World Wide Web (2015) 18:661–679
DOI 10.1007/s11280-014-0276-2

Ranked content advertising in online social networks

Weixiong Rao · Lei Chen · Ilaria Bartolini

Received: 19 July 2012 / Revised: 4 November 2013 /
Accepted: 13 January 2014 / Published online: 5 February 2014

Abstract Online social networks (OSNs) such as Twitter, Digg and Facebook have become popular. Users post news, photos and videos, etc. and followers of such users then view and comment the posted information. In general, we call the users who produce the information as the information *producers*, and the users who view the information as the information *consumers*. The recently popular targeted information advertising systems enable the producers to target users (i.e., consumers). A key problem of the advertising system is to efficiently find the top-k most desirable targeted users, who next will view the advertised information and perform potential e-commerce activities. Unfortunately, state-of-the-art solutions to find the top-k desirable targeted users in large OSNs incur high space cost and slow running time. In this paper, we focus on designing efficient algorithms to overcome such efficiency issues. Experimental results, over synthetic and real data sets, demonstrate the effectiveness and efficiency of our algorithms.

Keywords Top-k query · Shortest path distance · Efficiency

W. Rao (✉) · L. Chen
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China
e-mail: rweixiong@gmail.com

L. Chen
e-mail: leichen@cse.ust.hk

I. Bartolini
Department of Computer Science and Engineering, University of Bologna, Bologna, Italy
e-mail: i.bartolini@unibo.it

1 Introduction

Recently, online social networks (OSN) such as Twitter, Digg and Facebook have become popular. Users post news, photos and videos, etc. and followers of such users then view and comment the posted information. Beyond this, many OSNs nowadays host online applications. Via the hosted application, users advertise product promotion campaign, post job information, or invite partners of online games. Then, other users click the product catalog, apply for the job position, or participate the invited games. In general, we call the users who produce the information as the information *producers*, and the users who view the information as the information *consumers*.

Given the producers and consumers, the recently popular targeted information advertising systems [4, 13, 17, 21, 28, 30–32] enable the producers to target users (i.e., consumers) based on users demographics, profile information and online activities. The targeted users (i.e., consumers) then click the advertised information (including products, online games, news information, etc.), which matches the users' personalized interests. The information producers achieve potential benefits from the clicks and e-commerce activities performed by the consumers.

In this paper, we are interested in an effective and efficient solution to the advertising technique. First, for a specific advertiser, neither massively broadcasting postings information to all users nor randomly finding some target users should be conducted, since such approaches annoy users and makes the postings becoming spams. Thus, among all potential users in OSNs, finding only the top-k desirable users will avoid the spam. Unfortunately, the existing works either ignore the effect of social networks [17, 21, 28, 30] or leverage limited knowledge of social networks [4, 13, 31, 32] (e.g., only the topology of the social networks).

Second, the efficiency is also an important issue for the online advertising application. For example, we take Digg postings as the example of the advertised information. Among the most popular Digg postings, all of them become popular within 3 days; Digging votes, particularly in the initial several hours, are influential to become popular [34]. Therefore, an ideal solution should quickly find top-k desirable users as fast as possible. The targeted users then can view the fresh Digg posting as early as possible. Otherwise, a slow offline process delays the advertisement, and the targeted users have to view expired and then meaningless Digg postings.

Unfortunately, in large OSNs (which are frequently modelled as large $\underline{s}$ocial graphs), performing an efficient $\underline{a}$lgorithm to find the top-k most desirable targeted $\underline{u}$sers is a challenging problem (in short the SAU problem). State-of-the-art top-k solutions require the availability of sorted lists of partial attribute scores. In the context of the SAU problem, the maintenance of sorted lists requires either high space cost or slow running time. In addition, the previous works on social information advertising [4, 13, 31, 32] suffer from high overhead by solving the optimization problem involving the whole social networks.

To overcome the above issues, in this paper, we design efficient algorithms with the following contributions.

- We design an aggregated scoring function to measure the relevance between the advertised information and the targeted users (Section 2). The scoring function outperforms the approach only considering either the content similarity or structure similarity.
- To reduce the searching time, we propose a variant of NRA algorithm, i.e., NRA_SAU, to find the top-k most desirable targeted users. It creates sorted lists of user similarity on

the fly by incrementally running an adapted Dijkstra algorithm to reduce the overhead (Section 3).

- To further improve the running time, the batching algorithm, BAT_SAU, proposes two techniques to optimize NRA_SAU: pruning unnecessary candidates and using multiple phases of batch process (Section 4).
- The empirical study using both the synthetic and real data sets validates the proposed algorithm (Section 5) before we investigate related works (Section 6).

2 Preliminary

In this section, we first define the similarity of users, and then formulate the SAU problem, After that, we review the classic TA and NRA algorithm. Finally, we give alternative algorithms and show their limitations.

2.1 Similarity definition

In an OSN, users are registered to the OSN and maintain follow-up relations with other users. We use a social graph $\mathcal{G}$ to model the OSN. The graph $\mathcal{G}$ consists of user vertexes $\mathcal{V}$ and followup edges $\mathcal{E}$, where a vertex $u \in \mathcal{V}$ indicates a user registered to the OSN and the edge e_{ij} indicates the followup relation between the users u_i and u_j (where $1 \leq i \neq j \leq V$). We consider an undirected graph $\mathcal{G}$, i.e., if u_i is a friend of u_j, then u_j is also a friend of u_i. This property is common in real social networks, such as Microsoft Messenger social network and Facebook.

In the graph $\mathcal{G}$, each edge e_{ij} is associated with a weight to indicate the similarity between u_i and u_j (given in Section 2.1.1). Next, we extend the similarity to the general case that u_i and u_j are connected by a path (given in Section 2.1.2).

2.1.1 Similarity between neighbors

Given two neighbors u_i and u_j in a graph $\mathcal{G}$, the similarity between u_i and u_j, denoted by $\mathtt{sim}_{nbr}(u_i, u_j)$, covers the content similarity and the structure similarity as follows.

First, we compute the *<u>con</u>tent similarity* between u_i and u_j, i.e., $\mathtt{sim}_{con}(u_i, u_j)$, based on the history content items that u_i and u_j ever viewed (voted, or other user activities). If both u_i and u_j share more similar interests and viewed common items more times, we consider that the content similarity between u_i and u_j is higher. This is consistent with the item-based collaborative filtering approach [1]. Suppose $\mathcal{D}(u_i)$ denotes the set of information (e.g., Web pages) that u_i has ever viewed, and $|\mathcal{D}(u_i)|$ denotes the cardinality of $\mathcal{D}(u_i)$. Similar denotations occur for u_j. Then, we compute $\mathtt{sim}_{con}(u_i, u_j) = \frac{|\mathcal{D}(u_i) \cap \mathcal{D}(u_j)|}{|\mathcal{D}(u_i) \cup \mathcal{D}(u_j)|}$. For possible extension, we plug other similarity functions (like cosine similarity) and properties (like the popularity [3]) to measure the content similarity, without degrading the proposed scheme.

Next, we design the *<u>str</u>ucture similarity* between u_i and u_j, denoted by $\mathtt{sim}_{str}(u_i, u_j)$. Suppose that $\mathcal{N}(u_i)$ denotes the union of u_i and its neighbors (resp. $\mathcal{N}(u_j)$ denotes the union of u_j and its neighbors). We compute $\mathtt{sim}_{str}(u_i, u_j) = \frac{|\mathcal{N}(u_i) \cap \mathcal{N}(u_j)|}{|\mathcal{N}(u_i) \cup \mathcal{N}(u_j)|}$. Since u_i and u_j are neighbors, then $\mathcal{N}(u_i) \cap \mathcal{N}(u_j)$ is not null and $\mathtt{sim}_{str}(u_i, u_j) > 0$ holds. In addition, we apply other similarity functions, e.g., cosine distance, graph edit distance, etc. for $\mathtt{sim}_{str}(u_i, u_j)$.

