Socially Enhanced Search and Exploration in Social Tagging Networks

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Tom Crecelius

Max Planck Institut Informatik

Abteilung 5 : Datenbanken und Informationssysteme

max planck institut informatik

Saarbrücken, 2012

Dekan der Naturwissenschaftlich-Technischen Fakulät I	Prof. Dr. Mark Groves
Vorsitzender der Prüfungskommission	Prof. Dr. Jens Dittrich
Berichterstatter	Privatdozent Dr.-Ing. Ralf Schenkel
Berichterstatter	Dr. Sihem Amer-Yahia
Berichterstatter	Prof. Dr.-Ing. Gerhard Weikum
Beisitzer	Dr.-Ing. Klaus Berberich
Tag des Promotionskolloquiums	23.04.2012

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 16.02.2012

_____

(Tom Crecelius)

Abstract

Social tagging networks have become highly popular for publishing and searching contents. Users in such networks can review, rate and comment on contents, or annotate them with keywords (*social tags*) to give short but exact text representations of even non-textual contents. In addition, there is an inherent support for interactions and relationships among users. Thus, users naturally form groups of friends or of common interests.

We address three research areas in our work utilising these intrinsic features of social tagging networks.

- We investigate new approaches for exploiting the social knowledge of and the relationships between users for searching and recommending relevant contents, and integrate them in a comprehensive framework, coined *SENSE*, for search in social tagging networks.
- To dynamically update precomputed lists of transitive friends in descending order of their distance in user graphs of social tagging networks, we provide an algorithm for incrementally solving the all pairs shortest distance problem in large, disk-resident graphs and formally prove its correctness.
- Since users are content providers in social tagging networks, users may keep their own data at independent, local peers that collaborate in a distributed P2P network. We provide an algorithm for such systems to counter cheating of peers in authority computations over social networks.

The viability of each solution is demonstrated by extensive experiments regarding effectiveness and efficiency.

Kurzfassung

Im Internet sind soziale Netzwerke, die es erlauben Inhalte mit Anmerkungen zu versehen, inzwischen weit verbreitet und bei Anwendern gleichermaßen beliebt, um eigene Informationen zu veröffentlichen oder nach denen andere Benutzer zu suchen. Anwender können in diesen sozialer Netzwerken vorhandene Inhalte kritisieren, bewerten und kommentieren oder eben mit Schlagworten, d.h. mit sozialen Annotationen (engl. *social tags*) versehen. Ein weiteres Merkmal dieser sozialen Netzwerke ist es, dass Interaktionen und Freundshaftsbeziehungen zwischen Benutzern aktiv gefördert werden und sich so Anwender mit ähnlichen Interessen in Gruppen zusammenschließen.

Hieraus ergeben sich interessante Möglichkeiten für die Forschung. Wir sprechen drei Bereiche in dieser Arbeit an.

- Wir präsentieren mit *SENSE* ein umfassendes Rahmenwerk zur Suche in sozialen Netzwerken und stellen darin neue Ansätze zur Verbesserung von Suchergebnissen vor, die das gemeinschaftliche Wissen der Anwender und die Beziehungen zwischen den Anwendern nutzen.

- Zur kontinuierlichen Aktualisierung von Freundeslisten, stellen wir einen Algorithmus zur inkrementellen Lösung des kürzesten Wege-Problems zwischen allen Paaren von Knoten im Benutzergraphen sozialer Netzwerke vor.

- Soziale Netzwerke, die in einer verteilten P2P Umgebung betrieben werden, stehen dem Problem gegenüber, dass Benutzer-Peers versuchen können, Suchergebnisse zu beeinflussen. Wir stellen einen Algorithmus vor, der diesem Problem entgegentritt.

Summary

Over the years, more and more social tagging networks, like *Flickr.com*[1], *Delicious.com*[2], *Twitter.com*[3] or *LibraryThing.com*[4], have been emerging from the Internet and have become highly popular for publishing and searching contents, turning users from mere consumers into information providers. Users in such networks can easily publish own contents like photos or opinions, e.g. expressed in ratings, reviews or comments.

Moreover, users can annotate contents with arbitrary keywords, so called *social tags*, which allows for short but exact text representations of even non-textual contents. In this way, tags are collaboratively chosen by the users who publish the same contents (e.g. the same bookmarks, books, songs) or others who discovered them and find them worthwhile to annotate.

In addition, the provided services in social tagging networks inherently support and stimulate interactions and relationships among users. Thus, with a growing number of members, a more and more tight and huge network of users is established, and naturally, groups of friends or of common interests are formed.

These intrinsic features of social tagging networks offer great research possibilities. In this context, we address three challenging areas.

Firstly, we investigate new approaches for exploiting the social knowledge of users and the relationships between users for searching and recommending relevant content. Typically, items in social tagging networks like URLs or photos are retrieved by issuing queries that consist of a set of tags, returning items that have been frequently annotated with these tags. However, users often prefer a more personalised way of searching over such a 'global' search, exploiting preferences of and connections between users. Hence, we develop a comprehensive framework, coined *SENSE*, for socially enhanced search and exploration in social tagging networks. The framework includes a data model, defining the entities (e.g. users, tags, items like photos, etc.) and all relations being available in a multitude of social tagging networks, and allows for sophisticated scoring models, search strategies and efficient *top-k* algorithms. We apply the data model to three real world social tagging networks and implement two *top-k* algorithms in the context of *SENSE*. We discuss the distinctiveness of each algorithm, its scoring model and query processing, and show the effectiveness and efficiency of the respective approach in extensive experiments. *SENSE* has been implemented as prototype system for searching and exploring bookmarks, pictures and books on data retrieved from the social tagging networks *Delicious.com*, *Flickr.com* and *LibraryThing.com*. Hence, we introduce the system architecture of our *SENSE* prototype system.

Secondly, we provide a solution for dynamically updating transitive friendship lists of users in large social tagging networks when new users enter the system or new friendship connections are established. Friendship relations define a huge, directed and weighted graph where users, represented as nodes, are connected to their direct friends (i.e. the immediate neighbours) and transitive friends (i.e. the friends of friends), with edge weights defining the friendship strength of two immediate neighbours. In *SENSE*, we precompute lists of transitive friends in the order of shortest path distances in friend-

[1] http://flickr.com
[2] http://del.icio.us
[3] http://twitter.com
[4] http://librarything.com

ship graphs. Hence, when new users enter the social tagging network or new friendship connections between users are established, these updates to the friendship graph need to be reflected in the users' lists of nearest friends.

This problem is not limited to *SENSE*. Accessing transitive neighbours of a specific node in a given directed and possibly weighted graph in the order of increasing distances is a key building block of many algorithms for aggregated search in social networks, explorative search of connections in knowledge bases, or retrieval in linked documents. The problem of updating precomputed shortest path distances to transitive neighbours for all nodes in a graph is known as dynamic all pairs shortest distance (APSD) problem.

In the context of social tagging networks, we provide a novel algorithm for incrementally solving the APSD problem in large, disk-resident user graphs and formally prove its correctness.

Thirdly, we consider social tagging networks in the context of peer-to-peer (P2P) systems. The fact that users are also content providers in social tagging networks naturally motivates the idea that users keep own contents at their private computers and participate in a distributed network of P2P users instead. To make contents available and searchable to other users, the local and independent peers in a P2P network have to collaborate to form a distributed search engine. Of course, in such a distributed setting where users have to collaborate to identify interesting and authoritative information, the risk of misbehaving or cheating users arises.

We provide a distributed algorithm which allows us to compute authority scores over social networks in a Peer-to-Peer environment that correctly works even in the presence of cheating users.

The viability of our approach is demonstrated in experiments by mapping users and their data from the real world social tagging network *LibraryThing.com* to a simulated P2P network.

Zusammenfassung

Über die Jahre haben sich im Internet mehr und mehr soziale Netzwerke etabliert, die es erlauben in unkomplizierter Weise, eigene Inhalte ins Netz zu stellen oder nach neuen und interessanten Inhalten anderer Benutzer zu suchen. Beispiele hierfür sind *Flickr.com*[5], *Delicious.com*[6], *Twitter.com*[7] und *LibraryThing.com*[8]. Inhalte können hierbei sowohl eigene Dateien sein, z.B. digitale Fotos, oder auch Meinungen, z.B. durch Schreiben von Bewertungen, Rezensionen oder Kommentaren. Aufgrund sozialer Netzwerke haben sich somit Internet-Benutzer von bloßen Konsumenten zu Produzenten von Inhalten gewandelt.

Ein herausragendes Merkmal dieser sozialen Netzwerke ist es, dass Anwender vorhandene Inhalte mit Anmerkungen versehen, die jeweils nur aus einzelnen aber beliebigen Worten bestehen. So ist es möglich mittels kurzen und präzisen Schlagworten, selbst nicht-textuelle Inhalte geeignet zu repräsentieren. Das Annotieren von Inhalten ist dabei eine gemeinschaftliche Leistung vieler Anwender – nämlich von denen, die gleiche Inhalte ins Netz stellen (z.b. die gleichen Lesezeichen, Bücher oder Musik) und von denen, die sie im Netzwerk auffinden und es als lohnenswert erachten, diese zu annotieren. Aus diesem Grund werden die in Anmerkungen verwendeten Schlagworte als *soziale Annotationen* bezeichnet (engl. *social tags*). Des Weiteren bieten soziale Netzwerke eigens Dienste an, die die Interaktion und die Bildung von sozialen Zusammenschlüssen der Benutzer (z.B. in Freundschaften, Gruppen) aktiv unterstützen. Auf diese Weise entstehen mit steigender Anzahl an Benutzern ein enges und riesiges Netz von miteinander in Verbindung stehenden Anwendern. Hierdurch entwickeln sich über die Zeit hinweg eine Vielzahl an Gruppen von Freunden oder Anwendern mit gleichen Interessen.

Diese inhärenten Eigenschaften sozialer Netzwerke bieten herausfordernde Möglichkeiten für die Forschung. Wir sprechen in dieser Zusammenhang drei interessante Bereiche an.

Im ersten Teil dieser Arbeit entwickeln wir neue Ansätze, die das soziale Wissen von Anwendern und die Beziehungen zwischen den Anwendern nutzen, um bessere Suchergebnisse für Benutzeranfragen erzielen zu können. Inhalte in sozialen Netzwerken (z.B. Fotos, Lesezeichen, etc.) finden Anwender meist, indem sie eine Suchanfrage bestehend aus einer oder mehreren Schlagworten absenden. Üblicherweise setzen sich Suchergebnisse für diese Anfragen aus den Inhalten zusammen, die am häufigsten von allen Benutzen des Netzwerks mit eben jenen Schlagworten annotiert wurden. Eine personalisierte und um soziale Merkmale erweiterte Suche wird allerdings von Benutzern oftmals einer 'globalen' Suche über das gesamte Netzwerk bevorzugt.

Wir präsentieren mit *SENSE* ein umfassendes Rahmenwerk zur Suche in sozialen Netzwerken, die sowohl das gemeinschaftliche Wissen aller Anwender als auch die Vorlieben und Beziehungen der Anwender untereinander miteinbezieht. Hierzu definiert *SENSE* ein Datenmodell, das auf eine Vielzahl von sozialen Netzwerken und deren individuelle Elemente (z.B. Benutzer, Annotationen, Relationen, Inhalte wie Fotos, etc.) angewendet werden kann. *SENSE* erlaubt dabei die Definition ausgefeilter Auswertungsmodelle, neuartiger Suchstrategien und effizienter *top-k* Algorithmen. Wir zeigen

[5] http://flickr.com
[6] http://del.icio.us
[7] http://twitter.com
[8] http://librarything.com

die Anwendbarkeit des Datenmodells anhand dreier real existierender sozialer Netzwerke und implementieren in *SENSE* zwei verschiedene *top-k* Algorithmen. Wir stellen die Besonderheiten jeden Algorithmus, ihrer Auswertungsmodelle und Anfrageabwicklungen heraus und demonstrieren deren Effizienz und Effektivität in umfassenden Experimenten. Ebenso stellen wir die Systemarchitektur einer Prototyp-Implementierung von *SENSE* vor, die wir zur Suche von Lesezeichen, Bildern und Büchern in einem Auszug der Daten von *Delicious.com*, *Flickr.com* und *LibraryThing.com* entwickelt haben.

Im zweiten Teil dieser Arbeit präsentieren wir ein Lösung, um transitive Freundschaftslisten von Anwendern in großen sozialen Netzwerken dynamisch zu aktualisieren, wenn neue Anwender dem Netzwerk beitreten oder neue Freundschaftsverbindungen erstellt werden. Die Beziehungen zwischen Anwendern in großen sozialen Netzwerken definieren einen riesigen, gerichteten Graphen, in dem Anwender als Knoten repräsentiert werden und Kanten den Verbindungen zu ihren direkten Freunden entsprechen. Kantengewichte stellen hierbei dar wie eng das Freundschaftsverhältnis zwischen zwei direkten Freunden ist. In diesem Freundschaftsgraph sind Anwender indirekt mit allen transitiven Freunden (d.h. den Freunden der Freunde) verbunden.

In *SENSE* werden für die Benutzer sozialer Netzwerke Listen vorberechnet, die alle ihre transitiven Freunde enthalten. Diese Freunschaftslisten sind in absteigender Reihenfolge zu den kürzesten Wege-Distanzen im Freundschaftsgraphen sortiert. Treten dem sozialen Netzwerk neue Benutzer bei oder werden neue Freundschaften geschlossen, so müssen entsprechend die Freundschafslisten angepasst werden. Das Problem der Aktualisierung kürzester Wege-Distanzen ist allerdings nicht nur auf *SENSE* beschränkt. Eine Schlüsselkomponente in vielen Algorithmen zur aggregierten Suche in sozialen Netzwerken, zur Erkundung von Zusammenhängen in Wissensdatenbanken oder zur Suche in verknüpften Dokumenten ist es, die transitiven Nachbarn eines bestimmten Knotens in der Reihenfolge ihrer Entfernungen innerhalb eines gerichteten und potenziell gewichteten Graphen zu finden.

Das Problem der schrittweisen Aktualisierung vorberechneter kürzesten Wege-Distanzen zu den transitiven Nachbarn von allen Knoten in einem Graph wird als dynamisches *"all pairs shortest distance (APSD)"*-Problem bezeichnet. Wir stellen einen neuartigen Algorithmus im Rahmen sozialer Netzwerke vor, der das dynamische APSD-Problem in großen, Plattenspreicher-basierenden Freundschaftsgraphen inkrementell löst und beweisen die Korrektheit des Algorithmus.

Im dritten Teil dieser Arbeit betrachten wir soziale Netzwerke im Kontext von Peer-to-Peer (P2P) Systemen. Da Inhalte in sozialen Netzwerken von den Anwendern erstellt werden, können diese Inhalte auch auf den privaten Computern ihrer Besitzer belassen werden, wenn diese in einem verteilen Peer-to-Peer Netzwerk zusammengeschlossen sind. Um Inhalte anderen Anwendern zugänglich zu machen, bilden die lokalen und unabhängigen Benutzer-Peers eine verteilte Suchmaschine. Das heißt, in einem solchen System müssen Peers zusammenarbeiten, damit interessante Inhalte und zuverlässige Informationen gefunden werden können. Hierdurch entsteht natürlich das Risiko das sich Benutzer betrügerisch verhalten, um die Suchergebnisse nach eigenen Vorstellungen zu beeinflussen.

Wir stellen einen verteilten Algorithmus vor, der es in einem P2P Netzwerk mit betrügerischen Benutzer-Peers erlaubt, korrekte Berechnungen über die Wertigkeit von Inhalten sozialer Netzwerke durchzuführen.

Contents

Part
Outline

1 Overview

Social tagging networks provide an easy to use environment for users in the Internet to publish own data or information about various contents, to express opinions, and to interact with other users. Social tags are another key feature of social tagging networks that allow users to organise or describe data of interest by annotating contents in the network with arbitrary keywords. In this way, social tagging networks are filled with data, information and opinions of millions of users. Hence, there is a great potential for harnessing the "wisdom of crowds", with social interactions of individual users and user groups taken into account.

In our research studies, we utilise the particular features of social tagging networks to personalise search results and to explore interesting new data. Moreover, we deal with the problem of how to dynamically update friendship graphs in such networks. Another aspect of our research is to consider social tagging networks in the context of a distributed peer-to-peer (P2P) system instead of relying on a centralised architecture. In such a P2P environment, where independent peers collaborate to form a distributed search engine, special care has to be taken about malicious or cheating peers when computing authoritative information over social tagging networks.

1.1 Scope of Work

Our research presented in this work is threefold.

A We present a comprehensive social search framework, coined *SENSE*, including a data model representing contents and relations in social tagging networks, sophisticated search strategies and scoring models, and two efficient algorithms for searching and exploring items of interest to querying users in the network. Extensive experiments on data from real world social tagging networks show the quality and efficiency of our approach. Additionally, we discuss the system architecture of our prototype system implementing the *SENSE* framework.

Our research work in the area of social search and efficient *top-k* querying over social tagging networks has been published in [24], [38], [39], [40], [112], [111]

B Moreover, we discuss the problem of how to incrementally solve the all pairs shortest distance (APSD) problem for dynamically updating social friendship lists of users in large social tagging networks. Social friendship lists are precomputed lists of nearest neighbours in a weighted user graph of social networks. We present an algorithm that allows to maintain updates—which inserts new edges to the graph or increase edge weights—on-the-fly while queries for the users' *top-k* best friends are processed. Experiments on real world social tagging networks demonstrates the viability of our algorithm and a formal proof of correctness shows that queries always correctly identify the *top-k* best friends of users (for any value of *top-k*).

Our recent research work about retrieving and maintaining nearest neighbours in large, dynamic graphs of social tagging networks is currently in preparation for being published.

C Finally, we consider social tagging networks detached from a centralised view but cast in the distributed environment of a peer-to-peer (P2P) network where

users are represented by individual peers and have to collaborate to form a distributed search engine. Computing authoritative information in such a system is a challenging task in the presence of malicious peers trying to manipulate the outcome of the computation. We present an algorithm for countering cheating in P2P authority computations over social networks.

Our research work in the area of P2P has been published in [25], [26], [102], [117]

1.2 Structure of the Thesis

This thesis is structured in three parts according to the three areas covered by our research work.

Part A presents our research on social search and efficient *top-k* querying over social tagging networks. In Chapter 2, we motivate our research in this area (Section 2.1) and discuss related work (Section 2.2). Chapter 3 introduces our sophisticated *SENSE* framework. We present in Section 3.1 a unified data model capturing the various kinds of contents and activities typically found in a multitude of social tagging networks. In Section 3.2 three real world social tagging networks are described and presented in the context of our data model. In Section 3.3, we explicate the design decisions being fundamental to our *SENSE* framework. In Chapter 4, we then formally introduce the problem statement that is solved by our two *top-k* threshold algorithms implemented in the scope of *SENSE*. Our first algorithm, coined SOCIALMERGE, is discussed in Chapter 5. In Section 5.1 and Section 5.2 we present the details of its scoring model and explain its query processing, respectively, while in Section 5.3 various search strategies based on the algorithm's scoring model are defined. In Section 5.4, an experimental evaluation on two real world datasets shows the effectiveness and efficiency of our algorithm for each respective search strategy. Our second algorithm, coined CONTEXTMERGE, is discussed in Chapter 6. In Section 6.1, we first classify the task of searching in social tagging networks according to their information needs before providing detailed information about our algorithm's scoring model in Section 6.2 and its query processing in Section 6.3. The effectiveness and efficiency of CONTEXTMERGE is shown in Section 6.4 by extensive experiments on datasets retrieved from three real world social tagging networks. In Chapter 7, we finally present the system architecture of our *SENSE* prototype implementation.

Part B presents our research work on maintaining and searching the *top-k* nearest neighbours of users in large, disk-resident friendship graphs of increasing social tagging networks. In this respect, we developed a novel algorithm solving the all pairs shortest distance (APSD) problem. In Chapter 8, we motivate our research in this area (Section 8.1) and discuss related work (Section 8.2). Our research is then presented in Chapter 9. We give an overview over the problem in Section 9.1, define the setup in regard to social networks in Section 9.2, and formally introduce the problem statement in Section 9.3. Our basic algorithm for incrementally solving the all pairs shortest distance (APSD) problem is presented in Section 9.4. It works on precomputed inverted lists of nearest friends in friendship graphs of social networks that are maintained by only two basic operations, i.e. *update* and *merge*, while queries for a user's *top-k* nearest friends are processed. In Section 9.5 the associated data structures are defined and in Section 9.6, we address the difficulty of concurrency in regard to updates to the

graph while queries are processed. In Section 9.7 we explain when there is a need for an update operation and in Section 9.8, when there is a need for a merge operation. In Section 9.9 and 9.10 we introduce two alternative approaches, i.e. *EAP* and *LAP*, for implementing update and merge operations and discuss their respective advantages and disadvantages. In Section 9.11 we discuss improvements to both approaches. The chapter closes with extensive experiments on two real world social networks in Section 9.12 which show the performance of each approach. In Chapter 10 we formally prove the correctness of our basic algorithm.

Part C of this work presents our research on peer-to-peer (P2P) user networks where peers are collaborating for computing authoritative information over social tagging networks. In Chapter 11, we motivate our research in this area (Section 11.1) and discuss related work (Section 11.2). Chapter 12 presents our algorithm for countering cheating in P2P authority computations over social networks. In Section 12.1 first an overview over the problem is given before we formally define it in Section 12.2. In Section 12.3, we then present a distributed algorithm that is able to compute correct authority scores (PageRank scores) in a P2P network even in the presence of peers that try to maliciously manipulate the computation. In Section 12.4, we show the viability of our approach by experiments on data retrieved from a real social tagging network.

1.3 Notational Conventions

- The spelling in this work follows the rules of British English.
- The end of properties, lemmas, invariants, theorems and definitions are marked by the following symbol: ■
- The end of a proof is marked by the following symbol: □
- Pseudocode is depicted in the following way:
  - code lines are numbered.
  - lines with only comments are unnumbered.
  - comments are written in grey and preceded by two slashes, e.g. // comment.
  - commands regulating the program flow are written in RED, upper case letters, e.g. IF.
  - non-trivial method calls are highlighted by *blue*, italic, lower case letters, e.g. $U.update()$.
  - curly brackets are written in grey, e.g. { }

Part A
Search in Social Tagging Networks

2 Introduction

Online social networks like *Facebook.com*, *YouTube.com*, or *Last.fm* have established themselves as very popular and powerful services for publishing and searching content, as well as for finding and connecting to other users that share common interests. The contents in these social networks are typically user-generated and include, for example, personal blogs (e.g. *WordPress.com*), bookmarks (e.g. *Delicious.com*), and digital photos (e.g. *Flickr.com*).

A particularly intriguing type of content are user-generated annotations, so called *social tags*, that users affix to their own or to other users' content items. Social tagging, i.e. the collaboration of users to annotate content items, is an integral part of *social tagging networks*. Usually social tags are carefully selected and often semantically meaningful, hence, concise string descriptions of the involved content item and, therefore, allow for reasoning about the interests of the user who created or published the content, but also about the users who generated the annotations.

To illustrate the concepts of user provided contents and tags in social tagging networks, Figure 1 shows a screenshot of *LibraryThing.com*[6] displaying a user's library, i.e. books catalogued by a user. In *LibraryThing.com*, users create personal libraries and catalogue books, enriching book entries with meta data like ratings or *tags*. A detailed description of *LibraryThing.com* and other social tagging networks is given in Section 3.2.

In the following, we motivate our research in this area and give an overview of related work before introducing in Chapter 3 the basic framework for our *SENSE—Socially ENhanced Search and Exploration*—algorithms on social tagging networks. The formal problem statement follows in Chapter 4 and details on our two algorithms SOCIALMERGE and CONTEXTMERGE for enabling sophisticated search strategies and serving different information needs of users in social tagging networks in Chapter 5 and Chapter 6, respectively. For both algorithms we show their usefulness and efficiency in several experiments on datasets crawled from real world social tagging networks. Finally, in Chapter 7 we give details about the system architecture of our prototype implementation of the *SENSE* framework.

2.1 Motivation

The advent of online social networks like *Flickr.com*, *MySpace.com*, *Facebook.com*, or *YouTube.com* has changed the way users interact with the Internet. While previously most users were mere information consumers, those platforms are offering an easy and hassle-free way to also publish own content, turning users into content producers, too. Users are encouraged to share content items, e.g. photos or videos, opinions, to rate content, but also to explore the online network's community and to find people with similar interest profiles. In this sense, social networks not only have changed the way people interact with the Internet but also the way users interact with each other.

A key feature of social networks is that users are able to maintain lists of friends which facilitate communication between and notification of friends about latest content items. The size of such friendship lists is often considered as a gauge for a user's reputation in the network. While initially, many users populate their friendship lists with people they already know from the offline world or other online communities, as time goes by, they typically identify previously unknown users that share common interests and also add those users to their list of friends.

Figure 1: User provided content in the Social Tagging Network *LibraryThing.com*

Furthermore, in many social networks, like *Delicious.com* or *LibraryThing.com*, another key feature, coined social tagging , is the widely-used opportunity of all users to attach manually generated annotations, i.e. social tags, to content items. Most social tagging networks offer comfortable and intuitive ways to explore new content items based on these tags, e.g. via tag clouds. Therefore, social tagging has emerged as an important asset to explore the fast-growing communities in order to identify interesting content and users [52, 56, 70, 119, 118].

Hence, social tagging networks offer a great potential that can be utilised to improve the search experience of users in the network. We identified the following challenging research aspects:

- **Social Tags.** Social tags are usually carefully selected and often semantically meaningful and, thus, can be considered precise descriptions of content items, flavoured with the respective personal interest of the user who generated the tag. Since the multitude of users' opinions lead to a variety of tags for a single item, as a consequence, the tags describing an item are collaboratively chosen by those users who publish the same content items or who find them in the network, considering it worthwhile to tag, rate or to comment on them.

  Hence, the typically high quality of user-generated social tags suggests to leverage this "wisdom of the crowds" [122] for identifying and ranking high-quality and high-authority content. Ratings or comments expressing the user's opinions about an item can be taken in addition into account.

- **Social Relations.** The relationships among users can be taken into consideration for ranking search results, the intuition being that you trust the recommendations of your close friends more than those of your casual acquaintances. This situation resembles the paradigm of collaborative recommendation [67, 92] which applies data mining on customer-products and similar usage data to predict items that users are likely interested in.

  However, social networks offer the additional opportunity to exploit the socially enhanced "wisdom of the crowds" [122] by identifying valuable content that is recommended by friends. For this, the notion of friendship between two users can be deduced either directly (i.e. users selected as friends) or indirectly for transitive friends (i.e. friends of friends), and, taking social tagging into account, explicitly from user ratings and tag semantics (that express positive or negative opinions), or implicitly from similar tag usage patterns of users.

- **Personalised or Social Search.** While differing in the type of content social tagging networks focus on (e.g. blog entries, photos, videos, bookmarks, etc), they almost all work similarly. Initially, users register and join the community. Once registered, they start to produce information, ideally by creating or publishing their own content items and by adding tags or ratings, comments, etc., to their own or other users' content items already available in the network.

  Content items, e.g. URLs or videos, are typically searched for and retrieved by issuing queries that consist of a set of tags, returning items that have been frequently annotated with these tags. However, users often prefer a more personalised way of searching over such a 'global' search, exploiting preferences of and connections between users.

  To this end, a user can have different information needs when searching for different content items. Therefore, search in social networks should be flexible enough to express different information needs to enable a personalised and socially enhanced way of searching in the network.

- **Efficiency and Scalability.** The existing, traditional algorithms for searching on the web do not consider user relationships and the assets from social tagging, thus, fall short of being effective in social networks since they focus only on the content quality and disregard the social component. Prior methods for collaborative recommendation, on the other hand, do not provide the throughput scalability that one needs for running millions of daily ad-hoc queries in social communities such as *Flickr.com* - with more than six billion content items [86], many million users, and high dynamics.

  This makes a strong case for novel methods that exploit the different entities present in social networks (users, content items, tags) and their mutual relationships. However, the fast growth of communities and the very high rate of content production and tagging efforts calls for highly *efficient* and *scalable* methods.

Motivated by these research opportunities, we conceived a framework to cast the different entities of social tagging networks into a unified data model representing the mutual relationships of users, content items, and tags. It derives scoring functions for each of the entities and their relations, includes algorithms for the efficient retrieval of search results and addresses all the mentioned research aspects. We have performed an experimental evaluation of the quality of these scoring functions and efficiency of our

search algorithms on several real-world datasets crawled from the social tagging networks *Delicious.com*, *Flickr.com* and *LibraryThing.com*. Manual user assessments of the result quality suggest that our unified data framework delivers high-quality results on social networks while the retrieval performance is highly efficient.

2.2 Related Work

Social networks of the Web 2.0 style have received major attention in the recent literature, with focus on applying data mining methods on social relations and, most prominently, relations among tags. [56] provides an empirical study of the tagging behavior and tag usage in online communities. [69, 114] discuss methods for generating taxonomy-like relations among tags, so-called "folksonomies", based on statistical measures. Similar approaches have been applied to query-and-click logs, e.g. in [17], but none of this work considers social relations between different users. Identifying important and emergent tags and visualising them in so-called "tag clouds" (and corresponding time series) has been extensively explored; recent work along these lines includes [52, 64, 115]. The dynamics of social relations among users (e.g. the rate of making friends) has been studied, for example, in [12, 88, 123].

As for the exploitation of social tags for information retrieval, [14] discusses the challenges of searching and ranking in social communities. Various forms of community-aware ranking methods have been developed, mostly inspired by the well-known PageRank method [30] for web link analysis. [70] proposes *FolkRank* for identifying important users, data items, and tags. [133] compares different methods for identifying authoritative users with high expertise. [19] introduces *SocialPageRank*, to measure page authority based on its annotations, and *SocialSimRank* for the similarity of tags. [132] further extends this work by augmenting language models with tag similarities. [49] shows that explicit user tagging can help to improve precision of queries for Intranet search. The work of [68] provides an empirical analysis of how social bookmarking can influence web search, with both positive and negative insights. None of this prior work considers the impact of user-friendship strengths on the scoring of search results, and the problem of efficient query processing in the presence of such "social wisdom".

The work of [13] discusses efficient top-k processing of network-aware search queries in collaborative tagging sites by clustering users in groups of seakers and taggers with similar behaviour and defining cluster related upper score bounds for items. However, they do not consider the impact of social relations or tagging activities on the result quality.

Aspects of user communities have also been considered for peer-to-peer search, most notably, for establishing "social ties" between peers and routing queries based on corresponding similarity measures (e.g. similarities of queries issued by different peers). [25] has studied "social" query routing strategies based on explicit friendship relationships and behavioral affinity. [103] has developed an architecture and methods for "social" overlay networks that connect "taste buddies" with each other. [97] has proposed a community-enhanced web search engine that takes into account prior clicks by community members. [42] has proposed the notion of Peer-Sensitive ObjectRank, where peers receive resources from their friends and rank them using peer-specific trust values.

There is ample literature on collaborative filtering for recommender systems (e.g. [11, 66, 108, 110]), for example, to predict movies or e-commerce items that customers are likely to buy or to identify news that news-feed subscribers are likely interested in. In a nutshell, these methods aim to learn user preferences from the collective behaviour

- like purchases or tagging - of an entire community. Typically, statistical analysis and machine learning techniques are used offline for precomputations, and the actual run-time recommendations have got limited flexibility and cannot easily cope with high dynamics and ad-hoc interests of individual users (as expressed by an ad-hoc query). One of the notable exceptions is the work by [43] which addresses scalability issues when the number of users and items in a recommender system grows to many millions and both undergo fast changes. However, in contrast to our "social search" theme, this prior work considers only the space of user-item pairs and there is no notion of user-specific tags or annotations on items. Thus, our setting requires search over a three-dimensional user-tag-item space, as opposed to the two-dimensional user-item space of the previous work on collaborative filtering.

3 *SENSE* Framework

In this chapter, we discuss the data model used in our *SENSE—Socially ENhanced Search and Exploration*—framework for social tagging networks, introduce the basic design decisions being made for its implementation and present three real world social tagging networks that have been mapped to our data model and are the basis for the experimental evaluations given in Section 5.4 and 6.4.

3.1 Data Model

This section defines a unified set of abstractions, modelling the user-provided data and activities in social tagging networks. For this, entities occurring in social networks are cast into the model, representing different types and their mutual relationships.

3.1.1 Types of Entities

We identify three major types of entities in social tagging networks which are represented in our data model in the following way:

- **User U:** A user U in social tagging networks produces content either by creating or publishing own documents or by tagging existing content.
- **Document d:** A document d is a content item that is published by a user U, e.g. a blog entry, a bookmark or a photo, etc.
- **Tag t :** A tag t is a keyword used by a user U to annotate documents d and usually describes or categorises the respective document.

Additionally, social tagging networks exhibit various relationships, both within entities of the same type (*intra-entity relations*) and between entities of different types (*inter-entity relations*). Each relationship can be cast into a relational scheme and is given in detail in the following sections.

3.1.2 Intra-Entity Relations

Each of the three entity types exhibits some sort of relation between the entities of the same type.

User-User Relation: Friendship(U_1,U_2,$type$,s_f)

Friendship is a user-user relation between two users U_1 and U_2 with a weight s_f equal to the friendship strength of U_2 with respect to U_1. In *friendship* relations, we treat s_f as a pluggable building block; it may also be completely absent (or constant/equal for each pair of users). Quantitative measures for different friendship relations are given in Section 5.1 and 6.2.

The friendship relation can be defined in different forms; we therefore allow multiple types of friendship relations captured by the *type* attribute. A variety of friendship type definitions are possible, and in the following, we describe three intriguing types: social, spiritual and global friendship.

Definition 3.1 (**Social Friendship**). *A* friendship *of type* social *is defined by a user-provided relation which can be either symmetric or asymmetric; it assumes that such a relation exists only if the users know each other by some* social *interaction in real life or within the social network.* ∎

A key feature of social networks is to allow users to maintain an explicit list of friends. Hence, a straightforward way to create a social friendship relation with instances of the form that user U_1 considers user U_2 as a friend is done by the explicit act of U_1 adding U_2 to her friendship list.

Additional means of establishing *social* friendships include, for example, a user subscribing to another user's content (e.g. in *LibraryThing.com*, by adding other users' books to the own set of *interesting libraries*), or writing comments on a user's profile page or addressing others in messages (e.g. in *Twitter.com*, by mentioning a user in own *tweets*). Regardless of how direct friends are defined, we may consider the transitive closure of the friendship relation or a bounded set of transitive connections (up to some distance). The union over all users and social friendship relations defines the user or friendship graph of a social tagging network.

Definition 3.2 ((Social) Friendship Graph, User Graph)**.** *The graph representing all users in a social network with edges being defined between users according to their social friendship relations is called the* friendship graph *or* user graph *of the social network.* ∎

The friendship strength s_f of $type = social$, also denoted as *social friendship strength*, between two users could be finally derived from their distance in the underlying social friendship graph of the network, i.e. the length of the shortest path from one user to the other, and can function as measure of the trust of one user in another user. In addition, the friendship strength could be weighted by some semantic measurement like considering the overlap in the usage of tags, too, such that it is a combination of social friendship and mutual interests.

Definition 3.3 (**Spiritual Friendship**). *A* friendship *of type* spiritual *considers similar behaviour or activities of users. This type of friendships is a symmetric relation and does not assume that the users know each other but rather are* "Brothers in Spirit"*—hence, the chosen name for this type of friendship—expressing the overlap in thematic interests and being an indicator of users sharing common interests.* ∎

The spiritual friendship could, for example, be based on users' participation in thematic similar groups, or it could be based on similar tags being issued to documents or personal content items receiving similar tags from third parties. It could also be derived from mutual comments and ratings.

