NearBucket-LSH: Efficient Similarity Search in P2P Networks

Naama Kraus[1], David Carmel[2], Idit Keidar[1 2], and Meni Orenbach[1]

[1] Viterbi EE Technion, Haifa, Israel
[2] Yahoo Research, Haifa, Israel

Abstract. We present NearBucket-LSH, an effective algorithm for similarity search in large-scale distributed online social networks organized as peer-to-peer overlays. As communication is a dominant consideration in distributed systems, we focus on minimizing the network cost while guaranteeing good search quality. Our algorithm is based on Locality Sensitive Hashing (LSH), which limits the search to collections of objects, called buckets, that have a high probability to be similar to the query. More specifically, NearBucket-LSH employs an LSH extension that searches in near buckets, and improves search quality but also significantly increases the network cost. We decrease the network cost by considering the internals of both LSH and the P2P overlay, and harnessing their properties to our needs. We show that our NearBucket-LSH increases search quality for a given network cost compared to previous art. In many cases, the search quality increases by more than 50%.

1 Introduction

User *similarity search* in *Online Social Networks (OSNs)* is the task of effectively finding OSN users that are similar to a given user based on common *interests*. It is used for many applications including recommending new friends [23, 28], as well as for recommending content based on preferences of similar users [2]. In this work, we consider *Peer-to-Peer (P2P)* OSNs (e.g., [6, 9, 24, 21]), which offer increased scalability and avoid control by a single authority.

A similarity search algorithm in P2P OSNs faces several challenges: The algorithm should be decentralized in order to fit the P2P architecture. As network cost is a dominant consideration in P2P networks, the algorithm should be network-efficient, while preserving a good search quality. Furthermore, the similarity search should cope with the dynamic nature of OSNs: users join or leave, and users dynamically modify their interests. We present a similarity search algorithm in P2P OSNs that meets these requirements.

We base our algorithm on *Locality Sensitive Hashing (LSH)* [16, 14], which is a widespread randomized method for efficient similarity search in high-dimensional spaces. LSH hashes an OSN user into a succinct representation, where the hash values of similar users collide with high probability (w.h.p.). At a pre-processing stage, LSH maps users into collections of objects called *buckets* based on common hashes. Upon receiving a query, LSH limits the search to buckets to which the query is mapped;

these contain similar users w.h.p. We follow a variant of LSH, called MultiProb-LSH [20], which increases search quality by additionally searching *near buckets*, which are buckets that are similar to the query's bucket.

We present *NearBucket-LSH*, which integrates LSH into a P2P architecture. For our P2P overlay we use *Content Addressable Network (CAN)* [25], which is a good fit for a distributed LSH implementation. We use CAN to dynamically map and store LSH buckets within nodes, and refresh bucket contents once in a while in order to adjust to changes in the data. Upon search, we use CAN to locate the buckets to search in.

In P2P settings, searching additional buckets entails contacting additional nodes, which is a network-costly operation. We improve the network-efficiency when searching near buckets by exploiting the internals of CAN: We observe that in CAN, near buckets reside in a bucket's neighboring nodes, and thus contacting them incurs a low network cost. We further eliminate this network cost by caching near buckets in each CAN node. We show, both analytically and empirically, that the cache-based NearBucket-LSH provides the greatest search quality for a given network cost, compared to other approaches.

2 Model and Problem Definition

In this section we detail the model we consider. We formally define the notion of similarity search (Section 2.1), and provide details about P2P OSNs (Section 2.2).

2.1 User Similarity Search in OSN

An OSN user exposes an *interest profile*, which we represent as a non-negative weighted feature vector in a high d-dimensional vector space $V = (\mathbb{R}_0^+)^d$. The interests-weighting scheme may be arbitrary. A *similarity function* [8] measures the similarity between two user vectors. It returns a *similarity value* within the range $[0, 1]$, where a similarity value of 1 denotes complete similarity, and 0 denotes no similarity.

An *m-similarity search* algorithm accepts as an input a *query* vector $q \in V$. It returns a unique *ideal result set* of m user vectors that are most similar to q, according to the given similarity function. An *approximate m-similarity search* algorithm trades-off efficiency with accuracy. Given a query q, it returns an *approximate result set* of m user vectors, which may differ from q's ideal result set. We consider the commonly used *cosine similarity* function [22], also proposed in the context of similarity between OSN users [3].

