *iterations*. The number of hops for each walk is equal to the number of iterations.

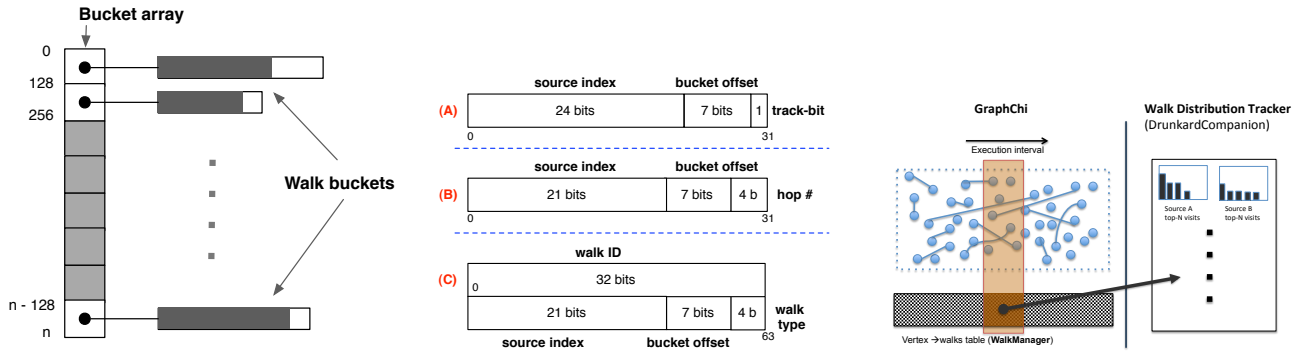
Algorithm 2: Batched operation of DrunkardMob

```
1  DrunkardMobDriverCallback(vertexArray) begin
2      walksForSet ←
       walkManager.allWalksAt(vertexArray)
3      updatedWalks ← []
4      foreach vertex ∈ vertices do
5          foreach w ∈ walksForSet[vertex] do
6              w' ← walkUpdateFunc(vertex, w)
7              updatedWalks.add(w')
8          end
9      end
10     walkManager.insert(updatedWalks)
11 end
```

For improved performance, DrunkardMob handles vertices in batched manner, i.e instead of processing one vertex a time, it handles vertices in large chunks. This allows `DrunkardMobDriver` to query walks from the `WalkManager` for many vertices at once, enabling more efficient data structures in the `WalkManager` (see below). The pseudocode for the batched operation is shown in Algorithm 2.

4.2 Efficient data structures

DrunkardMob assumes that the states of all walks is held in memory. To maximize the throughput (number of walks



(a) WalkManager buckets (b) Walk encoding (c) DrunkardMob + GraphChi

Figure 2: (a) WalkManager bucket data structure; (b) Examples of walk encodings. A and B use 32 bits to represent a walk. B uses 4 bits to encode the hop-count of the walk, thus allowing less different source vertices then A. Example C allows keeping track of each walk separately and uses 64 bits to represent each walk. (c) Schematic diagram showing the DrunkardMob algorithm implemented on GraphChi.

simulated in time), it is important to minimize the memory footprint of each walk. We will first discuss the mapping of vertices to walks and then describe how the walks themselves are encoded.

`WalkManager` continuously manages a mapping from vertices to walks: for each vertex, it knows the walks whose last hop is in that vertex. Simplest choice would be to define an array A of lists, where $A[i]$ contains a linked list or array-buffer of walk objects (alternatively, we could store the vertex-list pairs in an associative map). Unfortunately, as we are interested in working with very large graphs with potentially hundreds of millions or a few billion vertices, the memory required to store pointers to the lists for each vertex quickly becomes a dominating cost. Moreover, with high-level languages such as Java, each individual list object can take tens of bytes of memory [14]. Instead we divide the range of vertices $[0, n-1]$ to sufficiently long sub-intervals (such as 128). For each sub-interval we associate a *bucket*, which is implemented as a buffered array, see Figure 2(a). When `DrunkardMobDriver` retrieves walks for a range of vertices, the walks from the corresponding buckets (whose intervals intersect the query interval) are sorted by the vertex. Thus, each walk object needs to be associated with the *offset* from the first vertex of the bucket. If the bucket intervals have length 128, the offset requires in 7 bits (Figure 2(b)).