The friendship strength s_f of $type = spiritual$, also denoted as *spiritual friendship strength*, between two users represents the degree of mutual interest overlap by taking the mentioned behaviour and activities of users into account.

Definition 3.4 (**Global Friendship**). *A* friendship *of type* global *is defined by neglecting all kind of distinctions of users but treating all of them as one* global *community of like-minded users, i.e. each user is a friend of each other user and with an equal weight for the friendship strength s_f for all users.*

Concrete definitions of social, spiritual and global friendship relations are given in Section 5.1 and 6.2.

Tag-Tag Relation: TagSimilarity(t_1,t_2,$tsim$)

In social tagging networks, users frequently use more than one tag to describe a particular document, documents can be tagged by more than one user, and the same tag can be used on different documents. There is no restriction which terms can be used as tags. In fact, tags are freely selected annotations, and, given the natural diversity of users' opinions in social networks, it is often the case that different tags describe the same content item and may express (near-)synonyms (e.g. "feline" and "cat", "Web_2.0" and "Social_Web", etc.) or other kinds of semantically related concepts (e.g. hyponyms such as "dogs" and "German_shepherd", "search_engine" and "Google", etc.).

Determining the similarity of tags is a way of clustering tags with respect to their meaning. To this end, an ontology, a light-weight knowledge base that captures different types of "semantic" relations among tags (e.g. synonymy or specialisation / generalisation), could be exploited. An ontology may be provided by domain experts or imported from real ontologies, or they may be built by applying data-mining techniques to the tagging data of the social network. The latter case is more realistic for today's types of social tagging networks and is often referred to as "folksonomies" (folklore taxonomies) [70, 127]. Hence, the tag-usage statistics in the social network are harnessed to derive a weight corresponding to the tag similarity $tsim$ for two tags t_1 and t_2. Quantitative measures are given again in Section 5.1 and 6.2.

Document-Document Relation: Linkage(d_1,d_2,w)

In some applications, like in web search and PageRank computations [31, 99], documents also exhibit relations among themselves. In the case of web pages, this linkage is obvious and given by the hyperlink graph, with weights w often chosen proportionally to the outdegree of the pages. For other types of documents, different notions of links between two documents d_1 and d_2 and their weight w need to be defined; conceivable options include, for example, the geographic proximity of different photos when GPS information is available or may be based on associated feature vectors representing the documents.

3.1.3 Inter-Entity Relations

For our unified data model, we observe the following relations between entities of different types:

Document-Tag Relation: Content(d,t,$score$)

By annotating a document d with a tag t, users strongly associate the tag with the document so that the tag should be viewed as a strong indication about the document's content. We consider the value $score$ in the content relation as a weight associated with a document-tag pair to reflect how well that tag describes the document.

User-Document Relation: Rating(U,d,$rating$)

In many social communities, a user U can explicitly rate a document d, which is captured by a rating score $rating$. Another naïve instantiation of *Rating* is authorship of a content item, which (e.g. in the case of bookmarks) can be seen as an endorsement for the document. Alternatively, we can also derive a weight as rating score based on the tag usage of the user.

User-Tag-Document Relation: Tagging(U,t,d,$score$)

A tag t is naturally associated with the document d and user U who associates it with that document. Hence, *Tagging* is a *ternary* relation between users U, documents d, and tags t. In full generality, it can *not* be decomposed into three binary relations (users-docs, docs-tags, users-tags) without losing information. Nevertheless, binary-relation (or, equivalently, graph or matrix) representations for tagging are very popular in the literature on social networks for convenience.

Our approach preserves the full information and feeds it into a scoring model (see Sections 5.1 and 6.2).

3.1.4 Remarks

With the ingredients given above, our data model eventually allows for well-founded scoring and ranking models that go far beyond ad-hoc retrieval models for social networks which often include many hard-to-tune parameters.

Furthermore, it is important to note that the weights for all the relations introduced in our data model can be defined in many different ways. The examples we have provided present some of the alternatives, but are not meant to be exhaustive. Our data model and the upcoming scoring and ranking models presented in Section 5.1 and 6.2 are independent from changes to those functions.

Also note that this model is much richer than, e.g. the datasets in traditional recommender systems. In addition to the shown relations, we can easily add various kinds of aggregation views, for example, document-tag frequencies aggregated over all users.

Also note, while our data model captures all relationships that might occur in a social network, some of the introduced relationships might not exist for certain social networks. In *LibraryThing.com*, for instance, user interactions are mainly through bookmarking and tagging, whereas in *Flickr.com*, the vast majority of users has got "authored" contents, which in this case are the photos that they have published. Moreover, only few platforms for social networks would show the users' home locations and some might not facilitate any cross-references among individual items. A detailed analysis of three real world social tagging networks is given in Section 3.2.

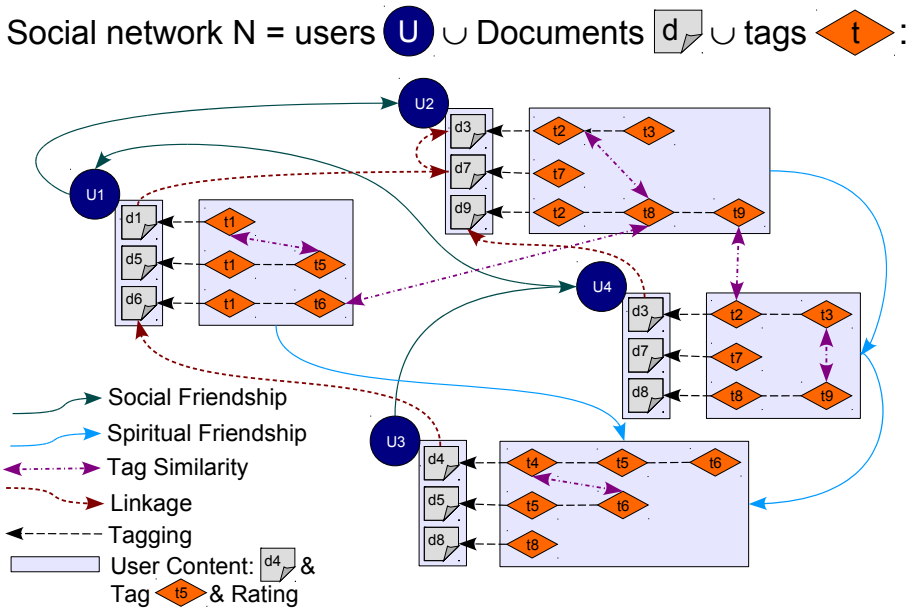


Figure 2: Example illustration of our data model

In any case, our data model is flexible enough for being applicable to most social tagging platforms. And in fact, as shown in Section 5.4 and Section 6.4, our experimental studies on three real world social networks utilise only a subset of our data model.

An example illustration for our data model with 4 users, their tagged documents and the relations between entities of the same and of different types, is given in Figure 2.

3.2 Datasets

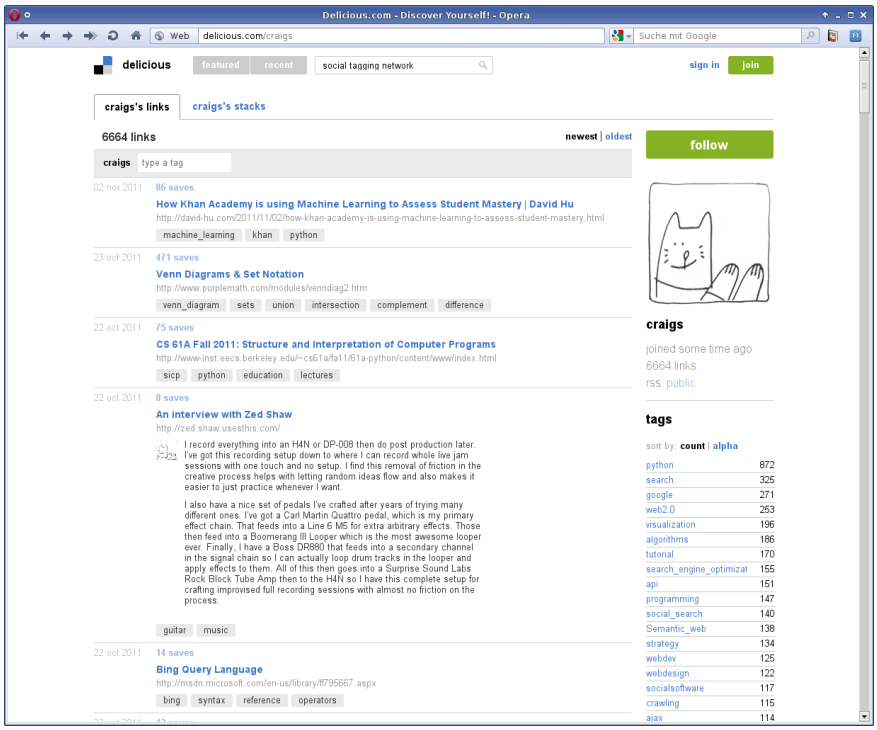
In this section, we introduce three real world social tagging networks that were the basis for obtaining the datasets used in all our experimental evaluations.

3.2.1 *Delicious.com*

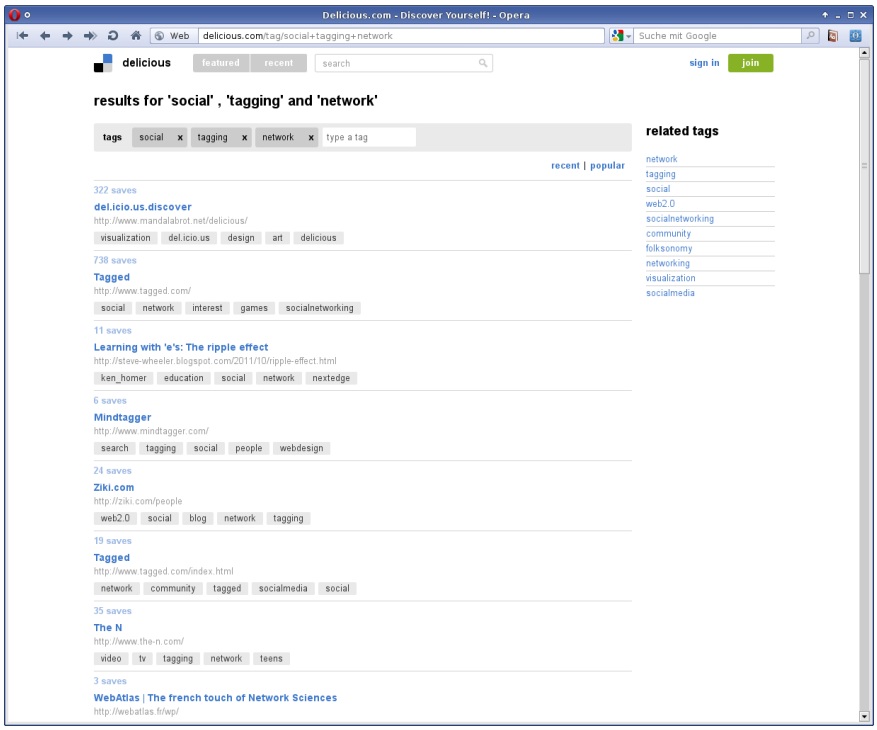
The social web bookmarking service *Delicious.com* [2] is a popular social tagging network with millions of registered users and over one hundred million unique URLs bookmarked (as of September 2007 [1], end of 2008 [8]).

By registering at *Delicious.com*, users can store and share own bookmarks, i.e. links to URLs of web pages, or discover bookmarked web pages of other users in the network. For this, each user has got a homepage in the social network, listing all her bookmarks. When saving a bookmark on her homepage in *Delicious.com*, a user can make notes on each web page, allowing for a content description or other related comments. Notes on web pages are shown in addition to the bookmark on the user's homepage. An example of a user homepage in *Delicious.com* (as of October 2011) is given in Figure 3a.

While bookmarks are shown in an non-hierarchical and unstructured flat list, the organisation of bookmarks is achieved by tags, individually chosen by each user. Tags



(a) Example for user homepage in *Delicious.com*



(b) Bookmarks in *Delicious.com* about "Social Tagging Network"

Figure 3: Example screenshots of *Delicious.com*

are usually assigned when a bookmark is saved in *Delicious.com* but also can be added at a later time. They are used to categorise or describe the content of an associated web page with single keywords. On the right hand side of a homepage, there is a list of all tags assigned to web pages which allows users or visitors of the homepage to navigate through all saved bookmarks and to locate web pages of interest. By clicking on the tags in this list, only the bookmarks annotated with the respective tag are displayed. In general, all bookmarks listed on a user's homepage are public and visible to everyone if not explicitly declared otherwise as private (on a per bookmark granularity).

Apart from the tags and notes used to describe a web page, the list of bookmarks additionally shows for each URL—and this is true for all URLs discovered in *Delicious.com*—how many users in *Delicious.com* saved the same web page on their homepages (but not necessarily annotated it with the same tags). Users are offered many ways for discovering bookmarks. For example, they can browse through most recent or popular bookmarked URLs or through most recent or popular used tags. For one or more given tags, *Delicious.com* lists all bookmarks annotated by any users with these tags. The latter allows to search for web pages of certain topics. An example is given in Figure 3b which is a screenshot of the results when searching for the tags "social", "tagging", and "networks".

Moreover, by clicking on the number showing how many users have saved the same bookmark, it is possible to navigate to the homepages of those users and browse through their list of bookmarks. In this way, it is easily possible to find other users with preferences matching personal interests. Finally, it is possible to remember those users on the own homepage to quickly have got access to their bookmarks.

At the time when we crawled the *Delicious.com* website, users could subscribe to another user's homepage or to only one or more single tags used by another user. In this way, either all bookmarks or only those annotated with a certain tag were listed (separately) on the own homepage, too, being updated when new bookmarks were saved by the subscribed user.

However, just recently, on 26th of September 2011, *Delicious.com* changed their terminology and slightly modified their functionality [7]. Instead of subscribing to a user's bookmark list, a user now can *follow* another user and instead of subscribing to single tags, a user can follow another user's *stack*. A *stack* allows for thematically grouping bookmarks by creating a stack for an arbitrary topic and assigning URLs to this stack. To give an example, a user can create a stack about "healthy recipes" and, in her sole discretion, then adds web pages to this stack, matching the given topic. Previously, a user achieved the same by adding the tag "healthy recipes" or ("healthy" and "recipes") to each URL belonging to this topic. Thus, the principle is unchanged: A user can choose "friends" to easily have got access to all or a group of bookmarks on a friend's homepage in *Delicious.com*.

Another change in *Delicious.com* is, that the bookmarks from users being followed are not listed anymore on the own homepage (like in the former case when a user subscribed to another user's homepage). Instead, one has to navigate to her homepage by clicking on her name in the list of followed users in order to see her bookmarks.

Matching the *SENSE* Data Model

Obviously, users and tags in *Delicious.com* match our data model as given and most relations between these entities apply in a straight forward way, too. Therefore, we only refer to mappings being non-obvious in *Delicious.com*.

- **Documents.** The user provided contents in *Delicious.com* are *bookmarks* and, thus, correspond to the *documents* in our Data Model.
- (social) **Friendship Relation.** Naturally, the subscription or following activity of users coincide with the friendship relation as defined in our data model: If a user U subscribes to / follows another user U_f or subscribes to one of her tags / follows another user's stack, U_f is considered a social friend of U.

  At the time we crawled *Delicious.com*, only the *subscription* functionality existed. Hence, in our dataset two users participate in a friendship relation when one user subscribed to the other user or to at least one of the other user's tags.
- **Document-Document Relation.** Bookmarks are links to web pages which include hyperlinks and, thus, the document-document relation is inherently defined on the hyperlink graph of web pages. However, as the set of web pages harvested by following the bookmarks from *Delicious.com* did not exhibit a reasonable amount of hyperlinks among them, we did not consider a document-document relation in our studies on *Delicious.com*.

The dataset from *Delicious.com* was retrieved by subsequent crawls and, thus, varies with different experiments. Therefore, the details about the employed dataset are given together with the description of the respective experimental setup in Section 5.4 and 6.4.

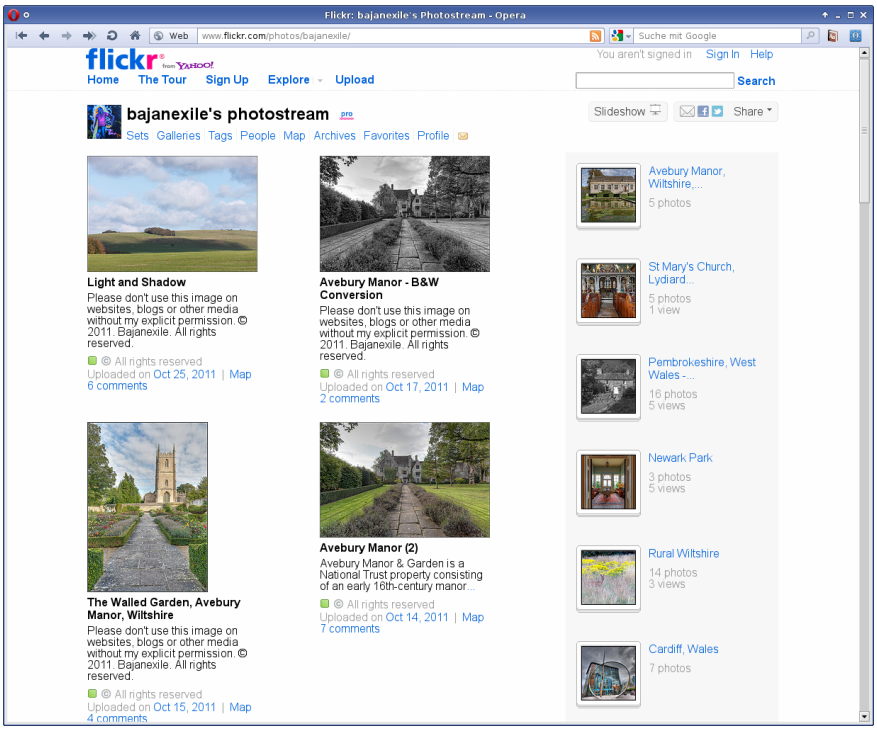
3.2.2 *Flickr.com*

The image and photo hosting service *Flickr.com* [5] is a huge social tagging network for sharing and embedding personal images and photographs with over 50 Million users and 6 Billion images (as of August 2011) [4, 3].

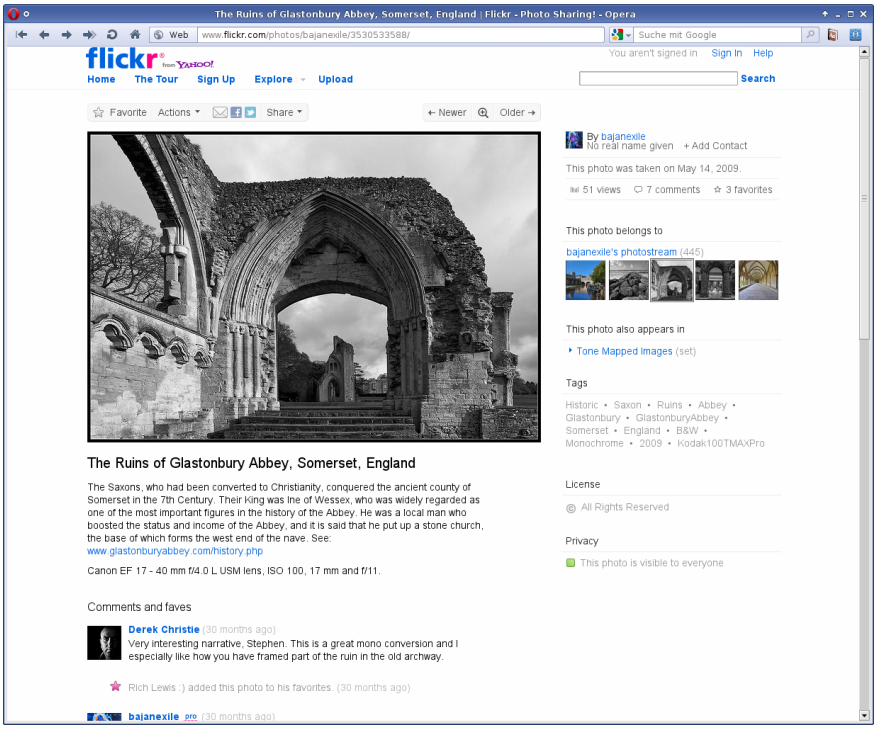
Typically, a user in *Flickr.com* uploads photos to her *Flickr.com* homepage where they are stored in a so called *photostream*. A photostream is basically the visual arrangement of all photos being uploaded by a user. Usually, uploaded photos are taken by the user herself and, thus, we consider a user to be the owner of all photos on her homepage in *Flickr.com*. The layout of the photostream can be configured to display photos in one or more columns and, if present, to list all *sets* of images on the left hand side of the homepage. A *set* is a way to thematically group photos. For this, a user can add a photo in one or more *sets*, arbitrarily named according to the user's own intentions. Figure 4a shows an example of a user's homepage in *Flickr.com*.

Other than by sets, a user can organise photos in her photostream by annotating them with one or more tags. The entirety assigned tags can be visualised as a complete list or, for the 150 most frequently used tags, as a *tag cloud*. A tag cloud shows the annotations in alphabetic order, arranged in an unstructured, continuous text flow but in different font sizes, depending on how frequent a tag has been used. The more frequently a tag is applied to different photos, the bigger its font size relative to all other tags in the cloud. In this way, the most popular tags of a user are perceived first among all others. By clicking on a tag, only the pictures annotated with that tag are shown in the user's photostream. Eventually, the same is achieved for one or more tags entered in a search bar on the top of the user's homepage: the photostream is limited to photos only matching the requested tags.

By default, photos uploaded to *Flickr.com* are public and can be viewed by all visitors of a user's homepage but permissions also can individually be altered on each photo. A public photo can easily be discovered by searching globally in *Flickr.com* with keyword queries matching associated tags or by choosing a tag in the global tag



(a) Example for user homepage in *Flickr.com*



(b) Page in *Flickr.com*for a single Photo

Figure 4: Example screenshots of *Flickr.com*

cloud of recent or most popular tags used in the network. Figure 4b is a screenshot of a *photo page*, showing a picture from a user's photostream. Aside from one or more tags, the owner can assign a title and a textual description to a photo which then is displayed on the same page. Moreover, a photo page lists other users in *Flickr.com* who added the shown photo to their favourites and users who added a comment on the page.

The description and title can only be changed by the user who uploaded a photo. However, by default, comments on a photo can be placed by all *Flickr.com* users and tags can be added by the owner and all her *contacts*, although, permissions for all users can be granted, too, or can be withdrawn for contacts. At any time, a user can add another user as a contact and, if wanted, marking her additionally as *friend* or *family member* to enable a contact to have got access to non-public photos, too.

Matching the *SENSE* Data Model

As with *Delicious.com*, users and tags in *Flickr.com* immediately match our data model and these entities also apply to most introduced relations as is. Hence, we only refer to mappings being non-obvious in *Flickr.com*.

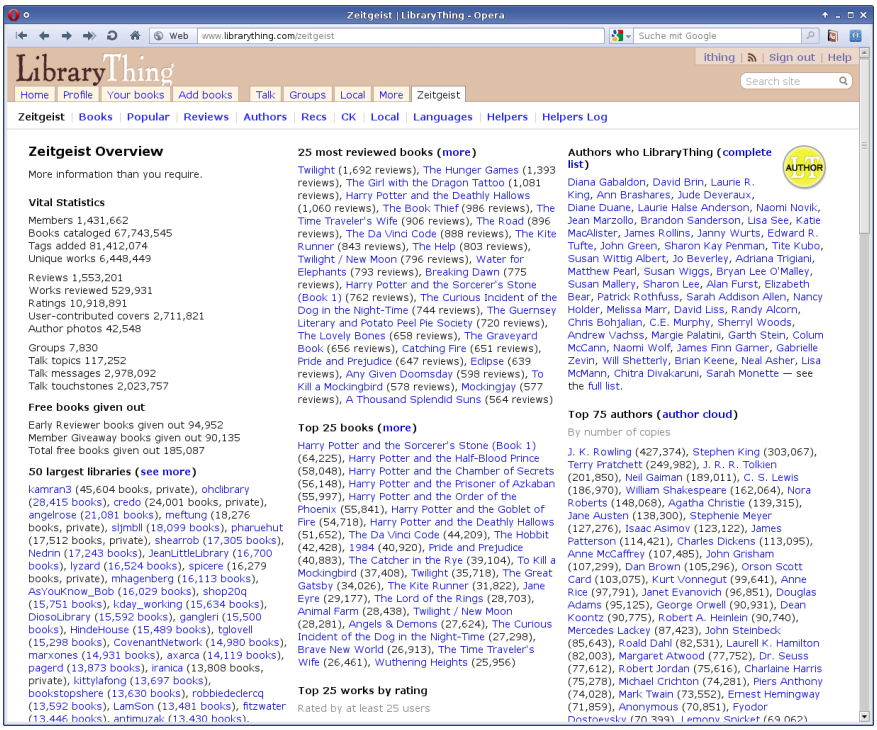
- **Documents.** The user provided contents in *Flickr.com* are typically self-taken *photos*. Hence, pictures and photos in *Flickr.com* correspond to the *documents* in our data model.
- (social) **Friendship Relation.** Naturally, all contacts of a user match the (social) friendship relation in our data model. However, since we discovered that user contacts are mainly known person in real life (e.g. family or friends), we decided to additionally consider a user to be a friend of someone when the user posted a comment on one of her photos: A user U_f is a (social) friend of a user U if U_f has been chosen as contact or if U posted a comment on one of U_f's photos.
- **Document-Document Relation.** Since there is no trivial mapping on the basis of the binary data of photos only, we do not consider a document-document relation in our experimental studies on *Flickr.com*.

As with *Delicious.com*, the dataset from *Flickr.com* was also retrieved by subsequent crawls and, thus, varies with different experiments, too. Therefore, the details about the employed dataset are given together with the description of the respective experimental setup in Section 5.4 and 6.4.

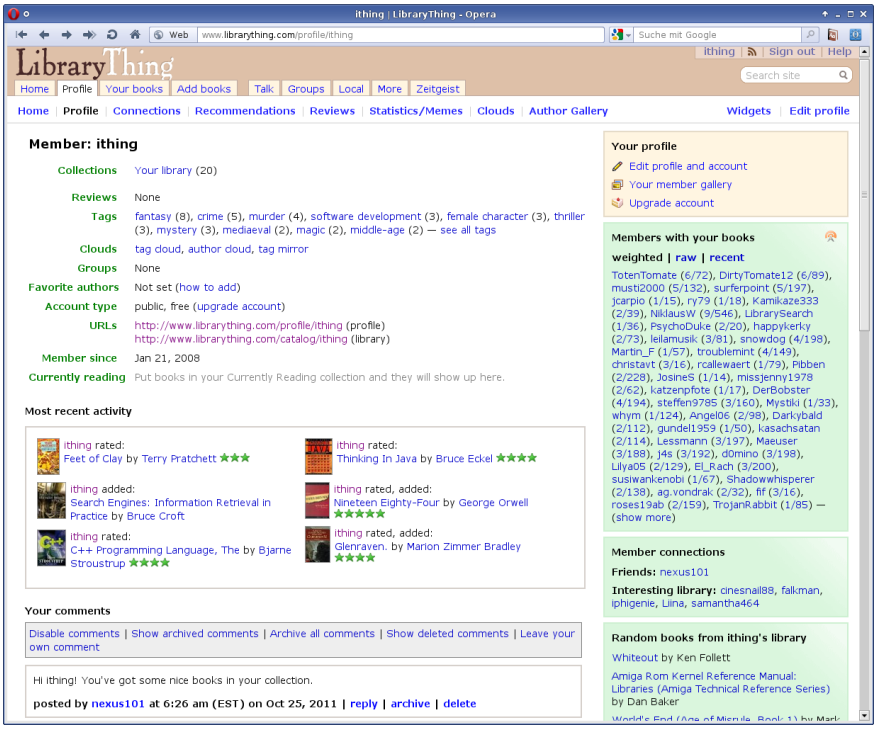
3.2.3 *LibraryThing.com*

The service provided by the social tagging network *LibraryThing.com* [6] allows its over 1.4 million users [9] to store and share information about books by cataloguing them in personal, virtual libraries. In total, over 69 million books [9] have already been catalogued by all users in the network (as of January 2012).

By registering with *LibraryThing.com*, a user can start creating her personal *library* by cataloguing own books or books she read and knows about. For this, the user can enter the title or an ISBN number of a book in *LibraryThing.com*, which in turn queries other web pages to retrieve possible candidates of books matching the user input. After the selection of a book from the suggested candidates, the user can add the book to its own library and enrich the corresponding book entry with several meta information. To categorise or organise her books, the user can annotate them with tags, e.g. allowing her



(a) Zeitgeist Overview of *LibraryThing.com*



(b) User Profile Page in *LibraryThing.com*

Figure 5: Example screenshots of *LibraryThing.com*

to list only the books in her library that match certain tags. Additionally, descriptions, comments and reviews can be written and attached by a user to book entries in her library. Moreover, a user can rate books with up to five stars, enabling her to quickly express and show visitors of her library if she likes a book or not. By default, books listed in a library are public to all visitors but they also can individually be declared as private such that no one but the creator of a library can see the corresponding book entries.

An example for a user's library has already been provided in Figure 1 of Section 2. It shows that in addition to the meta data (e.g. author, comments, tags, etc.) associated to each book, *LibraryThing.com* lists how many other users have included the same book in their libraries and how man users have written reviews for a book. By clicking on these numbers, one can navigate to the libraries of those users and read their reviews. *LibraryThing.com* offers several ways to find books or users who catalogued potentially interesting books in their library. Among other things, *LibraryThing.com* provides a web page called *Zeitgeist Overview*, see Figure 5a, which outlines the *zeitgeist* for books and users in the network. For example, it lists the users with the largest libraries or the books most often reviewed or catalogued in libraries, the most popular tags used and much more related information.

For interaction and establishing social connections, each user has got a profile page in *LibraryThing.com* which summarises available information about her and her library. Figure 5b shows the web page of an example profile. The user profile enables visitors to find out about the tags used on books in the user's library, from which authors the catalogued books are or about the recent activities of a user. To mingle with other members of *LibraryThing.com*, a user can create or join groups about any kind of topic in order to discuss with other users. Moreover, *LibraryThing.com* lists those users in the network who catalogued the same books and allows to write comments on a members profile page. Finally, interesting libraries that have been discovered in *LibraryThing.com* can be remembered on the profile page or the creators of libraries can be considered as interesting contacts (– Note: at the time we crawled *LibraryThing.com*, the feature of remembering other users as *contacts* was not yet available). Friends from real life who are members in *LibraryThing.com* as well or users in the network who have become real friends can also be listed on a user profile page, allowing users to easily stay in touch in *LibraryThing.com*.

Matching the *SENSE* Data Model

As in the previous cases, users and tags in *LibraryThing.com* immediately match our data model and also apply to most introduced relations of our data model. Hence, we only refer to mappings being non-obvious in *LibraryThing.com*.

- **Documents.** The user provided contents in *LibraryThing.com* are the books catalogued in their libraries and the meta data applied to them. Hence, the book entries in *LibraryThing.com* correspond to the *documents* in our data model.
- (social) **Friendship Relation.** Naturally, all member contacts like *friends*, *contacts* and users with *interesting libraries* listed in a user profile match the (social) friendship relation of our data model.

  Since the *contacts* feature in *LibraryThing.com* was not yet present at the time we crawled the social tagging network, only friends and users with interesting libraries*interesting library* were considered for the social friendship relation in

our dataset of *LibraryThing.com*: A user U_f is a (social) friend of a user U if U_f has been chosen as friend or if U regards U_f's library as interesting.

- **Document-Document Relation.** Although a document-document relation could be defined on the available textual context (e.g. reviews) or by taking the authors of books into account, there is no direct linkage between books. Therefore, we refrain from defining such a relation and did not consider document-document relations for our experiments on *LibraryThing.com*.

As with *Delicious.com* and *Flickr.com*, the dataset from *LibraryThing.com* was retrieved by subsequent crawls and, thus, varies with different experiments, too. Therefore, the details about the employed dataset are given together with the description of the respective experimental setup in Section 5.4 and 6.4.

3.3 Design Decisions

In this section, we discuss the design decisions made for representing the datasets retrieved from social tagging networks matching our data model and for implementing and integrating the model into the general framework of *SENSE*. The design decisions made are fundamental for both algorithms SOCIALMERGE and CONTEXTMERGE in the following Section 5 and 6, respectively. To this end, we introduce at this point also the concept of inverted lists.

3.3.1 Relational Database Schemas

As defined in Section 3.1.3 by our data model, *Tagging* is a ternary relation between users, tags and documents. For representing our data model as a graph with tags, documents and users being the nodes, the ternary relation needs to be broken into binary relations such that the edges in the graph can be defined between pairs of nodes accordingly. However, as already mentioned in Section 3.1.3, this is not possible without losing information.

One might argue that tags should be considered as attributes of links between users and documents instead of entities of their own, as in the social content graph suggested in [14]. However, in this case, either one would have to completely abandon the notion of tag similarity within the graph model (since edges between edge attributes are not well defined in graphs) or one would need to consider tag similarities separated from the graph model.

In our view, the notion of tag similarity is a particularly intriguing and promising asset for a powerful searching and ranking model. Hence, abandoning this part of our model is not desirable. However, we also want to preserve the ternary relation *user-tag-document* within one unified data model without losing any information. Therefore, we made the design decision of not using a graph representation. Instead, we implemented our data model based on a relational database system, using database tables for representing each relation. See Figure 2 in Section 3.1 for an example illustration of our data model.

3.3.2 Inverted Lists

The relations defined by our data model in Section 3.1 can be implemented in database tables as *(key,value)*-pairs. The *key* is a tuple of entities (– in our case, the users, tags or documents) which are involved in a relation and the *value* corresponds to the weight

defined by the respective relation (– in our case, the friendship strength, tag similarity, score, etc).

By creating an appropriate index, the entries in such database tables can be sorted in descending order of their values. In this way, the database tables follow the semantic of inverted lists where entries are not sorted by their keys but inversely, according to the respective key's value. Hence, we will use the notion of *inverted lists* as a synonym for the database tables implementing the same functionality.

By sorting database tables in this inverted way, entities can be sequentially fetched in descending order of their values without the need for way more expensive random accesses to the database and, thus, the tables can be used like inverted lists with typical *top-k* query processing algorithms.

Top-k query processing is a fundamental cornerstone of ranked retrieval of documents and many other modern applications. Ideally, an efficient query processor would not read the entire input (i.e. all *(key,value)*-pairs from the underlying relations) but should rather find ways of early termination when the k best results can be safely determined, using techniques like priority queues, bounds for partially computed aggregation values, pruning intermediate results, etc. These issues have been intensively researched in recent years (e.g. [32, 35, 44, 54, 61, 91, 98, 125]) and are well-understood.

Most *top-k* algorithms scan, i.e. sequentially read, inverted lists and aggregate precomputed per-term or per-dimension scores into in-memory "accumulators", one for each candidate document. The optimisations in the IR literature aim to limit the number of accumulators and the scan depth on the index lists in order to terminate the algorithm as early as possible. This involves a variety of heuristics for pruning potential result candidates and stopping some or all of the index list traversals as early as possible, ideally after having seen only short prefixes of the potentially very long lists. For this, it is often beneficial that the entries in inverted lists are kept in descending order of score values rather than being sorted by document identifiers.