2.2 P2P OSN and CAN

P2P networks are distributed systems organized as overlay networks with no central management. Nodes (also called *peers*) are autonomous entities that may join or leave at any time. P2P networks provide massive scalability, fault tolerance, privacy, anonymity, and load balancing (see [18] for a survey). We consider a P2P Online

Social Network [6, 9, 24, 21], in which users' content is distributed among nodes. Any node in the P2P OSN may initiate a similarity search query.

In our algorithm, we use CAN [25] as our overlay, which naturally fits a distributed LSH implementation, as we later show. CAN implements a self-organizing P2P network representing a virtual c-dimensional Cartesian coordinate space on a c-torus. The Cartesian space is dynamically partitioned into *zones*, which are distributed among CAN nodes. CAN implements a *Distributed Hash Table (DHT)* abstraction, which provides a distributed *lookup* operation that accepts a vector as key, and returns a node that owns the zone to which the vector belongs. Each node maintains a table of *neighbors*, which are nodes that own zones adjacent to its own. These tables are used for routing messages within CAN.

3 Background and Previous Work

Before diving into our algorithm, we provide essential background and overview previous work. In Section 3.1, we overview LSH [16, 14] and its space-efficient variant, MultiProb-LSH [20]. Section 3.2 discusses Layered-LSH [15, 4], which is a distributed LSH implementation that optimizes network cost. In addition to distributed solutions, there are also parallel LSH variants, e.g. [26]. However, these do not focus on improving network-efficiency, which is not of essence in a parallel setting. Other P2P similarity search methods have been proposed [5], in particular, Falchi et al. [12] use CAN as their overlay. However, these methods are not based on LSH, which is the focus of our work.

3.1 Locality Sensitive Hashing.

Locality sensitive hashing [16, 14] is a widely used approximate similarity search algorithm for high-dimensional spaces, with sub-linear search time complexity. LSH limits the search to vectors that are similar to the query vector w.h.p. instead of linearly searching over all vectors. This reduces the search time complexity at the cost of missing similar vectors with a some probability.

LSH uses hash functions that map a vector in the high dimensional input space $(\mathbb{R}_0^+)^d$ into a representation in a lower dimension $k << d$, so that the hashes of similar vectors are likely to collide. LSH executes a pre-processing (index building) stage, where it assigns vectors into buckets according to their hash values. Then, given a query vector, the similarity search algorithm computes its hashes and searches vectors in the corresponding buckets. The LSH algorithm is parametrized by k and L, where k is the hashed domain's dimension, and L is the number of hash functions used. Formally [7]: a *locality sensitive hashing* with similarity function sim is a distribution on a family $\mathcal{H}$ of hash functions on a collection of vectors, $h : V \to \{0, 1\}$, such that for two vectors u, v,

$$Pr_{h\in\mathcal{H}}\left[h(u) = h(v)\right] = sim(u, v). \qquad (1)$$

We use here a hash family $\mathcal{H}$ for angular similarity [7], which fits cosine-based similarity search [7]. In order to increase the probability that similar vectors are mapped to the same bucket, the algorithm defines a family $\mathcal{G}$ of hash functions, where each

$g(v) \in \mathcal{G}$ is a concatenation of k functions chosen randomly and independently from $\mathcal{H}$. In the case of angular similarity, $g : V \to \{0,1\}^k$, i.e., g hashes v into a binary *sketch vector*, which encodes v in a lower dimension k. For two vectors u, v, $Pr_{g \in \mathcal{G}} [g(u) = g(v)] = (sim(u,v))^k$, for any randomly selected $g \in \mathcal{G}$. The larger k is, the higher the precision.

In order to mitigate the probability to miss similar items, the LSH algorithm selects L functions randomly and independently from $\mathcal{G}$. The item vectors are now replicated in L hash tables, where each vector is mapped to L buckets. Upon query, search is performed in L buckets. This increases the recall at the cost of additional storage and processing. In order to reduce the storage cost, MultiProb-LSH [20] additionally searches in *near buckets* within the same hash table, which are buckets that slightly differ from the query's *exact bucket* $g(q)$, and have a high probability to contain vectors similar to the query.