The walk objects themselves are also represented compactly, typically as 32-bit words. Figure 2(b) shows three examples of how to encode information into 32-bit or 64-bit walk objects. Typically they encode the source vertex by using 24 bits to represent the index of the source vertex in the source vertex array. That is, this allows us to simulate walks from a maximum of $2^{24} = 16,777,216$ distinct source vertices. The sources thus need to be registered to `WalkManager` before the start of the simulation. Each walk needs to also encode a bucket offset to store the current position of the walk. Rest of the bits can be used to encode meta-data of the walk. For example, one bit can be used for a "track-bit" to set whether the walk should be counted in the visit frequency statistics (for example, we might not want to include the immediate connections of a vertex in the statistics).

The compact encoding of walks allows us to run up to 0.5 billion walks on a typical laptop with 8 gigabytes in memory using Java. With C/C++, we could probably execute significantly more walks due to lower runtime overheads.

4.3 Keeping track of walk visit frequencies

The next challenge is to keep track of the visit frequencies for the walks, for each source vertex separately. As shown In the class diagram (Figure 1), this is done by component `DrunkardCompanion`, of which we have two different versions:

- *Log each hop of the walk to a file or database* (files can be partitioned by sources) and analyze these log files separately. For PPR and related algorithms, this is efficient because each hop only requires storing the source vertex and current vertex identifier. The final analysis is done off-line.
- *Continuous in-memory tracking.* In this model we keep track of visit frequencies in-memory as they happen. The tracking can be done either locally in the same address space as the DrunkardMob simulation or remotely on a separate machine. Since walks are processed in batches, the walks can be efficiently sent over network in big packets.

In this paper, we concentrate in the in-memory tracking of walks. The main challenge is how to keep track of visit frequencies (for each source vertex separately), if the memory is limited. Formally, we want to keep track of the empirical distribution $\#\{v_i = w(i) \,|\, w \in \mathbb{W}_s, \forall i\}$, where $\mathbb{W}_s$ is the set of all simulated walks from source s. Alternatively, we might only be interested in the top k vertices visited by the walks for each source (not the actual frequencies).

In worst case, each hop visits a separate vertex and we need $O(\text{num-of-walks} \times \text{num-of-hops})$ of memory to store the visit frequencies. This problem is similar to the classic problem in data streams: estimating the top-K frequent items with limited memory. In our case, we can consider the stream of hops for walks from a given source s as a single data stream. For each source we maintain a vertex $\times$ visits mapping for some maximum of K elements. To limit the number of vertices in the map to K in principled manner we

use the FREQUENT algorithm described in [5], which was originally proposed by [25]. The idea is simple: when we record a new visit to vertex v, as long as the number of distinct vertices in the map is less than K, we either insert $(v, 1)$ into the mapping or increase the count for v by one if it already was being tracked. If the size of the map is already K and we were to insert a new value $(v, 1)$, we decrease the count of each vertex by one and remove all values that have a zero count. The intuition is that if the map contains a long tail, i.e many vertices with a count of one, the long tail will be "cut" when the capacity limit of K is exceeded.

Theoretical guarantees for this algorithm are given in [5]. In practice a sufficiently skewed distribution (such as Zipfian) of the visits is required to maintain good approximation. In many real-world graphs, the in-degree distribution of nodes is highly skewed, follows the power-law degree [9], and thus also the visits frequencies of walks concentrate to a small number of vertices. However, this may not be true for walks from all sources: some vertices may be less connected to the "hot" nodes than average nodes or reside in distinct subgraphs outside the core of the graph [22]. We leave further study of this issue as future work.

Our architecture allows programmers to easily plug in different implementations of `DrunkardCompanion` which could support different trade-offs in approximation quality than our solution.