Our algorithms SOCIALMERGE and CONTEXTMERGE operate on index structures corresponding to inverted lists, too, since both generally fall into the well-established framework of so-called threshold algorithms (TA) as well. They depend on impact-sorted inverted lists for efficient *top-k* query processing and require that score aggregation functions are monotonic (e.g. a weighted summation). Eventually, we employ variants of Fagin's Threshold Algorithm (TA) [54] with flexible scheduling of list scans ([124, 20]). Hence, the design decision to cast our data model into inverted lists has been crucial for both of our algorithms. The details about SOCIALMERGE and CONTEXTMERGE are given in Chapter 5 and 6, respectively.

4 Problem Statement

In order to introduce our SOCIALMERGE and CONTEXTMERGE algorithms and the scoring models used, we first formalise the notion of a query. In line with the free-text tagging of social networks, we define a *query* as follows.

Definition 4.1 (Query q_U)**.** *A query $q_U = \{t_0, \ldots, t_{n-1}\}$ is a set of query tags issued by a query initiator U to the social network. A query tag is a keyword, corresponding to a tag t_i used by some user in the network to annotate a document.* ∎

The result of a query is then defined as follows:

Definition 4.2 (Query Result R_U)**.** *The result R_U of a query q_U is a ranked list of documents, annotated by at least one of the query tags $t_i \in q_U$ or a tag t_e similar to t_i which is determined during the query processing by expanding a tag $t_i \in q_U$ to t_e. The result list is ordered according to a query-specific document score.* ∎

In particular, a query-specific document score enables *top-k* query processing for efficiently retrieving the k documents with the highest document scores in regard to a query. The definition is perfectly in line with the current querying model of popular search engines. However, in contrast to those search engines, the document scores used in our model—details are given in Section 5.1 and 6.2—also contain a social component: the, by definition, query-specific content-based score of a document is additionally *user-specific*, i.e. it depends on the social context of the query initiator.

Even though commercial search engines offer similar personalisation approaches, social tagging networks are the natural habitat to further explore and improve this idea since having the additional asset of knowing friendship relations and the friends' tagging behaviour. Moreover, by considering these additional assets for computing user-specific query results, queries become high-dimensional and traditional IR text-retrieval methods are bound to fail. The dimensions of a query are defined as follows:

Definition 4.3 (Query Dimensions)**.** *The involved components for computing the final score of a document with respect to a query are called the* query dimensions. ∎

Accordingly, high-dimensional queries involve a high number of components during the result computation.

Unlike in standard text retrieval, the dimensions of a query, when considering friendship relations in social tagging networks and in presence of our proposed scoring methods (see Section 5.1 and 6.2), are not only the tags in a query q_U. Instead, for each query tag t_i, the score of a document is additionally influenced by a user-specific score. Assuming m users in the system and a query with n tags, a query therefore has got $m \cdot n$ dimensions. If we additionally consider expansions of tags, the number of query dimensions will again increase a lot. Such high-dimensional queries cannot be efficiently handled by the existing variants of threshold algorithms (TA) for standard text retrieval, since usually, for each dimension a corresponding inverted list has to be precomputed and is eventually involved in the query processing.

In the following chapters, we introduce two different algorithms for efficiently retrieving content-based and user-specific query results for high-dimensional queries from social tagging networks. Both algorithms are based on the data model introduced in Section 3.1.

5 SOCIALMERGE Algorithm

Our first algorithm developed for the *SENSE* framework is called SOCIALMERGE. Subsequently, we introduce the associated scoring model followed by details about the query processing.

5.1 Scoring Model

The scoring model used with our SOCIALMERGE algorithm is based on the relations defined by our data model in Section 3.1. The score for a document depends on the

tags that have been used to annotate the document and on the users who have tagged the document. In this scoring model, we perform *semantic expansions* by considering tags that are similar to the keywords appearing in a query, and *social expansion* by preferring documents tagged by close friends. More formally, let be

$$q_U = \{t_0, \ldots, t_{n-1}\}$$

a query with query tags $t_0, \ldots, t_{n-1}$. We define the *social score* $ssc(q_U, d)$ of a document d with respect to a query q_U initiated by user U in the following way:

Definition 5.1 (Social Score $ssc(q_U, d)$)**.**

$$ssc(q_U, d) = \sum_{t_i \in q_U} sts(t_i, d, U)$$

■

where $sts(t_i, d, U)$ is the *single tag score* of a document d with respect to a query tag $t_i \in q_U$ and the user U who issued the query.

We define the single tag score $sts(t_i, d, U)$ as follows:

Definition 5.2 (Single Tag Score $sts(t_i, d, U)$)**.**

$$\begin{aligned} sts(t_i, d, U) = DR(d) \quad \times & \sum_{U_f \in \text{FLIST}(U)} s_f(U, U_f) \\ & \times \max_{t' \in \text{SIMTAGS}(t_i)} \{tsim(t', t_i) \cdot s_d(U_f, t', d)\} \\ & \times \quad UR(U_f) \end{aligned}$$

■

Before defining each of the components used in the definition of the single tag score $sts(t_i, d, U)$, we introduce the notion of friendship in regard to the scoring model used with our SOCIALMERGE algorithm.

With this scoring model, we use only the user-user relation

$$Friendship(U_1, U_2, type = social, s_f)$$

of type $social$ as presented in Section 3.1.2 and abstractly given in Definition 3.1. We implement the *social friendship* relation in our scoring model based on the friendship graph in social tagging networks (see Definition 3.2) by considering the shortest path distances of users in the graph. Since we are using a different notion of shortest path in later chapters, we explicitly define the shortest path for our SOCIALMERGE algorithm in the following (obvious) way:

Definition 5.3 (Shortest Path)**.** *Let be G the directed,* unweighted *friendship graph of a social tagging network and U and U_f two different users in G. The length of a path in G leading from U to U_f is equal to its number of edges. A path of shortest length leading from U to U_f is called a* shortest path *from U to U_f.* ■

Based on this definition, we now can define the distance between two users in the friendship graph.

Definition 5.4 (Distance of U_f wrt. U)**.** *Let be G the directed,* unweighted *friendship graph of a social tagging network with U and U_f being two different users in G. The distance $dist(U, U_f)$ of U_f with respect to U is equal to length of the shortest path leading from U to U_f in G if such a path exists, otherwise ∞, i.e.*

$$dist(U, U_f) = \begin{cases} 0 & \text{if } U_f = U \\ \infty & \text{if } \nexists \text{ path from } U \text{ to } U_f \\ \#\text{edges of } \pi & \text{if } \pi \text{ is a shortest path from } U \text{ to } U_f \end{cases}$$

■

We consider U_f as a *social friend* of U if a path from U to U_f in the graph exists. Furthermore, we denote with FLIST(U) the list of all *social friends* of a user U. Formally, we define the social friendship of two users as follows:

Definition 5.5 (Social Friendship)**.** *A user $U_f \neq U$ is a (social) friend of U if and only if there is a path from U to U_f in the friendship graph of a social network, i.e.*

$$\begin{aligned} U_f \text{ is a friend of } U &\iff U_f \in \text{FLIST}(U) \\ &\iff 0 < dist(U, U_f) < \infty \end{aligned}$$

We denote with FLIST*(U) the list of all transitive friends of U.* ■

The measure in our scoring model for the social friendship strength $s(U, U_f)$ of a user U_f in regard to a user U favours users that are closer to U in the friendship graph of the social network. The intuition is, that if U issues a query, the results from users close to U in the friendship graph are preferred because it is likely that a user is more interested in results from her friends or that she trusts them more than unknown users. We define the friendship strength as follows:

Definition 5.6 (Friendship Strength $s_f(U, U_f)$)**.**

$$s_f(U, U_f) = \begin{cases} \frac{1}{dist(U,U_f)^2} & \text{if } U_f \in \text{FLIST}(U) \\ \frac{1}{(|\mathcal{U}|)^2} & \text{if } U_f \notin \text{FLIST}(U) \text{ and } U_f \neq U \\ 0 & \text{if } U_f == U \end{cases}$$

where $|\mathcal{U}|$ is the number of all users in the network. ■

The friendship strength of a user U with respect to herself is set to 0, while the friendship strength for a social friend U_f is equal to the inverse of the square of the shortest path distance from U to U_f. If there is no path between two users in the friendship graph of a social network, the friendship strength is set to a constant which equals to the inverse of the square of the longest possible distance, that is, a path leading over all users.

After having cast the friendship relation defined in our data model into our soring model, we now define the remaining components of the single tag score $sts(t_i, d, U)$ given in Definition 5.2.

The value of $tsim(t, t')$ corresponds to the tag similarity of tag t' in regard to tag t and actually implements in our scoring model the tag-tag relation

$$TagSimilarity(t_1, t_2, tsim)$$

introduced with our data model in Section 3.1.2.

In our scoring model, the similarity of tags is based on the co-occurrence of tags in the document collection. Formally, it is defined as follows:

Definition 5.7 (Tag Similarity $tsim(t, t')$)**.** *The list of all tags t' which are similar to a tag t is denoted with* SIMTAGS*(t). The similarity of two tags t and t' is computed by the* Dice coefficient *on the set of documents in the social network tagged with both tags, i.e.*

$$tsim(t, t') = \frac{2 \cdot df(t \wedge t')}{df(t) + df(t')}$$

where $df(t \wedge t')$ is the document frequency for both tags t and t', i.e. the number of documents that are tagged with t and t'; $df(t)$ and $df(t)$ is the document frequency for the single tag t and t', respectively. ∎

Note: The most similar tag with respect to a tag t is t itself, i.e. $tsim(t, t) = 1$, and thus, is the entry in SIMTAGS(t) with the highest similarity value.

$UR(U)$ and $DR(d)$ define the rank of a user U in the friendship graph and the rank of a document d in the document graph of a social network, respectively. The user rank weights documents from users with a high reputation stronger, while the document rank generally boosts the single tag score $sts(t_i, d, U)$ for high authoritative documents in the social network. The user or document rank is equal to the *PageRank* [99] score of the respective entity defined by a random walk on the user or document graph, respectively, with a random jump probability $(1 - \epsilon)$ set to 0.15. Formally, the user rank is defined as follows:

Definition 5.8 (User Rank $UR(U)$)**.**

$$UR(U) = \frac{1 - \epsilon}{|\mathcal{U}|} + \epsilon \cdot \sum_{\forall U_i : U \in \text{DIRECTFRIENDS}(U_i)} \frac{UR(U_i)}{|\text{DIRECTFRIENDS}(U_i)|}$$

*where $|\mathcal{U}|$ is the number of all users in the network, $|$*DIRECTFRIENDS*$(U_i)|$ is the number of direct friends of U, i.e. the friends U_f with $dist(U, U_f) = 1$, and $\epsilon = 0.85$ is a damping factor.* ∎

Analogously, the document rank is defined as:

Definition 5.9 (Document Rank $DR(d)$)**.**

$$DR(d) = \frac{1 - \epsilon}{|\mathcal{D}|} + \epsilon \cdot \sum_{\forall d_i : d_i \to d} \frac{DR(d_i)}{outdegree(d_i)}$$

where $|\mathcal{D}|$ is the number of all documents in the network, $outdegree(d_i)$ is the number of outgoing links from d_i and $\epsilon = 0.85$ is a damping factor. ∎

Finally, the document score used in Definition 5.2 of the single tag score corresponds to the ternary user-tag-document relation

$$Tagging(U, t, d, score)$$

introduced with our data model in Section 3.1.3.

The score computation of a document with respect to a certain tag is based on a user-specific BM25 [105] formula. Its definition looks as follows:

Definition 5.10 (Document Score $s_d(U,t,d)$)**.** *The score of a document d that is tagged with a tag t by a user U is defined as*

$$s_d(U,t,d) = \frac{k_1 + tf_U(t,d)}{K + tf_U(t,d)} \cdot \log \frac{N_U - df_U(t) + 0.2}{df_U(t) + 0.5}$$

where

$$K = \left((1-b) + b \frac{length_U(d)}{avg_{(d' \text{ tagged by } U)}\{length_U(d')\}} \right)$$

and k_1 and b are constants, set to $k_1 = 1.2$ and $b = 0.5$. ∎

The value of $tf_U(t,d)$ corresponds to the number of times U tagged d with t, $df_U(t)$ corresponds to the number of times U tagged any document with t, N_U is equal to the total number of documents tagged by U, and $length_U(d)$ corresponds to the number of tags given to d by U.

Notes on remaining entity relations

By summing up the document score $s_d(U',t,d)$ over all users U' in the social network, the resulting value is independent of any user relation in regard to a querying user U. Hence, the aggregated document scores for a document d and tag t from the entire social tagging network implements in our scoring model the document-tag relation

$$Content(d,t,score)$$

as defined by our data model in Section 3.1. We make use of it in Section 5.3 for enabling a *semantic search* strategy (see Definition 5.12).

The scoring model used in our SOCIALMERGE algorithm neither implements the document-document relation $Linkage(d_1,d_2,w)$ (see Section 3.1.3) nor the user-document relation $Rating(U,d,rating)$ (see Section 3.1.2) as defined by our data model in Section 3.1. The reason is that the datasets crawled from real world social tagging networks (see Section 3.2) and used in our experimental evaluation presented in Section 5.4, do not exhibit such relations to the full extend or only could be harvested on a limited scale.

5.2 Query Processing

Given the scoring model presented in the previous Section 5.1, a naïve application of *top-k* threshold algorithms (TA), commonly used for text retrieval, is bound to fail as queries are high-dimensional (see Section 4 and Definition 4.3).

Our new SOCIALMERGE algorithm is designed for handling high-dimensional queries in social tagging networks. It explores the user-specific dimensions very carefully, opens additional dimensions on demand and only when it is clear that they will contribute to the score of the final results. Hence, there is no need to precompute inverted lists in *all* dimensions with our algorithm like in standard *top-k* threshold algorithms.

5.2.1 Preprocessing

Nevertheless, our SOCIALMERGE algorithm can and does make use of precomputed inverted lists (see Section 3.3 for more information about inverted lists) as most of the components used to compute the social score $ssc(q_U, d)$ (see Definition 5.1) *can* be precomputed and stored in lists in descending order of their score values. Our query processing algorithm eventually takes advantage of them for efficiently executing *top-k* queries.

We maintain only three different kinds of inverted lists which are accessed sequentially in descending order of scores:

- USERDOCS(U, t) contains for a user U and a tag t entries (d, w_d) with documents d tagged by U with tag t in descending order of their weighted document score $w_d = s_d(d, t) \cdot DR(d)$.
- FLIST(U) contains for a user U entries (U_f, w_f) with U's social friends U_f in descending order of their weighted friendship strengths $w_f = s_f(U, U_f) \cdot UR(U_f)$.
- SIMTAGS(t) contains for a tag t entries (t', sim) with all similar tags t' in descending order of $sim = tsim(t, t')$.

Note: For the reason of an unambiguous definition, we actually would need to include in the notion of w_d the tag t and user U used in USERDOCS(U, t), e.g. $w_d(U, t)$, and similarly, we would need to state w_f as $w_f(U, U_f)$ and also could not use an abbreviated notation for $tsim(t, t')$. However, to avoid further notational complexity, we will use the simplified versions w_d, w_f and sim as given above when their usage is clear from the respective context.

5.2.2 Notation

Let be,

- $|\text{FLIST}(U)|$ the number of social friends of a user U
- $pos(U)$ the current read position in FLIST(U) where $pos(U) = 1$ means that one friend has been read and processed from the list.
- $high_d(U, t_i)$ the last value of w_d read from the USERDOCS(U, t_i) list
- $high_f(U)$ the last value of w_f read from the FLIST(U) list
- $high_t(t_i)$ the last value of sim read from the SIMTAGS(t_i) list.
- $L(U_f) = \{\text{USERDOCS}(U_f, t_i), ...\}$ the set of lists USERDOCS(U_f, t_i) of user U_f and any tag t_i that have been opened during the query execution.
- $E(U_f, d) = \{\text{USERDOCS}(U_f, t_i), ...\}$ the set of lists USERDOCS(U_f, t_i) of a user U_f and any tag t_i in which the document d has already been discovered

Note that the values of $high_d(U, t_i)$, $high_f(U)$ and $high_t(t_i)$ are upper bounds for score values of unseen entries in each of the lists as they are sorted in descending order of the score values.

Furthermore, to simplify the description and for a better understanding of the query processing with our SOCIALMERGE algorithm, we assume that:

1. Scores in all lists are normalised to the interval $[0 \dots 1]$.

2. Friendship scores in FLIST(U) are normalised in such a way that the score within one list sums up to 1, i.e. single score values are divided by the sum over all values.

Hence, before the first entry is read from the associated list, $high_d(U, t_i)$, $high_f(U)$ and $high_t(t_i)$ can be initialised with 1.

5.2.3 Operation Mode

We start describing the query processing with our SOCIALMERGE algorithm by first neglecting its tag expansion part and, afterwards, extending the approach by adding tag expansion into the query processing.

Without Tag Expansion

To this end, we focus on computing the results for a simpler social score $ssc'(q_U, d)$ using a single tag score $sts'(t_i, d, U)$ without tag expansion and show later how the processing can be extended towards $ssc(q_U, d)$ by using $sts(t_i, d, U)$, integrating tag expansion. Hence, $sts'(t_i, d, U)$ is defined like $sts(t_i, d, U)$ but without the tag expansion component:

Definition 5.11 (Single Tag Score without Tag expansion $sts'(t_i, d, U)$)**.**

$$sts'(t_i, d, U) = DR(d) \quad \times \sum_{U_f \in \text{FLIST}(U)} s_f(U, U_f) \times s_d(U_f, t_i, d) \times UR(U_f)$$

■

With our SOCIALMERGE algorithm, the query processing starts for a query q_U from user U with query tags $\{t_0, \dots, t_{n-1}\}$ by opening a scan on FLIST(U) and reading its first entry (U_f, w_f). By doing this, $w_f = s_f(U, U_f) \cdot UR(U_f)$ is the highest weighted friendship strength of all friends of U.

Subsequently, the algorithm opens and then scans, i.e. sequentially reads, the lists USERDOCS(U_f, t_i) for each query tag t_i of the previously retrieved best friend U_f. The weighted document scores w_d of documents d read from these lists are multiplied by the weighted friendship strength w_f of U's friend U_f. The friend's document lists are continuously processed until the score contribution by the next best friend U'_f, being currently at the top of the FLIST(U) list, is higher than the contribution from any of the already read USERDOCS(U_f, t_i) lists. The condition to stop reading from the USERDOCS(U_f, t_i) lists of U_f is checked by comparing for any query tags t_i the current $high_d(U_f, t_i)$ values associated with these lists with $high_f(U) \cdot 1.0$, which is the maximal upper bound for scores of documents retrieved from the next best friend.

In the event of the stop condition becomes true, the scan on U's friendship list FLIST(U) reads the next entry (U'_f, w'_f) with the next best friend U'_f and the corresponding weighted friendship strength w'_f, such that, the algorithm can first open and then start reading from U'_f's USERDOCS(U'_f, t_i) lists for all query tags t_i. Afterwards,

```
1:  ssc'(q_U = (t_0, ..., t_{n-1}), d) {
      // Q_T, Q_C: Queue of top-k docs and candidates. Set_F: set of considered friends
2:    Q_C = Q_T = Set_F = {}
      // While candidates could make it into Q_T, enter the loop
3:    WHILE(max{bs(d) | d ∈ Q_C} > {ws(d) | d ∈ Q_T}) {
        // get best friend who is USERDOCS-lists should be opened next
4:      (U_f, w_f) =FLIST(U).next()
5:      high_f(U) = w_f
6:      Set_F = Set_F ∪ {U_f}
7:      ∀t_i ∈ Q : USERDOCS(U_f, t_i).open() ∧ high_d(U_f, t_i) = 1.0 * w_f
        // Read USERDOCS-lists until next friend may have better documents
8:      WHILE(max{high_d(U'_f, t_i) | t_i ∈ {t_0, ..., t_{n-1}} ∧ U'_f ∈ Set_F} ≥ high_f(U)){
          // get best document d_i with score w_i from any friend U'_f ∈ Set_F for all tags t_i
9:        FOR(t_i ∈ {t_0 ... t_{n-1}}) {
10:           (d_i, w_{d_i}) = USERDOCS(t_i, U'_f).next() {
              // weight the document score by the friendship strength
11:           w_{d_i} = w_{d_i} · w_f
12:           high_d(U'_f, t_i) = w_{d_i}
              // compute best and worst scores for d_i
13:           bs(d_i)=BestScoreUpdate(d_i)
14:           ws(d_i)=WorstScoreUpdate(d_i)
              // if d_i is better than some doc in Q_T, replace it and move
              // the previous doc in Q_C, replacing the weakest candidate
15:           IF (ws(d_i) > min{bs(d) | d ∈ Q_T})
16:             Update(Q_T,d_i)
17:           ELSE
                // otherwise, check if d_i is at least a candidate. If so,
                // insert it in Q_C
18:             IF (bs(d_i) > min{ws(d) | d ∈ Q_T} &&
19:                 bs(d_i) > min{ws(d) | d ∈ Q_C})
20:             Update(Q_C,d_i)
21:         }
22:       }
23:   }
      // return the top-k documents
24:   RETURN(Q_T)
25: }
```

Listing 1: SOCIALMERGE framework without tag expansion

among all already opened document lists of the friends seen so far, the one with the potentially highest score contribution for any query tag t_i is chosen to be processed next. The processing continues in this way until again the highest possible score estimation is contributed by the next best friend in U's friendship list FLIST(U), i.e. the score contribution is higher than from any opened USERDOCS(U_f, t_i) lists for any query tag t_i and already considered friends U_f. The scan on the list FLIST(U) can be limited by considering a certain threshold for the friendship strength of a next friend.

The pseudocode of this general framework of our SOCIALMERGE algorithm is given in Listing 1.

When not yet all dimensions are evaluated in order to compute the final score of a document that has been read from a friend's document list USERDOCS(U_f,t_i), upper and lower bounds are computed to estimate the best score $bs(d)$ and the worst score $ws(d)$ of a document d. The worst score of a document is easily computed by considering the score contributions from not yet known dimensions as 0. Unfortunately, it is not that trivial to estimate the document's best score.

Following standard procedures in threshold algorithms, to compute the best score of a document, the scores of already evaluated dimensions, resulting in the current worst score of the document, and upper bounds for scores of not yet discovered dimensions are accumulated. For this, the scores in the latter case must be assumed to be the maximum scores that could be achieved for a document in the remaining undiscovered dimensions. Hence, they are set to the highest known values, the ones read during the last access to the corresponding lists in those dimensions. With our data model and the precomputed inverted lists introduced previously in this section the estimation for the best scores can be done in as follows:

Given that U has got $|\text{FLIST}(U)|$ friends out of which $pos(U)$ have already been processed and that to some friend U_f belong at most n USERDOCS(U_f, t_i) lists for the query tags $t_0, \ldots, t_{n-1}$, then a total of

$$|\mathcal{L}| = (|\text{FLIST}(U)| - pos(U)) \cdot n$$

lists have not yet been opened. With $w_f = s_f(U, U_f) \cdot UR(U_f)$ being the weighted friendship strength of U_f in regard to U and with $ws(d)$ is the worst score of a document d, and, by using the notation introduced in Section 5.2.2, which denotes $E(U_f, d)$ as the set of document lists of U_f in which d has been discovered already, and $L(U_f)$ as the set of all currently opened document lists of U_f for the query tags t_0 to t_{n-1}, we can compute the best score for the document by:

$$\begin{aligned} bs(d) = \; & ws(d) && (1)\\ & + high_f(U) \cdot |\mathcal{L}| \cdot 1.0 && (2)\\ & + \sum_{\substack{\forall U_f \wedge list = \text{USERDOCS}(U_f, t_i):\\ list \in \{L(U_f) \setminus E(U_f, d)\}}} w_f \cdot high_d(U_f, t_i) && (3) \end{aligned}$$

The worst score $ws(d)$ in (1) is equal to the score contribution from the document lists of all processed friends of U in which d has already been discovered. The computation in (2) is equal to the maximal score contribution from the at most $|\mathcal{L}|$ not yet opened document lists of all not yet discovered friends U'_f in FLIST(U). The weighted friendship strengths of those friends U'_f cannot be greater than $high_f(U)$ and the maximal weighted document score in a list USERDOCS(U'_f,t_i) of U'_f is 1.0. Finally, the maximal possible score contribution from lists of known friends in which the document d

has not yet been seen is computed in (3). Since the friend U_f is already known, her weighted friendship strength w_f is also known and can be multiplied with the maximal weighted document score being possible for d in lists USERDOCS(U_f,t_i) where the document has not yet been seen. The weighed document score of d cannot be greater than $high_d(U_f, t_i)$ in such a list.

Like all threshold algorithms, our SOCIALMERGE algorithm terminates as soon as the best score for any of the candidates that do not belong to the current *top-k* results cannot exceed anymore the worst score of any of the current *top-k* documents. To this end, depending on its worst and best scores, a document is kept in one of two distinguished priority queues. A queue Q_T for documents belonging to the current or final *top-k* results and a queue Q_C for candidates that still could displace a document from the Q_T queue.

In Listing 1, the function $Update(Q, d)$ inserts or replaces a document d in queue Q and moves the replaced document d' out of the queue if appropriate. Is a document moved out of Q_T, it will be inserted in Q_C which again could replace a weaker candidate there. Documents moved out of queue Q_C can't make it into the *top-k* queue anymore and, therefore, are simply pruned.

Besides these round-robin-style sequential accesses (SA) to users' document lists for all query tags t_i, our SOCIALMERGE implementation can also perform random accesses (RA) to the index lists.

As RA are usually a lot more expensive than SA (in the order of 100 to 1,000 times for real systems), they have to be carefully selected and scheduled to avoid any unnecessary work. Our scheduling for RA follows the LAST heuristics from [20], i.e. our algorithm performs only SA until the estimated cost to perform *all* RA to remaining (and potentially not yet seen) candidates is at most as high as the cost for all SA done so far. The cost for not yet seen candidates is estimated by assuming a *virtual* document d_v, representing any unseen document appearing right at the front of the not yet seen part of all document lists. Our estimation for the number of RA is quite crude as we just sum, for all candidates (including d_v), the number of dimensions that have not yet been evaluated.

Including Tag Expansion

While the previously described framework of our SOCIALMERGE algorithm has explored the user dimensions, the computation of the single tag score $s'_t(t_i, d, U)$ has set aside tag expansion, i.e. the computation does not explore tags that are similar to the actual query tags. For enabling tag expansion in our framework, we leverage the *Incremental Merge Algorithm* [124] for efficient query expansion in text retrieval.

The framework shown in Listing 1 is unchanged for our final version that includes expansion of tags and essentially works like explained above. The only difference is that it does not immediately open for all query tags t_i all lists USERDOCS(U_f,t_i) for reading when a new friend is processed due to the scan on FLIST(U). Instead, it opens a *meta index list* META(U_f, t_i) that incrementally opens and merges the document lists USERDOCS(U_f,t_x) for tags t_x that are similar to t_i. The scores retrieved from these lists for a document d are weighted by the tag similarity $tsim(t_i, t_x)$ and only the first occurrence of a document is considered.

In more detail, the tag expansion of a tag t_i by reading from its meta index list

```
1:  cache = {}
2:  META(U_f, t_i).next() {
      // if similarity of next tag in SIMTAGS(t_i) is above some threshold..
3:    IF(high_t(t_i) > tsim-Threshold) {
      // keep opening new USERDOCS-list for the tag with next best tsim
      // when that could contain potentially a doc with highest score
4:      WHILE(high_t(t_i) > max{w_d|(d, w_d, t) ∈ cache}) {
          // get next best tag to open associated USERDOCS-list..
5:        (t_x, sim_x)=SIMTAGS(t_i).next()
6:        high_t(t_i) = sim_x
          // .. which may contain doc with potentially highest score ..
7:        (d, w_d)=USERDOCS(U_f, t_x).next()
8:        high_d(U_f, t_x) = w_d
          // .. even when weighted by tag similarity.
9:        w_d = w_d · sim_x
          // put doc into the cache for later checkout.
10:       do_cache(d, w_d, t_x)
11:     }
12:   }
      // get the doc with maximum score from cache
13:   (d, w_d, t_x) = max_{w_d}{(d, w_d, t)|(d, w_d, t) ∈ cache}
      // put the next doc from the associated USERDOCS-list in cache
14:   (d', w'_d) = USERDOCS(U_f, t_x).next()
15:   high_d(U_f, t_x) = w'_d
16:   w'_d = w'_d · tsim(t_i, t_x)
17:   do_cache(d, w_d, t_x)
      // the maximum of all high bounds for all opened USERDOCS-lists is the
      // current high bound for docs from U_f and query tag t_i
18:   high_t(U_f, t_i) = max{high_d(U_f, t_x) | USERDOCS(U_f, t_x)-list ∈ L(U_f)}
19:   RETURN((d, w_d))
20: }
```

Listing 2: Incremental merge algorithm with META(U_f, t_i) lists to include tag expansion in SOCIALMERGE

META(U_f, t_i) works as follows:

For a query tag t_i and a friend U_f, the meta index list META(U_f, t_i) is initialised by opening a scan on SIMTAGS(t_i) and reading its first entry $(t_x, tsim_x)$ with t_x corresponding to the most similar tag with respect to t_i and $tsim_x$ is equal to the tag similarity $tsim_x = tsim(t_i, t_x)$. In fact, after initially accessing the list SIMTAGS(t_i), t_x is equal to t_i since t_i is the first entry in the list with a similarity of 1.0. Afterwards, the corresponding document list USERDOCS(U_f, t_x) for tag t_x of the user U_f is opened for reading and the top most document d and its score w_d weighted by $tsim_x$ is read into a buffer. The computed score $w_d \cdot tsim_x$ of d serves eventually as $high_d(U_f, t_x)$ bound for the USERDOCS(U_f, t_i) list. The bound is necessary as usual to provide an accurate value for the next score contribution that can be retrieved from the list.

As the processing continues and pairs (d, w_d) of documents and scores are retrieved from META(U_f, t_i) by the main loop of our SOCIALMERGE algorithm, the documents and scores read from the meta index lists are always fetched from the (already opened) USERDOCS(U_f, t_x) list which currently can contribute a document with the highest score value. A new list is opened only, i.e. the tag expansion eventually happens, when a document from the USERDOCS(U_f, t'_x) list for the next similar tag t'_x could achieve a higher score value as any document of already opened lists USERDOCS(U_f, t_x) for all already expanded tags t_x. The next similar tag t'_x is determined by continuing reading the next entry from the list of similar tags SIMTAGS(t_i) for the original query tag t_i. This incremental merge algorithm uses a threshold for the similarity of related tags and stops when the similarity of the next tag falls below it.

As a consequence, it is possible that a document, tagged by a friend U_f with a tag t_x but not with the initial query tag t_i receives a higher document score than any other document actually tagged with t_i by users in the social network. It happens when the friendship strength and the tag similarity are high enough to achieve a better document score than any from other user's document lists USERDOCS(U'_f, t_i) for the query tag t_i.

The upper score bound $high_d(U_f, t_i)$ for the meta index lists META(U_f, t_i) used in the main loop of the SOCIALMERGE algorithm is the maximum over all high score bounds $high_d(U_f, t_x)$ of currently opened lists for expanded tags t_x and the upper bound $high_t(t_i)$ of the SIMTAGS(t_i) list. As all upper bounds are exact, it is guaranteed that the meta index lists always delivers entries in descending order of scores.

The pseudocode that determines the document with highest score from the meta index list META(U_f, t_i) is given in Listing 2.

5.3 Search Strategies

The scoring model introduced in Section 5.1 offers several interesting instances for evaluation and allows for comparing different search strategies which are derived from the final proposal of modelling *Single Tag Scores* (see Definition 5.2) by only considering certain components of the score computation. We define six search strategies which seamlessly can be used with our SOCIALMERGE algorithm since only the computation of $s_t(t_i, d, U)$ is modified as shown in the following.

For a query $q_U = \{t_0, \ldots, t_{n-1}\}$, a document d and a user U, the score of the document d with respect to the user U is defined by the sum over all single tag scores for each query tag as stated by Definition 5.1. However, the single tag score for a tag t_i, a document d and user U differs with respect to the applied search strategy.

Note: Since the social tagging networks introduced in Section 3.2 or the subset crawled from these networks do not provide a useful document-document relation, we

do not use the notion of the document rank (see Definition 5.9) for computing the single tag score (see Definition 5.2) with any of our strategies.

Semantic Search

When applying a semantic search, the results for a query are contributed by all users holding similar content to the query, i.e. users who used the query tags in describing their content. For a tag t_i, a retrieved document d is ranked by only using the introduced user-specific BM25 score.

Definition 5.12 (Semantic Search)**.** *Let be $\mathcal{U}$ the set of all users in a social tagging network. By applying a semantic search for a query $q_U = \{t_0, \ldots, t_{n-1}\}$, the single tag score for a document d with respect to a query tag t_i and user U is computed as follows:*

$$sts(t_i, d, U) = \sum_{U_j \in \mathcal{U}} s_d(U_j, t_i, d)$$

■

See Definition 5.10 for details about $s_d(U_j, t_i, d)$.

Social Search

With a social search strategy, only documents from social friends holding similar content to a query are considered for determining search results. For each query tag t_i, a retrieved result d is ranked using our user-specific BM25 score weighted by the friends' social friendship strengths:

Definition 5.13 (Social Search)**.** *By applying a social search for the query $q_U = \{t_0, \ldots, t_{n-1}\}$, the single tag score for a document d with respect to a query tag t_i and user U is computed as follows:*

$$sts(t_i, d, U) = \sum_{U_f \in \text{FLIST}(U)} s_f(U, U_f) \cdot s_d(U_f, t_i, d)$$

where FLIST*(U) is the social friendship list of U.* ■

See Definition 5.6 and 5.10 for details about $s_f(U, U_f)$ and $s_d(U_f, t_i, d)$, respectively.

Expanded Semantic Search

For enriching the search results, the expanded semantic search strategy uses a semantic search with tag expansion, i.e. tags with the largest similarity to the query tags are added to a query. For a query tag t_i or an expanded tag t_i', a result d is ranked using the user-specific BM25 score weighted by the tag similarity:

Definition 5.14 (Expanded Semantic Search)**.** *Let be $\mathcal{U}$ the set of all users in a social network. By applying an expanded semantic search for a query $q_U = \{t_0, \ldots, t_{n-1}\}$, the single tag score for a document d with respect to a query tag t_i and user U is computed as follows:*

$$s_t(t_i, d, U) = \sum_{U_j \in \mathcal{U}} \max_{t' \in \text{SIMTAGS}(t_i)} tsim(t', t_i) \cdot s_d(U_j, t', d)$$

where SIMTAGS*(t_i) is the list of all tags similar to t_i.* ∎

See the Definitions 5.7 and 5.10 for details about $tsim(t', t_i)$ and $s_d(U_j, t', d)$, respectively.

Expanded Social Search

The expanded social search strategy uses a social search with tag expansion which adds similar tags to a query for enriching search results. For each query tag t_i or an expanded tag t'_i, a document d from a social friend U_f is ranked using our user-specific BM25 score weighted by a the friendship strength U_f and the tag similarity:

Definition 5.15 (Expanded Social Search)**.** *By applying a expanded social search for a query $q_U = \{t_0, \ldots, t_{n-1}\}$, the single tag score for a document d with respect to a query tag t_i and user U is computed as follows:*

$$sts(t_i, d, U) = \sum_{U_f \in \text{FLIST}(U)} s_f(U, U_f) \cdot \max_{t' \in \text{SIMTAGS}(t_i)} tsim(t', t_i) \cdot s_d(U_f, t', d)$$

where SIMTAGS*(t_i) is the list of tags similar to t_i.* ∎

See the Definitions 5.6, 5.7 and 5.10 for details about $s_f(U, U_f)$, $tsim(t', t_i)$ and $s_d(U_f, t', d)$, respectively.