Based on the above $\mathtt{sim}_{con}(u_i, u_j)$ and $\mathtt{sim}_{str}(u_i, u_j)$, we compute $\mathtt{sim}_{nbr}(u_i, u_j) = \alpha \cdot \mathtt{sim}_{con}(u_i, u_j) + (1-\alpha) \cdot \mathtt{sim}_{str}(u_i, u_j)$, where α and $(1-\alpha)$, within the range [0.0, 1.0], indicate positive weights of $\mathtt{sim}_{con}(u_i, u_j)$ and $\mathtt{sim}_{str}(u_i, u_j)$ over the neighbor similarity, respectively. Since both $\mathtt{sim}_{con}(u_i, u_j)$ and $\mathtt{sim}_{str}(u_i, u_j)$ are positive numbers smaller than 1.0, we then have $0.0 \leq \mathtt{sim}_{nbr}(u_i, u_j) \leq 1.0$.

2.1.2 Similarity between pair of arbitrary vertices

Let's proceed to the general case that u_i and u_j are a pair of arbitrary vertices in $\mathcal{G}$. Following [22], we note that there exist multiple paths between u_i and u_j in the graph $\mathcal{G}$. Inside each of these paths, the edge connecting two neighbors u_i and u_{i+1} has a weight $\mathtt{sim}_{nbr}(u_i, u_{i+1})$. Then, for such a path, we compute the similarity between u_i and u_j by a product of the neighbor similarity values for all edges inside the path, i.e., $\prod_{path:u_i,u_{i+1},\ldots,u_j} \mathtt{sim}_{nbr}(u_i, u_{i+1})$. Based on the definition, when u_i and u_j are far away, $\mathtt{sim}(u_i, u_j)$ is small.

For each path between u_i and u_j in $\mathcal{G}$, there is an associated similarity value. Among the similarity values associated with all paths between u_i and u_j, the *maximal* one can be treated as the *optimal similarity* $\mathtt{sim}(u_i, u_j)$, and the path associated with the optimal similarity is called the *optimal path* between u_i and u_j. Note that, when there exists an infinite loop inside the path between u_i and u_j, using the maximal value makes sense to select the path associated with fewer edges without infinite loops (surely having larger similarity). Without special mention, $\mathtt{sim}(u_i, u_j)$ by default refers to the optimal similarity between u_i and u_j.

We use Figure 1 as an example to compute $\mathtt{sim}(u_i, u_j)$. There are three paths between u_1 and u_3: (i) $u_1 - u_3$, (ii) $u_1 - u_2 - u_3$ and (iii) $u_1 - u_2 - u_4 - u_3$. The 2-nd one is the optimal path with the highest similarity 0.24, i.e., the optimal $\mathtt{sim}(u_1, u_3) = 0.24$.

We note that the above similarity definition is consistent with the previous works [3, 22, 27] which use the *shortest path distance* to measure the similarity. Many real data sets show the shortest path distance is practically useful to compute the similarity between users. For example, the analyst result [23] based on real Microsoft Messenger network shows that a user is interested in finding content from those users that are similar to him/her in the social graph. The experimental result of wikipedia search [26] indicates that the distance of a query initiation point (the query context) to relevant web-pages is an important parameter in ranking search results. All these results practically verify the effectiveness of the shortest path distance.

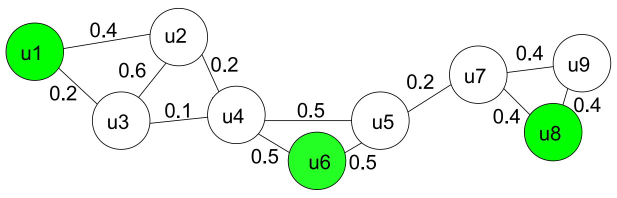


Figure 1 An example of a social graph

2.2 Problem statement

Based on the $\mathtt{sim}(u_i, u_j)$, we formally define the SAU problem to advertise a given content item d towards the top-k most desirable targeted users. We assume that the item d is associated with preference weights $weight(d, u_l)$ by I initial seed users u_l $(1 \le l \le I)$. The meaning of $weight(d, u_l)$ depends on the OSNs. For example, in the Digg.com, the value of $weight(d, u_l) = 1.0$ means that the user u_l ever digged the item d and otherwise $weight(d, u_l) = 0.0$ if u_l did not digg d. The seed users u_l and preference weights $weight(d, u_l)$ help exploring more potential users who are also interested in d. By $weight(d, u_l)$ and $\mathtt{sim}(u_l, u_i)$ for $1 \le i \ne l \le N$, the following *monotonic* score function measures the relevance score between u_i and d:

$$\textsc{score}(d, u_i) = \sum_{l=1}^{I} [weight(d, u_l) \cdot \mathtt{sim}(u_l, u_i)] \tag{1}$$

We define the SAU problem as follows. Among all users in the social network $\mathcal{G}$, we want to find the top-k desirable users with the highest $\textsc{score}(d, u_i)$. We denote $\mathcal{K}$ to be the top-k users (except those I seed users). Each user $u_i \in \mathcal{K}$ will have high potentials to be interested in the advertised content d. We focus on designing efficient algorithms for the SAU problem with low space cost and fast running time.

2.3 Review of the TA and NRA algorithms

For helping understand the proposed algorithm, we first review two classic top-k query algorithms (TA and NRA).

The top-k query algorithms assume the data is accessed by *sorted access*, *random access* and a *combination* of both. In sorted access, objects are *accessed sequentially* ordered by some scoring predicate, while for random access, objects are *directly accessed* by their identifiers. As a common assumption, the cost of random access is generally more *expensive* than that of sorted access [16]. There are two categories of top-k query algorithms in the literature: threshold algorithm (TA) and not random access (NRA) algorithm.

TA [10] scans multiple sorted lists and maintains an upper bound T for the overall score of unseen objects and a heap of overall scores for all seen objects. The upper bound T is computed by applying the scoring function to the last seen objects in different lists. The overall score of a seen object is computed by applying the scoring function to object's partial scores, obtained from different sorted lists. To obtain the overall scores, each new object in one of the lists is looked up in all other lists via the *random access*. All objects with total scores that are greater than or equal to T are reported. TA terminates after returning the k-th output.

When random access is not supported, NRA [10] exploits *sorted accesses* only. Instead of reporting the exact object scores, NRA produces the top-k answers using both the score lower bounds and upper bounds. The score lower bound of some object t is on t's known partial scores. Its score upper bound is obtained on t's known scores and the maximum possible values of t's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. When the score lower bound of t is not smaller than the score upper bounds of all other objects (including unseen objects), then t is safely reported as the next top-k object.

2.4 Baseline solution and limitations

Before presenting the proposed algorithms, we review alternative solutions by directly re-using the classic TA and NRA [10, 16] algorithms. First, any vertex user in a social graph theoretically has the possibility to become a top-k result. Therefore, we might (pre-)compute the pairwise similarity of any two vertex users before the TA and NRA [10, 16] are adopted over the sorted lists of the similarity $\mathtt{sim}(u_i, u_j)$ between any seed users u_l and every candidate user u_j. However, given a very large graph, the maintenance cost of the pairwise similarity, at the scale of $O(V^2)$, is expensive. For example, by the assumption that the pair of $\langle u_j, \mathtt{sim}(u_i, u_j)\rangle$ requires the data size of 40 bytes, the Facebook US user base alone with 103 million in 2009 has to maintain 400 petabytes space, i.e., 40 bytes $\times$ $(103 \times 10^6)^2$ $\approx 4 \times 10^{17}$ bytes. Such a dramatically huge storage cost is for the indexing structure alone, without involving any other production data. In addition, instead of maintaining the costly pre-computed pairwise similarity, the online computation of pairwise similarity, i.e., all-pairs shortest path distance problem, for example by the Dijkstra algorithm [8], simply requires the low space cost to maintenance neighbor similarity only (e.g., with the help of an adjacent list-based index). However, it incurs very high running time, at the scale of $O(E^2)$ or improved $O(V \log V + E)$.