4.3.1 Long random walks as many short ones

To advance each walk by one hop, we need to do a full pass over the graph. Since we assume the graphs to be very large, also full pass is expected to be rather expensive. DrunkardMob is therefore suited only for walks that are relatively short. However, the definition of PageRank [26] and its variations is based on an infinitely long random walk with resets (i.e jump to a random vertex or the source vertex in Personalized PageRank). Therefore it would be natural to simulate as long walk as possible to approximate the model. But because of the resets, a random walk $W_s^t \rightarrow W_s^\tau$ can be decomposed into a sequence of τ short walks ω_s^i with the same source vertex s (note, that τ is a random variable):

$$W_s^\tau = s \circ \omega_s^1 \circ \omega_s^2 \circ \cdots \circ \omega_s^\tau$$

The short ω_s^i walks have length $|\omega_s^i| \sim Geom(d)$ where d is the reset-probability (damping factor). The expected length of the short walk is $\frac{1}{d}$ hops and the expected fraction of short walks longer than k is $(1-d)^k$. For example, with the usual choice of $d = 0.15$, the expected length is approximately 6.7 hops. For a more detailed analysis, see Fogaras et. al. [10].

It therefore appears that a large number of short walks could approximate the PageRank and related models well. In the context of recommender systems, it may make sense to bias towards shorter walks[2]: on typical social networks and other natural graphs, the number of nodes in a $k-$hop radius increases extremely fast as the expected distance between any two nodes is very small [1], even less than the famous "six degrees of separation" result by Milgram [24]. For computing sensible recommendations, it thus makes sense to concentrate the graph exploration using random walks to a smaller radius of the source vertex.

[2]Our implementation of Personalized PageRank actually continues the short walk after a reset, from the source vertex, causing a large number very short walks to be simulated.

4.4 Implementation on GraphChi

GraphChi [19] is a recently proposed very efficient disk based graph processing system. It is able to execute *vertex-centric* graph computation algorithms on graphs with billions of edges on just a commodity PC or laptop. GraphChi's performance is based on a "Parallel Sliding Windows" technique [19] : The edges of the graph are partitioned into a set of P *shards* by their destination vertex, each shard representing a contiguous *interval* of vertex IDs, and then sorted by their source vertex. Now to load a subgraph corresponding to one interval of vertices, only the associated shard (in-edges) and $P-1$ contiguous blocks from the other shards (out-edges) are loaded from disk. Updating the values of edges to disk is symmetric. GraphChi can execute a full pass over the graph by doing only a $O(P^2)$ non-sequential disk accesses, enabling it to perform well on both hard drives and SSDs.

We chose to implement DrunkardMob on GraphChi, because its method of processing vertices in large contiguous intervals matches perfectly the batched operation of DrunkardMob (Algorithm 2). In principle, DrunkardMob does not require many of the features of GraphChi, and a simple procedure to load the graph one vertex a time from disk (in adjacency list format) would have sufficed. However, in addition to saving us development time, the benefit of using GraphChi is that DrunkardMob could work alongside any other graph algorithm.

Figure 2(c) shows the high level operation of DrunkardMob as implemented on GraphChi. GraphChi provides a batch of vertices (corresponding to the vertex intervals that define the shards) a time, and calls `DrunkardMobEngine` which is implemented as a standard GraphChi `update-function` [19]. When receiving a `beginSubInterval()` call from GraphChi, `DrunkardMobEngine` requests the walks currently visiting the vertices in the interval from `WalkManager`. This batch of walks is also sent to the `DrunkardCompanion`, which can be a local or remote component. Subsequently, GraphChi invokes the `DrunkardMobEngine` for each of the vertices in the interval separately: `DrunkardMobEngine` selects the walks for the given vertex and calls the user-defined **walk-update function** for each of the walks as shown in Algorithm 2. DrunkardMob uses efficiently all available processor cores because different vertices can be processed in parallel. Figure 3 shows the actual Java-code for the walk-update function for Personalized PageRank.

For improved performance, DrunkardMob utilizes the *selective scheduling* feature of GraphChi. Each vertex is associated a bit which is set if the vertex has any walks currently visiting it. This allows GraphChi to save memory and avoid loading inactive vertices from disk.