Social Search with User Rank

A social search with user rank strategy considers only results form social friends and weights them by the friends' user rank. For each query tag t_i, a document d contributed by a social friend U_f is ranked using the user-specific BM25 score weighted by the friendship strength of U_f and her user rank:

Definition 5.16 (Social Search with User Rank)**.** *By applying a social search with user rank for a query $q_U = \{t_0, \ldots, t_{n-1}\}$ from a user U, the single tag score for a document d with respect to a query tag t_i and user U is computed as follows:*

$$sts(t_i, d, U) = \sum_{U_f \in \text{FLIST}(U)} s_f(U, U_f) \cdot UR(U_f) \cdot s_d(U_f, t_i, d)$$

where FLIST*(U) is the friendship list of U.* ∎

See the Definitions 5.6, 5.10 and 5.8 for details about $s_f(U, U_f)$, $s_d(U_f, t', d)$ and $UR(U_f)$, respectively.

Expanded Social Search with User Rank

Finally, the strategy combining all the components (except for the document rank as explained in the beginning of this section) for computing the social score as provided in Definition 5.1 is called expanded social search with user rank. Query tags are expanded by similar tags and search results are considered only from social friends with additionally taking their user rank into account. For a query tag t_i or an expanded tag t'_i, a document d from a social friend U_f is ranked by using the user-specific BM25 score weighted by the friendship strength of U_f, her user rank and the tag similarity:

Definition 5.17 (Expanded Social Search with User Rank)**.** *By applying an expanded social search with user rank for a query $q_U = \{t_0 \ldots, t_{n-1}\}$, the single tag score for a document d with respect to a query tag t_i and user U is computed ans follows:*

$$\begin{aligned} sts(t_i, d, U) = \sum_{U_f \in \text{FLIST}(U)} & s_f(U, U_f) \\ & \times \quad UR(U_f) \\ & \times \quad \max_{t' \in \text{SIMTAGS}(t_i)} tsim(t', t_i) \\ & \times \quad s_f(U_f, t', d) \end{aligned}$$

where FLIST*(U) is the friendship list of U and* SIMTAGS*(t_i) is the list of tags similar to t_i.* ∎

See the Definitions 5.6, 5.7, 5.10 and 5.8 for details about $s_f(U, U_f)$, $tsim(t', t_i)$, $s_d(U_f, t', d)$ and $UR(U_f)$, respectively.

5.4 Experiments

For our experimental evaluation of our SOCIALMERGE algorithm and its different search strategies introduced in Section 5.3, we performed automated web crawls on the social tagging networks *Delicious.com* and *Flickr.com* (see Section 3.2), constituting our testing environment. Hence, the studies about our SOCIALMERGE search strategies are based on two real-world datasets with the following features:

- ***Delicious.com*:** The dataset harvested from *Delicious.com*

  comprises a total of $13,515$ users with $4,582,773$ bookmarks and $152,306$ friendship connections.

- ***Flickr.com*:** The dataset harvested from *Flickr.com* comprises a total of $2,274$ users, $1,357,424$ images and $72,703$ friendship connections.

For both datasets, we identified the most frequent tag pairs used in annotations on documents as our benchmark query workload. Typical example queries are "landscape, nature" or "insect, bug" for *Flickr.com* and "cooking, recipes" or "firefox, extension" for *Delicious.com*. For a full list of queries, please refer to the Tables 1 and 2. We employed the tag pairs for querying and measuring the efficiency and manual assessing the effectiveness of the corresponding search results for each search strategy. While the retrieval efficiency for both datasets and the effectiveness for *Flickr.com* have been evaluated on the full set of queries, the *Delicious.com* effectiveness assessments were only conducted with a subset of queries due to the higher effort of opening web pages as opposed to checking out photos.

Relevance Assessments

While many previous works focused on recall (i.e. the fraction of documents that is retrieved from the global set of relevant documents), we do not consider this a useful measure for social communities. Building the ground truth, i.e. the global set of relevant documents for a query, is virtually impossible in a social network though, as the notion of relevance is highly subjective and dependent from the query initiator and her personal context. To capture this, we have conducted a manual relevance assessment user study based on the following observations that are peculiar to social search.

Tag 1	Tag 2
sky	clouds
white	black
winter	snow
blackandwhite	bw
flowers	flower
ocean	beach
fall	autumn
woman	girl
selfportrait	me
birthday	party
green	blue
france	paris
yellow	red
urban	city
cats	cat
newyork	nyc
lights	night
d50	nikon
tokyo	japan
tree	trees
germany	deutschland

Tag 1	Tag 2
sun	sunset
sea	water
nature	landscape
color	colour
children	kids
concert	music
dinner	food
uk	england
thailand	bangkok
birds	bird
austin	texas
italy	italia
portrait	people
kid	child
new	york
face	eyes
closeup	macro
animal	pet
brazil	brasil
vegas	las
bride	groom

Tag 1	Tag 2
brick	wall
barcelona	spain
lomo	lca
light	luz
graffiti	streetart
soccer	football
naked	nude
auto	car
family	christmas
botanical	gardens
animals	zoo
pennsylvania	pa
insect	bug
happy	smile
leaves	leaf
mac	apple
taiwan	taipei
seoul	korea
horse	horses
architecture	buildings

Table 1: *Flickr.com* Queries

Tag 1	Tag 2
free	music
design	css
javascript	ajax
mac	osx
tutorial	howto
rails	ruby
linux	software
programming	code
tips	reference
lifehacks	productivity
video	youtube
education	learning
webdesign	inspiration
tagging	folksonomy
firefox	extensions
blogs	blogging
gallery	art
blog	internet
tools	webdev
plugin	wordpress
free	freeware
security	windows
cooking	recipes
cool	interesting
film	movies

Tag 1	Tag 2
culture	society
gtd	organization
maps	geo
php	development
information	visualization
humour	funny
green	environment
podcasting	podcast
recipe	food
wiki	wikipedia
python	django
television	tv
apple	macosx
source	open
computer	tech
technology	trends
health	fitness
list	links
clothing	fashion
xml	xslt
startup	entrepreneur
graphic	illustration
interface	ui
ideas	web2.0
shop	shopping

Tag 1	Tag 2
service	social
Mac	Apple
books	wishlist
daily	web
architecture	urbanism
USA	politics
search	Google
academic	theory
usa	news
Iraq	war
language	java
soa	webservices
investment	finance
conference	2007
sharing	hosting
investing	stocks
useful	todo
Software	Freeware
semanticweb	semweb
fiction	science
feminism	women
bike	cycling
knitting	pattern
events	nyc

Table 2: *Delicious.com* Queries

- The query results obtained from the system using a social search strategy depend on the query initiator, i.e. different users will obtain different results for the same query. This is obvious as the scoring model explores friendship links and tag similarities, which are user-specific.
- For judging the relevance of a result document with respect to a query and a certain social search strategy, the associated query initiator needs to be asked. For example, a picture of a person may only be relevant if she is known to the query initiator. However, when we execute our queries in the context of a user, we obviously have not got that user available to assess the subjective relevance of a result item, making it even harder to assess the relevance of documents as in standard information retrieval.

Hence, we conducted a user study as follows. First, the participants were shown a query. Next, one particular user that has previously used the query tags was randomly selected from the database as the (fictitious) query initiator. The participants are displayed exactly those documents (i.e. pictures or bookmarks) from that user that contain at least one of the query tags, in order to understand the personal context of the query initiator. This way, we try to overcome the aforementioned problem of subjectively assessing result qualities with the eyes of the query initiator. Next, the participants are displayed a 6-column result page that illustrates the top-10 results from each of our processing strategies under evaluation. The columns are not labelled and presented in random order. For each result item, the participant has to mark if the item is relevant to the query in the context of the query initiator or not. From these assessments, we finally compute the precision of the *top-10* documents (*Precision[10]*) as a measure of user satisfaction. The precision measure is computed as follows:

Definition 5.18 (Precision)**.**

$$precision = \frac{\#\ of\ relevant\ docs\ retrieved}{total\ \#\ of\ retrieved\ docs}$$

■

Retrieval Effectiveness

We have used the results from our user study to compare the retrieval effectiveness for all of the above strategies and for both datasets. The results are summarised in Table 3. First, we note that the precision results on the *Flickr.com* dataset are constantly higher than for the *Delicious.com* dataset. This is partly due to the fact that tags on *Flickr.com* are used much more descriptively than for *Delicious.com*, as one can clearly see from the queries (that, in turn, were derived from popular tag pairs). Though, mainly this is due to the more sophisticated context for the *Delicious.com* users, as presented to the participants of the user study, which made precision drop due to the better context information.

Semantic search already yields strong results, in particular for *Flickr.com*. Social expansion based on the friendship graph can further improve the precision remarkably, which makes a strong case for social search strategies.

On the other hand, tag expansiondid not yield the desired results in our experiments, but had an negative impact. Taking a closer look at the individual numbers indicates that the Dice coefficient does not do a good job of grouping tags. In effect, tag expansion often makes the query execution drift away from the original topic.

Finally, the user rank UR (see Definition 5.8) does not seem to have got any remarkable influence on the precision results. Again, taking a closer look at the individual numbers indicate that the nature of *PageRank*-style authority scores [99] assigning global (i.e. non query-specific) authority scores to users makes the effect on precision vanish for particular queries. In other words, a high user rank by design can not give an indication as to whether the authority has been accumulated w.r.t. the particular information need. This makes a strong case for considering personalised authority-style analyses that capture authority from the viewpoint of the query initiator and her personal interest profile [65, 73].

Approach	**P@10 - *Flickr.com***	**P@10 - *Delicious.com***
Semantic	72%	29%
Social	76%	37%
Expanded Semantic	72%	28%
Expanded Social	66%	36%
Social + UR	77%	33%
Expanded Social + UR	67%	31%

Table 3: Precision[10]

Retrieval Efficiency

For each query and each of our strategies under evaluation, we measure the following performance metrics:

- total number of sequential list accesses
- total number of random list accesses
- total number of dimensions (i.e. number of index lists opened)

We report the average over all queries separately for our *Flickr.com* and *Delicious.com* datasets in Table 4 and Table 5.

Approach	**#SA**	**#RA**	**#Dim**
Semantic	1530.69	4.69	2.0
Social	147.68	0.032	157.16
Expanded Semantic	1531.10	19.27	17.42
Expanded Social	738.0	5.92	2234.29
Social + UserRank	144.06	0.0	157.16
Expanded Social + UserRank	754.21	5.53	1935.66

Table 4: Performance Figures (*Flickr.com*)

The numbers show (especially on the *Flickr.com* dataset but also for the social search strategy with user rank on the *Delicious.com* dataset) that our social search

Approach	#SA	#RA	#Dim
Semantic	294.86	7.41	2.0
Social	778.82	8.38	34.54
Expanded Semantic	778.15	18.78	12.12
Expanded Social	1027.14	26.53	187.35
Social + UserRank	166.72	2.35	9.65
Expanded Social + UserRank	216.97	5.69	52.88

Table 5: Performance Figures (*Delicious.com*)

strategies are clearly competitive with the baseline, non-social semantic strategy. In particular, the exploration of more dimensions does not hurt the overall number of sequential or rand accesses, due to our sophisticated incremental merge strategy. Also, the careful scheduling of random accesses does not hurt the query efficiency either, as it is clearly limited to very low numbers. Combining all this with the shown effectiveness improvements in Table 3, our social search yields a better benefit/cost-ratio.

6 CONTEXTMERGE Algorithm

The manual assessment accomplished and described in Section 5.4 to evaluate the retrieval effectiveness of our SOCIALMERGE algorithm gave us insights about the correlation of the result quality with respect to our different search strategies introduced in Section 5.3 and the queries used for the evaluation. For some queries a certain search strategy achieved better results than for other queries. This raised the question if different queries maybe serve different information needs and if we can adjust the query processing according to the information needs of queries.

Based on the results achieved with our SOCIALMERGE algorithm (see Section 5) and based on the experience made during its experimental evaluation (see Section 5.4), we developed the CONTEXTMERGE algorithm with an improved scoring model which allows for a flexible adjustment to different information needs and the application of diversified search modes by the query processing.

In the following we present the different information needs that we identified in social tagging networks. Afterwards we describe in detail the scoring model, and how the query processing of our CONTEXTMERGE algorithm combines the requirements of different search modes, serving different information needs.

6.1 Information Needs in Social Tagging Networks

The way in which users interact and search in social tagging networks depends on their intents. Accordingly, with our CONTEXTMERGE algorithm, we classify the task of searching in social tagging networks in the following three categories, each expressing an associated, different kind of information need.

Global Search

In social tagging networks, typically documents, URLs or videos are retrieved by issuing queries that consist of a set of tags, returning results that have been frequently

annotated with these tags without taking any relation between users into account. We denote search tasks that neglect user relations but consider all users equally important as *global search* tasks. For a global search, it is not important which user can contribute documents to a query as long as results match a requested query. Hence, the global search fulfils a rather generic information need where queries can be answered by any user of the global network.

Example Scenario. Assume user U is active in some social tagging network about books, e.g. *LibraryThing.com*, and she needs to buy as a birthday present the fifth book of *Harry Potter* but does not know its title. To be sure to buy the correct book, she might issue the query *"Harry Potter Book 5"* to the social network she is involved in. From her point of view, it is not important which user provides the correct result to her since she just wants to have got an exact answer to her request. So, by taking the answer given by the majority of users, the information need of user U is likely to be fulfilled.

Social Search

Users often prefer a more personalised way of searching over a global search, exploiting preferences of and connections between users. A reason for this might be that a user does not care about results from unknown users maybe because a query is about documents containing information about social friends. Social friends are users in the social network known from real life or who became acquainted over time and are explicitly chosen as friends in the network. So, if a user for example is looking for photos, she might be not interested in photos of unknown people.

Another reason why a user might be more interested in results from social friends for certain queries is that the user is maybe interested in *recommendations* and she likely trusts recommendations of users she knows, i.e. her friends, and thus possibly sharing her preferences, more than those of some arbitrary, unknown users.

Example Scenario. Again consider the same user U from the example given for the global search. Assume U does not like *"Harry Potter"*-books which is the reason why she had to query for the title of the fifth book by a global search. However, further assume that in general U is interested in fantasy books and especially in books about magic and wizards. She just cannot stand *Harry Potter* books. To descry some new and interesting novels, she therefore might issue a query *"fantasy magic wizard"* to her social tagging network about books.

If a global search is used for such a query, it is likely that results are dominated by books about *"Harry Potter"* because the commercial success of its complete series shows that many people like *"Harry Potter"* books and, hence, would recommend them.

On the other hand, for such broad topics like *fantasy*, *magic* and *wizard*, it is conceivable that user U has chosen other users in the social network as friends that have got an overlap in their interests, thus, maybe also dislike *Harry Potter* books. In this case, a search restricted to the social context of user U, i.e. information contributed by U's explicit and transitive friends (her friends and the friends of her friends), promises to better fulfil U's information need for her query.

We denote search tasks that only consider results from a user's transitive friends in the social friendship graph of a social tagging network as *Social Search* tasks.

Spiritual Search

Another way to personalise search in social tagging networks is to exploit similar preferences, behaviour or activities of users and to consider the information within the user's interests already expressed in the network. This can be done for example by taking into account the users' overlap in tag usage, bookmarked pages, or commenting and rating activities.

We denote users who do not necessarily know each other but have got a common overlap in interests as *"Brothers in Spirit"* which in turn leads to the notion of *spiritual friends*. In contrast to social friends, which are explicitly chosen as friends by a user, spiritual friends can be considered implicitly related to a user due to their behavioural affinity. Hence, the name *Spiritual Search*, which denotes a search task that only considers results from spiritual friends in social tagging networks.

If a user has got some interests in a specific area of a broader topic, her social friends maybe have not got the same interests and, thus, a social search is likely not to retrieve desired results for a query. Neither, a global search would lead to good results because the majority of users is probably be interested in the more general areas of the topic. However, those users who share the same specific interests can, even if they do not know each other, recommend and contribute results in which the querying user most likely is interested in.

Example Scenario. Again consider the same user U from the example given for the global and social search. So, U does not like *Harry Potter*-books but is in general interested in fantasy novels about magic and wizards and so are her social friends. Further assume that U is additionally interested in topics about *Computer* and *Programming* and especially in the specific area of the small niche computer platform *Amiga*. When U tries to find programming books about her hobby computer platform, she maybe issues a query like *Computer Programming Amiga* to her social tagging network. Neither a global, and if U's social friends do not share U's passion for the same niche computer platform, also a social search might not lead to the desired results because they might be dominated by general books about programming. However, if there are any users in the social tagging network with similar preferences as U, they finally could fulfil U's information need.

This kind of personalised search that asks for recommendation-style results is very common in online communities, for example asking for books tagged by other users with similar interests, or searching for restaurants tagged by users from the same area and with similar preferences for food.

Note: The three example scenarios given for the social, spiritual and global search are not arbitrarily or artificially chosen for a just notional user U but indeed show the characteristics of a real user in the social tagging network *LibraryThing.com* (see Section 3.2.3). The prototype implementation of our *SENSE* framework, which applies the CONTEXTMERGE algorithm on a dataset crawled from *LibraryThing.com*, demonstrates this behaviour as described in our three example scenarios given above.

Further note that the information needs identified in social tagging networks and the associated *global*, *social* and *spiritual search* tasks perfectly match our notion of social, spiritual and global friendship as given in Definition 3.1, 3.3 and 3.4, respectively, of Section 3.1 introducing our data model and its entity relations.

In the following we introduce our improved scoring model, incorporating flexible adjusting of scores according to the different information needs and discuss our new

and sophisticated CONTEXTMERGE algorithm.

6.2 Scoring Model

The scoring model used with our CONTEXTMERGE algorithm can be tuned towards different aspects of social tagging networks and, thus, can be adjusted to match the information needs of users in the network. To this end, as with our SOCIALMERGE algorithm, scores for documents are *user-specific*. However, depending on the configuration of the scoring model, scores are based on the social and spiritual context of the query initiator. Moreover, it extends the traditional IR scoring models (tf-idf-based, probabilistic IR, language models) to search in social networks, with the following ingredients:

1. a measure for the importance of users, relative to the querying user,
2. a context-specific tag frequency relative to the querying user that reflects the relative importance of users which used a tag, and, optionally,
3. the expansion of query tags with related ("semantically similar") tags.

In our *context-specific scoring model*, tags given by close social or spiritual friends are weighted higher than tags given by users at a great distance in the respective friendship graph (defined by the respective friendship relation, see Definition 3.2) of the social network.

The scoring model implemented with our CONTEXTMERGE algorithm allows to specify the information need to perform a *social*, *spiritual* or *global* search (see Section 6.1). It also allows for a *hybrid* search, i.e. a weighted combination of any of the three search tasks.

6.2.1 Modelling Friendship Strengths.

For matching the different information needs applied to searches in social tagging networks (see Section 6.1), with the scoring model of our CONTEXTMERGE algorithm, we define the user-user relation

$$Friendship(U_1, U_2, type, s_f)$$

specified by our data model in Section 3.1 for *type=social*, *type=spiritual* and for *type=global*. The three different types of friendship relations correspond to the Definition 3.1, 3.3 and 3.4 of social, spiritual and global friendship. The associated three different quantifications of the *friendship strength* $s_f(U_1, U_2)$ of a user U_2 with respect to some other user U_1 form the core of our scoring model. Measuring the friendship strengths can be done in different ways, and our current implementation allows to switch between different definitions at run-time.

Before going into detail, we change the notation in regard to the friendship strength s_f defined by the user-user relation in order to easier distinguish the friendship strength for each type of friendship:

- $s_{so}(U, U_f)$ denotes the *social* friendship strength of U_f w.r.t U
- $s_{sp}(U, U_f)$ denotes the *spiritual* friendship strength of U_f w.r.t U
- $s_{gl}(U, U_f)$ denotes the *global* friendship strength of U_f w.r.t U

Eventually, the scoring model enables the different search tasks identified in social tagging networks by defining the friendship strength of users accordingly. Before introducing the formal definitions, we give a general view on each type of friendship strength.

- The *social* friendship strength $s_{so}(U, U_f)$, applied for social searches, is based on social measures like the inverse shortest path distance of U and U_f in the friendship graph of the social network, but may additionally include syntactic measures as used for the computation of the spiritual friendship strength.
- The *spiritual* friendship strength $s_{sp}(U, U_f)$ of a spiritual friend U_f with respect to a user U is tuned towards spiritual searches and computed by using a combination of syntactic measures such as overlap of tag usage, bookmarked pages, or commenting and rating activity.
- The *global* friendship strength $s_{gl}(U, U_f)$, used for global searches, gives equal weight to all users.

Each type of friendship strength is normalised over all users such that for all users U in the social network, the following is true:

$$\sum_{U_f} s_{xx}(U, U_f) \; = \; 1 \text{ with } xx \in \{so, sp, gl\}$$

The friendship strength $s_{xx}(U, U_f)\ with\ xx \in \{so, sp, gl\}$ may be viewed as the probability that with respect to a spiritual, social or global information need, respectively, documents from a random user U_f will be interesting to U. In this way, the importance of a user U_f, relative to a querying user U, can be quantified in a single friendship definition, aggregating the different types of friendship strengths. To this end, we introduce

- the (final) *friendship strength* $s_f(U, U_f)$, computed to evaluate a query, as the linear mixture of the three definitions of spiritual, social and global friendship strength.

The formal definition of the final friendship strength of two users is the then:

Definition 6.1 ((final) Friendship Strength $s_f(U, U_f)$)**.** *The final friendship strength of a user U_f with respect to a user U is defined as:*

$$s_f(U, U_f) = \alpha \cdot s_{so}(U, U_f) + \beta \cdot s_{sp}(U, U_f) + (1 - \alpha - \beta) \cdot s_{gl}(U, U_f)$$

with $\alpha, \beta \in [0 : 1]$ being configurable parameters, $\alpha + \beta \leq 1$ and $s_f(U, U_f) = 0$ if $U_f = U$. ∎

When setting $\alpha = 1$ and $\beta = 0$ the scoring model is tuned to social searches, when setting $\alpha = 0$ and $\beta = 1$ it is tuned to spiritual searches, and with $\alpha = 0$, $\beta = 0$ to global searches. However, any nontrivial combination is fine and reasonable. Moreover, we define the friendship strength for a user U with respect to herself to be zero, i.e. $s_f(U, U) = 0$. The reason is, that a user cannot contribute new results to own search queries. Thus, we want to ignore own documents in search requests.

Next, we define the different quantification, i.e. the spiritual, social and global friendship strength, used to compute the final friendship strength.

Spiritual Friendship.

The *spiritual friendship strength* $s_{sp}(U, U_f)$ of two users U and U_f can be computed by using a combination of syntactic measures such as overlap of tag usage, bookmarked pages, or commenting and rating activity.

In our implementation, the spiritual friendship strength is based only on the overlap in the set of tags both users use.

Definition 6.2 (Spiritual Friendship Strength $s_{sp}(U_i, U_j)$)**.** *For any two users U_i and U_j, we define their spiritual friendship strength by computing the Dice coefficient of their tag sets:*

$$s_{sp}(U_i, U_j) = \frac{2 \times |tagset(U_i) \cap tagset(U_j)|}{|tagset(U_i)| + |tagset(U_j)|}.$$

where $tagset(U)$ is the set of tags used by U. ∎

The spiritual friendship strength s_{sp} is a symmetric measure, i.e $s_{sp}(U_i, U_j) = s_{sp}(U_j, U_i)$ for two users U_i and U_j, with $0 \leq s_{sp}(U_j, U_i) \leq 1$ for all users U_i, U_j.

Social Friendship.

The *social friendship strength* $s_{so}(U, U_f)$ is based on social measures like the inverse shortest path distance of U and U_f in the friendship graph of the social network (see Definition 3.2) but may additionally include syntactic measures like the spiritual friendship strength.

In our implementation, we first compute an overlap-based similarity, similar to the spiritual friendship strength but only for directly connected users U, U_f (i.e. there is an edge from U to U_f) in the friendship graph of the social networks.

Definition 6.3 (Direct (social) Friend)**.** *A user U_f who is directly connected with U over an edge $U \rightarrow U_f$ in the friendship graph of a social tagging network is called a* direct (social) friend *of U.* ∎

The overlap-based similarity defines then the weight of the edges between direct social friends and is again computed as the Dice coefficient of the sets of tags used by a user U and her direct friend U_f.

Definition 6.4 (Overlap of Tags for Direct Social Friends $O(U, U_f)$)**.** *For U_f being a direct friend of U, we define their tag overlap by the Dice coefficient of their tag sets:*

$$O(U, U_f) = \frac{2 \times |tagset(U) \cap tagsat(U_f)|}{|tagset(U)| + |tagset(U_f)|}.$$

where $tagset(U)$ is the set of tags used by U. ∎

From this definition immediately follows $0 \leq O(U, U_f) \leq 1$.

We then extend Definition 6.4 to users U and U_f that are indirectly connected by more than one edge in the friendship graph, over one or more paths leading from U to U_f, by aggregating the overlap in tag usage for each pair of direct friends along each path and picking the path with the highest aggregated overlap measure. To this end, the overlap measure for direct friends can be averaged, multiplied, or multiplied weighted by (linear or dampened) distance, etc.

In our implementation, we compute the social friendship strength as follows:

Definition 6.5 (Social Friendship Strength $s_{so}(U, U_f)$)**.**

$$s_{so}(U, U_f) = \begin{cases} 0 & \text{if } \nexists\, \pi : U_0 \dots U_k \\ \max_{\pi: U_0 \dots U_k} \prod_{i=0}^{k-1} O(U_i, U_{i+1}) & \text{otherwise.} \end{cases}$$

where π is a path from U=U_0 to U_k=U_f with edges $(U_i \rightarrow U_{i+1})$ and $0 \leq i \leq (k-1)$. ∎

This definition of social friendship flavours, as in the case of the SOCIALMERGE algorithm, users at small distances. However, in contrast to our SOCIALMERGE algorithm, now the social friendship strength is not defined over the number of edges but over the weight of the edges for all paths between two users. Since the resulting weight becomes smaller the more edges are multiplied, it is likely that the path defining the friendship strength is also "short" with respect to the number of edges. Hence, with our CONTEXTMERGE algorithm, we want to define the shortest path as follows:

Definition 6.6 (Shortest Path)**.** *Let be G the directed,* weighted *friendship graph of a social tagging network with edge weights in the interval $[0:1]$. We define the length of a path in G from a user U to a user U_f as the inverse of the product over all of its edge weights.*

Hence, a shortest path from U to U_f is a path with the maximal product over its edge weights. ∎

Given the Definition 6.5 of social friendship strengths, with our CONTEXTMERGE algorithm, the shortest path between a user U and his friend U_f is now, as in the case of SOCIALMERGE, equal to the path defining their friendship strength $s_{so}(U, U_f)$.

Note: Other types of direct social friendship strength and aggregation over paths can be easily plugged into the model and our implementation. For directed friendship graphs, i.e. a user U_f is a friend of a user U but not necessarily the other way around, the measure of social friendship strength is asymmetric, i.e. $s_{so}(U, U_f) \neq s_{so}(U_f, U)$

Global Friendship.

The *global friendship strength* $s_{gl}(U, U_f)$ used for global searches gives equal weight to all users in the social network.

Definition 6.7 (Global Friendship Strength $s_{gl}(U_i, U_j)$)**.** *For any pair of users U_i and U_j, the global friendship strength is equal, i.e.*

$$s_{gl}(U_i, U_j) = \frac{1}{|\mathcal{U}|}$$

where $\mathcal{U}$ is the set of all users in the social network. ∎

6.2.2 Modelling Context Scores

With having defined the friendship strength of users, matching the three information needs in social networks, we next introduce the scoring model for a document d and a tag t in the *context* of the information needs and the friendship relations of a user U.

In contrast to the scoring model presented in Section 5.1 used in our SOCIALMERGE algorithm, the importance of other users in a social network for contributing results to a query, is not only measured by their social friendship strengths with respect to the

query initiator but depends on all defined friendship relations, dynamically weighted according to the information need of the querying user. Hence, we call the weighted social, spiritual and global friendship relations to be the *context* of a user, issuing a query.

Context Frequency.

To reflect the friendship strengths of friends (of any kind) who tagged a document that may be of interest to the querying user, we introduce the notion of *context frequency*.

To this end, we first define for a user U, a tag t and a document d the *user-specific tag frequency* $tf_U(t,d)$ and the *global tag frequency* $TF(t,d)$.

Definition 6.8 (User-Specific Tag Frequency $tf_U(t,d)$)**.** *For a user U, a tag t and a document d, we define the user-specific tag frequency $tf_U(t,d)$ as the number of times U annotated document d with tag t.* ∎

Note: In most of today's social tagging platforms, typically $tf_U(t,d) = 1$, i.e. document d has been tagged with t once by U, or $tf_U(t,d) = 0$, i.e. document d has not been tagged at all with t by U. However, it is conceivable that quantitative ratings are factored into this measure or user feedback leads to non-binary $tf_U(t,d)$ values.

Definition 6.9 (Global Tag Frequency $TF(t,d)$)**.** *For a tag t and a document d, we define the global tag frequency $TF(t,d)$ to be equal to the number of times all users together in a social network annotated d with t:*

$$TF(t,d) = \sum_{U' \in \mathcal{U}} tf_{U'}(t,d)$$

where $\mathcal{U}$ is the set of all users in a social tagging network. ∎

In our scoring model, the context frequency $cf_U(t,d)$ replaces the standard IR term frequency $tf(t,d)$ and considers the social, spiritual and global friendship strengths of users tagging a document. Formally, it is defined as follows:

Definition 6.10 (Context Frequency $cf_U(t,d)$)**.** *The context frequency for a tag t and a document d, relative to a user U, is defined as*

$$cf_U(t,d) = \sum_{U' \in \mathcal{U}} s_f(U,U') \cdot tf_{U'}(t,d).$$

where $\mathcal{U}$ is the set of all users in the network and $tf_{U'}(t,d)$ denotes the user-specific tag frequency of U' for tag t and document d. ∎

By plugging the definition of the friendship strength $s_f(U,U_f)$ into the above formula, we obtain

$$cf_U(t,d) = \sum_{U' \in \mathcal{U}} (\alpha \cdot s_{so}(U,U') + \beta \cdot s_{sp}(U,U') + (1-\alpha-\beta) \cdot s_{gl}(U,U')) \cdot tf_{U'}(t,d)$$

By rewriting the equation, we can separate the parts being bound by each friendship type and when we additionally substitute $s_{gl}(U, U')$ with its definition (see Definition 6.7), we obtain

$$cf_U(t,d) = \alpha \cdot \sum_{U' \in \mathcal{U}} s_{so}(U,U') \cdot tf_{U'}(t,d) + \beta \cdot \sum_{U' \in \mathcal{U}} s_{sp}(U,U') \cdot tf_{U'}(t,d) + \frac{1-\alpha-\beta}{|\mathcal{U}|} \cdot \sum_{U' \in \mathcal{U}} tf_{U'}(t,d)$$

With $TF(t,d)$ being the *global tag frequency* as defined in Definition 6.9, we finally obtain

$$cf_U(t,d) = \alpha \cdot \sum_{U' \in \mathcal{U}} s_{so}(U,U') \cdot tf_{U'}(t,d) + \beta \cdot \sum_{U' \in \mathcal{U}} s_{sp}(U,U') \cdot tf_{U'}(t,d) + \frac{1-\alpha-\beta}{|\mathcal{U}|} \cdot TF(t,d)$$

For simplifying the notation in subsequent sections, we additionally introduce the slightly modified context frequency $cf'_U(t,d)$, which is defined as

Definition 6.11 ((modified) Context Frequency $cf'_U(t,d)$)**.**

$$\begin{aligned} cf'_U(t,d) = \; & \alpha \cdot |\mathcal{U}| \cdot \sum_{U' \in \mathcal{U}} s_{so}(U,U') \cdot tf_{U'}(t,d) \\ & + \beta \cdot |\mathcal{U}| \cdot \sum_{U' \in \mathcal{U}} s_{sp}(U,U') \cdot tf_{U'}(t,d) \\ & + (1-\alpha-\beta) \cdot TF(t,d) \end{aligned}$$

■

Note that $cf_U(t,d)$ and $cf'_U(t,d)$ are proportional

$$cf_U(t,d) \propto cf'_U(t,d)$$

and that we can refer to the context frequency for a user U, tag t and document d as

$$cf'_U(t,d) = |\mathcal{U}| \cdot cf_U(t,d)$$

Finally, we can observe that the modified context frequency $cf'_U(t,d)$ can be split into a *global* part

$$(1-\alpha-\beta) \cdot TF(t,d),$$

being independent of the querying user and corresponding to a weighted global tag frequency $TF(t,d)$, and a *user-specific* frequency

$$\alpha \cdot |\mathcal{U}| \cdot \sum_{U' \in \mathcal{U}} s_{so}(U,U') \cdot tf_{U'}(t,d) + \beta \cdot |\mathcal{U}| \cdot \sum_{U' \in \mathcal{U}} s_{sp}(U,U') \cdot tf_{U'}(t,d),$$

which depends on the social and spiritual friendship strengths of users with respect to the query initiator. We will make use of this decomposition in Section 6.3 to efficiently process queries.

Note: The weighted global tag frequency $TF(t,d)$ is agnostic to any user relations as all users are regarded to be equally important for computing this measure. Hence, it

is an important building block in the definition of our context frequency and defines in our scoring model the document-tag relation

$$Content(d, t, score)$$

as specified in our data model in Section 3.1.3. Accordingly, the user-specific portion in the definition of $cf_U(t, d)$ implements the ternary user-tag-document relation

$$Tagging(U, t, d, score)$$

as defined by our data model.

Next, we use our notion of context frequency for defining the score of a document d for a single tag t with respect to a user U.

Single Tag Context Score.

To compute the single tag context score $sts(t, d, U)$ of a document d with respect to a single tag t relative to the querying user U, we use a scoring function in the form of a simplified BM25 [105] score. Unlike the original BM25 formula, our model has no notion of document length because the number of tags assigned to a document does not vary as much as the length of text documents. Furthermore, we replace in BM25 the standard term frequency $tf(t, d)$ for a tag t and a document d by our user-specific context frequency $cf_U(t, d)$. We formally define the single tag context score as follows:

Definition 6.12 (Single Tag Context Score $sts(t, d, U)$)**.** *The score of a document d with respect to a tag t and querying user U is defined as:*

$$sts(t, d, U) = \frac{(k_1 + 1) \cdot |\mathcal{U}| \cdot cf_U(t, d)}{k_1 + |\mathcal{U}| \cdot cf_U(t, d)} \cdot idf(t)$$

where k_1 is a tunable coefficient and $idf(t)$ is the inverse document frequency of tag t. ∎

Note from Definition 6.11 that $|\mathcal{U}| \cdot cf_U(t, d) = cf'_U(t, d)$. The inverse document frequency [77] is defined for a tag t as follows:

Definition 6.13 (Inverse Document Frequency $idf(t)$)**.** *For a tag t, the inverse document frequency is defined as:*

$$idf(t) = \log \frac{|\mathcal{D}| - df(t) + 0.5}{df(t) + 0.5}$$

with $df(t)$ denoting the number of documents that were tagged with t by at least one user and $|\mathcal{D}|$ is equal to the number of all documents in the social network. ∎

By plugging the definitions of the modified context frequency $cf'_U(t, d)$ and the friendship strength $s_f(U, U_f)$ into the formula for computing the single tag context score of a document, we obtain

$$sts(t, d, U) = idf(t) \cdot \frac{(k_1+1)(1-\alpha-\beta)TF(t, d)+\alpha \sum_{U'} ..s_{so}()..+\beta \sum_{U'} ..s_{sp}()..}{k_1+(1-\alpha-\beta)TF(t, d)+\alpha \sum_{U'} ..s_{so}()..+\beta \sum_{U'} ..s_{sp}()..}$$

From this, it can be observed that we can obtain the global score of a document d with respect to a single tag t that is agnostic to any user relations, by choosing $\alpha = 0$ and $\beta = 0$ and, thus, only considering the global tag frequency $TF(t, d)$ and the inverse document frequency $idf(t)$ in the computation of the single tag context score $sts(t, d, U)$.