3.2 Layered LSH.

In P2P networks, buckets are distributed over the overlay nodes. Contacting a near bucket involves performing a DHT lookup of its node, which incurs high network cost. Prior art [15, 4] suggests *Layered-LSH*, which maps buckets to nodes using a second LSH, such that near buckets are assigned to the same node w.h.p. Queries now access a single node holding the desired buckets, which reduces the network cost. In Section 5.1, we show that in the case of cosine similarity, Layered-LSH is equivalent to the basic LSH for an appropriate choice of k.

4 Algorithm

We describe NearBucket-LSH, our network-efficient P2P user similarity search that is based on MultiProb-LSH. We construct a dedicated overlay above the CAN infrastructure, and exploit its internals for reducing search network cost when searching near buckets.

The Overlay. We use a k-dimensional CAN (i.e., $c = k$) to store and lookup LSH buckets in a decentralized manner. For simplicity, we assume that $N = 2^k$, where N denotes the number of CAN nodes. Each CAN node owns the zone of a single k-dimensional binary vector v representing some LSH sketch vector, and maintains the bucket of user vectors that are mapped to v by some hash function $g \in \mathcal{G}$. We name such a node the *bucket node* of v. The bucket node provides a local similarity search facility over its locally stored user vectors. The local search time is typically proportional to the searched bucket size [14]. The internal bucket data-structure and local search implementation are orthogonal to this research.

Each CAN node in our overlay has k neighbors; the i-th neighbor of node v owns a vector u that differs from v in the i-th entry only. Routing a message from node v to one of its neighbors requires a single hop, i.e., a single message. Routing a message from an arbitrary source node v to an arbitrary target node u, entails modifying the binary vector entries that differ between u and v. Two vectors of length k, differ in $k/2$ entries in expectation, and thus, the expected path length is $k/2$ hops[3].

The L hash functions $g = \{g_1, \cdots, g_L\}$ are randomly selected from $\mathcal{G}$ a priori. They are given to the distributed algorithm as a configuration parameter, and are known to all bucket nodes. CAN supports multiple hash functions [25], which we use for supporting multiple g_i's and mapping each user vector into L bucket nodes.

Bucket Maintenance. Our algorithm constructs and refreshes the buckets continuously, in a decentralized manner. Each user periodically re-hashes its vector using LSH into L sketch vectors. It then performs DHT lookups to locate the corresponding bucket nodes, and sends them the fresh user vector. Note that the user vector may or may not have changed since the previous update message.

We do not construct buckets a priori. Rather, bucket construction is triggered by vector update messages. A CAN node becomes an active bucket node when it first receives a notification of some user vector. Since user vectors change dynamically, their hashes change accordingly. Obsolete vectors that are not refreshed for a certain predefined length of time are garbage-collected from bucket nodes.

Query Processing. Algorithm 1 depicts NearBucket-LSH query processing procedure: Each P2P node may trigger an m-similarity search request for an input query q. The initiating node, denoted n, hashes q into L sketch vectors $g_i(q)$, looks-up the corresponding exact bucket nodes, and sends them m-similarity search requests in parallel. Once a query request reaches some exact bucket node $g_i(q)$, the node performs a local similarity search in its own bucket, and also forwards the request to the k near bucket nodes that differ from $g_i(q)$ in exactly one entry. All (exact and near) bucket nodes send back a set of up to m results to node n. Node n merges the result sets and returns a final m-result set to the caller.

As a CAN node maintains a table of k neighbors that differ from it in exactly one entry, these neighbors hold the desired near buckets. Thus, contacting a near bucket node costs a single message, and a total of kL messages per query. We further eliminate these additional messages by caching k near buckets at each CAN node. In order to maintain fresh caches, each node periodically sends its bucket to its neighbors. The cache requires an additional storage of size kB at each node, where B is the average bucket size.

It is possible to cache all k near buckets or any subset of them. For the purpose of the analysis and evaluation in the next sections, we refer to the following two extremes: we name NB-LSH a NearBucket-LSH that does not use caching at all, and CNB-LSH a NearBucket-LSH that caches all k near buckets. In addition, we refer in LSH to the basic LSH algorithm, which completely avoids searching near buckets.

[3] Note that in a general c-dimensional CAN of N nodes, the expected routing length is $c/4\left(N^{1/c}\right)$ [25], which equals $k/2$ for $c = k$ and $N = 2^k$.