5. CASE STUDY: TWITTER'S WHO-TO-FOLLOW

To demonstrate the viability of DrunkardMob to large-scale applications, we implemented a complete recommendation engine for the microblogging service Twitter following the description in recently published paper by Gupta et. al. [12]. The Twitter folllow-graph is a directed graph where users are represented by vertices and an edge (u, v) exists if user u *follows* user v's postings. The purpose of Who-to-Follow (WTF) service is to compute for each user a set

```
public void PPWalkUpdate(int[] walks,
     ChiVertex vertex, DrunkardContext drunkardCtx) {
    for(int walk : walks) {
        if (rand.nextDouble() < RESET_PROBABILITY) {
            drunkardCtx.resetWalk(walk, false);
        } else {
            int next = vertex.getOutEdgeId(
                    rand.nextInt(vertex.numOutEdges()));
            drunkardCtx.forwardWalkTo(walk, next, true);
        }
    }
}
```

Figure 3: Walk-update function for Personalized PageRank for DrunkardMob's Java implementation.

of recommendations for other users she could follow. The algorithm presented in [12] works in three steps.

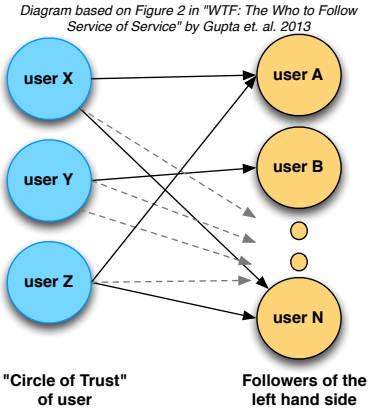


Figure 4: Bipartite graph used for computing recommendations for a Twitter user using the SALSA algorithm [21]. Diagram is based on Figure 2 in [12].

For each user: **(1)** Simulate an egocentric random walk and pick N most frequently visited vertices as the "Circle-of-Trust" (CoT) for the user. **(2)** Construct a bipartite graph by placing CoT vertices on the left side and a subset of the users they follow on the right side (see Figure 4a). **(3)** Compute SALSA algorithm on this graph and choose the top K scored nodes on the right side as recommendations for the user.

All details are not given in [12], so our algorithm might differ slightly from theirs. First step is equivalent to approximating the Personalized PageRank [26] and choosing the top ranked vertices. In our experiments, we chose $N = 200$ top vertices as the CoT for each user. Using DrunkardMob we simulate the egocentric random walks for a large number of users in parallel, using DrunkardMob. For the second step, we query the followers for each user in the CoT and count the number of common followings among CoT and eliminate all that have less than four common followers (this somewhat arbitrary filtering was done to improve performance). We can query followers efficiently by imposing a sparse index over the GraphChi shard files and finding followers for many users simultaneously. It is out of scope of this paper to describe the graph query functionality of GraphChi in more detail. Finally, in the third step we execute SALSA [21] algorithm in-memory on the constructed bipartite graph. Steps 2 and 3 are done for each user separately, but we use multithreading to compute several recommendations in parallel.

Experiment: We run the Twitter recommendation algorithm on the *twitter_rv* graph, which is the almost complete Twitter graph from year 2010[3], using a Mac Mini (8GB, SSD, two cores) and a MacBook Pro laptop (8GB, SSD, 4 cores). On the Mac Mini we computed recommendations for 60,000 users a time, with average time of 2 hours and 52 minutes for a batch. For the MacBook Pro we computed 100,000 recommendations with mean time of 1 hour and 50 minutes. In both cases over 85% of the time was spent in steps 2 and 3.

6. EXPERIMENTS

All experiments were done on DrunkardMob implemented on top of Java-version of GraphChi. The graphs we used for the experiments are listed in Table 1.

Graph name	Vertices	Edges
live-journal [2]	4.8M	69M
domain [27]	26M	370M
twitter_rv[18]	65M	1.5B
uk-2007-05 [6]	106M	3.7B
yahoo-web [27]	1.4B	6.6B
Twitter follow-graph (Sep 2012)	-	>20B

Table 1: Experiment graphs. The exact size of Twitter's follow graph cannot be disclosed.