Tag Expansion.

Even though related users are likely to have tagged related documents, they may have used different tags to describe them. It is therefore essential to allow for an expansion of query tags to semantically related tags. A simple way to account for this would be to statically expand the query with a fixed number of similar tags; however, experiments on text IR have shown that this can lead to topic drifts and search results that are inferior to those of the unexpanded, original query [27]. Instead, we adopt the *careful expansion* approach proposed in [124] that considers, for the score of a document, only the best expansion of a query tag, not all of them.

For this, we implement in our scoring model the tag-tag relation

$$TagSimilarity(t_1, t_2, tsim)$$

introduced in our data model in Section 3.1 by defining for all pairs of tags t_1, t_2 the similarity function $tsim(t_1, t_2)$, with $0 \leq tsim(t_1, t_2) \leq 1$, as specified in the relation.

In the implementation used with our CONTEXTMERGE algorithm, the similarity between two tags is determined, similarly as in the case of our SOCIALMERGE algorithm, by the co-occurrence of tags in the entire document collection. However, instead of computing the Dice coefficient on sets of documents (as in the SOCIALMERGE case, see Definition 5.7), we estimate conditional probabilities for the co-occurrence of tags.

Definition 6.14 (Tag Similarity $tsim(t, t')$)**.** *The similarity of two tags t and t' is defined as:*

$$tsim(t, t') = P[t'|t] = \frac{df(t \wedge t')}{df(t)}$$

where $df(t)$ is the number of documents tagged with t and $df(t \wedge t')$ is the number of documents tagged with both tags t and t' (but possibly by different users). ∎

However, other measures such as *SocialSimRank* from [19] could be easily incorporated as well.

Eventually, the single tag context score $sts^*_U(t, d, U)$ with tag expansion for a document d with respect to a tag t and relative to a querying user U is defined as follows:

Definition 6.15 (Single Tag Context Score with Tag Expansion $sts^*(t, d, U)$)**.** *For a document d, the score of a document with respect to a tag t and a user U is defined as:*

$$sts^*(t, d, U) = \max_{t' \in \text{SIMTAGS}(t)} tsim(t, t') \cdot sts(t', d, U)$$

where SIMTAGS*(t) is the list of all tags t' similar to tag t.* ∎

Context Score for Queries.

Finally, the *context score* for an entire query q_U with multiple query tags $t_0, \ldots, t_{n-1}$ is the sum of the per-tag context scores:

Definition 6.16 (Context Score $csc^*(d, q_U = \{t_0, \ldots, t_{n-1}\})$)**.** *For a query q_U with query tags $t_0, \ldots, t_{n-1}$ from a user U, the final score of a document is computed as:*

$$csc^*(d, q_U) = \sum_{t_i \in q_U} sts^*(t_i, d, U)$$

∎

Note that the context score assumes a non-conjunctive query evaluation. However, it can easily be extended to conjunctive evaluation by setting the context score $csc^*(d, q_U = \{t_0, \ldots, t_{n-1}\}) = 0$ when for at least one query tag t_i, the single tag context score $sts^*(t_i, d, U)$ is equal to 0.

Remark: With the scoring model for our CONTEXTMERGE algorithm, we neither implement the document-document relation $Linkage(d_1, d_2, w)$ (see Section 3.1.3) nor the user-document relation $Rating(U, d, rating)$ (see Section 3.1.2) as defined by our data model in Section 3.1. The reason is that the datasets crawled from real world social tagging networks (see Section 3.2) and used in our experimental evaluation presented in Section 6.4, do not exhibit such relations to the full extend or only could be harvested on a limited scale.

6.3 Query Processing

In this section, we introduce the CONTEXTMERGE algorithm to efficiently evaluate the *top-k* matches for a query, using the context score defined with our scoring model in Section 6.2. As our SOCIALMERGE algorithm, the CONTEXTMERGE algorithm generally falls into the well-established framework of threshold algorithms over impact-ordered inverted lists [54, 15] for efficient *top-k* query processing. However, as the context score depends on the user who submits a query, it is impossible to precompute per-tag scores for each document and each user. Standard algorithms that rely on scanning inverted lists cannot be applied here. Instead, CONTEXTMERGE makes use of information that is available in social tagging networks anyway, namely, lists of documents tagged by a user and numbers of documents tagged with tags. It incrementally builds context frequencies by considering users that are related to the querying user in descending order of social or spiritual friendship strengths, computes upper and lower bounds for the context scores from these frequencies, and stops the execution as soon as it can be guaranteed that the best k documents have been identified.

6.3.1 Preprocessing

Our CONTEXTMERGE algorithm makes use of up to five different kinds of preprocessed inverted lists, depending on the instantiation of the algorithmic framework, which are built at indexing time and accessed mostly sequentially at querying time. Additional random accesses to look up the value of an entity in a list are possible, but more expensive than sequential accesses in terms of access cost.

- DOCS(t)**:** A *global document list*, containing for a tag t, the documents d tagged by at least one user with t and the corresponding *global* tag frequencies $TF(t, d)$ (see Definition 6.9). The documents in the list are sorted in descending order of $TF(t, d)$.

- USERDOCS(U, t)**:** A *user document list*, containing for a user U and a tag t, the (unsorted) set of documents d tagged by U with t and their user-specific tag frequency $tf_U(t, d)$ (see Definition 6.8) which is often equal to 1 in most of today's social tagging networks.

- FLIST$_{so}$(U)**:** A *social friendship list*, containing for a user U all social friends U_f and their social friendship strengths $s_{so}(U, U_f)$, sorted in descending order of the social friendship strengths.

See Definition 6.5 for details about the social friendship strength $s_{so}(U, U_f)$.

- FLIST$_{sp}$(U) **:** A *spiritual friendship list*, containing for a user U all spiritual friends U_f and their spiritual friendship strengths $s_{sp}(U, U_f)$, sorted in descending order of the spiritual friendship strengths.

  See Definition 6.2 for details about the spiritual friendship strength $s_{sp}(U, U_f)$.

- SIMTAGS(t)**:** A *tag similarity list*, containing for a tag t all similar tags t' with their tag similarity $tsim(t, t')$ multiplied by the inverse document frequency of t', and sorted in descending order of $tsim(t, t') \cdot idf(t')$. We denote with $tsim_w(t, t') = tsim(t, t') \cdot idf(t')$ the *weighted tag similarity*.

  See Definition 6.14 for details about the tag similarity $tsim(t, t')$, and Definition 6.13 for details about $idf(t)$.

Note: In contrast to the document lists used with our SOCIALMERGE algorithm, the inverted lists used with the CONTEXTMERGE algorithm contain *no* precomputed document scores but only tag frequencies.

6.3.2 Notation

Let be,

- $high_d(t_i)$ the last value of $TF(t, d)$ read from the DOCS(t_i) list
- $high_{so}(U, t_i)$ the last value of $s_{so}(U, U_f)$ read from the FLIST$_{so}$(U) list for t_i
- $high_{sp}(U, t_i)$ the last value of $s_{sp}(U, U_f)$ read from the FLIST$_{sp}$(U) list for t_i
- $high_t(t_i)$ the last value of $tsim_w(t, t')$ read from the SIMTAGS(t_i) list

Note: The values of $high_d(t_i)$, $high_{so}(U, t_i)$, $high_{sp}(U, t_i)$ and $high_t(t_i)$ are upper bounds for score values of unseen entries in each of the lists since those are sorted in descending order of the score values.

Furthermore, to simplify the description and for a better understanding of the query processing with our CONTEXTMERGE algorithm, we assume that all friendship lists are normalised:

Definition 6.17 (Normalized Friendship Lists)**.** *The friendship strengths of users in the lists* FLIST$_{so}$*(U) and* FLIST$_{sp}$*(U) are normalised in such a way that the score within one list sums up to 1, i.e. the friendship strength values are divided by the sum over all values.* ∎

6.3.3 Operation Mode

The pseudocode shown in Listing 3 depicts the general structure of our query processing framework including tag expansion. However, for a better understanding, we first describe our CONTEXTMERGE algorithm without tag expansion, i.e. in Listing 3, Line 30 ($l == Taglist$) is never true. We first consider only the user expansion while processing a query, i.e. in Listing 3, Line 19 ($l == Social$) or Line 23 ($l == Spiritual$) is true. Afterwards we extend the explanation by including tag expansion.

```
1:  ContextMerge(U, Q = {q_0, ..., q_{n-1}}, α, β) {
      // Initialisation of data structures for all query tags q_i
2:    FOR i = 0 TO n − 1 {
3:      Social[i][0] = FLIST_so(U)
4:      Spiritual[i][0] = FLIST_sp(U)
5:      Global[i][0] = DOCS(q_i)
6:      high_gl(U,q_i) = DOCS(q_i)[0].getTF()
7:      high_so(U,q_i) = FLIST_so(U)[0].getStrength()
8:      high_sp(U,q_i) = FLIST_sp(U)[0].getStrength()
9:      Taglist[i] = SIMTAGS(q_i)
10:     tag[i][0] = q_i
11:     exp[i] = 0
12:   }
      // Initialisation of data structures in first dimension for q_i
13:   Q_T = {}
14:   Q_C = {}
      // Main loop for retrieving query results for U
15:   DO{
16:     FOR b = 0 TO batchsize {
17:       FOR i = 0 TO n − 1 {
            // Choose list and dimension by computing upper score bounds
18:         l, j =ChoseNextList(tag,exp,Taglist,α,β)
19:         IF (l == Social) {
20:           (U_f, s) = Social[i][j].next()
              // read docs, compute scores and add them to Q_T or Q_C if appropriate
21:           Q_T, Q_C = read(USERDOCS(U_f))
22:           high_so(U, tag[i][j]) = s
23:         }ELSE IF (l == Spiritual) {
24:           (U_f, s) = Spiritual[i][j].next()
              // read docs, compute scores and add them to Q_T or Q_C if appropriate
25:           Q_T, Q_C = read(USERDOCS(U_f))
26:           high_sp(U, tag[i][j]) = s
27:         }ELSE IF (l == Global) {
              // read docs, compute scores and add them to Q_T or Q_C if appropriate
28:           Q_T, Q_C = read(Global[i][j])
29:           high_gl(U, tag[i][j]) = lastTFReadFrom(Global[i][k])
30:         }ELSE IF (l == Taglist) {
              // open a new dimension and initialise data structures
31:           exp[i] + +
32:           tag[i][exp[i]] = Taglist[i].next()
33:           Social[i][exp[i]] = FLIST_so(U)
34:           Spiritual[i][exp[i]] = FLIST_sp(U)
35:           Global[i][exp[i]] = DOCS(tag[i][exp[i]])
36:           high_gl(U,tag[i][exp[i]]) = DOCS(tag[i][exp[i]])[0].getTF()
37:           high_so(U,tag[i][exp[i]]) = FLIST_so(U)[0].getStrength()
38:           high_sp(U,tag[i][exp[i]]) = FLIST_sp(U)[0].getStrength()
39:         }
40:       }
41:     }
42:     CheckRandomAccesses()
43:   }WHILE(CheckTermination()==No)
44: }
```


Listing 3: CONTEXTMERGE framework

The user expansion is achieved by processing in addition to the global document list DOCS(t_i), the social and spiritual friendship lists FLIST$_{so}(U)$ and FLIST$_{sp}(U)$ for adding documents from the (expanded) social and spiritual friends into the query processing.

Without Tag Expansion

To compute the *top-k* results for a query $q_U = \{t_0, \ldots, t_{n-1}\}$ submitted by a user U, our CONTEXTMERGE algorithm sequentially scans for all query tags t_i the global document list DOCS(t_i) and U's friendship lists FLIST$_{so}(U)$ and FLIST$_{sp}(U)$ in an interleaved way. Documents from user document lists USERDOCS(U_f, t_i) of social or spiritual friends U_f are read only when for a query tag t_i the user U_f is the next best friend in one of U's friendship lists.

The algorithm maintains a queue Q_C of candidate documents seen during the scans and a queue Q_T of current *top-k* documents, and terminates as soon as none of the candidates can make it into the *top-k* queue. To improve efficiency, the CONTEXTMERGE algorithm additionally performs random accesses to the inverted lists if appropriate to look up values for selected documents. Details are given later in this section.

To limit the number of disk accesses, the USERDOCS(U_f,t_i) lists for users U_f and query tags t_i are opened on demand, namely when the document score that can be read for a document from a USERDOCS(U_f,t_i) list is greater than the global score that can be retrieved for a document from the DOCS(t_i) list. To this end, the algorithm reads initially for each query tag t_i the document score of the top entry in DOCS(t_i) and the friendship strength of the top friend in each of the social and spiritual friendship lists FLIST$_{so}(U)$ and FLIST$_{sp}(U)$. The values are the initial settings for $high_d(t_i)$, $high_{so}(U, t_i)$ and $high_{sp}(U, t_i)$, respectively, and are used to compute upper bounds for document scores as described in the following.

In each iteration of the main loop, the CONTEXTMERGE algorithm performs a batch of $batchsize$ list accesses, where in each access a number of documents are read that potentially belong to the final *top-k* results with the highest document scores. Since no precomputed scores are stored in any inverted list of documents, but just tag frequencies, the score contributions and associated upper bounds have to be computed from each list at run-time. For this, while processing a query of a user U, our algorithm maintains for each dimension in $high_d(t_i)$ the last value read from the list DOCS(t_i) for tags t_i, and in $high_{so}(U, t_i)$ and in $high_{sp}(U, t_i)$ the last value read from U's social friendship list FLIST$_{so}(U)$ and spiritual friendship list FLIST$_{sp}(U)$, respectively.

Upper Score Bounds for Document Lists.

The upper bound max_gl for the score contribution read from the next entry of the global document list DOCS(t_i) for a query from a user U with a tag t_i can be computed by evaluating the single tag context score $sts(t_i, d, U)$ (see Definition 6.12) with $TF(t_i, d) = high_d(t_i)$ and neglecting the user-specific part by setting its score contribution to 0, i.e,

$$max_{gl}(t_i) = \frac{(k_1 + 1) \cdot (1 - \alpha - \beta) \cdot high_d(t_i)}{k_1 + (1 - \alpha - \beta) \cdot high_d(t_i)} \cdot idf(t_i)$$

Analogously, the upper score bounds $max\_so(t_i)$ and $max\_sp(t_i)$ for documents from U's social or spiritual friend at the next entry in U's social or spiritual friend-

ship list are computed by setting the score contributions from the global and spiritual, or global and social part, respectively, in $sts(t_i, d, U)$ to 0, considering only the remaining (social or spiritual) contribution. For this, the friendship strength $s_{so}(U, U_f)$ or $s_{sp}(U, U_f)$, respectively, of the next best friend U_f has to be assumed maximal, as well as the user-specific tag frequency $tf_{U_f}(t_i, d)$ (see Definition 6.8) that possibly could be read for document d from U_f's user document list USERDOCS(U_f, t_i) (see Section 6.3.1) for tag t_i.

The friendship strength of the next best social or spiritual friend cannot be higher than $high_{so}(U, t_i)$ or $high_{sp}(U, t_i)$, respectively, or the user had been found earlier in the corresponding friendship list. With $maxtf(t_i)$ being the maximal user-specific tag frequency, i.e. the maximal number of times a user has tagged a document with t_i, over all users and documents for a given query tag t_i, we finally can compute the upper score bounds $max\_so(t_i)$ and $max\_sp(t_i)$ as follows:

$$max\_sp(t_i) = \frac{(k_1+1) \cdot \alpha \cdot |\mathcal{U}| \cdot high_{sp}(U, t_i) \cdot maxtf(t_i)}{k_1 + \alpha \cdot |\mathcal{U}| \cdot high_{sp}(U, t_i) \cdot maxtf(t_i)} \cdot idf(t_i)$$

and

$$max\_so(t_i) = \frac{(k_1+1) \cdot \beta \cdot |\mathcal{U}| \cdot high_{so}(U, t_i) \cdot maxtf(t_i)}{k_1 + \beta \cdot |\mathcal{U}| \cdot high_{so}(U, t_i) \cdot maxtf(t_i)} \cdot idf(t_i)$$

Remark: Usually, $maxtf(t_i) = 1$ in today's social tagging networks. The required values of $idf(t_i)$ for computing the upper bounds can be fetched once during the initialisation of the execution.

Choosing the appropriate List.

Our CONTEXTMERGE algorithm greedily selects the list which has got the highest expected score for the next document to be read. The implementation is shown in Listing 4: *ChooseNextList*().

If a friendship list FLIST$_{so}(U)$ or FLIST$_{sp}(U)$ is selected, its next entry is read to determine U's friend U_f with the next highest friendship strength and all the documents from U_f's document list USERDOCS(U_f,t_i) for query tag t_i are completely read. Note that document lists USERDOCS(U, t_i) of users for a certain tag t_i are short compared to the global list of all documents tagged with the same tag t_i. Moreover, in most of today's social tagging applications, all documents in a user's document list have got the same tag frequency $tf(t_i) = 1$. Hence, it is reasonable to read all of the documents from a user in one shot.

If the next selected list is a global document list DOCS(t_i), a configurable number of entries are read from it. The DOCS(t_i) lists are usually much longer than the user document lists USERDOCS(U, t_i) because the former globally contain the documents for a tag t_i of all users in the network.

Further Note: The algorithm can be optimised if α or β is set to extreme values: For $\alpha + \beta = 1$, no DOCS(t_i) lists for any query tag need to be opened as the execution can be limited to the social and spiritual context of U; analogously, if $\alpha = \beta = 0$, there is no need to consider any lists of friends, so just the DOCS(t_i) for $t_i \in \{t_0, \ldots, t_{n-1}\}$ are read and our CONTEXTMERGE algorithm behaves like a standard *top-k* algorithm.

```
1:  ChoseNextList(tag[], exp[], Taglist[], α, β) {
      // get maximal upper bound for docs in new dimension
2:    Max_tl =upperBound(Taglist[i].peek())
3:    IF (α + β < 1)
        // get dimension j with maximal upper bound for docs from global list
4:      (Max_gl, j) =upperBound(∀_{j=0..exp[i]}DOCS(tag[i][j]),α, β)
5:    IF (α + β = 0){
6:      IF (Max_gl ≥ Max_tl){
7:        RETURN ("Global",j)
8:      }ELSE{
9:        RETURN ("Taglist")
10:     }
11:   }ELSE{
12:     U_so=FLIST_so(U).peek()
13:     U_sp=FLIST_sp(U).peek()
        // get dimension k and l with maximal upper bound for docs from friend's list
14:     (Max_so, k) =upperBound(∀_{k=0..exp[i]}USERDOCS(U_so,tag[i][k]),α, β)
15:     (Max_sp, l) =upperBound(∀_{l=0..exp[i]}USERDOCS(U_sp,tag[i][l]),α, β)
16:     IF (Max_tl > max{Max_gl, Max_so, Max_sp}){
17:       RETURN ("Taglist")
18:     }ELSE {
19:       IF (α + β < 1 && Max_gl ≥ max{Max_so, Max_sp}){
20:         RETURN ("Global",j)
21:       }ELSE {
22:         IF (Max_sp ≥ Max_so){
23:           RETURN ("Spiritual",l)
24:         }ELSE {
25:           RETURN ("Social",k)
26:         }
27:       }
28:     }
29:   }
30: }
```

Listing 4: *ChoseNextList* method

Candidate Management Q_C, Q_T.

With our CONTEXTMERGE algorithm, candidates for the query result are collected while scanning the inverted lists and maintained in two disjoint priority queues Q_T and Q_C, one for the currently considered *top-k* documents and another one for the candidate documents that still could make it into the final *top-k* results, respectively. During the processing of a query $q_U = \{t_0, \ldots, t_{n-1}\}$, our algorithm maintains for each *top-k* or candidate document d_j the following information:

- $E_{gl}(d_j) = \{\text{DOCS}(t_i), ...\}$:

  the set of global document lists DOCS(t_i) for tags t_i in which the document d_j has already been discovered.

- $E_{so}(U, d_j) = \{\text{USERDOCS}(U_f, t_i), ...\}$:

  the set of lists USERDOCS(U_f, t_i) of U's social friends U_f and any tag t_i in which the document d_j has already been discovered.

- $E_{sp}(U, d_j) = \{\text{USERDOCS}(U_f, t_i), ...\}$:

  the set of lists USERDOCS(U_f, t_i) of U's spiritual friends U_f and any tag t_i in which the document d_j has already been discovered.

- $r_{so}(d_j, t_i)$:

  the number of times document d_j has been discovered for tag t_i in document lists USERDOCS(U_f,t_i) of social friends U_f.

- $r_{sp}(d_j, t_i)$:

  the number of times document d_j has been discovered for tag t_i in document lists USERDOCS(U_f,t_i) of spiritual friends U_f.

- $TF(t_i, d_j)$:

  the global tag frequency of d_j, read from the global document list DOCS(t_i).

- $uf_{so}(d_j, t_i) = \sum_{\text{USERDOCS}(U_f,t_i) \in E_{so}(U,d_j)} s_{so}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$:

  the weighted sum over the user-specific tag frequencies for document d_j, read from user document lists USERDOCS(U_f, t_i) of social friends U_f.

- $uf_{sp}(d_j, t_i) = \sum_{\text{USERDOCS}(U_f,t_i) \in E_{sp}(U,d_j)} s_{sp}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$:

  the weighted sum over the user-specific tag frequencies for document d_j, read from the user document lists USERDOCS(U_f, q_i) of spiritual friends U_f.

- $ws(d_j)$:

  the worst score of the document d_j which is a lower bound for the total score of d_j and is computed from the values seen so far for d_j,

- $bs(d_j)$:

  the best score of the document d_j which is an upper bound for the total score of d_j.

Worst Scores.

To compute the worst score $ws(d_j)$ of a candidate d_j, for each query tag t_i, the single tag context score $sts(t_i, d_j, U)$ (see Definition 6.12) is evaluated by using only the already discovered score contributions for computing the context frequency $cf_U(t_i, d_j)$ (see Definition 6.10)

1. The global term frequency is set to 0 for all dimensions in which d_j has not yet been discovered, i.e.

$$TF(t_i, d_j) = 0 \text{ for } \textsc{Docs}(t_i) \notin E_{gl}(d_j)$$

2. In the user-specific context part, the weighted sum over user-specific tag frequencies is used only on values that have already been read from the friends' document lists (instead of the weighted sum over the tag frequencies from all users' document lists), i.e.

$$uf_{so}(d_j, t_i) \text{ is used instead of } \sum_{U_f \in \mathcal{U}} s_{so}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$$

and

$$uf_{sp}(d_j, t_i) \text{ is used instead of } \sum_{U_f \in \mathcal{U}} s_{sp}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$$

such that the context frequency used for computing the worst score is the following:

$$cf'_U(t_i, d_j) = \begin{cases} \alpha \cdot |\mathcal{U}| \cdot uf_{so}(d_j, t_i) & \\ \quad + \beta \cdot |\mathcal{U}| \cdot uf_{sp}(d_j, t_i) & \text{if } \textsc{Docs}(t_i) \notin E_{gl}(d_j) \\ (1 - \alpha - \beta) \cdot TF(t_i, d_j) & \\ \quad + \alpha \cdot |\mathcal{U}| \cdot uf_{sp}(d_j, t_i) & \\ \quad + \beta \cdot |\mathcal{U}| \cdot uf_{so}(d_j, t_i) & \text{otherwise.} \end{cases}$$

Note that in conjunctive evaluation, the worst score of a candidate remains 0 until, for each query tag t_i, the document has been seen in $\textsc{Docs}(t_i)$ or in one of the $\textsc{UserDocs}(U_f,t_i)$ lists read for spiritual or social friends U_f.

Best Scores.

To compute the best score $bs(d_j)$ of a document d_j, we have to add to its worst score the maximal score contribution from lists in which d_j has not yet been seen. For this, we evaluate the context frequency $cf_U(t_i, d_j)$ (see Definition 6.10) in the following way:

1. We estimate an upper bound for the global document list $\textsc{Docs}(t_i)$, i.e.

$$TF(t_i, d_j) = high_d(t_i) \text{ for } \textsc{Docs}(t_i) \notin E_{gl}(d_j)$$

2. For the user-specific context part, we need to estimate the additional contribution $C_{so}(t_i, d_j)$ and $C_{sp}(t_i, d_j)$ for a tag t_i and document d_j from users that

have not yet been seen and potentially are social or spiritual friends, such that we can replace, as in the case of the worst score, the weighted sum by the already computed part of the score plus an upper bound for the not yet known contribution, i.e.

$$uf_{so}(d_j, t_i) + C_{so}(t_i, d_j) \text{ is used instead of } \sum_{U_f \in \mathcal{U}} s_{so}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$$

and

$$uf_{sp}(d_j, t_i) + C_{sp}(t_i, d_j) \text{ is used instead of } \sum_{U_f \in \mathcal{U}} s_{sp}(U, U_f) \cdot tf_{U_f}(t_i, d_j)$$

such that the context frequency used for computing the best score is the following:

$$cf'_U(t_i, d_j) = \begin{cases} (1-\alpha-\beta) \cdot high_d(t_i) & \\ \quad + \alpha \cdot |\mathcal{U}| \cdot (uf_{sp}(d_j, t_i) + C_{so}(t_i, d_j)) & \\ \quad + \beta \cdot |\mathcal{U}| \cdot (uf_{so}(d_j, t_i) + C_{sp}(t_i, d_j)) & \text{if } \textsc{Docs}(t_i) \notin E_{gl}(d_j) \\ & \\ (1-\alpha-\beta) \cdot TF(t_i, d_j) & \\ \quad + \alpha \cdot |\mathcal{U}| \cdot (uf_{sp}(d_j, t_i) + C_{so}(t_i, d_j)) & \\ \quad + \beta \cdot |\mathcal{U}| \cdot (uf_{so}(d_j, t_i) + C_{sp}(t_i, d_j)) & \text{otherwise.} \end{cases}$$

Computing $C_{so}(t_i, d_j)$ and $C_{sp}(t_i, d_j)$. As the algorithm considers users in descending order of their friendship strengths, we know, for users U_f who have not yet been considered for the score computation t_i, that

$$s_{so}(U, U_f) \leq high_{so}(U, t_i)$$

and

$$s_{sp}(U, U_f) \leq high_{sp}(U, t_i)$$

Hence, denoting with

$$mass_{so} = \sum_{\substack{\textsc{UserDocs}(U_f, t_i) \\ \in E_{so}(U, t_i)}} s_{so}(U, U_f)$$

and

$$mass_{sp} = \sum_{\substack{\textsc{UserDocs}(U_f, t_i) \\ \in E_{sp}(U, t_i)}} s_{sp}(U, U_f)$$

the sum of social and spiritual friendships strengths of users already considered for t_i, respectively, and by $maxtf$ the maximal tag frequency of any user for any document and tag in the collection (which again is usually 1), we can estimate the maximal remaining contribution by unseen social and spiritual friends as

$$C_{so}(t_i, d_j)_{(mass)} = (1 - mass_{so}) \cdot maxtf$$

and

$$C_{sp}(t_i, d_j)_{(mass)} = (1 - mass_{sp}) \cdot maxtf$$

because $s_{so}(U, U_f)$ and $s_{sp}(U, U_f)$ are normalised to a sum of 1.

Another way of estimating the contribution of unseen users can be achieved if additionally the maximal number $max_U(t_i)$ of users is known who tagged any document with t_i. The maximal contribution from unseen social or spiritual friends for d_j and t_i is then at most

$$C_{so}(t_i, d_j)_{(max_U)} = (max_U(t_i) - r_{so}(d_j, t_i)) \cdot maxtf \cdot high_{so}(U, t_i)$$

and

$$C_{sp}(t_i, d_j)_{(max_U)} = (max_U(t_i) - r_{sp}(d_j, t_i)) \cdot maxtf \cdot high_{sp}(U, t_i)$$

The value of $max_U(t_i)$ can be read during the initialisation of the algorithm.

Furthermore, the score estimation can be made more precise if the global tag frequency $TF(t_i, d_j)$ (see Definition 6.9) is known because in this case, $max_U(t_i)$ can be replaced by $TF(t_i, d_j)$ for estimating the additional score contribution $C_{so}(t_i, d_j)$ and $C_{sp}(t_i, d_j)$. The total amount of times a document d_j was tagged with t_i by any user in the social network is often much smaller than the total number of all users $max_U(t_i)$ who ever used t_i on any document. Should $TF(t_i, d_j)$ be not (yet) known but documents from the global document list DOCS(t_i) are read during the query execution, i.e. $(1 - \alpha - \beta) > 0$, then $high_d(t_i)$ can be used instead of $TF(t_i, d_j)$ since the latter cannot be higher than $high_d(t_i)$ or the correct global tag frequency would have already been read during the scan of the corresponding global document list DOCS(t_i). Hence,

$$C_{so}(t_i, d_j)_{(TF)} = \begin{cases} (TF(t_i, d_j) - r_{so}(d_j, t_i)) \cdot high_{so}(U, t_i) & \text{if } TF(t_i, d_j) \text{ is known} \\ (high_d(t_i) - r_{so}(d_j, t_i)) \cdot high_{so}(U, t_i) & \text{otherwise.} \end{cases}$$

and

$$C_{sp}(t_i, d_j)_{(TF)} = \begin{cases} (TF(t_i, d_j) - r_{sp}(d_j, t_i)) \cdot high_{sp}(U, t_i) & \text{if } TF(t_i, d_j) \text{ is known} \\ (high_d(t_i) - r_{sp}(d_j, t_i)) \cdot high_{sp}(U, t_i) & \text{otherwise.} \end{cases}$$

Finally, the computation of the upper bound for the best score of a document d_j for a query tag t_i eventually can be made most precise during the query processing, by taking the minimum of the above mentioned estimations, i.e.

$$C_{so}(t_i, d_j) = \min\{C_{so}(t_i, d_j)_{(mass)},\ C_{so}(t_i, d_j)_{(max_U)},\ C_{so}(t_i, d_j)_{(TF)}\}$$

and

$$C_{sp}(t_i, d_j) = \min\{C_{sp}(t_i, d_j)_{(mass)},\ C_{sp}(t_i, d_j)_{(max_U)},\ C_{sp}(t_i, d_j)_{(TF)}\}$$

Termination Test.

For the termination test, the following information is derived and maintained at each step during the query execution:

- *min_ws*$_k$:

  the minimum worst score of the current *top-k* documents. It serves as the stopping threshold.

- *max_bs* :

  the maximum best score that any currently unseen document can get. It is computed by assuming a *virtual* document d_v, representing any unseen document, which appears right at the front of the not yet seen part of all document lists. Its best score $bs(d_v)$ is then computed in each dimension by setting $TF(t_i, d_v) = high_d(t_i)$ and estimating the contribution from not yet seen users like previously described.

Finally, the $CheckTermination()$ method of our CONTEXTMERGE algorithm can safely indicate its termination while yielding the correct *top-k* results, when the maximum best score of a document in the candidate queue and the best score *max_bs* of any unseen document is not larger than *min_ws*$_k$. Additionally, whenever the best score of a document in the candidate queue is not higher than *min_ws*$_k$, this candidate can be pruned from the queue. Thus, the memory footprint of the execution is kept low as unneeded candidates can be removed early in the process. To further limit the CPU overhead of testing the candidates, the $CheckTermination()$ test is only performed after a full batch of scan steps, and only enabled after the maximal best score *max_bs* of the unseen, virtual document d_v is not greater than *min_ws*$_k$.

Random Accesses.

In addition to sequential accesses (SA) to the index lists, our CONTEXTMERGE algorithm can also perform random accesses (RA) to the index lists in order to look up missing scores of candidates.

However, it is not feasible to check all user document lists USERDOCS(U_f, t_i) of not yet seen users U_f for a document d_j, as this would require to explore the full range of potentially many thousands of transitive friends. Therefore, the only type of RA applied by CONTEXTMERGE is RA to global document lists DOCS(t_i) to look up the global tag frequency of a document for a tag t_i. This serves two purposes: first, it can reduce the gap between the document's worst score and best score values because $TF(t_i, d_j)$ is then known exactly for one more tag; second, the estimation of the score contribution from the remaining friends gets more precise as $TF(t_i, d_j)$ can be used in the estimation instead of $max_U(t_i)$ as previously described.

As RA are much more expensive than SA (in the order of 100 to 1,000 times for real systems), they have to be carefully selected and scheduled to avoid any unnecessary work. Our scheduling for RA follows the *LAST* heuristics from [20]: our algorithm performs only SA until the estimated cost to perform *all* RA to remaining candidates is at most as high as the cost for all SA done so far. We estimate the number of RA by summing up, for all candidates, the number of query dimensions (i.e. original query tags) that have not yet been evaluated.

Including Tag Expansion

Tag expansion adds another dimension that needs to be combined with the user expansion dimension in our CONTEXTMERGE algorithm. In Listing 3, we represent these additional dimensions by multidimensional arrays:

- $DOCS[i][j]$ for global documents for expanded tags t_j from t_i,
- $FRIENDS_{so}[i][j]$ for social friends in the expanded tag dimension t_j,

- $FRIENDS_{sp}[i][j]$ for spiritual friends in the expanded tag dimension t_j.

Assume for a query q_U from user U with query tag t_i, the global document list DOCS(t_i), and the social and spiritual friendship lists FLIST$_{so}$(U) and FLIST$_{sp}$(U), respectively, are opened and stored in the arrays

$$DOCS[i], FRIENDS_{so}[i] \text{ and } FRIENDS_{sp}[i],$$

Conceptually, the tag expansion for a query tag t_i means that for each tag t_j similar to a query tag t_i a new dimension is opened, i.e.

$$DOCS[i][j], FRIENDS_{so}[i][j] \text{ and } FRIENDS_{sp}[i][j],$$

representing the global document list DOCS(t_j), and for tag t_j, new instantiations of U's friendship lists of social and spiritual friends, respectively.

In summary, the user expansion for a query tag t_i is realised in the implementation of our CONTEXTMERGE algorithm by introducing in addition to the array representing the global document list for t_i, i.e. $DOCS[i]$, new arrays for representing a user's social and spiritual friends, i.e. $FRIENDS_{so}[i]$ and $FRIENDS_{sp}[i]$, whereas the tag expansion is realised by adding to the existing arrays another dimension j, representing the global document list and friendship lists in the new dimension of the expanded tag t_j.

Extended Candidate Management. The upper score bounds for these additional lists are computed like the bounds for lists without tag expansion, but are additionally multiplied with the tag similarity of the expanded tag t_j, i.e. $tsim(t_i, t_j)$, read from the associated tag similarity list SIMTAGS(t_i). As a consequence, the candidate management needs to be extended for each current *top-k* and candidate document in order to maintain not only the information used to compute best and worst scores of documents in regard to a query tag t_i but also to each similar tag t_j. Following the $\max$-semantics of our tag expansion score, worst scores and best scores of candidates are estimated by the maximum worst score and best score over each query tag and its similar tags weighted by the associated tag similarity $tsim(t_i, t_j)$.

However, it would be very inefficient to directly include the lists of all similar tags in the query processing. Instead, the number of expansion tags per original query tag are limited (e.g. by a limit of 10) with our CONTEXTMERGE algorithm and lists for similar tags are incrementally added to the processing on the fly.

To do this, the algorithm maintains for each query tag t_i the list SIMTAGS(t_i) of tags t_j similar to t_i and includes these lists in the selection process of the next list to open. Listing 4 shows the framework of the corresponding $ChooseNextList()$ method.

To compute the upper bounds of scores for documents that could be retrieved when considering a similar tag from the list SIMTAGS(t_i), our CONTEXTMERGE algorithm first reads the next entry $(t_j, tsim_w(t_i, t_j))$ from the list without actually processing the list to its subsequent entry and looks up the value of $idf(t_j)$.