Algorithm 1 NearBucket-LSH Query Processing

```
 1: function QUERY($q$)                                        ▷ At the query node
 2:     pforeach $g_i \in g$ do                                ▷ A parallel foreach
 3:         $v_i \leftarrow g_i(q)$
 4:         $n_i \leftarrow DHT$.LOOKUP($v_i$)                  ▷ Lookup bucket node
 5:         $n_i$.SENDREQ($SimSearchNB, q, n$)                 ▷ Send request
 6:     end pforeach
 7:     $hits \leftarrow$ collect results from bucket nodes
 8:     return top $m$ hits                                    ▷ Rank and return top $m$
 9: end function

10: function SIMSEARCHNB(q, n)                                 ▷ Query $q$ from $n$
11:     $res \leftarrow Bucket$.LOCALSIMSEARCH($q$)            ▷ Local search
12:     $n$.SENDRES($res$)                                     ▷ Send back result
13:     pforeach $j \in \{1, \cdots, k\}$ do                   ▷ A parallel foreach
14:         $n_j \leftarrow Neighbors.j$                       ▷ Extract the $j$-th neighbor
15:         if $Bucket_j$ is cached then                       ▷ Neighbor's bucket is cached
16:             $res \leftarrow Bucket_j$.LOCALSIMSEARCH($q$)  ▷ Local search
17:             $n$.SENDRES($res$)                             ▷ Send back result
18:         else
19:             $n_j$.SENDREQ($SimSearch, q, n$)               ▷ Forward request
20:         end if
21:     end pforeach
22: end function

23: function SIMSEARCH($q$, $n$)                               ▷ Query $q$ from $n$
24:     $res \leftarrow Bucket$.LOCALSIMSEARCH($q$)            ▷ Local search
25:     $n$.SENDRES($res$)                                     ▷ Send back result
26: end function
```

5 Theoretical Analysis

We theoretically analyze an algorithm's capability of retrieving similar objects, and show the superiority of NearBucket-LSH to successfully retrieve similar objects for a given network cost.

5.1 Success Probability Formulation

The basic building block in our analysis is the *success probability* [20] of an algorithm A to find object y that has a similarity value s to query object q, under a random selection of $g \in \mathcal{G}$. We denote this success probability by $SP(A, s)$.

LSH. Let $LSH(k, L)$ denote the angular-LSH algorithm with parameters k and L, and let s denote the angular similarity between query q and searched object y. According to the LSH theory [7], for a randomly selected $h \in \mathcal{H}$:

$$Pr_{h \in \mathcal{H}}\left[h(q) = h(y)\right] = s, \text{ and } Pr_{h \in \mathcal{H}}\left[h(q) \neq h(y)\right] = (1 - s). \tag{2}$$

$LSH(k, L)$ searches in L exact buckets independently, thus, it finds y in any of these buckets with probability:

Proposition 1

$$SP(LSH(k, L), s) = 1 - \left(1 - s^k\right)^L .$$

NearBucket-LSH. We define *b-near buckets* to be buckets that differ from an exact bucket in $0 \leq b \leq k$ entries (note that a 0-near bucket is an exact bucket). The success probability of finding y in a b-near bucket of $g(q)$ is:

$$s^{k-b}(1 - s)^b. \tag{3}$$

As our vectors are non-negative, their angular similarities s satisfy that $s \in [0.5, 1]$. This implies that $\forall s, (1 - s) \leq s$, and therefore, for $0 \leq b_1 < b_2 \leq k$, $s^{k-b_2}(1 - s)^{b_2} \leq s^{k-b_1}(1 - s)^{b_1}$, thus:

Proposition 2 *The success probability when searching in a b_1-near bucket is greater or equal to the success probability when searching in a b_2-near bucket, for any $0 \leq b_1 < b_2 \leq k$. Hence, NearBucket-LSH's selection of k 1-near buckets is optimal, with respect to any other k buckets selected for search, in addition to the exact bucket.*

The exact bucket and its near buckets are disjoint, as an object is mapped to exactly one bucket according to a specific g. NearBucket-LSH searches in L exact buckets each along with its k 1-near buckets. Thus,