Consequently, the offline and online approaches above incur either high storage cost or high running time. This issue is essentially caused by overly maintaining or computing unnecessary users similarity (i.e., the pairwise similarity). In the following sections, we leverage the adjacent list-based index, and compute the similarity as needed, such that the storage cost and running time is trivial.

3 NRA_SAU

The proposed algorithm essentially is a variant of NRA with the sorted lists created on the fly by incrementally running the Dijkstra algorithm. Depending on whether or not the NRA condition is met, the Dijkstra algorithm computes the similarity items if and only if needed. In this section, we first adapt the Dijkstra algorithm to create the sorted lists on the fly (Section 3.1), and use the adapted algorithm to answer the SAU problem by a variant of NRA (Section 3.2).

3.1 Adapted Dijkstra algorithm

We adapt the Dijkstra algorithm to output a vertex w together with the similarity $\mathtt{sim}(u_l, w)$ regarding to a given seed user u_l. For convenience, we call the adapted algorithm *Dijk_sorted_sim*. As a unique property, when we call the function *Dijk_sorted_sim* by k rounds, the number k of outputted vertexes w are sorted by *descending* order of the similarity $\mathtt{sim}(u_l, w)$.

We first give an observation. Given a seed user u_l and one of its neighbors, say u_i, we claim that $\mathtt{sim}(u_l, u_i)$ must be larger than $\mathtt{sim}(u_l, u_j)$, where u_j is any neighbor of u_i (except that u_j is also u_l's neighbor). It is because each similarity value is inside the range [0.0, 1.0]. Given $0.0 \le \mathtt{sim}(u_i, u_j) \le 1.0$, we then have $\mathtt{sim}(u_l, u_j) = \mathtt{sim}(u_l, u_i) \times \mathtt{sim}(u_i, u_j) \le \mathtt{sim}(u_l, u_i)$.

The basic idea of *Dijk_sorted_sim* is as follows. Suppose that in the last round, the vertex u_i is the output vertex of *Dijk_sorted_sim*; the set $\mathcal{H}$ maintains the candidate vertexes (except u_i) with larger similarity regarding to u_l than other existing vertexes. Then among all members inside both $\mathcal{H}$ and u_l's neighbors, we choose the vertex w having the largest similarity $\mathtt{sim}(u_l, w)$ as the output of $Dijk\_sorted\_sim$ in the current round. Following the above observation, the vertex w must have the largest similarity $\mathtt{sim}(u_l, w)$ among all existing vertexes (except those previous output vertexes). In this way, the output vertexes w of $Dijk\_sorted\_sim$ are sorted by descending order of the similarity $\mathtt{sim}(u_l, w)$.

Algorithm 1 Dijk_sorted_sim (seed user u_l, graph $\mathcal{G}$, heap $\mathcal{H}$, set $\mathcal{K}$)

```
1:  if $\mathcal{H}$ is null then Initiate $\mathcal{H}$, and insert all pairs $\langle \mathcal{N}_j(u_l), \mathcal{N}_j(u_l).sim \rangle$ to $\mathcal{H}$;
2:  if $\mathcal{H}$ is empty then return null;
3:  pop the head item $w$ of $\mathcal{H}$, and add $w$ to $\mathcal{K}$;
4:  for all items $w'$ in $\mathcal{N}(w)$ do
5:      if $w'$ is not in the heap $\mathcal{H}$ and not in $\mathcal{K}$ then
6:          Insert $\langle w', \mathtt{sim}(u_l, w) \times \mathtt{sim}_{nbr}(w, w') \rangle$ into $\mathcal{H}$;
7:      else if $\mathtt{sim}(u_l, w) \times \mathtt{sim}_{nbr}(w, w') > \mathtt{sim}_{inh}(u_l, w')$ then
8:          $\mathtt{sim}_{inh}(u_l, w') = \mathtt{sim}(u_l, w) \times \mathtt{sim}_{nbr}(w, w')$;
9:      end if
10: end for
11: return $w$;
```

For the input parameters of $Dijk\_sorted\_sim$ in Algorithm 1, besides the given seed user u_l and the social graph $\mathcal{G}$, we use the parameter $\mathcal{H}$ (i.e., a maximal heap structure) to maintain the candidate output vertexes, and the parameter $\mathcal{S}$ as the set to record the previously outputted vertexes. Inside the algorithm, if $\mathcal{H}$ is null (i.e., we run $Dijk\_sorted\_sim$ in the first time), then the output vertex must be inside n_l's neighbors and we add such neighbors to $\mathcal{H}$. After that, among all members inside $\mathcal{H}$, the head item with the largest similarity is the output vertex w. After that, the remaining vertexes inside $\mathcal{H}$ and the new neighbors w' of the output vertex w are together merged into $\mathcal{H}$ as the candidate output vertexes for the next round. In this way, we simply return the head item $\mathcal{H}$ as the output of the next round.

During the above merge, we consider the following cases:

(i) *w' is not processed previously*: Then lines 5-6 directly add the neighbor vertex w' to $\mathcal{H}$ together with the similarity $\mathtt{sim}(u_l, w) \times \mathtt{sim}_{nbr}(w, w')$. Note that such a similarity may not be the (non-optimal) similarity associated with a path between u_l and w' which intermediately goes through w. The short name inh indicates $\mathtt{sim}_{inh}(r, w')$ is the similarity value maintained $\underline{in}$side the $\underline{h}$eap $\mathcal{H}$.

(ii) *w' has been already added to $\mathcal{H}$*: Then, lines 7-8 update $\mathtt{sim}_{inh}(r, w')$, if the new similarity between r and w', associated with a new path between r and w' intermediately going through w, is larger than the current $\mathtt{sim}_{inh}(r, w')$. The intuition is that among multiple paths between r and w', the one with the largest similarity is optimal.

(iii) *w' has already returned from $\mathcal{H}$, and added to $\mathcal{K}$*: It is unnecessary to handle this case.

Now we can verify that the complexity of Algorithm 1 is $O(N_w)$ where N_w is the number of the output vertex w's neighbors.

3.2 Answering SAU

We use a variant of the NRA algorithm, NRA_SAU in Algorithm 2, to answer the SAU problem. Given I seed users u_l, NRA_SAU calls the above $Dijk\_sorted\_sim$ as needed to create I sorted lists of similarities with respect to the seed users. With no random access, we leverage the sorted lists to approximate the lower bound $\text{SCORE}_{lb}(d, u_i)$ and upper bound $\text{SCORE}_{ub}(d, u_i)$, respectively (we will give the approximation in the final part of this section). After that, if the NRA stopping condition, i.e., the k-th largest lower bound of the candidates for final results is smaller than the upper bound of any non-candidate, we then return the candidates with the top-k largest lower bounds as the final results.