The maximal score contribution achieved when reading the next similar tag t_j in SIMTAGS(t_i) is estimated by computing the maximum over the score contributions from the global document list DOCS(t_j) and U's social or spiritual friendship lists FLIST$_{so}$(U) or FLIST$_{sp}$(U) for tag t_j. The so computed maximal score for a document with an expanded tag t_j similar to t_i is finally weighted by the tag similarity $tsim(t_i, t_j)$.

As shown in Listing 3, only if the method $ChooseNextList()$ chooses the tag similarity list SIMTAGS(t_i) to be proceeded to its next entry (i.e. Line 30, ($l == "Taglist"$) is true), our CONTEXTMERGE algorithm expands the processing by adding the next tag dimension to $DOCS[i][j]$, $FRIENDS_{so}[i][j]$ and $FRIENDS_{sp}[i][j]$ for the tag t_j similar to t_i. By this dynamic handling of tag expansions, the computation of worst scores for candidates must take into account that the actual score of a document for a query tag t_i is the maximum over all scores from opened lists for similar tags t_j. In addition, the best score computation needs to consider that not all lists for expanded tags may have been opened already. For each query tag t_i where the tag expansion limit has not yet been reached, it therefore computes the maximal score that *any* document can get for the next similar tag t_j in SIMTAGS(t_i). The best score of a document is then the maximum best score it can obtain in all opened lists and from the next tag expansion. Note that this bound is only correct because the entries in SIMTAGS(t_i) are sorted by $tsim(t_i, t_j) \cdot idf(t_j)$.

6.4 Experiments

To evaluate our CONTEXTMERGE algorithm and its scoring model we ran experiments on two different general scoring configurations, denoted by a *social-context configuration* and a *full-context configuration*.

First, with our *social-context configuration*, we consider only social friendship relations of users combined with the global view on the friendship graph in social tagging networks, trimming the score of a document (see Definition 6.15) by using only the social and global friendship strengths (see Definition 6.5 and 6.7) of users for computing the context frequency (see Definition 6.10) with respect to a user U, tag t and document d. Hence, with the definition of

$$cf_U(t,d) = \sum_{U' \in \mathcal{U}} (\alpha \cdot s_{so}(U,U') + \beta \cdot s_{sp}(U,U') + (1-\alpha-\beta) \cdot s_{gl}(U,U')) \cdot tf_{U'}(t,d)$$

we're using a fixed $\beta = 0$ and only varying $\alpha \in [0,1]$ instead.

Afterwards, with our *full-context configuration*, we compute the context frequency in its entirety, including also the spiritual friendship strengths of users in the computation by varying $(\alpha + \beta) \in [0,1]$ as defined in Definition 6.1.

We evaluate the effectiveness of our CONTEXTMERGE algorithm with both configuration setups by computing precision and normalised discounted cumulative gain (NDCG) [72] for the *top-10* results with relevance assessments of result documents in three rating levels: 0 (non-relevant), 1 (relevant) and 2 (highly relevant). Details on how the relevance assessments of documents are accomplished are given in Section 6.4.1 and 6.4.2.

Hence, we compute

- the *precision* for the *top-10* results, treating both ratings of 2 and 1 as relevant. The precision measure is defined in Definition 5.18 as follows:

$$precision = \frac{\#\text{relevant docs retrieved}}{\text{total } \#\text{retrieved docs}}$$

- the *normalised discounted cumulative gain (NDCG)* [72] for the top-10 results.

  DCG aggregates the ratings (2, 1, or 0) of the results with geometrically decreasing weights towards lower ranks:

$$DCG \propto \sum_{rank\ i} \frac{2^{rating(i)} - 1}{log_2(1+i)}$$

  which is then normalised into NDCG by dividing by the DCG of an *ideal* result, i.e. first all results with rating 2, followed by all results with rating 1, followed by results with rating 0. Formally, let be DCG_I the DCG value of an *ideal* ranking, then NDCG is defined as follows:

  Definition 6.18 (Normalized Discounted Cumulative Gain)**.**

$$NDCG = \frac{1}{DCG_I} \cdot \sum_{rank\ i} \frac{2^{rating(i)} - 1}{\log_2(1+i)}$$

■

  NDCG is a widely adopted standard measure in IR.

6.4.1 Social-Context Configuration

We evaluate the effectiveness of our scoring model and the efficiency of the CONTEXTMERGE algorithm using a social-context configuration on three datasets retrieved from three real world social tagging networks (see Section 3.2).

- ***Delicious.com*:** The dataset that we harvested by crawling the social bookmarking service *Delicious.com* contains a total of $12,389$ users, $175,754$ bookmarks with $2,781,096$ tags, and $152,306$ friendship connections.

- ***Flickr.com*:** For our experiments on the photo hosting network *Flickr.com*, we obtained a dataset with a total of $52,347$ users, $10,000,000$ images annotated with $29,111,183$ tags, and $1,293,777$ friendship connections. In addition to explicit friends defined by the user (which rarely happens in *Flickr.com*), we also considered two users as friends if one of them commented a photo uploaded by the other.

- ***LibraryThing.com*:** By crawling the social book cataloguing service *LibraryThing.com*, we retrieved a third dataset with a total of $9,986$ users, $6,453,605$ books with $14,295,693$ tags, and $17,317$ friendship edges. As users in *LibraryThing.com* typically use tags that consist of multiple terms, we extracted the terms from the tags and considered the set of terms used by a user for a book as if the book was tagged with these terms. The friendship relation is here again defined in a broader way, and consists of explicit friends and users marked as having interesting libraries.

For detailed information on the social tagging networks *Delicious.com*, *Flickr.com* and *LibraryThing.com*, see Section 3.2.

Relevance Assessments

Finding a good set of queries and relevant results for them is not an easy task. Even though there has been some work on evaluating queries in social networks, most notably by Bao et al. [19] who use DMOZ categories as global ground truth, such methods don't fit our user-centric search task. Here, it is not sufficient to consider global relevance measures, as the notion of relevance is highly subjective and dependent on the initiator of the query and her personal context. For example, a photo of a person may only be relevant if she is known to the query initiator. However, when we execute our queries in the context of a user taken from our collections, it is not possible to ask this user to assess the subjective relevance of a result item.

To solve this problem, we propose two independent evaluation methods, a *user-specific ground truth* and a *user study* with manual relevance assessments for selected queries.

- **User-Specific ground truth.** For a given query $q_U = \{t_0, \ldots, t_{n-1}\}$ from a user U, we consider as user-specific ground truth the union of all documents which were tagged with all query tags by U or any of her direct social friends. Our decision to consider also documents of friends as part of the ground truth is based on the fact that these are also documents that the user has got direct access to and it is likely that the user has seen, and agreed with the tags assigned. Since documents on the ground truth set contain tags from the query, we evaluate the queries on a residual collection where query tags by U and her friends are removed as they are known to lead towards relevant results. Note that this introduces a penalty for the social-context configuration of our algorithm as the transitive friends with highest friendship strengths cannot contribute relevant results by definition. Queries for the ground truth were randomly selected from tag pairs with medium frequency in the corpus (between $1,000$ and $2,000$), the query initiator was chosen randomly among users that have previously used the query tags and have at least one friend in the collection. This process yielded 150 queries for *Delicious.com* and 184 queries for *LibraryThing.com*; note that this method of identifying ground truth cannot be applied to the *Flickr.com* dataset because by the nature of corresponding social tagging network, there is almost no overlap in the photos that users tag.

- **User Study.** Our user study is done on the *LibraryThing.com* and *Flickr.com* dataset. For *Delicious.com* we found the manual relevance assessment much more difficult and a too time consuming task for people not knowing the bookmarked web pages because of the high effort of browsing through many bookmarks, navigating to the associated websites and reading and understanding their contents in order to hopefully get an idea of the interests of a query user and to know if a resulting bookmark might meet or not meet those interests. For our user study on the *LibraryThing.com* dataset, we asked five colleagues to register with *LibraryThing.com*, tag at least 20 books there, and choose some friends among other users. They then suggested 28 queries related to the books they tagged. For *Flickr.com*, we collected 40 queries from other colleagues and randomly selected a (fictitious) query initiator among the users in our crawl from *Flickr.com* who used all query tags at least once on the same photo. We then ran the queries with different configurations of our algorithm and pooled the results. In the assessment phase, a volunteer (which was the query initiator for *LibraryThing.com*)

was shown, in addition to the results for the query from the pool, the documents (i.e. books or photos) from the query initiator that contain at least one of the query tags in order to understand the personal context of the query initiator. In this way, we try to overcome the problem of subjectively assessing result qualities with the eyes of the query initiator. The participant then marks each result as highly relevant, relevant, or non-relevant in the context of the query initiator without knowing which configuration generated the result.

Retrieval Effectiveness

Results for the User Studies. The results from the user study are shown in Table 6. For both the *Flickr.com* and the *LibraryThing.com* dataset set the NDCG improved by increasing α, but decreased when setting α to 0. This shows that while the semantic aspect is more important than the social aspect for these specific datasets, the social component helps improving the search result, in particular for *Flickr.com*. This can also be seen with the precision[10] in Table 7, which for *Flickr.com* starts at 0.50 for $\alpha = 1.0$, increases to 0.54 for $\alpha = 0.1$ and drops to 0.40 for $\alpha = 0.0$.

Results for the User-Specific Ground Truth. The ground truth based experiments (Table 8) show very similar results. Again, result quality improves when decreasing α, but drops when ignoring the social aspect and setting $\alpha = 0$. For *Delicious.com* the social aspect seems to be more important than for the other datasets, here the optimal value is $\alpha = 0.2$. Overall search effectiveness clearly benefits from integrating social scores.

Without Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Flickr	0.39	0.42	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.36
LibraryThing	0.61	0.65	0.65	0.66	0.67	0.66	0.66	0.68	0.70	0.72	0.71

With Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Flickr	0.42	0.40	0.40	0.40	0.40	0.39	0.39	0.40	0.40	0.40	0.36
LibraryThing	0.61	0.63	0.64	0.65	0.65	0.65	0.65	0.67	0.69	0.72	0.71

Table 6: NDCG[10] for varying α, manual assessments

Without Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Flickr	0.50	0.53	0.53	0.53	0.53	0.53	0.54	0.54	0.54	0.54	0.40
LibraryThing	0.65	0.67	0.68	0.69	0.69	0.69	0.79	0.70	0.72	0.75	0.75

With Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Flickr	0.54	0.53	0.53	0.53	0.53	0.52	0.52	0.53	0.53	0.52	0.41
LibraryThing	0.64	0.65	0.67	0.68	0.68	0.68	0.68	0.69	0.72	0.75	0.74

Table 7: Precision[10] for varying α values, manual assessments

Without Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Delicious	0.15	0.25	0.27	0.33	0.34	0.35	0.35	0.37	0.39	0.39	0.36
LibraryThing	0.29	0.42	0.49	0.54	0.53	0.55	0.56	0.59	0.60	0.63	0.62

With Tag Expansion											
α	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.7	0.2	0.1	0.0
Delicious	0.16	0.25	0.28	0.36	0.36	0.36	0.36	0.39	0.40	0.37	0.36
LibraryThing	0.30	0.41	0.46	0.53	0.52	0.53	0.55	0.57	0.59	0.61	0.60

Table 8: Precision[10] for varying α values, ground truth experiments

Retrieval Efficiency

Another main concern in this work has been the efficiency and scalability of the query processing. To assess the efficiency of our CONTEXTMERGE algorithm, we evaluated the two sets of queries used for the ground-truth based evaluation on the *LibraryThing.com* and *Delicious.com* datasets, and in addition a set of 164 queries on the *Flickr.com* dataset that were computed similarly to the others. The algorithm was implemented in Java, with the index lists stored in an Oracle 10g database. The experiments were done on an Windows-based server machine with four single-core Opteron CPUs and 16GB of main memory. We measured wall-clock runtimes and abstract cost in terms of disk accesses, where the cost for a random access was 100 and the cost for a sequential access 1. We compared our algorithm with a standard join-then-sort algorithm that reads all index lists involved in the query execution into memory, uses an in-memory hash join to combine entries for the same document, and finally does an in-memory sort of the candidate set to compute the *top-k* results.

For each collection, we performed a large variety of experiments to explore the space of possible values of α, thresholds for maximal tag expansion, and conjunctive vs. disjunctive evaluation. However, we limit the discussion to selected results with conjunctive evaluation since results with disjunctive evaluation generally followed the same trends. Figure 6 depicts the average abstract cost per query for the three collections and selected values of α, evaluated with CONTEXTMERGE and the baseline without tag expansion. It is evident that our highly efficient *top-k* algorithm has got at most half the cost of the baseline algorithm for most values of α, and shows even higher savings for the *LibraryThing.com* collection. Table 9 shows some additional details for the experiments on *LibraryThing.com*; it is evident from the table that wall-clock runtime of our algorithm is also at least 50% better than that of the baseline (which is also the case for the other two collections).

The Figure 7 shows the abstract cost for the same setup but now with tag expansion up to a limit of 10 similar tags. Note that the bars for the *LibraryThing.com* baseline experiments have been cut at $350,000$. Here, the effectiveness of our dynamic tag expansion is clearly evident, as it saves factors of 3 to 5 compared to the baseline method which needs to completely scan the lists of all 10 related tags for each query tag. Table 9 again shows details for some experiments on *LibraryThing.com*; here, our highly efficient method manages to reduce runtime by up to an order of magnitude over the baseline. The column *avg.#exp* shows the average number of similar tags considered per query. Whereas the baseline methods needs to consider all tags, our self-throttling expansion technique requires only very few similar tags.

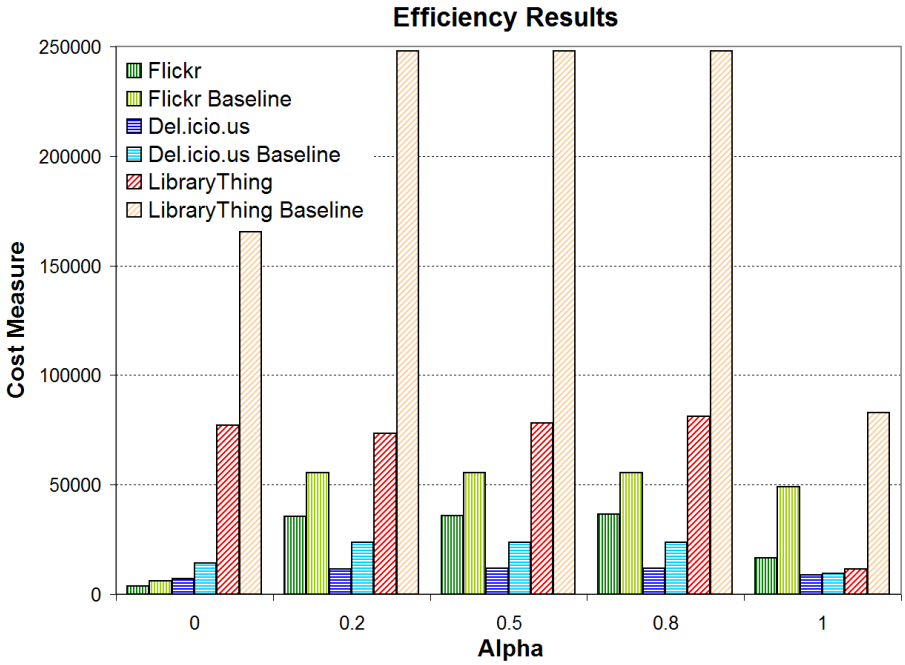


Figure 6: Average Execution Cost without tag expansion

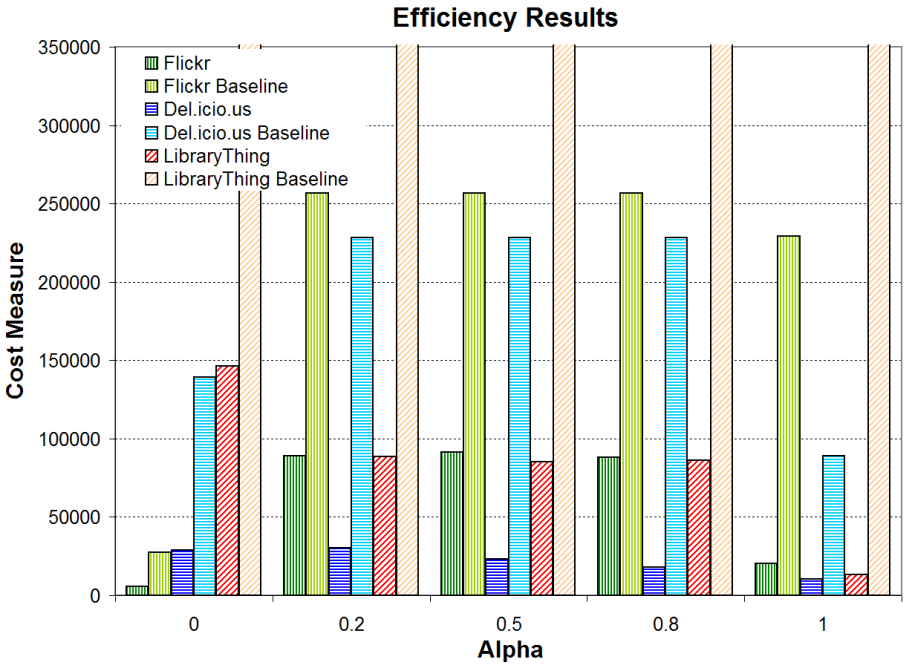


Figure 7: Average Execution Cost with tag expansion

Configuration	time[s]	avg.#SA overall	avg.#SA to DOCS	avg.#SA to USERDOCS	avg.#RA overall	#avg. #exp
$alpha = 1.0$						
CONTEXTMERGE	0.70	70,588	0	70,588	65	0
CONTEXTMERGE w/ tag exp.	1.37	126,772	0	126,772	194	7
Baseline	1.43	165,352	0	165,352	0	0
Baseline w/ tag exp.	6.10	67,0405	0	67,0405	0	20
$alpha = 0.5$						
CONTEXTMERGE	0.68	76,012	21,834	54,178	23	0
CONTEXTMERGE w/ tag exp.	0.84	78,808	22,063	56,745	64	2
Baseline	2.0	248,093	82,742	165,351	0	0
Baseline w/ tag exp.	9.92	1,118,554	448,149	670,405	0	20
$alpha = 0.0$						
CONTEXTMERGE	0.14	11,341	11,341	0	1	0
CONTEXTMERGE w/ tag exp.	0.21	12,223	12,223	0	9	1
Baseline	0.59	82742	82,742	0	0	0
Baseline w/ tag exp.	3.43	448,149	448,149	0	0	20

Table 9: Efficiency details for LibraryThing with conjunctive evaluation

6.4.2 Full-Context Configuration

With a full-context configuration, our CONTEXTMERGE algorithm considers the spiritual, social and global friendship strengths (see Definition 6.2, 6.5, 6.7) of users with respect to a query initiator for computing scores of documents as defined by our scoring model in Section 6.2.

Retrieval Effectiveness

To study the effectiveness of our fully context-specific scoring model, taking social and spiritual friendship relations into account, we performed again experiments with data extracted from partial crawls of the social book cataloguing service *LibraryThing.com*. We concentrated on *LibraryThing.com* only, because it is the most interesting social tagging network among the ones introduced in Section 3.2, i.e. *LibraryThing.com*, *Delicious.com* and *Flickr.com*. We found from the previous experiments in Section 5.4 and 6.4.1 that the social aspects in *Delicious.com* to be rather marginal, as most bookmarked pages are of fairly high quality anyway; so a user does not benefit from her friends' recommendations more than from the overall community. Moreover, manual relevance assessment from the viewpoint of an unknown user in *Delicious.com* is difficult as described in Section 6.4.1. *Flickr.com*, on the other hand, has recently grown so much that the tagging quality seems to be gradually degrading, also due the fact that only the owner of a photo or her friends usually can provide tags since, by the nature of photography and the photo hosting service *Flickr.com* in particular, there is almost no overlap in photo collections of users. Hence, mostly only the owner of a photo indeed annotates a photo with tags and these are sometimes relatively unspecific and given to an entire series of photos (e.g. vacation July 2007) instead of photos being individually tagged. *LibraryThing.com*, in contrast, features intensive tagging of a quality-controlled set of items, namely, published books, and its users have built up rich social relations. Finally, books are a matter of subjective taste, so that social relations do indeed have got high potential value. You trust your friends' taste, not necessarily their "technical" expertise.

For our studies, we extracted a dataset from the *LibraryThing.com* website including $11,717$ users who together own or have read $1,289,128$ distinct books with a total of $14,738,646$ tagging events (including same tags for the same book by different users), and $17,915$ social friendship relations (see Definition 3.1). For the latter, we used the *LibraryThing.com* notion of friends (where users mutually agree on be-

user 1	**user 2**	**user 3**
thailand travel	web learning	time traveler
asia guide travel	mountain climbing	leonardo vinci
technology enhanced learning knowledge management	kali death	english grammar
multimedia metadata standards	buddha	romance prague
knowledge management media theory	houdini	brazilian literature
social network analysis theory	science illusion magic	shalespeare play
multimedia social software	mystery magic	stephanie plum
	religion irony humor	search engines
	yakuza	spanish literature
	hitman	portuguese literature
		harry potter
		wizard

user 4	**user 5**	**user 6**
religion god world	information retrieval	sf nebula winner
challenge theory	probability statistics	fantasy politics
imagination fantasy science	database system	fantasy dragaera
drama story novel	transaction management	sf nuclear war
magic fantasy	data mining	fantasy malazan
india philosophy	software development	
fantasy story		
novel family life		
science fiction future		

Table 10: Queries of the user study

ing friends) and the notion of referring to an "interesting library" (see Section 3.2.3). The users included 6 users from our institute who have been contributing to *LibraryThing.com* for an extended time period and have made various social connections. These 6 users ran queries on the dataset and assessed the quality of the results in a user study.

Note that such human assessment is indispensable for this kind of experiments, and in our setting it was crucial that a query result was assessed by the same user who posed the query. Altogether, our 6 test users ran 49 queries, shown in Table 10.

Query results were computed for a variety of values of α and β, with and without tag expansion. The results from all runs for the same query were pooled; all of them together were shown to the corresponding user in random order (in a browser-based GUI), and the user assessed the quality of each result by assigning one of three possible ratings:

α \ β	0.0	0.2	0.5	0.8	1.0
0.0	0.666	**0.698**	**0.688**	**0.682**	0.680
0.2	0.661	0.678	**0.686**	**0.690**	n/a
0.5	0.637	0.657	0.663	n/a	n/a
0.8	0.612	0.647	n/a	n/a	n/a
1.0	0.549	n/a	n/a	n/a	n/a

Table 11: Precision[10] for all users

α \ β	0.0	0.2	0.5	0.8	1.0
0.0	0.546	**0.572**	**0.568**	0.565	0.565
0.2	0.564	**0.572**	**0.579**	**0.581**	n/a
0.5	0.539	0.552	0.559	n/a	n/a
0.8	0.515	0.546	n/a	n/a	n/a
1.0	0.465	n/a	n/a	n/a	n/a

Table 12: NDCG[10] for all users

0 = irrelevant or uninteresting

1 = relevant and interesting

2 = super-relevant and very appealing

Results that the user already knew, that is, books that she has got in her own library, are always discarded.

The Tables 11 and 12 show the precision and NDCG values for different choices of the configuration parameters α and β, without any form of tag expansion. These are micro-averaged results over all test users. Values printed in boldface are results that were significantly better than the baseline case ($\alpha = \beta = 0$) according to a statistical t-test with test level 0.1 [109].

The results show that both social (increasing α) and spiritual (increasing β) processing can improve the result quality. This holds for each of these two directions individually, and the combined effect is even better with a typical maximum at $\alpha = 0.2$ and $\beta = 0.8$. It may seem that the improvements, for example, from an NDCG value of 0.546 for the baseline to 0.581 for the best case, are not impressive. However, one has to keep in mind that differences in such effectiveness measures generally tend to be small in IR experiments as opposed to efficiency differences (e.g. response times) in the DB literature; we emphasise that the gains are statistically significant [109]. Moreover, it is worth pointing out that for some individual users (i.e. micro-averaging over the queries of one user only) or for individual queries the gains are higher. For example, Tables 13 and 14 show the results for $user2$ and $user5$, aggregated over their individual queries issued and evaluated by the users themselves. Here the NDCG value increased from the baseline value of 0.67 to 0.73 for $\alpha = 0.2,\ \beta = 0.8$, or from 0.51 to 0.72 for $\alpha = 0.5,\ \beta = 0.5$, respectively.

As anecdotic evidence, the query "science illusion magic" posed by $user2$ strongly benefited from the user's social relations: with global scoring alone, many good results

α \ β	0.0	0.2	0.5	0.8	1.0
0.0	**0.68**	0.69	0.69	0.68	0.69
0.2	0.68	0.68	0.71	**0.73**	n/a
0.5	0.62	0.62	0.64	n/a	n/a
0.8	0.59	0.60	n/a	n/a	n/a
1.0	0.59	n/a	n/a	n/a	n/a

Table 13: NDCG[10] for user2

α \ β	0.0	0.2	0.5	0.8	1.0
0.0	**0.51**	0.64	0.65	0.66	0.668
0.2	0.67	0.70	0.71	0.70	n/a
0.5	0.67	0.70	**0.72**	n/a	n/a
0.8	0.65	0.71	n/a	n/a	n/a
1.0	0.63	n/a	n/a	n/a	n/a

Table 14: NDCG[10] for user5

were missed; with spiritual scoring alone, the results drifted towards a big "Harry Potter" cluster which was not what the user wanted; only the combination of social and spiritual similarity gave the excellent results that the user appreciated (which included novels such as "Prestige", "Labyrinths", "Invisible Cities").

Table 15 shows the NDCG results with tag expansion enabled, aggregated over all 49 queries of the user study. Across the entire query mix of all users, the tag expansion did not achieve significant improvements over the results without tag expansion but again, for individual users such as $user2$ there were noticeable gains. For example, the query "Yakuza" created the expansion tags "Cosa Nostra", "Triads", and "nightclub" (among the top-5 expansions); the first and second expansion could have been expected (and created also by an ontology-based method), but the third expansion really reflected tag co-occurrences and implicitly the contents of the kinds of novels that the user wished to discover.

From a cognitive viewpoint, it is desirable that query results in social tagging networks exhibit some *diversity*. For interesting and surprising discoveries, you want to benefit from the natural diversity of cultures and tastes in your social network. (Even computer geeks should have some friends who are not in the IT business or in com-

α \ β	0.0	0.2	0.5	0.8	1.0
0.0	0.545	0.565	0.565	0.563	0.565
0.2	0.561	0.573	0.581	0.582	n/a
0.5	0.538	0.550	0.554	n/a	n/a
0.8	0.506	0.540	n/a	n/a	n/a
1.0	0.459	n/a	n/a	n/a	n/a

Table 15: NDCG[10] with up to 5 expansions per tag

puter science.) Using only spiritual similarities among users would bear the risk of not exploring items widely enough. To study this hypothesis, we also measured a notion of diversity among the relevant results of a given query (rating 1 or 2 for precision, rating 2 for strong precision). We computed the pairwise cosine dissimilarity (i.e. $1-$cosine similarity) for the results (inspired by, but not directly related to notions of result incoherence and query ambiguity [41]). To this end, each result document d was viewed as a feature vector or frequency distribution over tags, with the frequency of tag t set to the total number of taggings with t assigned to d aggregated over all users. In most cases, these diversity measures were maximal at α values around 0.5; for non-social recommendations with $\alpha = 0$ the diversity was much lower. This clearly shows the importance of social relations in diversity-aware recommendations.

6.4.3 Lessons Learnt and Open Issues

We have developed a comprehensive framework for socially enhanced search, ranking, and recommendation. Our experimental evaluation exhibits interesting results and indicates the potential of exploiting social-tagging information for scoring and ranking. However, the results reveal mixed insights, and thus also underline the need for further investigating this line of research.

The combination of social and spiritual scoring nicely improved the results of certain queries or users, but also led to result degradation in other cases. On average, there is a significant gain but it is not as impressive as one could have hoped for. It seems that categorising queries and identifying the query types that can benefit from social and spiritual relations is the key to a robust solution that would choose non-zero values for α and β only when benefits can be expected. In our user study, the queries seem to fall into the following four categories:

1. Queries with a purely *global* information need that perform best when $\alpha = \beta = 0$; examples are "Houdini", "search engines", "English grammar" or "Harry Potter band 5", all fairly precisely characterised topics with objectively agreeable high-quality results.

2. Queries with a subjective-taste and thus *social* aspect that perform best when $\alpha \approx 1$; an example is the query "wizard" or "fantasy magic wizard". This query produces a large number of results but the user may like only particular types of novels such as "Lord of the Rings", for which "wizard" is a relatively infrequent tag overall but was frequent among that user's friends.

3. Queries with a spiritual information need that perform best when $\beta \approx 1$; an example is the query "Asia travel guide" or "Computer programming Amiga"where one can harness the aggregated expertise of the entire user community without consideration of social relations.

4. Queries with a mixed information need that perform best when $\alpha, \beta \approx 0.5$; an example is the query "mystery magic".

Our future work aims at developing a principled understanding of query properties and their potential for socially-enhanced recommendation. Other issues that are worthwhile addressing include the *temporal evolution* of tagging and social relations (see, e.g. [18, 52]) and the notion of *diversity* in query results and recommendations (see, e.g. [85]). For interesting and surprising discoveries, you want to benefit from the natural diversity of cultures and tastes in your social network.

7 System Architecture

We implemented our *SENSE* framework with its algorithms SOCIALMERGE and CONTEXTMERGE. The basic architecture is depicted in Figure 8 for the CONTEXTMERGE algorithm, the instantiation of *SENSE* with the SOCIALMERGE algorithm is similar.

In a first step, the data already available in social tagging networks is harvested and imported in *SENSE* by creating for each tag t the global document lists DOCS(t) and for each user U, the user document lists USERDOCS(U, t) needed for our CONTEXTMERGE algorithm (see Section 6.3.1). Next, for all tags t, the tag similarity lists SIMTAGS(t) for tags t' similar to t are precomputed from the data available in a social tagging network, as well as the social and spiritual friendship lists FLIST$_{so}$(U) and FLIST$_{sp}$(U) (see Section 6.3.1) for all users U. In the case of SOCIALMERGE the corresponding inverted lists USERDOCS(U, t), FLIST(U) and SIMTAGS(t) are precomputed (see Section 5.2.1).

The imported and precomputed index lists are finally fed into our social merge algorithm which sequentially scans them for answering queries from a user. The user interface for submitting queries is realised as a Java Servlet hosted by a Tomcat server. Results from our CONTEXTMERGE (or SOCIALMERGE) algorithm are sent back to the Tomcat server and visualised again in a Java Servlet.

We fully implemented the *SENSE* framework with its CONTEXTMERGE algorithm in a prototype application, establishing a hybrid personalised search and exploration system where a user can perform spiritual, social, or global searches in social tagging networks or hybrid combinations of them. We imported three different datasets retrieved from the real world social tagging networks *Flickr.com*, *Delicious.com* and *LibraryThing.com*, and which were basis of our experimental evaluation in Section 6.4. Section 3.2 gives an overview over each of these social tagging networks.

Any user of one of the three social tagging networks who happen to be also found in our imported datasets can immediately log in to the system and submit queries in her own context. However, for the purpose of showcasing *SENSE*, we also implemented a way to enable search in the context of any other user whose data was imported into *SENSE*. For this, our prototype system allows to first choose among the available datasets from *Flickr.com*, *Delicious.com* or *LibraryThing.com*, and then to enter a query with one or multiple tags in order to find an appropriate user for issuing a query containing these tags. The corresponding user interface is shown in Figure 9a. The *SENSE* prototype system generates a list of candidates to be the initiators of the query, based on the overlap of the tags used in the query with the tags used on the documents of users, and in addition, on the number of direct social friends users have got in the social tagging network. Thus, users in the candidates list have got at least one document that is annotated with the query tags and at least one social friend. The number of documents shows how knowledgeable is the candidate about the query subject, and the number of friends shows how well connected is the candidate to the community. These two parameters have an impact on the query results, thus, they are the two main criteria of choosing a candidate as query initiator.

Once a user has been chosen from the provided list of candidates to log in to the system, queries can be submit in the context of that user.

Submitting queries in the context of different users also allows to study the influence of the match of the query and user profile (huge vs. few or even no tag overlap) and the size of the user's friend network on query performance and result quality.

A query is evaluated as spiritual, social or global search (using buttons in the user

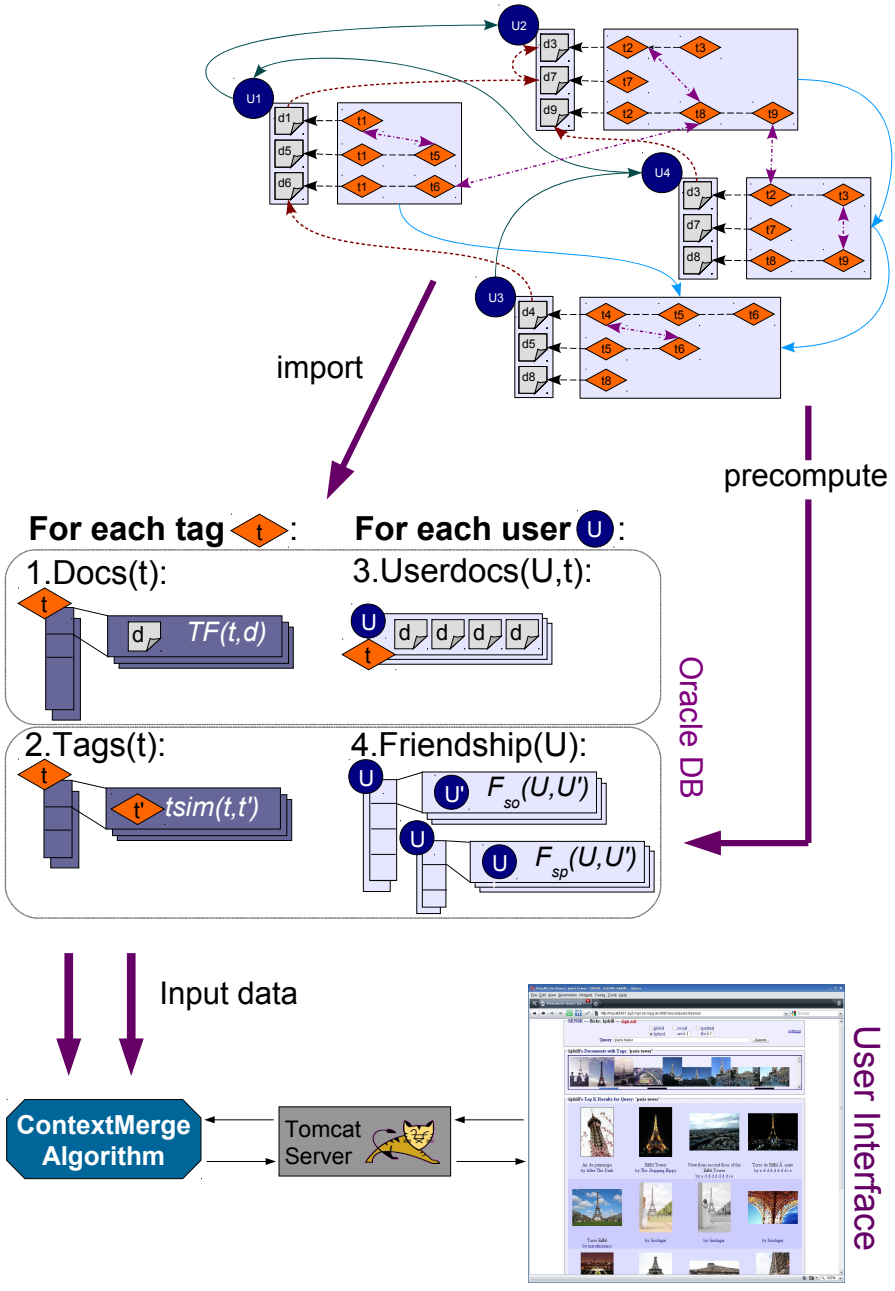
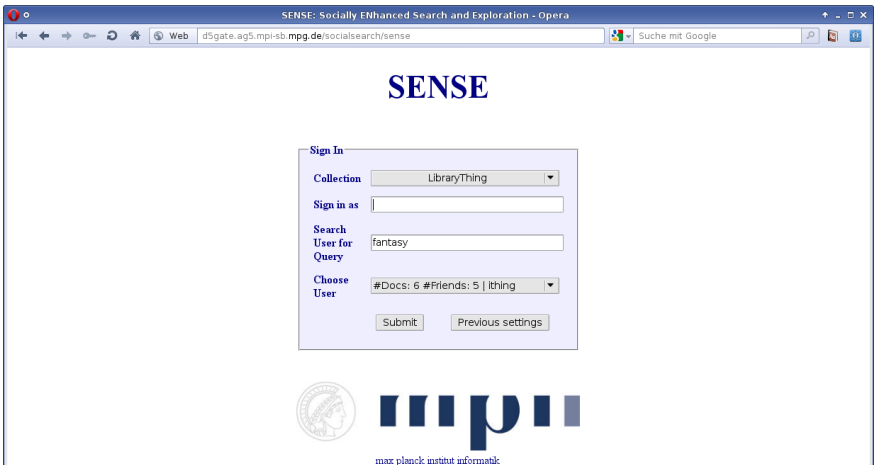


Figure 8: System Architecture of *SENSE* with our CONTEXTMERGE algorithm

interface that set the parameters to predefined values) or as hybrid query by explicitly specifying values for the parameters α and β according to the scoring model as described in Section 6.2. By changing the appropriate settings, our implementation also allows for enabling or disabling tag expansion for individual searches. Figure 9b shows the corresponding user interface.