Proposition 3

$$SP(\textit{NearBucket-LSH}(k, L), s) = 1 - (1 - (s^k + ks^{k-1}(1 - s)))^L.$$

Layered-LSH. We show that for the angular similarity, Layered-LSH is equivalent to the basic LSH. Layered-LSH maps near buckets to the same node w.h.p., which can be achieved by using Hamming-based LSH [14, 8] as follows. Let g_{ang} be the angular-LSH used for mapping vectors to buckets. By definition, g_{ang} is a concatenation of h_i angular-LSH functions. Let g_{ham} be the Hamming-LSH used for mapping buckets to nodes. Hamming-based LSH hashes a binary vector to another binary vector of a lower dimension k, by randomly and independently selecting k entries of the input vector. In our case, this resorts to randomly and independently selecting k entries from $g_{ang}(v)$, each of which corresponds to some $h_i \in \mathcal{H}$. We get that $g_{ham}(g_{ang}(v))$ maps v to a node according to k randomly selected $h \in \mathcal{H}$ functions, which is equivalent to using the angular-LSH with parameter k.

5.2 Success Probability Comparison

We use Propositions 1 and 3 to compare the success probabilities of LSH, Layered-LSH, and NearBucket-LSH. We compute an algorithm's success probability as a function of the cosine similarity between the query and the searched object[4]. As Layered-LSH is equivalent to LSH, we refer to both as LSH in this discussion. For the purpose of the demonstration, we present graphs for selected k and L values. Note however that we observed the same trend for other k and L values; we omit the respective graphs from this text.

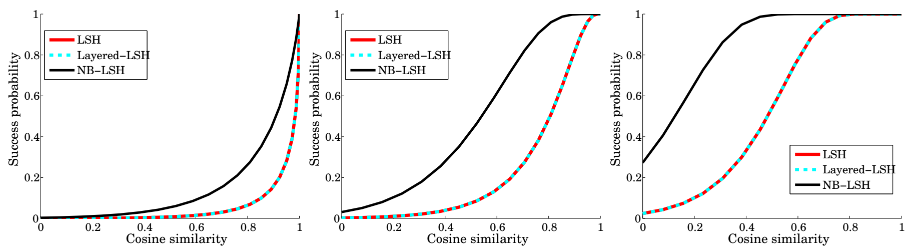
Constant Number of Hash Functions. We compare LSH and NearBucket-LSH for a constant L. Figure 1 depicts their success probabilities for $k = 12$ and for increasing L values of 1, 10, and 100. As the graphs demonstrate, the success probability of NearBucket-LSH is greater than or equal to the success probability of LSH for all similarities, for a constant L This stems from the fact that NearBucket-LSH searches in kL additional near buckets, which increases its success probability.

Network Efficiency. As we have seen, for a constant L, NearBucket-LSH increases the success probability of LSH at the cost of contacting additional buckets. We proceed to analyzing the success probability as a function of the network cost. We measure the network cost by the average number of messages per query. We distinguish between the cached (CNB-LSH) and non-cached (NB-LSH) versions of NearBucket-LSH.

The first column of Table 1 summarizes the number of bucket nodes contacted (and searched) by each of the algorithms, for given k and L parameters. Looking up an exact bucket node requires an average of $k/2$ routing hops, and contacting a neighbor node costs one message. The second column in Table 1 summarizes the average number of messages per query, for given k and L parameters.

Figure 2 depicts success probability for $k = 12$ and an increasing network costs of 18, 180, and 1800 average number of messages. The graphs illustrates that, thanks to the low network cost of searching near buckets, NearBucket-LSH, (and more notably CNB-LSH), improves LSH's success probability for all similarity values, for a constant

[4] We transform cosine similarity into angular similarity and then apply the success probability formulas.



(a) $L = 1$ (b) $L = 10$ (c) $L = 100$

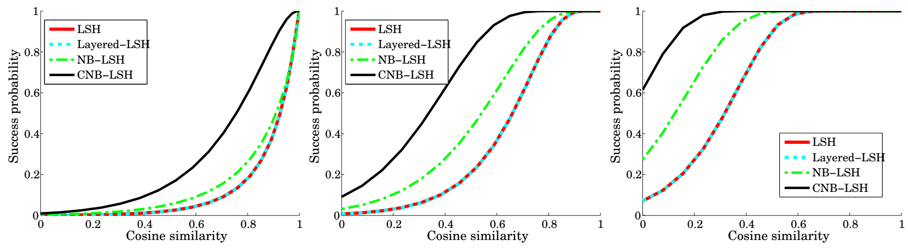
Fig. 1. Analytical success probability as a function of L ($k = 12$). NearBucket-LSH guarantees a greater or equal success probability compared to LSH and Layered-LSH, as it searches in more buckets (namely, near buckets). The gap increases as L increases.