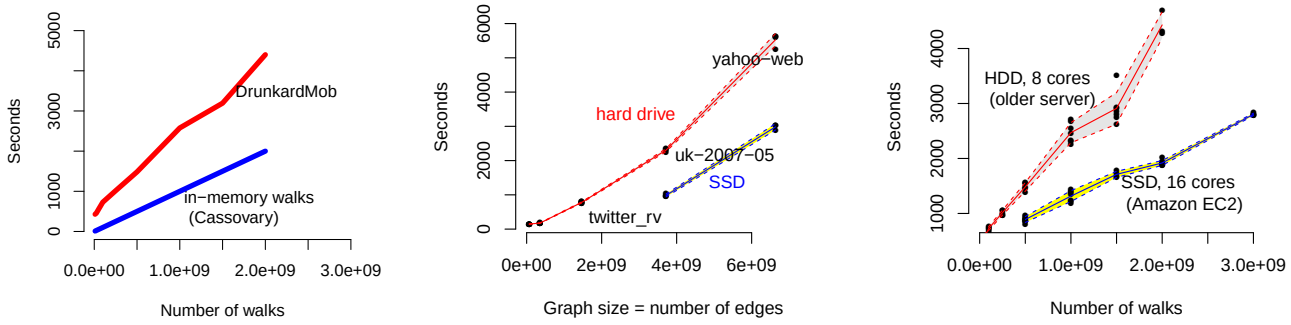
6.1 Large-scale experiments

We run DrunkardMob to compute estimate of Personalized PageRank for almost 15 million users on Twitter's full follow-graph in Sept 2012[4]. The exact size of the graph is not public information, but we can disclose that it had more than 20 billion follower-edges. For each source we started 900 walks and run six iterations. The total number of walks was approximately 13.1 billion.

The experiment was run using two server machines with 144GB of RAM, an SSD and 24-core Intel Xeon 2.4GHz CPUs. One server executed DrunkardMob on GraphChi, while the other one was used to run DrunkardCompanion.

Approximate running times of our results are provided in Table 2. The times have been scaled as-if the graph would have 20 billion edges. It is interesting to note that the algorithm runs much faster when the source vertices are the first 15 million vertices than when vertices from the middle of the ID range are chosen. This skew is probably explained by the fact that Twitter issues user IDs in order: the early Twitter users have smaller IDs than users who have registered later. Also, many of the celebrities are early users of Twitter, and simply have had more time to collect followers than later users. Based on this, we postulate random walks started from early users concentrate faster around the popular nodes, allowing DrunkardMob to operate on smaller set of active vertices. Unfortunately, we did not have opportunity to study this phenomenon further.

[3]Unfortunately, we did not have access to the complete Twitter follow-graph for this experiment.
[4]This experiment was conducted during author's internship at Twitter Inc in Fall 2012



(a) Comparison to in-memory walks (b) Running time / graph size (c) Running time / number of walks

Figure 5: (a) Running time of DrunkardMob and in-memory graph walks with Cassovary. Both experiments were made on a server with hard drive, 32 gigabytes of memory and 8 cores. Cassovary's timings are extrapolated for clarity. (b) Running time of six iterations of DrunkardMob for 200 million walks on various graphs (Mac Mini, hard drive and SSD). (c) Running time on the `twitter_rv` graph when the number of walks is varied. Upper line is for a server with a normal hard drive, while lower is for a high-performance server with an SSD. Time increases approximately linearly with the number of walks and the graph size. Shaded areas show the standard deviation.

Source vertex IDs	Type	Runtime (1 iter.)
0 - 14.6M	unweighted	47 min
14.6M - 29.2M	unweighted	100 min
0 - 14.6M	weighted	55 min
14.6M - 29.2M	weighted	98 min
100M - 114.6M	weighted	91 min
50M - 64.6M	weighted	97 min
200M - 214.6M	weighted	88 min

Table 2: Running time of DrunkardMob on the full Twitter follow-graph. The numbers have been scaled to approximate results on a graph with 20B edges. For each source, 900 walks were issued.

Interestingly weighted and unweighted versions of DrunkardMob have approximately the same running times. For fast weighted sampling of edges we used the alias method by Kronmal and Peterson [17]. It is plausible that the weighted sampling causes the walks to concentrate faster on popular nodes, accelerating the DrunkardMob operations and hiding the increased cost of weighted sampling.

This experiment shows that DrunkardMob with GraphChi can scale to one of the biggest graphs in the industry, on just a single machine. We also run DrunkardMob on the full Twitter graph on a MacBook Pro laptop with SSD and 8 gigabytes of RAM, and could execute half a billion walks simultaneously in approximately one hour / iteration (these experiments were run without a DrunkardCompanion, and are thus not comparable with the results in Table 2).