Finally, the results for a query are computed and listed in a configurable number of columns as shown in Figure 10. In addition, the query initiator's own documents matching the query tags are displayed in a scrollable frame of the size of a single row at the top of the result page. In this way, when executing a query in the context of some unknown user, it is possible to understand the user's interests in regard to the query tags.

When clicking on a result document, a browsable *user cloud* and *tag cloud* is shown, visualising the most influential users and their tags that contributed the highest scores to the result. By clicking on a user, the query initiator can browse all the user's



(a) User Interface in *SENSE* for choosing the dataset, finding a user for a query and log in to the system

(b) User Interface in *SENSE* for entering a serach query and setting the maximal tag expansion

Figure 9: Query Interface of our Prototype System *SENSE*

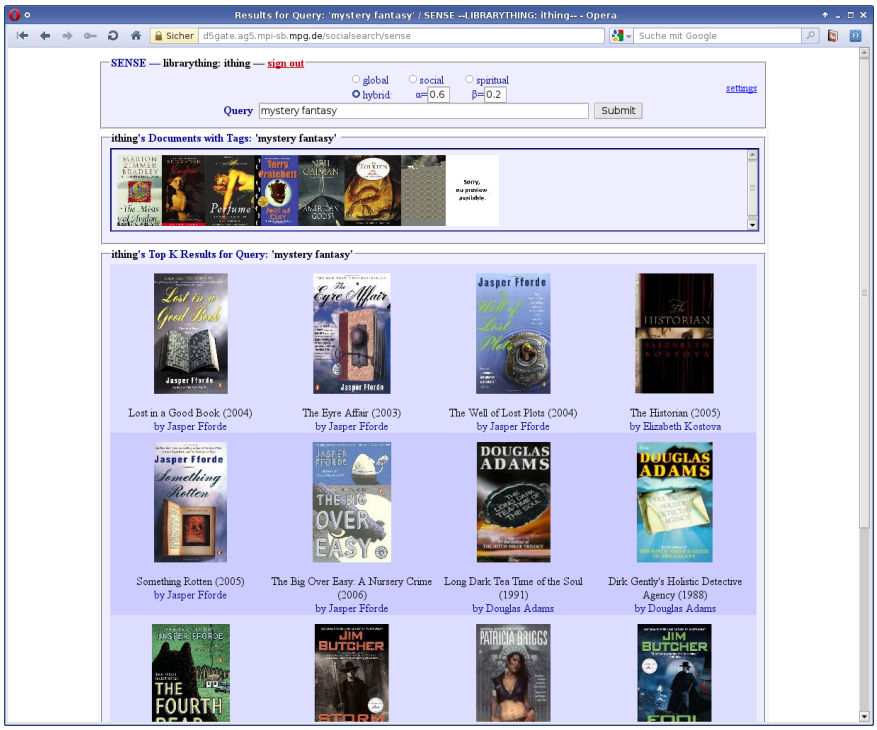


Figure 10: Query User Interface in *SENSE* and Result List for *LibraryThing.com*

documents and tags just like in the social tagging network itself. This allows to get some idea why these results were generated. The query evaluation usually takes less than one second for our imported datasets.

Part B
Dynamic Updates in User Networks

8 Introduction

After having introduced in Part A our *SENSE* framework for socially enhanced search and exploration of contents in social tagging networks, a remaining question is how to dynamically update the social friendship graph of users in social networks when new users enter the system or new friendship connections between users are created.

Since social friendship relations in *SENSE* are based on the shortest path distances of users in the friendship graph of social tagging networks, we are introducing in Part B of this work a novel algorithm for solving the *dynamic All Pairs Shortest Distance* (dynamic APSD) problem in large graphs.

8.1 Motivation

An increasing amount of applications need to manage large graphs with millions of nodes and billions of edges. Examples include user graphs in social networks such as *Facebook.com*, *MySpace.com*, or *Twitter.com*, knowledge graphs in large knowledge bases such as DBPedia [16] or YAGO [121], or keyword search in large databases. A common problem that many applications need to solve is accessing transitive neighbours of a specific node in the order of increasing distance; it is a key building block of algorithms for aggregated search in social networks [33, 111], explorative search of connections in knowledge bases [80], or retrieval in linked documents [57].

Likewise, our *SENSE* framework presented in part A relies on retrieving documents from the users' transitive friends according to their distances in the friendship graph of a social tagging network in which users are represented as nodes and friendship relations between users as weighted edges.

For small graphs that fit into main memory, incrementally retrieving neighbours of a node can be efficiently implemented by running Dijkstra's algorithm [48]. However, this is not a valid option for large, disk-resident graphs as each access to a node corresponds to one disk access. An efficient solution to this problem is to precompute for each node in the graph the distances to all other nodes, and maintain them in a list of nodes ordered by their distances.

As shown for our SOCIALMERGE and CONTEXTMERGE algorithm introduced in Chapter 5 and 6 of Part A, such precomputed lists are crucial for the effectiveness of both algorithms and are employed for storing for each user a list of social friends based on their shortest path distances in the friendship graph of a social tagging network.

While for static graphs that do not change precomputing this information can be expensive if graphs are large, it is an offline, one-time operation. After the precomputation is done, it requires only a few disk accesses to load a list and access all neighbours of a node.

This solution, however, turns prohibitively expensive when the graph changes over time. When a new edge from a node U is inserted or the weight of an existing edge starting at a node V changes such that the distance to the edge's destination node is reduced, the precomputed nearest neighbour list of U or V, respectively, needs to be recomputed: new nodes may have appeared in that list or the distance of already connected nodes may have decreased. Figure 11a depicts an example graph and 11b the corresponding initial neighbour list for node U containing its transitive neighbours and their shortest path distances. In this example, the length of a path is computed by summing up its edge weights. The dashed lines indicate two graph updates: a new edge with weight 1 is inserted at node U and the weight of the edge starting at V is reduced

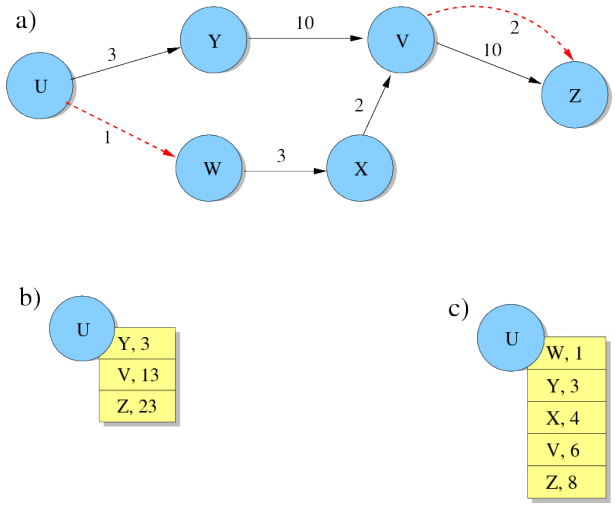


Figure 11: Example graph: A single inserted edge at U changes its entire nearest neighbour list and a decreased edge weight at V affects almost the entire graph because of changes in shortest paths for all predecessor nodes of Z.

from 10 to 2. As a consequence, nodes W and X are now connected with U through the new edge and appear as additional entries in U's neighbour list. Furthermore, the path to V and Z through the new edge is now shorter than the previously existing shortest path (i.e. the distance to V and Z changes from 13 and 23 to 6 and 8, respectively). Hence, the corresponding entries in U's nearest neighbour list change as shown in Figure 11c. More severely, the nearest neighbour lists of *all nodes preceding an updated node* V may have to be recomputed because a new edge or a changed weight may affect the distances from these nodes to any node reachable through that edge. In our given example the shortest path distance to node Z changes indeed for all other nodes in the graph due to the decreased weight of the edge starting at V.

Of course, the problem is evident especially in popular, fast growing social tagging networks, and thus, our SOCIALMERGE and CONTEXTMERGE algorithm suffer from it, too, due to their need of sequentially accessing friendship lists sorted according to the shortest path distances of friends. When new users register with social tagging networks or new friendship connections between users are established, these updates to the friendship graph need to be reflected in the users' lists of nearest friends.

This problem, known as *dynamic All Pairs Shortest Distance* (dynamic APSD), has been intensively studied in the literature for only inserting / decreasing or deleting / increasing of edge weights in graphs or both (see Section 8.2 for an overview). However, this existing work focuses on maintaining in-memory data structures for graphs that fit into main memory, and therefore cannot be applied for large, disk-resident graphs. Additionally, the proposed algorithms attempt to update all affected precomputed information as soon as a new edge is inserted. In most applications, this proactive update is not needed as some lists may not be read for a long time, and additional updates in the future may happen before the list is accessed again due to a certain query to the system.

In this work, we propose an algorithm for incrementally maintaining precomputed

nearest neighbour lists under edge addition and changing edge weights that decrease the distances of paths. Updates to the lists are deferred as long as possible, namely to the point where a query attempts to read an entry from a list which is not up to date. Any query that does not touch updated edges will efficiently read the precomputed list of neighbours.

With regard to the *SENSE* framework presented in Part A of this work, our algorithm for incrementally maintaining neighbour lists, i.e. friendship lists in *SENSE*, has been developed with friendship graphs of social networks in mind where the insertion of friendship connections happen much more frequent than their deletion. Moreover, we are aiming in our work at operations on graphs that introduce "better" paths (to friends in social networks), not on operations that reduce the quality of existing paths.

We provide two alternative versions of the algorithm, where the first incrementally maintains friendship lists by updating complete or fixed lengths prefixes of lists on demand, and the second integrates updates by dynamically determining when the processing of a list can be stopped, i.e. reading a variable number of entries from each list.

8.2 Related Work

The majority of work on indexing graphs has focused on compactly storing reachability information. Here, the proposed approaches include 2-hop covers [37, 113], 3-hop covers [75, 76], and storing tree- and non-tree edges of a graph separately [126]. Only a few proposals exist for efficiently updating such an index [29, 113].

Efficiently representing distances in such a compact index is a difficult problem, and mainly two classes of solutions have been proposed: adding distance information to 2-hop covers [28, 36, 37, 113], and building a shortest-path index by storing multiple BFS trees [130]. Gubichev et al [58] use path sketches to compute approximate distances and shortest paths between two nodes. Only [113] considers the problem of incremental index maintenance.

The problem of maintaining all-pairs shortest path information under certain classes of graph updates (like increasing or decreasing edge weights) is also an active topic in the algorithms community [21, 47, 55, 96, 46, 83, 104]. Here, solutions usually focus on graphs that fit in main memory. King [83] proposes a fully dynamic algorithm for APSP in directed graphs with integer edge weights bounded by b, which requires constant time to determine the distance between two arbitrary nodes. It provides amortized cost of $O(n^2\sqrt{bn})$ for a series of updates in a graph with n nodes [84], and it can be modified to retrieve neighbors of a node in ascending order of distance. Frigoni et al. [55] propose an algorithm for fully dynamic APSP in directed graphs with real weights; here, the amortized complexity of an update is $O(m \cdot \log n)$ in a graph with n nodes and m edges. Demetrescu and Italiano [47] experimentally compare several algorithms for the dynamic all-pair shortest path problem, namely [46, 83, 104] on random and real-world graphs of up to 3,000 nodes. Unlike these proposals, our method assumes that the graph is too big to store in memory and needs to be kept on disk, so traversing the graph as these methods do is not viable here. To the best of our knowledge, Meyer's work [96] is the only that considers graphs stored on external memory; however, it does not precompute any information, but performs a BFS traversal of the graph to determine node distances.

The problem of incremental maintenance of precomputed transitive closures and distances in databases was considered in [50, 100]. They update the complete set of precomputed information after a new edge was inserted or an existing edge was deleted.

It is incremental in the sense that it does not need to recompute from scratch, but can modify the existing information. In contrast, our proposed approach is also incremental, but considers only a subset of the precomputed information affected by an update, and defers maintenance to the point where queries access information that is potentially out-of-date.

9 Maintaining APSP Distances in Large Graphs

This section introduces our algorithm for incrementally maintaining shortest distances of shortest paths in directed, weighted graphs under edge insertion and changing edge weights that decrease the distances of paths.

9.1 Overview

For each node in a graph, we store an inverted index list (see Part A, Section 3.3) that represents all reachable nodes in the graph. A list entry contains a node identifier and the weight associated to the shortest path to that node.

We associate a weight to all paths in order to avoid the confusion about the different notions of a shortest path given in Definition 5.3 and 6.6 in the Chapters 5 and 6 of Part A introducing our SOCIALMERGE and CONTEXTMERGE algorithms, respectively. In both cases, the definition of a shortest path aims at maximising the friendship strength of two users being connected in a friendship graph of a social tagging network. Therefore, we define the weight of a shortest path as follows:

Definition 9.1 (Shortest Path Weight)**.** *Independent of the details of how the length of a path between nodes is defined, we define that the weight associated to a path has to be higher the shorter a path is. Hence, a shortest path between two nodes is a path with maximal weight.*

In the case a path actually becomes shorter by decreasing its edge weights (e.g. by summing over edge weights as in the example given in Section 8.1 or in Definition 5.3 which implicitly assumes edge weights of 1 for all edges) then, to match the definition 9.1, the weight associated to a path can be defined by the inverse of the path length. In the contrary case, that paths become longer by decreasing edge weights (as in Definition 6.6), the weight of a path can be defined to be equal to the path's length.

Definition 9.1 allows us to talk synonymously about shortest paths and paths with maximal weights. Hence, the presented algorithm applies to graphs with updates that increase or decrease edge weights as long as they are always reducing path distances.

Inverted list of nodes are accessed only sequentially by our algorithm for maintaining all pair shortest distances. The rationale to restrict ourselves to only sequential access index lists is two-fold:

1. *Efficiency:* When fetching data from a comparably slow storage backend, a sequential access is much faster than a random access which first has to search for the location at the storage backend in order to eventually access the data. When doing only sequential accesses, elements at the storage backend are fetched one by one in the order they have been stored. Hence, there is no need to search for a data item because after retrieving the first one, the position of the next item is already known.

In our algorithm, we exploit the efficiency of sequential accesses to maintain shortest distances in directed graphs.

2. *Applicability of TA:* The popular family of so-called threshold algorithms (TA) operate on score-sorted index lists and assume that the score aggregation function is monotonic (e.g. a weighted summation). We employ variants of Fagin's Threshold Algorithm (TA) [53] with flexible scheduling of list scans for our social search engine *SENSE* which operates on social tagging networks and utilises shortest path distances between users of the corresponding friendship graph to search for and explore available contents (see Part A of this work).

   By using only sequential accesses on inverted lists for maintaining weighted, shortest distances, we can keep the data structures needed by TA up-to-date while processing these lists. Consequently, there is no need to recompute inverted lists from scratch.

Dynamic Updates

Typically, a social network grows dynamically over time. Users in such a network can easily choose other users as new friends which corresponds to an insertion of a new and possibly weighted edge into the social friendship graph of the network. Additionally, an edge insertion can also cause an increase in the weight of the shortest paths between a large number of users because of a closer connection over the newly inserted edge.

We call such changes to the social network *dynamic updates* to contrast the fixed weights and edges in a static graph where all connections are precomputed and do not change until the entire graph is reconstructed again.

In the following sections, we introduce our algorithm for incrementally maintaining weighted, shortest path distances in directed graphs under edge insertion and increasing path weights. For each node in the graph, we store a corresponding inverted list that represents all its reachable nodes. A list entry contains a node identifier and the associated weight of the shortest path to that node. The list is sorted in descending order of all weights such that the nearest node is the top entry in the list.

Since our algorithm has been designed with data from social networks in mind, we will use the notation and terminology given in Section 9.2 in the remaining sections and chapters of Part B.

9.2 Social Network Setup

In our social network setup, a node in the associated friendship graph corresponds to a user of the network and the users' social friendship connections are represented by edges in the graph. Based on the shortest path distances between users, and in presence of dynamic (friendship) updates (see Section 9.1), we formally define in the following the notion of dynamic friendship graphs in social networks.

Dynamic Friendship Graph

We introduce the basic data structure used in this social network setup, representing the inverted lists required by our incremental APSD algorithm, and the terminology

and notation used in the following discussion of the algorithm for dynamically updating users' lists of transitive friends in social networks. The terminology and definitions are based on the data model introduced in Part A, Section 3.1. Hence, the friendship graph of social tagging networks is defined by the social friendship relations of users (see Part A, Definition 3.1) inherent to the network.

Definition 9.2 (User U)**.** *Each user U of a social network is represented by a node in the corresponding friendship graph.* ∎

The friendship relations between users define the friendship edges in the graph:

Definition 9.3 (Friendship Edge (U, U_f) & Friend U_f)**.** *A direct edge $e = (U, U_f)$ in the friendship graph represents the friendship relation between two users U, U_f and is called a friendship edge. U_f is called a friend of U.* ∎

Definition 9.4 (Dynamic Friendship Graph G)**.** *We consider $G = (G_0, \ldots, G_t, \ldots)$ a sequence of directed, weighted friendship graphs G_t for $t > 0$ with a constant node set $V = \{U_0, \ldots, U_{n-1}\}$, and a varying set E_t of edges and weights w_t. The nodes correspond to the users in a social network and the weighted edges to the friendship edges between pairs of users. The index t in the sequence of friendship graphs correspond to the version of the graph at time t.*

A friendship graph $G_t = (V, E_t)$ with friendship edges E_t at time t does not contain any self edges. For each edge $e = (U_i \rightarrow U_j) \in E_t$, there is an edge weight $0 < w_t(e) < 1$. Two consecutive friendship graphs G_{t-1} and G_t in the sequence differ either by

- *the weight of a friendship edge that was increased in G_t, i.e. $w_{t-1}(e) < w_t(e)$ for exactly one edge e and $w_t(e') = w_{t-1}(e')$ for all other edges $e' \neq e$, or*
- *a single friendship edge $e = (U \rightarrow U_f)$ was added to G_{t-1}.* ∎

Note: The restriction to a constant set of users has been made only to simplify the discussion. The retrieval of the nearest friends for newly added users without friendship edges is trivial as there are no connections to friends at all. Hence, those users could be ignored for the definition of G_t and G_{t+1}. However, as soon as there's a new edge for or to such a node, the problem description complies to the one mentioned above.

We call the new or increased friendship edge in G_t, for $t > 0$, a friendship update on U at time t. It can be thought of as a user U_f in G_{t-1} is becoming a friend or a better friend of U at time t.

Definition 9.5 (Friendship Update $(U \rightarrow U_f)$)**.** *We consider the insertion of a new friendship edge $(U \rightarrow U_f)$ or the change of a weight of an existing friendship edge $(U \rightarrow U_f)$, representing the difference in the sequence of friendship graphs from G_{t-1} to G_t, a* friendship update *on U with respect to U_f at time t.*

Each friendship update is defined by a unique timestamp $TS_e(U \rightarrow U_f)$ corresponding to time t of the occurrence of the update. ∎

With this construction of a friendship graph, we now can define the friendship strengths, the shortest paths and the distances between users. The definitions follow the ones given in Chapter 6 of Part A for our CONTEXTMERGE algorithm.

We define the friendship strength between two direct friends to be a value between 0 and 1 and being equal to the weight of the corresponding friendship edge.

Definition 9.6 ((Direct) Friendship Strength $s_{f,t}(U \to U_f)$)**.** *We define the* direct friendship strength $s_{f,t}(U \to U_f)$ *of a direct friend* U_f *of a user* U *in* G_t *as*

$$0 < s_{f,t}(e) = w_t(e) < 1$$

where $e = (U \to U_f) \in E_t$*, i.e.* $s_{f,t}(e)$ *is equal to the weight* $w_t(e)$ *of the edge* $e = (U \to U_f)$ *in* G_t*.* ∎

The friendship strength along a certain path in the friendship graph is computed as the product over all the path's edge weights:

Definition 9.7 ((Indirect) Friendship Strength $s_{f,t}(U_1 \to \cdots \to U_n)$)**.** *We define the* indirect friendship strength $s_{f,t}(U_1 \to \cdots \to U_n)$ *of a user* U_n *wrt. user* U_1 *for a path* $(U_1 \to \cdots \to U_n)$ *as*

$$s_{f,t}(U_1 \to \cdots \to U_n) = \prod_{1 \leq i < n} s_{f,t}(U_i \to U_{i+1})$$

∎

With reference to Definition 9.1, we define the weight associated to each path as follows:

Definition 9.8 (Weight of Path π)**.** *The weight of a path* $\pi = (U_1 \to \cdots \to U_n)$ *leading from* U_1 *to* U_n *is equal to the indirect friendship strength* $s_{f,t}(\pi)$ *computed for* π*.*

Finally, we define the friendship strength for arbitrary users in the graph as follows:

Definition 9.9 ((Maximal) Friendship Strength $s_{f,t}(U, U_f)$)**.** *Let* $\Pi_t(U, U_f)$ *be the set of paths* $\pi = (U \to \cdots \to U_f)$ *in* G_t*. We define the* friendship strength *or* maximal friendship strength $s_{f,t}(U, U_f)$ *of an arbitrary user* $U_f \neq U$ *wrt. a user* U *to be equal to the maximal weight over all paths from* U *to* U_f *if there is such path,* 0 *otherwise and* 1 *if* $U_f = U$*:*

$$s_{f,t}(U, U_f) = \begin{cases} 1 & \text{if } U_f = U \\ 0 & \text{if } \Pi_t(U, U_f) = \emptyset \\ \max\{s_{f,t}(\pi) \mid \pi \in \Pi_t(U, U_f)\} & \text{otherwise.} \end{cases}$$

∎

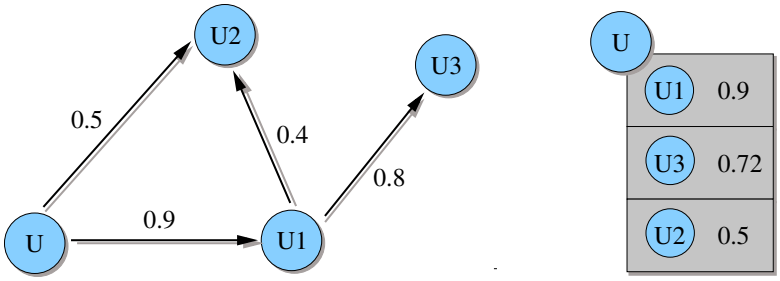
Based on the friendship definitions between users, we define the shortest path in the friendship graph of a social network as follows:

Definition 9.10 (Shortest Path)**.** *We define the length of a path* π *from* U *to* U_f *in* G_t *to be the inverse of the indirect friendship strength* $s_{f,t}(\pi)$ *computed for path* π *or* 0 *if* $U_f = U$*.*

The shortest path *from a user* U *to a user* U_f *in* G_t *is the path with maximal weight from* U *to* U_f *in* G_t*.* ∎

Hence, the shorter a path starting at U and ending at U_f, the greater its weight and the stronger is the friendship strength of U_f wrt. U.

In compliance with the notion of a dynamic friendship graph given in Definition 9.4, we define for each user U a *dynamic* friendship list of users being friends of U with respect to a given time t.



(a) Example friendship graph G_0 (b) Friendship list of U wrt. G_0

Figure 12: Example of a friendship graph G_0 with users U, U_1, U_2 and U_3 in (a) and the corresponding friendship list $Flist_{U,0}$ for U in (b).

Definition 9.11 (Dynamic Friendship List $Flist_{U,t}$)**.** *We define the* dynamic friendship list $Flist_{U,t}$ *of a user U as the list of pairs (U_f, s) for all transitive friends U_f of U in G_t in descending order of s with $s = s_t(U, U_f)$ is equal to the (maximal) friendship strength of U_f with respect to U.*

Figure 12a depicts an initial friendship graph G_0 consisting of four users U, U_1, U_2 and U_3 with no friendship updates. The weight of the shortest path is determined by multiplying the edge weights. In Figure 12b the corresponding dynamic friendship list $Flist_{U,t}$ at time $t = 0$ of user U is sketched.

9.3 Problem Statement

In this section we introduce the notion of a query and the APSD problem that has to be solved in the context of a user U in a social network.

Definition 9.12 (Query $\mathcal{Q}_U$)**.** *A query $\mathcal{Q}_U$ issued in the context of a user U of a social network aims at retrieving U's best friends in descending order of their friendship strengths until all or an arbitrary number of* top-k *friends has been retrieved. The value of* top-k *can be determined during a query or specified beforehand.* ∎

Remember: The length of the shortest path (see Definition 9.10), hence the APSD problem, is based on the friendship strength of users.

To define the result of a query in the presence of friendship updates (see Definition 9.5), we first have to define the point in time when a query is issued in the network.

Definition 9.13 (Query Time $t(\mathcal{Q}_U)$)**.** *When a query $\mathcal{Q}_U$ is issued in the context of a user U, the time $t(\mathcal{Q}_U)$ of the query is equal to the maximal timestamp $TS_e = t$ of any friendship update (see Definition 9.5) occurred so far for any user in the dynamic friendship graph $G = (G_0, \ldots, G_t)$ of a social network. Friendship updates at a later time t', such that G_t is transformed into $G_{t'}$, are not considered for a query at query time $t(\mathcal{Q}_U) < t'$.* ∎

Finally, the result of a query is equal to the dynamic friendship list of best friends (or a certain prefix of it) when for computing the friendship strengths of users only the friendship updates up to the query time are taken into account.

Definition 9.14 (Query Result $R(\mathcal{Q}_U)$)**.** *For a query $\mathcal{Q}_U$ at query time $t(\mathcal{Q}_U)$, the query result $R(\mathcal{Q}_U)$ is equal to U's dynamic friendship list of best friends at time $t(\mathcal{Q}_U)$ sorted in descending order of their friendship strengths, or to a prefix of it of size* top-k *as given with the query $\mathcal{Q}_U$, i.e.*

$$R(\mathcal{Q}_U) = Flist_{U,t} \text{ with } t = t(\mathcal{Q}_U)$$

or

$$R(\mathcal{Q}_U) = Flist_{U,t}[0:i] \text{ with } t = t(\mathcal{Q}_U) \text{ and } i = \text{top-k}$$

■

9.4 The Basic Algorithm

Our algorithm handles friendship updates in social friendship graphs that either insert new friendship edges or increase the friendship strengths of users by incrementally solving the *All Pairs Shortest Distance* (APSD) problem. For this, a user's friendship list is sequentially processed and any changes to the friendship graph are incorporated on-the-fly when certain conditions indicate a requirement for an appropriate action to correct the subsequent friendship list entry.

The algorithm is composed only of two basic operations, i.e. an *update operation* and a *merge operation*, to achieve the correct maintenance of friendship updates in social networks:

1. $U.update()$: An update operation on a user U computes for all new friendship updates $(U \rightarrow U_f)$ the friendship strength s_f of U_f with respect to U and incorporates the pair (U_f, s_f) in U's friendship list. The friendship strength s_f corresponds to the weight of the new or updated direct edge $(U \rightarrow U_f)$.

2. $U.merge(U_f)$: The merge operation effectively propagates the information about friendship updates from friends U_f to U in order to adjust the friendship strengths of already known or completely new friends.

When a friendship update $(U \rightarrow U_f)$ occurs in a social network, the update operation on user U is postponed until there is the need to find her best friend. That is the case when U submits a query to the social network or a user U_q submits a query and U is a friend of U_q and U_q's next best friend after U has to be identified. In the same way, merge operations are deferred until it is necessary to merge in new information found in a friend's friendship list.

Our algorithm can be characterised in the following way: Friendship updates in a social network are handled on-the-fly and processed on demand and only when needed such that the actual maintenance is deferred to the latest point in time.

The pseudocode of our basic algorithm is given in Listing 5. The algorithm is called in the context of a querying user U and it successively determines the *top-k* friends by processing U's friendship list, accomplishing checks for update and merge operations for each previously processed user in the list and doing the appropriate operation if necessary in order to guarantee that indeed the next best friend can be found next in the list.

A call to $U_f.update()$ and $U.merge(U_f)$ for an arbitrary user U_f can only happen when U_f is part of a shortest path from U to her *top-k* friends and hence, U_f is one of the *top-k* friends.

```
1:  U.get_Friends(top-k) {
2:    R_topk=new List<User,s_f>()
3:    IF (U.needs_update()) U.update()
4:    i=0 //find U's next (i.e. i-th) best friend
5:    DO {
        // remember U's next best friend and friendship strength in R_topk
6:      R_topk.add((U_f, s_f)=U.getFriend(i))
7:      IF(++i==topk || U_f==NULL) BREAK
8:      IF(U_f.needs_update()) U_f.update()
9:      IF(U.needs_merge(U_f)) U.merge(U_f)
10:     IF(i > tp_U) tp_U = i //tp_U is a bookkeeping variable
11:   }WHILE(true)
12:   RETURN(R_topk) //return U's top-k friends
13: }
```

Listing 5: $U.get\_Friends$(*top-k*)

The method call $U.getFriend(i)$ determines the *i-th* friend in U's friendship list and returns it. Basically, it is used to retrieve the *next best friend* of U as i is always incremented by one. Moreover, we can easily replace that method by an iterator $U.getNext()$ on U's friendship list which does not need an argument i but, instead, internally keeps track on the current positions in $Flist_U$ for each querying user who tries to process U's friendship list. The real implementation of our algorithm is indeed done in this way.

However, when later more details about our algorithm are given, it is handy to keep in mind which position of a user's friendship list is currently processed or that a step from position i to $i+1$ has to be done next. In addition, it simplifies the description when we have to clarify that our algorithm proceeds from a certain entry with a certain friend U_f to the next position in U's friendship list, i.e. $pos_U(U_f)$ to $pos_U(U_f)+1$. Finally, we also need sometimes to access the *very best friend* of a user U and then, we easily can use the same method $U.getFriend(0)$ to retrieve the top element of U's friendship list.

Before going into the details of how we maintain the all pair shortest path distances by just applying update and merge operations on friendship lists, we introduce the basic operation mode of our algorithm.

9.4.1 Basic Mode of Operation

The query processing in the context of a user U is based on her initial friendship list that is precomputed over the friendship relations defined in G_0 (see Definition 9.4) and is stored at a disk-resident storage backend, e.g. a database. The query result for U is then dynamically built in main memory by starting with an empty friendship list which is filled by entries sequentially retrieved from the version located at the storage backend.

Is a query issued at time $t(\mathcal{Q}_\mathcal{U}) = 0$, that is before any update to G_0 is applied, the in-memory list can be filled one by one from the storage backend and returned as result. Nothing else has to be done; the in-memory list can be finally discarded. The

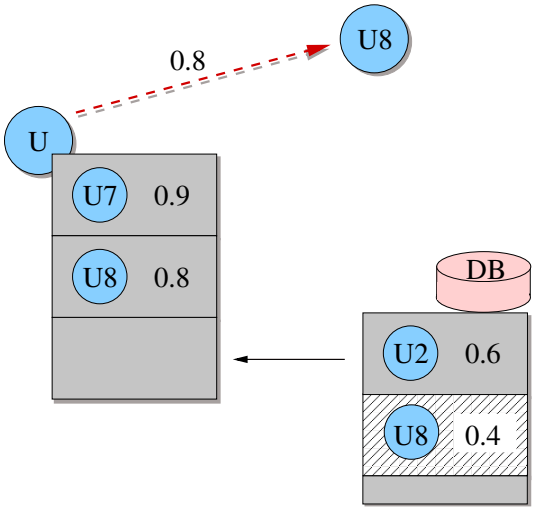


Figure 13: Example for an in-memory list being built by reading from a storage backend and by a new friendship update.

same is true for a query at a time $t(\mathcal{Q}_\mathcal{U}) > 0$ that does not need to modify anything in U's friendship list.

In any other case, only the in-memory part of a friendship list is modified during a query, while the storage backend is continued to be read by sequential accesses only. As a consequence, users that become new or better friends to U and, thus, are inserted in the in-memory friendship list, e.g. due to a new friendship edge, might be retrieved at a later iteration step of our algorithm from the storage backend for a second time. In this case, the entry with the weaker friendship strength can be simply discarded since we are only considering improvements of shortest paths.

In Figure 13 an example is given to illustrate this fact. A query on user U builds up a new list $Flist_U$ in main memory by reading from a previous version stored at a database DB. Assuming there is no friendship update for U, the first entry $(U_7, 0.9)$ is just read from the database and put into the list in main memory. However, as in the case given in this example, if there is a friendship update $(U \rightarrow U_8)$ with friendship strength 0.8, an entry $(U_8, 0.8)$ is created first in $Flist_U$ and then the entry $(U_7, 0.9)$ is read from DB. Since U_7 is a better friend than U_8, the entry is inserted in front of $(U_8, 0.8)$ which *moves downwards* in $Flist_U$ by one position. Afterwards, the DB is continued to be sequentially read and the next entry can be simply appended to $Flist_U$. At this stage, we can observe that there is already a previous entry for U_8 in DB_U with a smaller friendship strength 0.4. Thus, there are two entries for U_8 discovered during the query processing. Though, for computing the result of a query, we simply can discard the entry with the smaller friendship strength and go on by sequentially reading from DB.

Once a query has been finished, the modified list in main memory needs to be written back to the storage backend (when ignoring caching techniques at this point). The in-memory friendship list contains entries for all friends retrieved during the query from its disk-resident counterpart, plus potentially additional entries with new friends due to update and merge operations. All new entries and all entries that have changed during the query processing need to be written back to the storage backend.

Writing back entries that were read from disk and modified during a query is cheap since they can be quickly found and replaced as their position at the storage backend is already known. All these entries are part of the already read prefix at the storage backend of size at most of the query result (that is, when all results correspond to already known friends) and there can only be the need to change their position within this prefix. Remember: a path can only become shorter by friendship updates. Hence, users can only move up to the top of the friendship list. If no new user was added in the in-memory list in addition to the ones read from the storage backend or the complete list was retrieved, the in-memory list can even simply replace in its completeness the part that has been read from the database.