	Number of nodes contacted per query	Average number of messages per query	Number of vectors stored in a node	Number of vectors searched per query
LSH	L	$\frac{1}{2}kL$	B	LB
Layered-LSH	L	$\frac{1}{2}kL$	B	LB
NB-LSH	L(1+k)	$1\frac{1}{2}kL$	B	$L(k+1)B$
CNB-LSH	L	$\frac{1}{2}kL$	$(k+1)B$	$L(k+1)B$

Table 1. Summary of costs of similarity search in CAN-based LSH variants for given k, L LSH parameters.

average number of messages. Note that one could further extend NearBucket-LSH to search in near buckets that differ from the query's bucket in more than one entry. The success probability of such buckets decreases (Proposition 2), whereas the network cost in NB-LSH and the storage cost in CNB-LSH increases compared to 1-near buckets. Thus, searching additional buckets is expected to be less effective.

Other Considerations. Our work focuses on minimizing the network cost, which is a dominant cost in P2P networks. For completeness, we present in the third and fourth columns of Table 1 other costs which tradeoff with network-efficiency. We denote the average bucket size by B. In terms of storage capacity, NB-LSH preserves the same space complexity as LSH and Layered-LSH. CNB-LSH increases the space complexity due to caching, while being more network-efficient than NB-LSH. Both NearBucket-LSH variants search over a larger number of vectors than LSH, implying more processing work per query. As our algorithm searches the buckets in parallel, and the average bucket size is equal in all algorithms, this does not affect the query latency.



(a) network cost: 18 (b) network cost: 180 (c) network cost: 1800

Fig. 2. Analytical success probability as a function of network cost for $k = 12$. NB-LSH exploits the low lookup cost of near buckets in CAN, and increases LSH's and Layered-LSH's success probability for a given network cost. CNB-LSH further saves messages by caching near buckets, and achieves the greatest success probability for a given network cost.

6 Evaluation

We empirically evaluate our algorithm on three real world OSN datasets of varying sizes, and demonstrate the superiority of CNB-LSH over other approaches.

6.1 Search Quality Measures

Recall. (at m) is defined as follows [20]:

Definition 1 (recall at m) *Given a query q, let $I_m(q)$ denote its ideal m-result set. Let $A_m(q)$ denote the approximate m-result set of q returned by some algorithm A. An algorithm's recall for query q is the fraction of results from q's m-ideal result set that are returned by A:*

$$recall@m(A, q) = \frac{|A_m(q) \cap I_m(q)|}{|I_m(q)|}. \tag{4}$$

An algorithm's recall, $recall@m\,(A)$, is the mean of the queries' recall averaged over a query set Q.

Normalized Cumulative Similarity. We measure an algorithm's precision by comparing the similarity scores of its m-result set to those of the ideal m-result set. We define the following ratio, which we name the *normalized cumulative similarity (NCS)*:

Definition 2 (NCS at m) *Given a query q,*
let $CumSim(I_m, q)$ denote the sum of the similarity values to q of the results in q's ideal m-result set.
Let $CumSim(A_m, q)$ denote the sum of the similarity values to q of the results in q's m-result set of a given algorithm A. Then,

$$NCS@m(A, q) = \frac{CumSim(A_m, q)}{CumSim(I_m, q)} \tag{5}$$

We measure the NCS of an algorithm, $NCS@m(A)$, by averaging it over the query set Q.

Note that $CumSim(I_m, q) \geq CumSim(A_m, q)$, and both are positive. Therefore, $NCS@m(A) \in [0, 1]$.

6.2 Methodology

Datasets. We use three real-world publically-available datasets of OSNs [29]:

– *DBLP* [11], the computer science bibliography database: Authors are users, and venues are interests. We use a crawl of 13,477 interests, and 260,998 users that have at least one interest.
– *LiveJournal* [17] blogging-based OSN: Users publish blogs and form interest groups, which users can join. The LiveJournal crawl consists of 664,414 such groups, which we consider as user interests. There are 1,147,948 users with at least one interest.
– *Friendster* [13] online gaming network: Similarly to LiveJournal, Friendster allows users to form interest groups, which we consider as interests. The dataset consists of 1,620,991 interest groups, and 7,944,949 users with at least one interest.