Algorithm 2 NRA_SAU(I raters r_l with $1 \leq l \leq I$, graph $\mathcal{G}$, number k)

```
1: Initiate $I$ heaps $\mathcal{H}_l$ for seed users $r_l$, and the top-$k$ candidate heap $\mathcal{K}$;
2: For each $u_l$, insert all items of $\mathcal{N}(r_l)$ to the heap $\mathcal{H}_l$;
3: **while** $MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i)) < MAX_{\forall u_i \in \mathcal{G}\setminus\mathcal{K}}(\text{SCORE}_{ub}(d, u_i))$ **do**
4:     $w \leftarrow$ the head item with the max. score over $I$ heaps $\mathcal{H}_l$;
5:     call $Dijk\_sorted\_sim(u_{lw}, \mathcal{G}, \mathcal{H}_{lw}, \mathcal{K})$
6: **end while**
7: **return** the top-$k$ items with the largest lower bounds in $\mathcal{K}$ as is;
```

The pseudo-code of NRA_SAU is given by Algorithm 2. Line 1 uses $\mathcal{H}_l$ (a maximal heap structure) to represent the sorted list with respect to each seed user u_l, and $\mathcal{K}$ to represent the set of the candidates for final results. The NRA stopping condition is shown in line 3. Once the stopping condition is not met, among the heap items in the I sorted lists (i.e. the heaps $\mathcal{H}_l$), line 4 chooses the item w having the largest value $sim(u_{lw}, w) \times weight(u_{lw}, d)$. For such an item w, we denote u_{lw} and $\mathcal{H}_{lw}$ to be the associated seed user and the heap with such w, respectively. After that, in line 5, the function $Dijk\_sorted\_sim(u_{lw}, \mathcal{G}, \mathcal{H}_{lw}, \mathcal{K})$ adds the output vertex (i.e., the head item w) to the set of candidates $\mathcal{K}$, and updates the existing sorted list $\mathcal{H}_{lw}$. Since the output vertexes $Dijk\_sorted\_sim$ are sorted by descending order of the similarities, the approximated upper bound becomes gradually smaller and the k-th largest lower bound becomes larger. As a result, when $Dijk\_sorted\_sim$ outputs vertexes as needed, the stopping condition will be met finally.

Finally we set the lower bound $\text{SCORE}_{lb}(d, w)$ and upper bound $\text{SCORE}_{ub}(d, w)$ for the output vertex w in line 4 as follows. Though we might set $\text{SCORE}_{lb}(d, w) = \mathtt{sim}(u_{lw}, w) \times weight(d, u_{lw})$, this lower bound is too loose. For improvement, we observe that w, though chosen from $\mathcal{H}_{lw}$, may appear inside another heap $\mathcal{H}_{lw'}$ ($\neq \mathcal{H}_{lw}$) associated with the seed user lw'. In this case, we set a tighter lower bound $\text{SCORE}_{lb}(d, w)= \mathtt{sim}(u_{lw}, w) \times weight(d, u_{lw}) + \mathtt{sim}_{inh}(u_{lw'}, w) \times weight(d, u_{lw'})$. Next, to set $\text{SCORE}_{ub}(d, w)$, when $\mathtt{sim}(u_{lw'}, w)$ is missed, we replace the missed $\mathtt{sim}(u_{lw'}, w)$ by $\mathtt{sim}(u_{lw'}, u_{l'w'})$, where $u_{l'w'}$ is the head item of $\mathcal{H}_{lw'}$. Thus, we set the upper bound $\text{SCORE}_{ub}(d, w)$ based on the replaced $\mathtt{sim}(u_{lw'}, u_{l'w'})$.

4 BAT_SAU

Though the NRA_SAU algorithm works correctly to answer the SAU problem, there exist potentials for improvement. In this section, we first introduce two ideas to overcome the issues of NRA_SAU (Sections 4.1 and 4.2), then present a BAT_SAU algorithm (Section 4.3) and finally give a running example (Section 4.4).

4.1 Pruning unnecessary candidates

Recall that when NRA_SAU calls $Dijk\_sorted\_sim$, the lines 4-6 in $Dijk\_sorted\_sim$ process all neighbors w' of w and add them, if not appearing, to $\mathcal{H}_w$. Nevertheless, in case that w' is not a top-k result, adding w' to $\mathcal{H}_w$ is useless and incurs more cost.

To overcome the above shortcoming, we expect that the neighbor w' at least has the possibility to become the final top-k result. If w' has no possibility to become a top-k result, we will not add it to $\mathcal{H}_w$. Thus, before adding w' to $\mathcal{H}_w$, we ensure that the criteria $\text{SCORE}_{ub}(d, w') \geq MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$ holds, i.e., $\text{SCORE}_{ub}(d, w')$ is at least larger than the k-th largest lower bound $\text{SCORE}_{lb}(d, u_i)$ among all candidates $u_i \in \mathcal{K}$. This criteria surely prunes those neighbors that are not the top-k results, which is proven as follows.

By contradiction, we assume that there exists a vertex with $\text{SCORE}_{ub}(d, w') < MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$. Then we infer that for the lower bound $\text{SCORE}_{lb}(d, w')$, the following statement

$$\text{SCORE}_{lb}(d, w') \leq \text{SCORE}_{ub}(d, w') < MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$$

holds. Following the stopping condition of Algorithm 2, we verify that a real top-k result u_i must satisfy $MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i)) > MAX_{\forall u_i \in \mathcal{G} \setminus \mathcal{K}}(\text{SCORE}_{ub}(d, u_i))$. Thus, for any vertex w', if the condition $\text{SCORE}_{ub}(d, w') < MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$ holds, the vertex w' is not a top-k result, and we safely prune it.

4.2 Multiple phases of batch process

NRA_SAU essentially is a sequential approach: each iteration of the `while` loop fetches an item w from $\mathcal{H}_w$ and invokes an update on $\mathcal{K}$. For N iterations, we then have N items w and N sequential updates on $\mathcal{K}$. Since each update involves the cost $O(log|\mathcal{K}|)$, the overall cost for a large N is corresponding non-trivial.

To this end, we propose a batch approach to reduce the update cost, by fetching multiple items as a batch from each heap $\mathcal{H}_w$. Now suppose we have fetched N batches of items fetched from all of I heaps $\mathcal{H}_w$, and we propose two following techniques to optimize the items which are then used to update $\mathcal{K}$.

First, suppose that an item w appears in multiple, say two, batches. We next merge the two batches of items together and thus reduce the number updates on $\mathcal{K}$.

Secondly, since the output vertexes w of $Dijk\_sorted\_sim$ are sorted by descending order of $sim(u_w, w)$, among a batch of items fetched from $\mathcal{H}_l$, the last one must be associated with the *smallest* similarity inside the batch. Therefore, we utilize the smallest similarity to set the tightest bounds of $\text{SCORE}_{lb}(d, u_i)$ and $\text{SCORE}_{ub}(d, u_i)$.

Algorithm 3 BAT_SAU(I seed users u_l, graph $\mathcal{G}$, number k)

```
1:  The same as lines 1-2 of Algorithm 2;
2:  while $MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i)) < MAX_{\forall u_i \in \mathcal{G} \setminus \mathcal{K}}(\text{SCORE}_{ub}(d, u_i))$ do
3:      if this is the first phase then
4:          Fetch the top-$k$ items $w$ from each $\mathcal{H}_l$;
5:      else
6:          $T_1 \leftarrow MAX_{\forall u_i \in \mathcal{G} \setminus \mathcal{K}}(\text{SCORE}_{ub}(d, u_i)) - MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$;
7:          Fetch the items $w$ with $\mathtt{sim}(u_l, w) \geq \frac{T1}{I \times weight(u_l, d)}$ from each $\mathcal{H}_l$;
8:      end if
9:      for each unique fetched item $w$ do
10:         update $\mathcal{K}$ by the item $w$;
11:         lines 4-10 of $Dijk\_sorted\_sim(u_w, \mathcal{G}, \mathcal{H}_w, \mathcal{K})$
12:     end for
13: end while
14: return the top-$k$ items with the largest lower bounds in $\mathcal{K}$ as is;
```

4.3 Algorithm details

The proposed BAT_SAU (Algorithm 3) uses the above techniques to improve NRA_SAU. The key of this algorithm is how to set the size of a batch. We use two phases of fetches, and each phase adaptively tunes the size of the batch.