6.2 Scalability and Performance

Comparison to in-memory walks: We compared DrunkardMob with Cassovary[5], a high-performance in-memory graph library by Twitter which is able to load relative large graphs into memory. Cassovary is implemented in Java/Scala, similarly to ours, allowing a fair comparison. In our experiments, we found Cassovary to be roughly 30% faster in simulating walks than DrunkardMob on a small graph (`live-journal`) and 2 times faster on the larger `twitter_rv.net` graph. Timings for the latter are shown in Fig. 5a.

Scalability: Figure 5 shows the running time of DrunkardMob when the graph size or number of walks is varied. in Fig. 5b we plot the running time as function of graph size. The experiments was conducted on a Mac Mini with 8 GB of memory and an SSD. For each experiment we simulated a total of 200 million walks. In Figure 5c we used the `twitter_rv.net` graph and varied the number of sources and the number of walks simulated from each source. The higher curve is on an old (from year 2008) 8-core Intel Xeon 3.4GHz server with 32GB RAM and a hard drive; lower curve is for a high performance Amazon *hi1.4xlarge* instance with 16 virtual cores, 64GB of RAM and an SSD drive. The latter is faster, but both show close to linear scalability as the number of walks increases. We found that the overhead of Java's garbage collection becomes the main bottleneck when the the number of walks is very large.

On sufficiently large problems, the performance is close to linear in both the graph size and the number of walks being simulated. Based on these experiments, we can conclude that DrunkardMob can simulate random walks in similar time as an in-memory algorithm, but can process much bigger graphs, as it is not limited by the amount of RAM.

7. CONCLUSIONS AND FUTURE WORK

We presented DrunkardMob and demonstrated how it could be used to simulate very large scale random walk simulations on some of the biggest graphs available, on just a single computer. Compared to previous work which have used similar approaches, DrunkardMob on GraphChi provides a generic platform for implementing algorithms based on walks (not necessarily random) on graphs. Programming walks for DrunkardMob is as easy as it is for an in-memory

[5] `https://github.com/twitter/cassovary`

graph: programmer provides a simple walk-update function and the system takes care of the rest.

There are several remaining challenges for future work: First, could DrunkardMob be implemented on top of a distributed graph system such as GraphLab [11]? We also would like to study whether we could increase the scalability of DrunkardMob by utilizing the disk also to store the walk status array. Finally, we would like to extend the DrunkardMob to naturally support evolving graphs in a principled manner, by implementing the incremental technique proposed in [4].

Acknowledgments
This work was initiated during author's internship at Twitter Inc during Fall 2012. Author is advised by Carlos Guestrin and Guy Blelloch. We thank Pankaj Gupta, Matthew Gardner and Mayank Mohta for helpful feedback and encouragement. Funded by ISTC Cloud Computing and NSF IIS-1258741

8. REFERENCES

[1] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *Proceedings of the 3rd Annual ACM Web Science Conference*, pages 33–42. ACM, 2012.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'06. ACM, 2006.

[3] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD Conference*, pages 973–984, 2011.

[4] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010.

[5] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems (TODS)*, 35(4):26, 2010.

[6] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[7] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 69–78. ACM, 2008.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of the 6th USENIX conference on Operating systems design and implementation*, OSDI'04, pages 10–10, San Francisco, CA, 2004. USENIX.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[10] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[11] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. OSDI'12, Hollywood, CA, 2012.

[12] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter.

[13] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Folkrank: A ranking algorithm for folksonomies. *Proc. FGIR*, 2006, 2006.

[14] IBM. http://www.ibm.com/developerworks/java/library/j-codetoheap/.

[15] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 397–406. ACM, 2009.

[16] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279. ACM, 2003.

[17] R. A. Kronmal and A. V. Peterson Jr. On the alias method for generating random variables from a discrete distribution. *The American Statistician*, 33(4):214–218, 1979.

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

[20] S. Lee, S.-i. Song, M. Kahng, D. Lee, and S.-g. Lee. Random walk based entity ranking on graph for multidimensional recommendation. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 93–100. ACM, 2011.

[21] R. Lempel and S. Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001.

[22] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008.

[23] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[24] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[25] J. Misra and D. Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

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

[27] Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002, 2012. `http://webscope.sandbox.yahoo.com/`.