All datasets contain anonymous user ids and interest information. We filtered out users having no interest.

Parameters. We set $k = 10$ in DBLP, $k = 12$ in LiveJournal and $k = 15$ in Friendster. We follow previous art [4, 15] that uses k values between 10 and 20, and bucket sizes of a few hundreds [14]. Thus, we have 1,024 buckets in DBLP, 4,096 in LiveJournal, and 32,768 in Friendster. The average bucket size is approximately 250 vectors in all datasets. We set m, the number of search results, to 10.

Creating Sketch Vectors. We construct users' weighted interest vectors according to the dataset at hand. We weight each interest I based on its inverse frequency in user vectors [1]: $w(I) = ln(\frac{N_u}{N_I+1}) + 1$, where N_u denotes the total number of users, and N_I denotes the number of users having interest I. The user vector entry v_i is zero or $w(I)$ according to whether the user is associated with specific interest I. We use TarsosLSH's [27] for mapping vectors into LSH buckets.

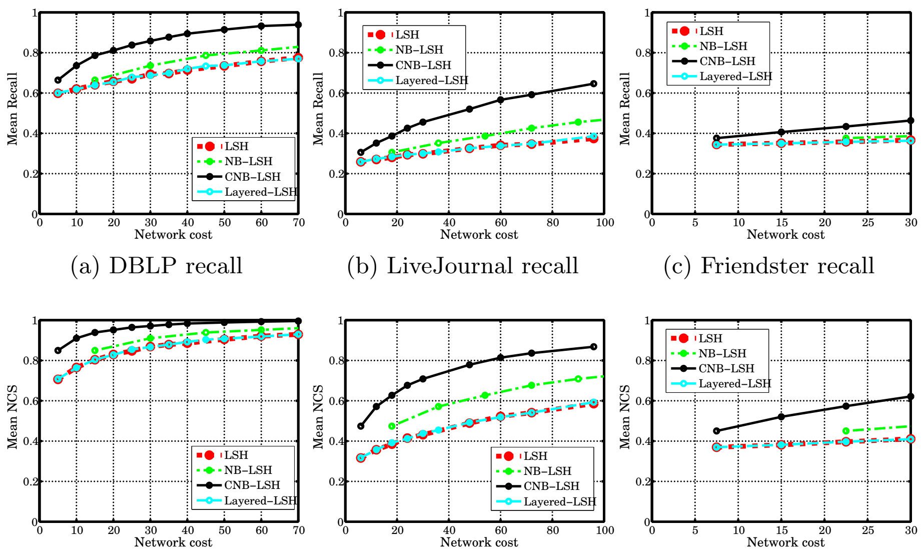
Simulator. We implement a simulator of our CAN-based overlay using Apache Lucene 4.3.0 [19] centralized search index. We simulate distributing user vectors in bucket nodes by indexing vectors by their hash values (sketch vectors). The hash is then used for looking up a specific bucket node, and local similarity search is performed by limiting the search to the selected bucket (using Lucene's Filter mechanism). We additionally use Lucene to compute the ideal result set of a given query, by executing the query over the whole dataset. We score results according to the cosine similarity.

Evaluation Set. We construct a query set of 3,000 randomly sampled users. For each query q, we retrieve its ideal result set, as well as the result sets according to the algorithms we compare. For each dataset, we measure recall and precision over the query set in use.

6.3 Search Quality Results

Figure 3 illustrates our experimental results as a function of network cost. As in Section 5.2, we measure the network cost by the average number of messages per query according to Table 1. We increase the network cost by gradually increasing L, which increases search quality for all datasets as expected. We use larger values of k for larger datasets in order to preserve a common average bucket size. This ensures that local search takes the same time, and the cache sizes are identical. The larger k is, the lower the success probability is, thus, we expect a decrease in search quality when the dataset size increases, which is indeed demonstrated in the graphs.

The three datasets show a similar trend. Layered-LSH's search quality equals that of the basic LSH as expected. NearBucket-LSH (both cached and non-cached) demonstrates an increase in search quality compared to LSH and Layered-LSH, which is achieved by searching in additional near buckets stored at neighboring nodes or the node itself. For example, in LiveJournal (second column), LSH requires an average of 96 messages per query in order to achieve 0.59 precision, whereas CNB-LSH achieves a precision of 0.57 using only 12 messages. CNB-LSH also improves recall significantly, for example, achieving a 0.59 recall using 72 queries, compared to a recall of 0.35 for LSH. In all cases, NB-LSH is between LSH and CNB-LSH.