Phase 1 (lines 3-4): In the first phase, we fetch a batch of the top-k items u_i from each heap $\mathcal{H}_l$ (with respect to the seed user u_l). Next, in lines 9-12, the total $k \times I$ items u_i are fetched from the I heaps $\mathcal{H}_l$, and update the heap $\mathcal{K}$ and $\mathcal{H}_l$. After that, before continuing the next iteration of the **while** loop, we validate whether or not the NRA stopping condition is satisfied. If the stopping condition is satisfied, the NRA_SAU algorithm directly returns the final top-k results. Otherwise, the following *phase 2* continues.

Phase 2 (lines 6-7): This phase first sets a threshold T_1 by the gap between the two items in the stopping condition, i.e., $T_1 = MAX_{\forall u_i \in \mathcal{G} \setminus \mathcal{K}}(\text{SCORE}_{ub}(d, u_i)) - MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$. Using the gap consistently follows the claim in Section 4.1, such that only those necessary neighbors, if and only if needed, are considered as the candidates. By the threshold T_1, we use T_1/I as the criteria to select more items as a batch from each heap $\mathcal{H}_l$. Using the division T_1/I is because each of the I seed users u_l contributes to $\text{SCORE}(d, u_i)$. Based on the selection criteria T_1/I, we then select those items u_i from $\mathcal{H}_l$ with $\mathtt{sim}(u_l, u_i) \geq T_1/(I \times weight(d, u_l))$. After all such items over the I heaps are fetched, we next update the heap $\mathcal{K}$, and renew the top-k stopping condition. If the renewed top-k stopping condition is still dissatisfied, we reset T_1 as before and again fetch more batches of items. This phase continues until the top-k stopping condition is satisfied and the final top-k results are found.

Note that during the update of the heap $\mathcal{H}$ in line 10, we use the last fetched item inside a batch to set the lower and upper bounds $\text{SCORE}_{lb}(d, w)$ and $\text{SCORE}_{ub}(d, w)$. This ensures the top-k stopping condition is satisfied as early as possible.

4.4 A running example

By using Figure 1 (having the three seed users u_1, u_6 and u_8) as the example social graph, we illustrate how BAT_SAU is able to find the top-3 users. We assume $weight(d, u_l) = 1.0$ for each seed user.

In Figure 2a, for the seed users u_1, u_6 and u_8, the associated column (i.e., the heap pertaining to the seed user) contains the pairs of candidates and the similarity values between the seed user and the candidate users. The pairs are sorted by the similarity values. For example, in the column of u_1, the first pair $\langle u_2, 0.4\rangle$ indicates the similarity value 0.4 between the candidate user u_2 and the seed user u_1.

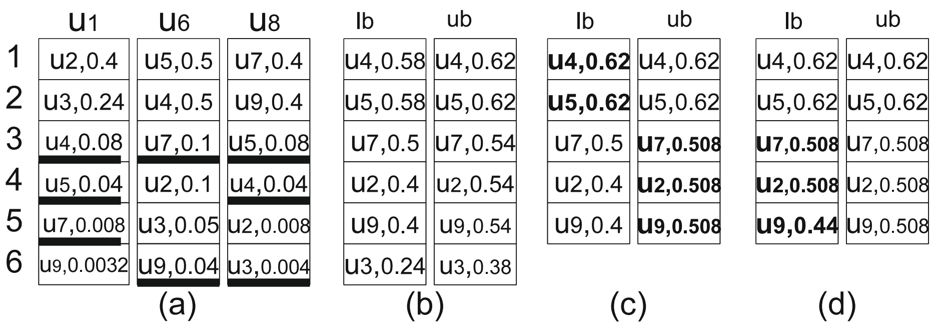
Following BAT_SAU, the Phase 1 first fetches the top-3 items (above position 3) pertaining to each seed user. Given the three seed users u_1, u_6 and u_8, we fetch 9 items but with only six distinct users u_2, u_3, u_4, u_5, u_7 and u_9. Thus we use 6 insertion operations, instead of 9 operations to update the heap $\mathcal{K}$ by together with their bounds $\text{SCORE}_{lb}(d, u_i)$ and $\text{SCORE}_{ub}(d, u_i)$ given by Figure 2b. For example, the lower bound of relevance score for u_4 is 0.58 ($= 0.08 + 0.5$), and the upper bound is 0.62 ($= 0.08 + 0.5 + 0.04$). Next, due to $\text{SCORE}_{ub}(d, u_3) = 0.38 < MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i)) = 0.5$, we discard u_3 because it cannot be a top-3 result.

Next, Phase 2 sets $T_1 = MAX_{\forall u_i \in \mathcal{G}\setminus\mathcal{K}}(\text{SCORE}_{ub}(d, u_i)) - MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i))$ $= 0.54 - 0.5 = 0.04$. Thus, we fetch the candidates with similarity larger than $T_1/3 = 0.04/3 = 0.013$. Next, as an example, the vertex u_5 at position 4 of the heap $\mathcal{H}_1$ is fetched. After that, $\mathcal{K}$ is updated with new lower bounds and three upper bounds (highlighted by the bold font of Figure 2c).

Then, Phase 2 again updates $T_1 = 0.508 - 0.5 = 0.008$, and fetches the candidates with similarity larger than $T_1/3 = 0.008/3 = 0.0023$. For example, in the heap $\mathcal{H}_1$, the user u_7 at position 5 is fetched. Then, in Figure 2d, the heap $\mathcal{K}$ is updated with three new lower bounds, and the stopping condition $MIN_{\forall u_i \in \mathcal{K}}(\text{SCORE}_{lb}(d, u_i)) = MAX_{\forall u_i \in \mathcal{G}\setminus\mathcal{K}}(\text{SCORE}_{ub}(d, u_i))$ $= 0.508$ holds. The top-3 results u_4, u_5 and u_7 are returned.

5 Experimental evaluation

We implement the main memory version of NRA_SAU and BAT_SAU in Java 1.6.0, and run the experiments on a Debian Linux server of 2.6.22 version with a 3.00GHz Intel Xeon CPU and 16 GB of RAM.



(a)

	u1	u6	u8
1	u2,0.4	u5,0.5	u7,0.4
2	u3,0.24	u4,0.5	u9,0.4
3	u4,0.08	u7,0.1	u5,0.08
4	u5,0.04	u2,0.1	u4,0.04
5	u7,0.008	u3,0.05	u2,0.008
6	u9,0.0032	u9,0.04	u3,0.004

(b)

lb	ub
u4,0.58	u4,0.62
u5,0.58	u5,0.62
u7,0.5	u7,0.54
u2,0.4	u2,0.54
u9,0.4	u9,0.54
u3,0.24	u3,0.38

(c)

lb	ub
u4,0.62	u4,0.62
u5,0.62	u5,0.62
u7,0.5	**u7,0.508**
u2,0.4	**u2,0.508**
u9,0.4	**u9,0.508**

(d)

lb	ub
u4,0.62	u4,0.62
u5,0.62	u5,0.62
u7,0.508	u7,0.508
u2,0.508	u2,0.508
u9,0.44	u9,0.508

Figure 2 Running example

The data sets include the real Digg trace [34] and synthetic social graphs with Eppstein Power Law distribution [9]. With the real data, we prove both effectiveness and efficiency of our algorithms. In addition, synthetic data helps better support the efficiency.

(i) The real Digg Secondary Trace (ST): In the Digg, users and their friends and fans form the Digg social network. When users submit stories, the stories are first placed on the upcoming stories section. Other users next browse recently submitted stories on the upcoming stories section and digg what they like the best. We call the event that a user digged a story as a *digg event*. Zhu [34] leveraged the Digg APIs to crawl the Digg social network. For each user in the crawled social network, the associated digg events by such a user were also fetched. The ST trace contains 580,228 users and 4,569,331 Digg events from March 17, 2009 to April 16, 2009. Based on the Digg trace log, we compute both the content similarity and the structure similarity. In terms of the content similarity, we sort the Digg events by the associated timestamps, and use the early Digg events (e.g., with a rate of 0.001 of the whole Digg events) as the history information to compute the content similarity $\mathtt{sim}_{con}(u_i, u_j)$.

For the storage cost to store the Digg ST trace, the adjacent list used by the SAU algorithms (including NRA_SAU and BAT_SAU) needs only 156.7 MB. Yet, the storage cost to maintain the similarity of all pairs of users (used by the offline NRA algorithm) needs around 7746.5 GB. Note that the Digg ST trace contains only a subset of the whole users in the Digg system. Given real users in the Digg system, the storage cost incurred by maintaining the similarity of all pairs users, at scale of V^2, is significantly large.

(ii) We generate synthetic graphs by the known open source graph project JUNG V2 [19]. We use two parameters V (the total number of vertices) and e (the average number of friends per vertex) to generate Power Law graphs. With only the generated graphs, we have to compute the similarity $\mathtt{sim}_{nbr}(u, v)$ only by the structure similarity $\mathtt{sim}_{str}(u, v)$.

5.1 Effectiveness

We first demonstrate that the SAU scheme itself is indeed useful before we show its efficiency. We use the Digg trace to evaluate the effectiveness of SAU and compare it with two approaches: (i) the adaptation of the collaborative filtering (CF) approach [14] to our solution framework, and (ii) the social-aware advertising (ADV) [32]. For the adapted CF approach, we only change our algorithms by setting the content weight $\alpha = 1.0$ and thus $\mathtt{sim}_{nbr}(u, v) = \mathtt{sim}_{con}(u, v)$. In this way, based on the similarity between users to digg the stories, the adapted CF finds the top-k users sharing the most similar taste with such seed users. For the ADV, we create cohesive subgraphs and then estimate the liking of the subgraph members to be interested in the advertised content. Note that, unlike ADV, there is no product category information in the SAU problem and we do not consider the category information of all digg stories.

We compare the CF and ADV with SAU in terms of effectiveness. In detail, we first consider the precision. For a given item d, we want to find the top-k users who will dig it. Based on the Digg trace, for each digg event d, we can find all users who truly digged it. We denote such users, as the ground truth, by $\mathcal{U}$. Among the set of k users returned by CF, ADV, and/or SAU (denoted by $\mathcal{K}$), not all of them are inside $\mathcal{U}$. Assume that the subset of users $\mathcal{R} \subseteq \mathcal{K}$ appear in $\mathcal{U}$. We define *precision* as the number of returned interested users over k, that is $|\mathcal{R}|/k$.

Besides the precision, another metric is recall. Because the SAU problem returns the fixed number of k users anyway, we might compute the recall by $|\mathcal{R}|/k$. Nevertheless, in our context of top-k processing which always returns k users, the only metric that does make sense is the precision at a level k of recall (named, P at k).

In this experiment, we use four parameters: the rate of history Digg events used to compute $\texttt{sim}_{con}(u, v)$, the number I of seed users, the top-k number, and the weight α of content similarity. By default, these parameters are set as: history rate = 0.001, I = 100, top-k = 100, and $\alpha = 0.5$.

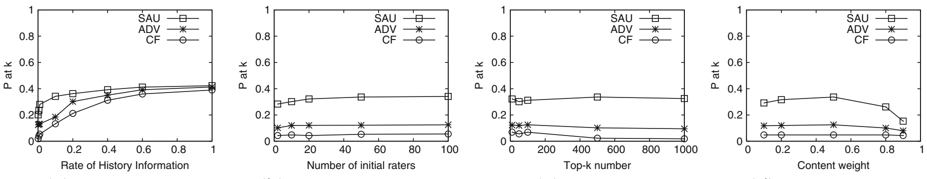
First, we study the effect of history rate. As shown in Figure 3a, a higher rate of history Digg events leads to higher precision for three schemes. The SAU approach has a highest precision because it considers both the structure similarit and content similarity. For the CF approach, it computes the content similarity only based on old history information, and cannot guarantee a very high precision in term of new posted items. Moreover, the ADV algorithm does not consider the weight information of social graphs and cannot achieve the highest precision as the SAU approach.

Second, in Figure 3b, we change the number I of seed users from 2 to 100. When I increases, the precision and recall of three schemes are higher. It is obvious because more seed users indicate the view point of the majority of users in the social network. The reason that SAU outperforms the ADV and CF schemes is due to the low default history rate (= 0.001). As a result, a very little history information is available to the CF scheme, which then cannot precisely compute the similarity of users.

Next, Figure 3c shows the effect of the top-k number. A larger top-k number leads to lower precision and recall of CF, yet the precision and recall of SAU and ADV are relatively stable. It is because the structure similarity is independent on the top-k number, and the precision and recall of SAU remain relatively stable. The ADV still leverages the social graph to discover cohesive subgraphs, also independent of the top-k number. Instead, the CF does not leverage the structure similarity, and a larger top-k cannot guarantee a larger $|\mathcal{R}|$. Since the precision and recall, computed by $|\mathcal{R}|/k$, are reverse to the value of k, a larger k leads to lower precision and recall.

Finally, Figure 3d plots the effect of the content weight α. As shown in this figure, neither the smallest content weight $\alpha = 0.1$ nor the largest one $\alpha = 0.9$ leads to the highest precision and recall of SAU. Instead, for $\alpha \in [0.5 - 0.8]$, the associated precision and recall have the largest value. It means that neither using the content similarity nor structure similarity alone can achieve the best effectiveness, and therefore it is necessary to combine both of them.

We note that the precision in Figure 3 is not very high, less than 0.4. The main reason is that the trace does not record the users who only viewed, not voted, the stories in Digg. Nevertheless, our above results, in particular the Figure 3d, show the benefits of the work to consider both the structure and content similarity.



(a) History Rate (b) Number of Seed Users (c) Top-k Number (d) Content Weight

Figure 3 Effectiveness (P: precision at k)

5.2 Efficiency

After demonstrating the effectiveness of SAU, based on both real Digg data set and the synthetic data set, we now focus on the efficiency of the proposed algorithms. We measure the efficiency by the average running time per content.

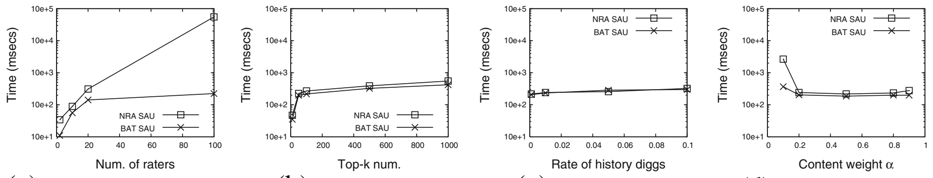
We compare the two SAU algorithms with the online NRA, the adapted CF and the ADV approaches. For the online NRA scheme (i.e., the baseline solution presented in Section 2), we first use the Dijkstra algorithm (implemented by JUNG v2) to compute the similarity between each rater and all other $(V-1)$ vertices. After that, the computed similarity values are maintained as a sorted list. Given I raters, the Dijkstra algorithm is executed with I times to maintain I sorted lists. Based on I sorted lists, the NRA algorithm next solves the top-k problem.

5.2.1 Experimental results on Digg trace

First Figure 4a shows the effect of the number I of raters. When I varies from 2 to 200, the running time of the two SAU algorithms is increased. We note that in the relevance score function, all I raters contribute to the relevance score $\text{SCORE}(d, u_i)$. Thus, we intuitively treat that the computation of $\text{SCORE}(d, u_i)$ involves a dimensionality of I, and a larger I leads to more running time. Since BAT_SAU carefully selects those candidates (by following the techniques in Sections 4.1 and 4.2) to make the stopping condition in line 2 quickly satisfied, the associated time of BAT_SAU becomes much smaller than the one of MRA_SAU.