(a) DBLP recall (b) LiveJournal recall (c) Friendster recall

(d) DBLP precision (e) LiveJournal precision (f) Friendster precision

Fig. 3. Search quality as a function of the average number of messages per query, for three real world datasets: DBLP, LiveJournal, and Friendster ($k = 10$, $k = 12$, $k = 15$, respectively). For all datasets, CNB-LSH provides the greatest search quality as a function of the network cost, according to two metrics: recall and precision.

7 Conclusions and Future Work

We presented NearBucket-LSH, a network-efficient LSH algorithm for P2P OSNs, which provides good search quality. We first analytically showed that, for angular similarity, our choice of searched near buckets is optimal, that is, near buckets that differ in a single entry from the query's bucket are more likely to contain similar vectors than other near buckets. We then showed, both mathematically and empirically, that one may dramatically lower the additional network cost for searching in these buckets by exploiting CAN's internal structure and judicious caching.

Our proposed overlay focuses on angular-LSH, which fits OSN similarity search. It would be of an interest to extend our overlay to support other LSH families such as l_p-LSH, which map vectors to hashes in $\mathbb{Z}^k$ [10]. We expect such an extension to naturally fit our CAN overlay: According to l_p-LSH, Near buckets are computed by adding $\{-1, +1\}$ to an entry of a given hash vector [20]. Thus, when constructing a CAN over $\mathbb{Z}^k$, a near bucket's node resorts to the current node or its neighbor [25], which follows our design.

Acknowledgments

Naama Kraus is grateful to the Hasso-Plattner-Institut (HPI) for the scholarship for doctoral studies.

References

1. L. A. Adamic and E. Adar. Friends and neighbors on the web. *SOCIAL NETWORKS*, 25:211–230, 2001.
2. G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005.
3. A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec. Effects of user similarity in social media. WSDM '12, pages 703–712, 2012.
4. B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *CIKM '12*, pages 2174–2178, 2012.
5. M. Batko, D. Novak, F. Falchi, and P. Zezula. Scalability comparison of peer-to-peer similarity search structures. *Future Generation Comp. Syst.*, pages 834–848, 2008.
6. S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. PeerSoN: P2P social networking - early experiences and insights. In *SNS '09*, pages 46–52, March 31, 2009.
7. M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02*, pages 380–388, 2002.
8. F. Chierichetti and R. Kumar. Lsh-preserving functions and their applications. In *SODA '12*, pages 1078–1094, 2012.
9. L. A. Cutillo, R. Molva, and M. Önen. Safebook: A distributed privacy preserving online social network. In *WOWMOM*, pages 1–3, 2011.
10. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04*, pages 253–262, 2004.
11. DBLP. http://www.informatik.uni-trier.de/ ley/db/.

12. F. Falchi, C. Gennaro, and P. Zezula. A content-addressable network for similarity search in metric spaces. In *DBISP2P'05*, pages 98–110, 2005.
13. Friendster. http://www.friendster.com/.
14. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99*, pages 518–529, 1999.
15. P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT '09*, pages 744–755, 2009.
16. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC '98*, pages 604–613, 1998.
17. Livejournal. http://www.livejournal.com/.
18. E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.
19. Lucene. http://lucene.apache.org/core/.
20. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB '07*, pages 950–961, 2007.
21. M. Mani, A.-M. Nguyen, and N. Crespi. Scope: A prototype for spontaneous p2p social networking. In *PerCom Workshops*, pages 220–225, 2010.
22. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge university press, 2008.
23. M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, pages 415–444, 2001.
24. R. Narendula, T. G. Papaioannou, and K. Aberer. Towards the realization of decentralized online social networks: An empirical study. In *ICDCS Workshops*, pages 155–162, 2012.
25. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, New York, NY, USA, 2001.
26. N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, pages 1930–1941, 2013.
27. TarsosLSH. https://github.com/jorensix/tarsoslsh.
28. R. Xiang, J. Neville, and M. Rogati. Modeling relationship strength in online social networks. WWW '10, pages 981–990, 2010.
29. J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *MDS '12*, pages 3:1–3:8, 2012.