In addition, for the online NRA algorithm, for example, given $I = 200$, it uses 8289 folds and 102 folds of the time used by BAT_SAU and NRA_SAU. The adapted CF approach is essentially the SAU algorithm only with the weight $\alpha = 1.0$, and uses the almost same time as the SAU algorithm. The ADV approach needs to solve the fractional 0-1 knapsack problem involving the whole social network and incurs 1513 folds and 18 folds of the time by BAT_SAU and NRA_SAU. The SAU algorithms achieve the obviously better result than the online NRA and ADV algorithms, and the almost same results to the adapted CF approach, and we then do not plot the results of the online NRA algorithm and adapted CF approach in the figures.

Second, we study the effect of the top-k number in Figure 4b. When k grows from 10 to 1000, the running time of three algorithms is all slightly increased. It is because (i) the running time of the SAU algorithms is dominated by the time of maintaining the sorted top-k candidates (instead of finding the real top-k results when the top-k candidates are ready) in an online manner, and (ii) k is significantly smaller than the number of maintained top-k candidates.



(a) Number I of initial raters (b) Top-k number (c) History Rate h (d) Content weight α

Figure 4 Efficiency study on Digg trace

Next, in Figure 4c, we vary the rate h of history Digg events to compute the content similarity. In this figure, a larger h leads to slower running time of the both SAU algorithms (with 1.81 and 1.95 folds of growth). It is because given a larger h, i.e., more history information, more users, who ever voted Digg events, appear in the history information. With more such users, they are more diversely distributed in the social graph. Therefore, the real top-k users are more possibly scattered in the social graph, and the SAU algorithms need more traversals inside the graph in order to find the real top-k users. This leads to more running time.

Finally, Figure 4d studies the effect of the weight α of content similarity. A larger α leads to less running time of both SAU algorithms. It is because a larger α means less importance of the structure similarity, and thus such real top-k users are more centralized around the initial raters. This next leads to less traversals inside the social graph and less running time of the SAU algorithms.

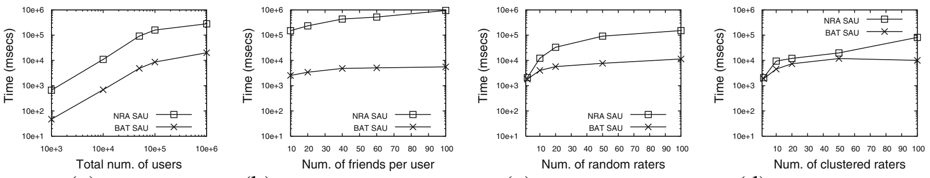
5.2.2 Experimental results on generated graphs

In this section, we study the efficiency of two SAU algorithms over generated graphs. The used parameters include the number of vertices V, average number of friends per vertex, and distribution of raters (i.e., clustered or random).

We study the effect of graph structure by varying the number of vertices V and the average number of neighbors e, and plot the experimental results in Figure 5. First, when V grows, the running time of the two SAU algorithms. It is because more vertices require more cost to select the optimal path and then to compute the optimal similarity. Second, we study the effect of average number of neighbor per vertex, e, by varying e from 10 to 100. In Figure 5b, with the growth of e, the running time of two NRA_SAU and BAT_SAU algorithms grows by around 6 and 2 folds, respectively. Thus, more neighbors e lead to exploring more candidate paths until the optimal path to compute $\mathtt{sim}(u, v)$ is found, thus resulting in slower running time.

To study the effect of raters, we consider whether the initial raters are clustered or randomly distributed inside the generated graph. Given the random manner, we randomly pick I vertices from the generated graph as the I initial raters; given the clustering manner, we pick I initial raters with Gaussian hops inside the generated graph. The clustered raters are thus distributed with bias inside the graph. In this experiment, we vary the number I of raters, and measure the running time of three algorithms.

Figure 5c–d plot the running time of the SAU algorithms, where I raters are random and clustered inside social graphs, respectively. In both figures, when I grows from 2 to 100, the running time of the two SAU algorithms is increased, consistent with the results from the Digg data set.



(a) Num.of vertices (b) Avg. num. of friends per vertex (c) Num. of random raters (d) Num. of clustered raters

Figure 5 Efficiency study of synthetic data: structure of graphs

In addition, when we compare Figure 5c with Figure 5d, we find that given the clustered raters, the two SAU algorithms in Figure 5d use less running time than the ones in Figure 5c. For example, we consider $I = 100$; given the NRA_SAU algorithm, the running time over the clustered raters is around 0.465 times of the running time over the random raters; with respect to the BAT_SAU algorithm, the running time over the clustered raters is around 0.622 times of the running time over the random raters. This is because given the clustered raters, the top-k candidates, located at the vertices similar to the raters, are associated with high relevance scores, and the two SAU algorithms select fewer candidates. Thus, clustered raters lead to less running time.

5.3 Discussion

In terms of the effectiveness, the SAU approach combines the benefits of the social similarity and content similarity, and achieves higher precision (and recall). Instead adapted CF approach considers only the content similarity, and the ADV approach leverages the social graphs but without structure similarity.

In terms of the efficiency, the running time of the SAU algorithm is dominated by seeking and maintaining top-k candidates in the sorted lists. The BAT_SAU algorithm uses the batch process to avoid costly updates of the candidates maintained inside heaps, and carefully selects the candidates based on the gap T_1. Instead, the classic TPUT algorithm [6] uses the lower bound of the top-k candidates as the threshold to select candidates. TPUT, using the lower bound of the top-k candidates, optimizes the network traffics and latency by three phases of selecting candidates. In particular, the 3rd phases of TPUT can use the random access to find the missed values during the 2nd phase. However, the three phases in TPUT by using the threshold of the top-k candidate are inapplicable to our problem since the random access is not allowed in our case. Moreover, the running example as shown in Figure 2b directly indicates that using the threshold 0.7 of the top-k candidates (u_4, u_5 and u_7) does not at all select any candidates. In addition, in order to practically improve the efficiency, it might be possible to set the limit of the top-k number, for example 1000. In this case, for every user u_l in the graph, we pre-compute the top-k (=1000) largest similarity $sim(u_l, w)$ between a candidate w and each user u_l with the space cost linear to the top-k number. The pre-computation equivalently saves the efforts required for the *Phase 1* of line 4 in Algorithm 3. After that, we can similarly repeat the remaining steps in Algorithm 3 until the final top-k results are found.

Finally, given a directed graph $\mathcal{G}$, we easily extend the proposed algorithms as follow. Following the classic Dijkstra algorithm on directed graphs, we choose only those candidate nodes inside the paths outgoing from raters. After that, we find more candidates to ensure the top-k stopping condition holds, until the final top-k results are found.

6 Related work

Targeted information adverting enables advertisers to target users based on user information such as personal interests, profile information and online behaviours. The previous works [17, 21, 28, 30] used the boolean subscriptions to model the user information and studied the efficiency of the algorithm to match the advertised information against the subscriptions. Bartolini et al., Herlocker, Yang and Dia and Yang et al. [4, 13, 31, 32] analyzed the topology structure of OSNs and leveraged only limited social relations of users (e.g., only the topology of the social networks). Unlike such solutions, we consider the relations of users in OSNs, define the relevance score function between the user information and advertised content, and define efficient algorithms to answer the proposed SAU problem.

Different from the targeted information advertising, recommendation systems select a set of meaningful content to attract a specific user. Based on the content-based filtering [11] and collaborative filtering [5, 14] techniques, the recommendation systems exploit the content similarity and use similarity to improve the recommendation quality. Next, the recent social-aware content query [3, 22] enhanced the content search with the help of social graph models. Yahoo! advertising targeting options [29] developed techniques to provide accurate and trustaware recommendations for social network users.

In terms of top-k query processing, it is a fundamental operation in modern information retrieval (IR) and database systems. The classic top-k algorithms [16] frequently assume the availability of sorted lists of partial scores with respect of each dimension and the missed scores are accessed via either sorted accessed or random accessed or controlled random probes. In the context of SAU, the sorted lists are unavailable and the proposed algorithms therefore create the sorted lists on the fly. Similar to BAT_SAU, the previous work [24] first used the technique of batching of list accesses in combination with candidate pruning only after each batch, and [6, 18] adopted multiple phases of candidates fetch. Nevertheless, [24] did not give specific solutions to set the batch size, whereas [6, 18] assumed that the random access fetched the missed scores.

Finally, we investigate algorithms that speedup the Dijkstra algorithm for the shortest path [2, 12, 15, 25]. Amer-Yahia et al. [2] gave an efficient implementation of the Dijkstra algorithm by using the radix heap structure, and gave a time bound of $O(m + n \log C)$. However, it requires the edge is associated with a nonnegative integer cost bounded by C. Ukkonen [25] presented a deterministic linear time and linear space algorithm. However, it is specially designed for the undirected single source shortest paths problem with *positive integer* weights. Differing from the assumption in [2, 25], the similarity of two neighbors in our problem is a numeric value between [0.0, 1.0], instead of an integer, and thus the algorithms given by [2, 25] are inapplicable. In addition, our work is only interested in the top-k vertices v in the graph with the highest similarity $\mathtt{sim}(u, v)$ for a given vertex u. Instead, [12, 15] computed the shortest distance (i.e., the similarity) of all pairs of vertices. Similar to the original Dijkstra algorithm, they overly computed the shortest distances of many unnecessary pairs of vertices.

7 Conclusion and future work

This paper studies the SAU problem to find the top-k desirable targeted users. To design efficient SAU algorithms, we propose to store the similarity only for neighbor users with low space cost, select the desirable candidate users with low overhead, and prune useless candidates for fast running time. To evaluate the proposed algorithm, we use the real Digg trace and synthetic data set for experiments. The experiment results validate that the SAU (i) effectively achieves higher quality than the existing solutions CF and ADV, and (ii) uses significantly less running time than state-of-the-art.

As future work, we consider new potential applications in recently popular Crowdsourcing systems, for example, finding the most appropriate contributor to answer a specific question [7], such as question in Yahoo! Answers. In addition, though the index (i.e., adjacent list) used by the proposed algorithms leads to low space cost, it is still possible that the adjacent list of a nowadays massive graph is out of the capacity of main memory of a standard server. We plan to solve the SAU top-*k* problem upon the on-going work SAKY-OMI [20, 33], a very efficient SSD-based general graph analytical system, which can use the exposed APIs to implement various graph analytical algorithms including PageRank, shortest path distance, connected comments, etc.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Trans. Knowl. Data Eng. **17**(6), 734–749 (2005)
2. Ahuja, R.K., Mehlhorn, K., Orlin, J.B., Tarjan, R.E.: Faster algorithms for the shortest path problem. J. ACM **37**(2), 213–223 (1990)
3. Amer-Yahia, S., Benedikt, M., Lakshmanan, L.V.S., Stoyanovich, J.: Efficient network aware search in collaborative tagging sites. PVLDB **1**(1), 710–721 (2008)
4. Atazky, R., Barone, E.: Advertising and Incentives Over a Social Network, Aug. 30. US Patent App. 11/512,595 (2006)
5. Bartolini, I., Zhang, Z., Papadias, D.: Collaborative filtering with personalized skylines. IEEE Trans. Knowl. Data Eng. **23**(2), 190–203 (2011)
6. Cao, P., Wang, Z.: Efficient top-k query calculation in distributed networks. In: PODC, pp. 206–215 (2004)
7. Chiang, M.-F., Peng, W.-C., Yu, P.S.: Exploring latent browsing graph for question answering recommendation. World Wide Web **15**(5–6), 603–630 (2012)
8. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
9. Eppstein, D., Wang, J.Y.: A steady state model for graph power laws. In: 2nd International Workshop Web Dynamics (2002)
10. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. **66**(4), 614–656 (2003)
11. Ferman, A.M., Errico, J.H., van Beek, P., Sezan, M.I.: Content-based filtering and personalization using structured metadata. In: JCDL, p. 393 (2002)
12. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: SODA, pp. 156–165 (2005)
13. Hadija, Z., Barnes, S.B., Hair, N.: Why we ignore social networking advertising. Qual. Mark. Res. Int. J. **15**(1), 19–32 (2012)
14. Herlocker, J.L., Konstan, J.A., Borchers, A., Riedl, J.: An algorithmic framework for performing collaborative filtering. In: SIGIR, pp. 230–237 (1999)
15. Holzer, M., Schulz, F., Wagner, D.: Engineering multilevel overlay graphs for shortest-path queries. In: ACM Journal of Experimental Algorithmics, p. 13 (2008)
16. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top- query processing techniques in relational database systems. ACM Comput. Surv. **40**(4) (2008)
17. Machanavajjhala, A., Vee, E., Garofalakis, M.N., Shanmugasundaram, J.: Scalable ranked publish/subscribe. PVLDB **1**(1), 451–462 (2008)
18. Michel, S., Triantafillou, P., Weikum, G.: Klee: a framework for distributed top-k query algorithms. In: VLDB, pp. 637–648 (2005)
19. O'Madadhain, J., Fisher, D., White, S., Boey, Y.-B.: JUNG: The Java Universal Network/Graph Framework. http://jung.sourceforge.net
20. Roy, A., Nilakant, K., Dalibard, V., Yoneki, E.: Mitigating i/o latency in ssd-based graph traversal. In: University of Cambridge, Computer Laboratory, Technical report (UCAM-CL-TR-823) (2012)
21. Sadoghi, M., Jacobsen, H.-A.: Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In: SIGMOD Conference, pp. 637–648 (2011)
22. Schenkel, R., Crecelius, T., Kacimi, M., Michel, S., Neumann, T., Parreira, J.X., Weikum, G.: Efficient top-k querying over social-tagging networks. In: SIGIR, pp. 523–530 (2008)
23. Singla, P., Richardson, M.: Yes, there is a correlation: from social networks to personal behavior on the web. In: WWW, pp. 655–664 (2008)
24. Theobald, M., Weikum, G., Schenkel, R.: Top-k query evaluation with probabilistic guarantees. In: VLDB, pp. 648–659 (2004)

25. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. J. ACM **46**(3), 362–394 (1999)
26. Ukkonen, A., Castillo, C., Donato, D., Gionis, A.: Searching the Wikipedia with contextual information. In: CIKM, pp. 1351–1352 (2008)
27. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., de Castro Reis, D., Ribeiro-Neto, B.A.: Efficient search ranking in social networks. In: CIKM, pp. 563–572 (2007)
28. Whang, S., Brower, C., Shanmugasundaram, J., Vassilvitskii, S., Vee, E., Yerneni, R., Garcia-Molina, H.: Indexing boolean expressions. PVLDB **2**(1), 37–48 (2009)
29. Wu, J., Chen, L., YU, Q., Han, P., Wu, Z.: Trust-aware media recommendation in heterogeneous social networks. In: World Wide Web, pp. 1–19 (2013)
30. Yahoo! advertising targeting options. http://advertising.yahoo.com/central/marketing/targeting.html
31. Yang, W.-S., Dia, J.-B.: Discovering cohesive subgroups from social networks for targeted advertising. Expert Syst. Appl. **34**(3), 2029–2038 (2008)
32. Yang, W.-S., Dia, J.-B., Cheng, H.-C., Lin, H.-T.: Mining social networks for targeted advertising. In: HICSS (2006)
33. Yoneki, E., Roy, A.: Scale-up graph processing: a storage-centric view. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES ’13, pp. 8:1–8:6. ACM, New York (2013)
34. Zhu, Y.: Measurement and analysis of an online content voting network: a case study of digg. In: WWW 2010, pp 1039–1048 (2010)