We have to take more care when there are new users in the in-memory friendship list, too. In this case, these users have to be correctly sorted into the disk-resident friendship list. However, there is no need to look for already existing entries at the storage backend to adjust location and friendship strengths. The reason is that even if those apparently new users are located at some later position at the storage backend, too, and therefore potentially users are stored multiple times on disk (with different friendship strengths), the additional entries with weaker friendship strengths can be discarded from the disk-resident list in the same way as previously described by subsequent queries that happen to read the same friend for a second time.

In this way, a friend that becomes a better friend over time automatically *"moves up"* in the friendship list of a querying user by just sequentially accessing the storage backend.

9.4.2 Explanatory Note

As explained in Section 9.4.1, technically, it is not difficult to built friendship lists in main-memory from a previous disk-resident version which is only sequentially accessed.

However, for a better understanding of the description of our algorithm, we assume in the following, that the friendship list of a *querying* user U is always completely available in main memory. In this way, we can easily describe tasks like: *"U_f becomes a better friend and moves up to some position i"* without repeating again and again the trivial but technical details mentioned above of how to identify and discard a user U_f that has been inserted due to a friendship update in the in-memory part of the friendship list and later is discovered for a second time.

Thus, assuming the friendship list is completely in main memory nicely eases the description of how our algorithm works for maintaining shortest path distances in friendship graphs of social networks, and therefore helps for a better understanding. If in the following certain aspects should differ for data being located in main memory or partly at a storage backend, then of course, we direct the attention back to the technical details.

In general, with our algorithm, we always only sequentially process lists at a storage backend without any random access to entries not being already located in main memory. As just mentioned, only for the purpose of an easier understanding, an exception is made for this description only when trying to clarify the point that a user in a friendship list becomes a better friend due to a friendship update, and hence, moves up to an earlier list position. In our actual implementation, no random but only sequential accesses are applied in the way as described in Section 9.4.1.

Furthermore, when talking in subsequent sections about a known or already processed part of a friendship list, we always refer to the prefix of a friendship list which

is already located in main memory. In contrast, the not yet known or processed part of a friendship list can be located both in memory and at a storage backend, depending on the current state of our algorithm.

Before going deeper into the details, we first introduce some required data structures for the bookkeeping of applied and outstanding friendship updates and define the notations used in later sections.

9.5 Data Structures, Definitions and Notation

Next, we introduce the data structures, definitions and notations used in the description of our algorithm.

Basic Data Structure

The basic data structure used by our algorithm in order to maintain all pair shortest distances in friendship graphs is an inverted list. We refer to this data structure with regard to a user U by using the following notation:

Definition 9.15 (Friendship List $Flist_U$)**.** *For each user U, $Flist_U$ denotes the inverted list of transitive friends of U incrementally built by our algorithm. The list is sorted in descending order of the users friendship strength. An entry in $Flist_U$ is a pair $fl_e = (U_f, s_f)$ with s_f is the* friendship strength *of U's friend U_f.* ∎

Accordingly, U's best friend is listed at the first position of her friendship list, i.e. $Flist_U[0]$, the second best friend at the second position, i.e. $Flist_U[1]$, and so on.

In Chapter 10, we will prove the correctness of our algorithm by showing that for each query at any time t, $Flist_U[0:i] = Flist_{U,t}[0:i]$ (see Definition 9.11) in each iteration step i of our algorithm.

For the decision when to do an update and when to do a merge operation, we need some bookkeeping data structures which we define in the following. For this, we first introduce the notion of a pending friendship update.

Definition 9.16 (Pending Friendship Update)**.** *We call a friendship update $(U \to U_f)$* pending *if a friendship update on U with respect to U_f occurred but has not yet been considered for recomputing U_f's friendship strength and position in U's friendship list $Flist_U$.* ∎

Global Bookkeeping of Friendship Updates

For each new or updated edge in the friendship graph of a social network, we need to remember when that friendship update (see Definition 9.5) occurred.

Definition 9.17 (Timestamp TS_e and TS_{max} of Friendship Updates)**.** *Each friendship update $(U \to U_f)$ (see Definition 9.5) in G, transforming G_{t-1} into G_t is remembered via its timestamp $TS_e = t$ and stored in a global data structure OP called* operation map*. The timestamp of the latest friendship update occurred in G is remembered in TS_{max}.* ∎

This maintenance of TS_e is easily achieved by uniquely numbering new friendship updates in increasing order of appearance.

The set of pending friendship updates $(U \rightarrow U_f)$ is stored together with their unique timestamps TS_e for each user U in a global operation map OP and removed again as soon as the update is accomplished.

Definition 9.18 (Operation Map OP)**.** *For each user U, the set of friendship updates $(U \rightarrow U_f)$ which have not yet been processed for U are stored in a* operation map *OP, such that $OP[U] = \{(TS_e, U_f), ...\}$ with $(U \rightarrow U_f)$ is a pending friendship update on U which occurred at time TS_e, i.e.*

$$OP[U] = \begin{cases} \{op_e = (TS_e, U_f) \mid & \exists (U \rightarrow U_f) \in G_t \text{ with } t = TS_e \text{ and} \\ & s_{f,t}(U \rightarrow U_f) > s_{f,t-1}(U \rightarrow U_f)\} \\ or\ NULL \end{cases}$$

■

Local Bookkeeping on each user's Friendship List:

Furthermore, we need to remember the time when a user's friendship list has been updated for the last time due to an update or merge operation.

Definition 9.19 (Timestamp TS_U of U's Friendship List)**.** *For each user U, we assign a timestamp TS_U to the user's friendship list. TS_U is equal to the timestamp TS_e of the last processed friendship update for U in OP after an update operation on U is performed or to the timestamp of the last friendship update known by U's friend U_f which is propagated to U when performing a merge operation on U with U_f. Prior to any update or merge operation, TS_U is equal to* 0*.* ■

Hence, the value of TS_U corresponds to the time for which all friendship updates with respect to U have been seen.

Finally, we need to remember for a given timestamp the last entry known to be correct in a friendship list.

Definition 9.20 (Timestamp Validity Pointer tp_U)**.** *The* timestamp validity pointer *tp_U is an offset into U's friendship list and points to the position up to which all friends are correctly identified with respect to timestamp TS_U.* ■

Auxiliary Functions

Moreover, for the readability of the pseudocode of our algorithm, we introduce some auxiliary functions and methods on friendship lists. However, our algorithm ensures that they are only applied on the in-memory prefix of a friendship list which has been already sequentially retrieved from the storage backend. Hence, these functions are executed quickly and do not effect the need of sequentially reading from friendship lists at the storage backend.

$pos_U(U_f)$: For the sake of a simpler notation, let $pos_U(U_f)$ denote an auxiliary function that returns the index of U_f's position in U's friendship list and -1 if U_f has not yet been seen in (the main memory based prefix of) $Flist_U$.

$findpos_U(s)$: For the same reason, let $findpos_U(s)$ denote an auxiliary function that returns an index i in U's friendship list where an entry with friendship strength s could be inserted to preserve the correct ordering in (the main memory based prefix of) $Flist_U$.

$Flist_U.set((U_f, s), i)$: Let denote $Flist_U.set((U_f, s), i)$ an auxiliary method on U's friendship list that inserts a user U_f with friendship strength s before the current *i-th* entry in (the main memory based prefix of) $Flist_U$.

$Flist_U.remove(i)$ Let denote $Flist_U.remove(i)$ an auxiliary method on U's friendship list that removes the *i-th* entry in (the main memory based prefix of) $Flist_U$.

Next, we define the way how friendship updates and queries in dynamic friendship graphs occur.

9.6 Queries & Friendship Updates

Note: From Definition 9.17 follows that the query time $t(\mathcal{Q}_U)$ is set to the current value of TS_{max}, i.e. $t(\mathcal{Q}_U) = TS_{max}$, when a query $\mathcal{Q}_U$ is issued in the context of a user U.

Hence, a query $\mathcal{Q}_U$ submitted at query time $t(\mathcal{Q}_U)$ (see Definition 9.13) in the context of a user U is supposed to retrieve the *top-k* best friends of U with respect to time TS_{max} when the query started (see Definition 9.14).

If friendship updates are permitted while a query is not yet completed, the *top-k* friends of a user U can change while U's friendship list is traversed. Hence, a query started at time TS_{max} could identify friends at later positions in $Flist_U$ that actually were not yet known at time TS_{max} but emerged because of friendship updates at time $TS'_{max} > TS_{max}$. Moreover, the first best friends already identified for a query at time TS_{max} could also change while a query is still proceeding due to new friendship updates not yet existent at time TS_{max} but at time $TS'_{max} > TS_{max}$. Hence, the retrieved query results would not be well-defined because neither they correspond to the *top-k* friends at time TS_{max} nor at time TS'_{max}.

To circumvent this problem, we can keep the original, well-defined semantic (see Definition 9.14) of retrieving the *top-k* friends for a query with respect to a certain time TS_{max} by keeping copies of friendship lists for the timestamp TS_{max} while any query for this time is still active, i.e. there is still a query with a matching query time which is not yet completed. Hence, friendship updates during a query are applied only to the latest version of friendship lists (not to their copies) and new queries only access the most up-to-date version of friendship lists available at query time, too. A copy of a friendship list can be removed again as soon as no query is active that started at a query time equal to the TS_{max}-timestamp for the friendship list copy.

A simple example is given in Figure 14. Assume a query started on U at query time $t(\mathcal{Q}_U) = TS_{max} = 3$. While the query is processed a friendship update on U at time $t = 4$ occurs. Moreover, a second query on U is issued while the first one still is processed. In this case, the second query needs to work on a copy of the friendship list, corresponding to the version at time $ts_{max} = 4$ in the figure. The friendship update is applied to the new list only, while the previous list version still exists as it is to serve the former query. Once, the former query ends and no other query related to the same timestamp $ts_{max} = 3$ exists, the older friendship list version can be discarded and the newer remains the only existing one. Although copies of friendship lists are only needed for a very short time, since queries are supposed to be answered quickly, maintaining copies of lists can be much more complex as in this simple example. If

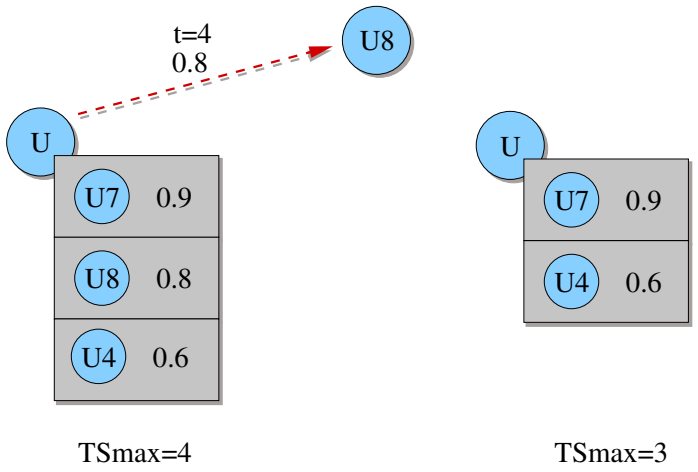


Figure 14: Example for using copies of friendship list. A query processes $Flist_U$ at time $ts_{max} = 3$. While the query is active a new update at time $ts_{max} = t = 4$ arrives and is applied to a copy of U's friendship list. Once, the query on the list at time $TS_{max} = 3$ is finished, and no other query with such a timestamp is active, that list can be discarded while the newer list remains.

during the processing of a friendship list a merge operation with some other user's friendship list is needed, then, this list must have the same timestamp ts_{max}. Hence, multiple copies of friendship lists per query might be needed for several concurrent queries and friendship updates. However, the details are only complex from a technical point of view. Eventually, by working with copies of friendship lists, we ignore updates for a query until the end of the query.

Therefore, in favour of a better understanding and since it does not change the intrinsic functionality of our algorithm, we neglect in the following the details of keeping and removing copies of friendship lists, but instead, limit friendship updates to appear only when no query is active. As a consequence, the timestamp TS_{max} can be assumed to be fixed during each query.

Next, we give detailed information about our basic algorithm and describe when there is a need for an update or merge operation.

9.7 Check for Friendship Updates

When there is at least one pending friendship update for a user U and our algorithm needs to access U's friendship list, an update operation on U has to be applied first. Therefore,

Definition 9.21 ($U.needs\_update()$)**.**

$$
\begin{aligned}
U.needs\_update() \;&==\; True \\
&\iff \exists\, t' \wedge e = (U \to U_f) : TS_U < t' \\
&\qquad \wedge\, e \in G_{t'} \wedge (e \notin G_{t'-1} \vee w_{t'-1}(e) < w_{t'}(e)) \\
&\iff \exists\, e = (U \to U_f) : TS_U < TS_e(U \to U_f) \\
&\iff \exists\, (U_f, TS_e) \in OP[U] : TS_U < TS_e
\end{aligned}
$$

■

Hence, the main loop of our algorithm only initiates an update operation when there is at least one friendship update in the operation map $OP[U]$ for U or in $OP[U_f]$ for the currently processed friend U_f in U's friendship list.

The pseudocode of $U.needs\_update()$ is shown in Listing 6.

```
1:  U.needs_update() {
2:    IF ( ∃ e = (U, U_f) with (U_f, TS_e) ∈ OP[U] )
3:      RETURN(true)
4:    ELSE
5:      RETURN(false)
6:  }
```

Listing 6: $U.needs\_update()$

9.8 Check for Update Propagation

While processing a friendship list of a user U, for each friend U_f in $Flist_U$, the friend's friendship list has to be checked for potential updates since she was discovered for the last time in U's friendship list. If that is the case, a merge operation propagates the friend's friendship updates to U. Therefore,

Definition 9.22 ($U.needs\_merge(U_f)$)**.**

$$U.needs\_merge(U_f) == True \iff \underbrace{TS_U < TS_{U_f}}_{(1)} \text{ or } \underbrace{(pos_U(U_f) \geq tp_U \wedge TS_{U_f} > 0)}_{(2)}.$$

■

The decision for a merge operation is based on the timestamp TS_U of U's friendship list, the timestamp TS_{U_f} of the friendship list of U's currently processed friend U_f and U's timestamp validity pointer tp_U. The two conditions for a merge operation are based on the following intuition:

(1) Whenever a friend U_f in U's friendship list is found and the timestamp of her friendship is greater than the one of U's friendship list, then the currently processed friend U_f must have seen some more recent updates than U herself. Hence, we have to merge these updates from U_f's friendship list into U's friendship list to be able to find the correct next best friend.

(2) Let U_f be the friend located at the position pointed to by U's timestamp validity pointer tp_U. Then, U_f is the last friend in U's friendship list who is correctly known to be among the *top-k* best friends.

When subsequently the next best friend is identified and the timestamp TS_{U_f} of U_f's friendship list is greater than 0, there might be a friendship update merged into U_f's friendship list that has not been merged yet into U's friendship list.

Actually, we cannot really know because tp_U tells us only up to which entry there are correct friendship entries but there is nothing known about users at positions beyond tp_U. Furthermore, a timestamp greater than 0 means, there was at least one update or merge operation on U_f since it initially was precomputed over G_0.

Therefore, we need to merge the friendship lists of U and U_f in order to not miss any friendship updates even if the merge operation may not be necessary.

In reference to our basic algorithm shown in Listing 5, the pseudocode for the method $U.needs\_merge(U_f)$ is given in the following Listing 9.8:

```
1: U.needs_merge(U_f) {
     // (1) The timestamp of U_f's friendship list is newer or (2) i > tp_U
     // and U_f's friendship list was updated at some time, i.e. TS_{U_f} > 0
2:   IF(TS_U < TS_{U_f} || ((i:=pos_U(U_f)+1) > tp_U && TS_{U_f} > 0))
3:     RETURN(true)
4:   ELSE
5:     RETURN(false)
6: }
```

Listing 7: $U.needs\_merge(U_f)$

When there is a merge operation on a user U with a user U_f but no merge operation is necessary because all the information about transitive friends of U_f are already known in U's friendship list or are not relevant for U (because shorter paths not leading over U_f exist for U), we call the merge operation *redundant*.

Definition 9.23 (Redundent Merge Operation)**.** *A merge operation on a user U with a friend U_f is* redundant *when that merge operation does not change anything in U's friendship list $Flist_U$.* ∎

For example, a merge operation on U with U_f is redundant when that merge operation was already previously accomplished and since then, nothing relevant for U with respect to U_f has changed in the friendship graph.

Note: If U_f has been updated just before the check of $U.needs\_merge(U_f)$ then this method always returns true because the update on U_f causes the timestamp of U_f's friendship list to be greater than the one of U's. Hence, U's and U_f's friendship lists will be merged. As a consequence, when within the loop of our basic algorithm, there is a need to update U_f's friendship list then it will also cause a merge operation on U with her friend U_f.

In the following, we describe further details of our algorithm which can be implemented in different ways with different approaches for the update and merge operations. We introduce two approaches that considerably differ in the way all method calls work. Moreover, although both approaches are based on the same basic algorithm and retrieve the same true *top-k* friends for each user U in the network, they vary considerably with respect to their workload.

9.9 EAger Propagation (*EAP*) Approach

With our algorithm, we want to postpone friendship updates in a social network until a search for the *top-k* friends of a querying user U makes it necessary to consider all new information relevant for one of the shortest paths from U to these best friends.

With the eager propagation (*EAP*) approach, however, once we decide to update a friendship list $Flist_U$ of a user U, i.e. when a check for an update or merge condition indicates the need for it, we eagerly propagate all available, updated information

from other users to U and modify $Flist_U$ accordingly. Update and merge operations immediately inserts all new or updated information into a user's friendship list.

Next, we introduce the operations $U.update()$ and $U.merge(U_f)$ defining the *EAP* approach.

9.9.1 $U.update()$

An update operation on a user U identifies and removes all new or updated edges from $OP[U]$, determines among all these friendship updates the maximum timestamp $maxTS_e$ and correctly incorporates the newly discovered path information in U's friendship list.

For this, the weight for each new or updated edge (U, U_f) is computed and potentially corresponds to the new maximum friendship strength of U_f with respect to U. However, when U_f has been previously discovered over a different but higher weighted path, i.e. U_f is found in (the memory based prefix of) $Flist_U$ among U's best friends, the new or updated edge is not the new shortest path from U to U_f and simply can be discarded. Otherwise, if a friendship update leads to a shorter path, U_f's friendship strength has to be adjusted and the corresponding entry in $Flist_U$ has to be *moved up* to its correct position in U's friendship list. If U_f is not yet a friend of U at all, i.e. U_f is not found in $Flist_U$, a new entry for U_f can be created and simply be sorted into U's friendship list.

Note: If parts of a friendship list are located at a storage backend, a new entry inserted in the main memory based prefix of the list can later *move downwards* to its true position (in main memory) while the friendship list is further processed over time. When eventually the very same user is found at the storage backend for a second time, we either discard the new or the old entry depending on the higher friendship strength. When the entry from the storage backend is discarded, the user actually becomes a better friend and *moves up* in the friendship list (at the storage backend). For explanations about how to *move* entries by sequential accesses only, see Section 9.4.1. Nevertheless, once an entry is inserted in a friendship list, it can never move further upwards in the list without another friendship update since users which are already better friends cannot lose any friendship strength by update or merge operations.

The user U_f finally will reach her correct position among the *top-k* friends (if she indeed belongs to them) due to appropriately setting the *timestamp validity pointer* tp_U. In the case of an update operation, we conservatively set $tp_U = 0$. By setting tp_U to 0, it must be true that all entries in $Flist_U$ up to tp_U are correct because all direct friends are already in U's friendship list—either at their correct position or, in case the part of $Flist_U$ stored at the storage backend is not yet up-to-date, at a possibly too good position but not the first one. Hence, the *very best friend* of U can only be found by a direct edge in the friendship graph and $Flist_U[0]$ will always correctly contain U's best friend.

Next, TS_U can safely be set to $maxTS_e$ because all friendship updates for U up to time $maxTS_e$ have been already included in $Flist_U$.

In addition, after inserting a new friend U_f into U's friendship list, we have to merge U_f's friendship list into U's because there could have previously been friendship updates for U_f which are relevant to U, too. The merge operation on U with U_f will propagate all relevant information in U's friendship list.

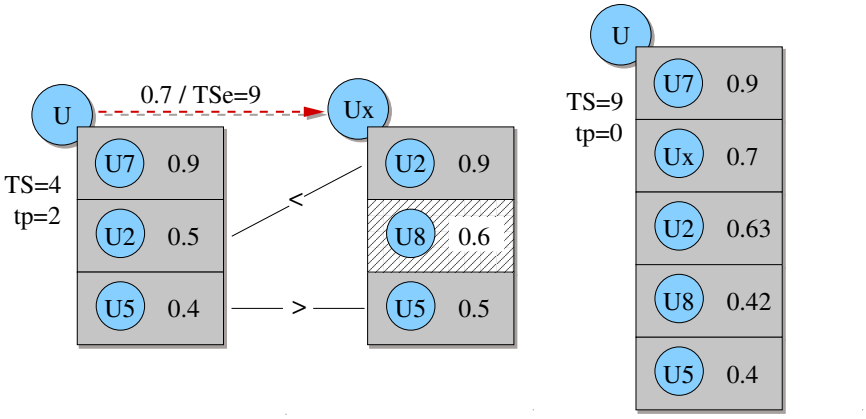
The pseudocode for an update operation on U is shown in Listing 8.

```
1:    U.update() {
        // find the timestamp for newest edge update
2:      maxTS_e = max{TS_e|(U_f, TS_e) ∈ OP[U]}
        // place all new friends correctly in Flist_U
3:      FOREACH ( e = (U, U_f) with op_e = (U_f, TS_e) ∈ OP[U] ) {
4:        OP[U].remove(op_e)
          // compute friendship strength s for e = (U, U_f)
5:        s_new = strength(U, U_f)
          // if i < 0 then U_f has not yet been seen in Flist_U
6:        i = pos_U(U_f)
7:        IF (i ≥ 0) {
          // otherwise, we check if the new path to U_f is shorter
8:          fl_e = (U_f, s) = Flist_U[i]
9:          IF (s_new > s)
10:           // if so, we remove the old entry in Flist_U
11:           Flist_U.remove(i)
12:         ELSE
              // if not, there is no improvement. Hence,we discard it
13:           RETURN
14:       }
15:       fl_e = (U_f, s_new)
          // find the position where to insert U_f in U's friendship list ..
16:       i=findPos_U(s_new)
          // .. and insert (U_f, s_new) at that position
17:       Flist_U.set(fl_e, i)
          // merge friendship lists to not miss potential new friends
18:       U.merge(U_f)
19:     }
        // U is now up-to-date, there are no more new edges
20:     TS_U = maxTS_e
        // after the update, U's best friend is for sure at Flist_U[0]
21:     tp_U = 0
22:   }
```

Listing 8: $U.update()$

```
1:  U.merge(U_f) {
2:    //get the friendship strength s_1 of U_f with respect to U
3:    i = pos_U(U_f)
4:    (U_f, s_1) = Flist_U[i]
      // merge each entry of U_f's friendship list into U' list
5:    FOREACH ( (U_ff, s_2) ∈ Flist_{U_f} ) {
        // when U is also a friend of U_f (==cycle), we can skip this entry.
6:      IF(U_ff == U) CONTINUE
7:      // compute weight w of the path from U to U_ff over U_f
      w = s_1 * s_2
8:      i = findpos_U(w)
9:      p_i = pos_U(U_ff)
        // check if U_ff has previously been found over a different path
10:     IF (p_i ≥ 0) {
          // U knows U_ff already, check if new path is shorter
          // U_f and s_1 are no longer needed. Fill with new values
11:       (U_f, s_1) = Flist_U[p_i]
12:       IF (w > s_1)
            //new path is shorter, remove wrong entry
13:         Flist_U.remove(p_i)
14:       ELSE
            //new path is longer, discard it
15:         CONTINUE
16:     }
        //insert new friend or updated friendship strength
17:     Flist_U.set((U_ff, w), i)
18:   }
      // we merged in all infos known of both lists
19:   TS_U=max(TS_U,TS_{U_f})
      // U's list can only have changed after the position of U_f
20:   tp_U=pos_U(U_f)+1
21: }
```

Listing 9: $U.merge(U_f)$



(a) Example for a friendship update $(U \rightarrow U_x)$ with initial friendship lists given for U and U_x

(b) Resulting $Flist_U$ after applying the update operation on U.

Figure 15: In (a) an example for a friendship update $(U \rightarrow U_x)$ is given with initial friendship lists for U and U_x. (b) shows the resulting friendship list for U after the update operation on U (which includes a merge operation on U with U_x)

9.9.2 $U.merge(U_f)$

A merge operation on U with U_f identifies all users U_{ff} in U_f's friendship list who are not yet known to U or whose friendship strength increases because the path over U_f to U_{ff} is shorter than a previously discovered path to U_{ff}. The pseudocode for a merge operation U is given in Listing 9.

Our algorithm ensures that we only do a merge operation on U with U_f when a friend U_f has been found at its correct position in U's friendship list and there is no more friendship update for U_f. It also means that friends taken from U_f's friendship list have got a weaker friendship strength with respect to U than U_f and, thus, will be inserted in U's friendship list at a position later than U_f.

In compliance with Definition 9.7, the indirect friendship strength for a user U_{ff} from U_f's friendship list with respect to U (for the path from U to U_{ff} over U_f), is computed by multiplying the weight of the path from U to U_f with the weight of the path from U_f to U_{ff}, i.e. $s_f(U, U_{ff}) = s_f(U, U_f) \cdot s_f(U_f, U_{ff})$.

If a user U_{ff} found in U_f's friendship list is already known to U, we have to check if the new path over U_f is shorter. If not, we simply can discard the longer path. Otherwise, we remove the old entry with U_{ff} from U's friendship list and appropriately insert it again with the updated friendship strength.

As soon as we have merged the complete lists, we can increase the timestamp TS_U of U's friendship list to the highest timestamp of both lists. All the latest information with respect to that timestamp is now available in U's friendship list.

The timestamp validity pointer tp_U is set to the next entry after U_f in U's friendship list because U_f's friend with the highest friendship strength could at best be inserted at exactly that position and, hence, could have displaced the user at that position.

An example for an update operation $(U \rightarrow U_x)$, including a merge operation on U with U_x, is given in Figure 15. Initially, the timestamp of U's friendship list is $TS_U = 4$ and the timestamp validity pointer $tp_U = 2$. Then, an update operation is applied on U

for a friendship update $(U \rightarrow U_x)$ with a new friendship strength 0.7 at time $TS_e = 9$. Figure 15a sketches this setup and depicts the friends and their friendship strengths in both users' friendship lists. It can be observed that the two users U_2 and U_5 in U_x's friendship list are also friends of U while the remaining friend U_8 of U_x is not yet known to U. Figure 15b shows the resulting friendship list of U after the update operation on U has been applied which also involves a merge operation on U with U_x. The resulting friendship list is achieved as follows: First, the update operation inserts the new friend U_x together with her friendship strength 0.7 in U's friendship list. Next, the merge operation with U_x replaces the entry with U_2 in U's friendship list since the path from U to U_2 over U_x is the new shortest path. U_2's new friendship strength is computed by multiplying the friendship strength of U_x with respect to U and the one of U_2 with respect to U_x. Hence, U_2's friendship strength increased to $0.63 = 0.7 * 0.9$ due to the friendship update for U. However, there is no change for the second user U_5 known to both users U and U_x since her friendship strength, 0.4, is already greater than the weight of the path over U_x, i.e. $0.35 = 0.7 * 0.5$. Finally, the new friend U_8 is merged from U_x in U's friendship list. The update operation is completed by setting the timestamp of U's friendship list to the timestamp of the friendship update, i.e. $TS_U = 9$, and by setting U's timestamp validity pointer tp_U to 0.

9.9.3 $U.getFriend(i)$

With our *EAP* approach, this method is very simple because all work for correcting friendship lists is accomplished by $U.update()$ and $U.merge(U_f)$. Furthermore, we only process U's friendship list sequentially and, thus, when this method is called, necessary update and merge operations have already been done. Hence, the timestamp validity pointer tp_U is always equal to i and, in this case, we know that the i*-th* entry in U's friendship list is correct. Therefore, this method only needs to return the friend found at the i*-th* position which is shown by the following pseudocode in Listing 10:

```
1: U.getFriend(i) {
2:     (U_f, s)=Flist_U[i]
3:     RETURN((U_f, s))
4: }
```

Listing 10: $U.getFriend(i)$

9.9.4 Friendship Graphs with Cycles

With *EAP*, a merge operation on U with a friend U_f immediately merges all those entries with users in U_f's friendship list who are not yet known to U or who become better friends of U when considering the path leading over U_f. No other entries are considered for inserting in U's friendship list. Moreover, by multiplying the friendship strengths of friends more than once, by definition, a friend cannot become a better friend, and in particular, a merge operation never merges U herself into her own friendship list because U is, of course, known to herself and the friendship strength cannot increase, too (—U's friendship strength to herself is 1 according to Definition 9.9). Hence, after a merge operation, all paths with cycles starting in U and leading over U to some user U_f have already been discarded with respect to the current timestamp as U_f is already known over a shorter path. The same is true for possible subsequent

merge operations on U_f with U. Therefore, cycles in friendship graphs do not cause any problems with the *EAP* approach and do not require any special attention.

9.9.5 Disadvantage of *EAP*

Although the *EAP* approach of our APSD algorithm works nicely when all friendship lists of users that are processed during a query can be loaded quickly into main memory or when they actually can be completely kept there, it considerably slows down when all user entries of very long friendship lists are fetched one by one from a comparatively slow storage backend like a database.

Even though our algorithm implementing *EAP* only starts updating and inserting new edges into friendship list when necessary—that is, when a query leads to a friend with new and relevant information—and also only for those users who are involved in the process of answering a query, yet complete lists have to be processed from the first entry to the last for both users involved in a merge operation.

By eagerly propagating update information, our algorithm does not miss any transitive friends but takes all information from all users found during a query into account – from the very best friend at the first position to the user farthest away but who is still connected in the friendship graph of a social network. Since we only merge lists once for a given timestamp, we could miss transitive friends otherwise.

9.9.6 Improvement by Limitation: *fixed-size EAP*

An idea for improving the *EAP* approach is to either fix the value of *top-k* for all queries or to introduce a second, fixed value *max-k* which functions as an upper bound on the value of *top-k* for all queries. Afterwards, we limit the entries in friendship lists being considered during a merge operation to the first *top-k* or *max-k* entries in both lists. The intuition here is that users at later positions cannot become one of the *top-k* friends (for all values of *top-k*$\leq$*max-k*) because all friends at earlier positions are already known to be better. We call this modified *EAP* approach *fixed-size EAP*.

However, *fixed-size EAP* only applies when it can be guaranteed that a query never has to retrieve more than the specified fixed number of friends. Once a fixed value of *top-k* or *max-k* has been chosen, there is no way to dynamically extend the search to more than this fixed number of friends without computing erroneous results as most likely, the knowledge about friends at later positions in friendship lists are incomplete. Hence, the friendship lists for all users have to be recomputed from scratch in such a case.

9.10 LAzy Propagation (*LAP*) Approach

The idea of the lazy propagation (*LAP*) approach of our algorithm for incrementally maintaining shortest path distances between all pairs of users in a friendship graph of a social network actually carries the general idea of postponing friendship updates (as already done in *EAP*) to the next level: We further postpone work that is necessary to keep friendship lists up-to-date, and, at all times, only do as little work as possible while avoiding the introduced limitations (see Section 9.9.6) and disadvantages (see Section 9.9.5) of *EAP*. Hence, once they are necessary, friendship updates of users are only lazily propagated to other users.

To be more precise, with the *LAP* approach, we want to avoid complete merges of friendship lists but instead only want to repeatedly merge single entries until we have

identified all users who potentially could become one of the *top-k* friends. For this, the work done by calls to $U.update()$ and $U.merge(U_f)$ is deferred until it is really necessary to be done, namely, when $U.getFriend(i)$ is called. Moreover, the amount of work is limited to that part which immediately has to be done.

Usually, the requested *top-k* friends form only a small prefix of a user's comparably long friendship list which contains all reachable users in the friendship graph of a social network. Therefore, it is not necessary to maintain the correctness of friendship lists beyond the first *top-k* positions as long as we still can do that later when required, i.e. when a query asks for larger number of *top-k* friends.

With *LAP*, the execution of an update and especially of a merge operation differs considerably compared to the *EAP* approach (see Section 9.9). *LAP* implements the following principles:

1. $U.update()$: As with *EAP* (see Section 9.9.1), an update operation on U retrieves from the operation map OP for U all relevant new or updated edges. However, in contrast to *EAP*, even during the query processing, U's friendship list is not immediately modified with those friendship updates. Instead, the friendship strength together with a reference to the corresponding updated friend U_f is stored in a priority queue PQ_U of "*next best friend*"-candidates (see Section 9.10.1). The head of the queue always points to the candidate with the highest friendship strength.

2. $U.merge(U_f)$: Again, as with *EAP* (see Section 9.9.2), a merge operation on U with U_f effectively gathers the information about friendship updates from U's friend U_f in order to propagate to U shortest path distances of friends found on a path over U_f. However, in contrast to *EAP*, the friendship list of U_f is not completely merged into U's but only a reference to U_f's best friend, i.e. the first entry in U_f's friendship list, together with her friendship strength in regard to U is kept in a priority queue PQ_U of "*next best friend*"-candidates for U (see Section 9.10.1).

9.10.1 Additional Bookkeeping

In contrast to our first approach and as mentioned earlier, we need an additional data structure which keeps pointers to friends' friendship lists which have not yet been completely merged.

In order to postpone updates and merges of friendship lists, and, most important, to avoid complete list merges, we maintain for each user U a priority queue PQ_U of *"next best friend"*-candidates.

Definition 9.24 (*"Next Best Friend"*-Candidate)**.** *A user U_{ff} is a* "next best friend"*-candidate of U when U's friendship list $Flist_U$ is processed during a query and U_{ff} is a candidate for being inserted at position i in $Flist_U$ while $i-1$ friends of U have already been identified.*

Definition 9.25 (Priority Queue PQ_U of U's "*next best friend*"-candidates)**.** *For each user U, a priority queue PQ_U contains triples $pq_t = (U_f, U_{ff}, s)$ where U_f is an already known friend of U and U_{ff} is a friend of U_f, or $U_{ff} = U_f$. Hence, U_{ff} is a* "next best friend"*-candidate of U with friendship strength s, corresponding to the weight of the shortest, known path from U to U_{ff} leading over U_f. The head of the priority queue PQ_U is always the candidate with the highest friendship strength s.*

In other words, U_{ff} is U's friend's friend and s corresponds to the weight of a path from U over U_f to U_{ff} which potentially is indeed the shortest path from U to U_{ff}.

Remark: When discovering due to a friendship update a new path for U to a friend U_f which only consists of a single, direct edge, not the best friend of U_f is a candidate for U's next best friend but U_f herself (because the friendship strength of U_f may have increased due to the new or updated edge). Hence, we set $U_{ff} = U_f$ for the corresponding entry in PQ_U.

A candidate U_{ff} is indeed the next best friend of U, when it turns out that the currently known shortest path over U_f to U_{ff} is in fact a shortest path, and the path is not yet known in $Flist_U$, and no shortest path to another candidate in PQ_U has a higher weight.

Since the friendship strength s remembered in an entry in PQ_U is equal to the weight of the shortest path from U to U_{ff} leading over U_f, it is in any case a lower bound for the actual friendship strength of U_{ff} with respect to U and timestamp TS_U.

In our algorithm the following characteristics for PQ_U are always true:

- U_{ff} is a friend of U_f or equal to U_f. In either case, U_{ff} is a candidate for the *"next best friend"* of U.
- When U_{ff} is U_f's friend, she can be found in U_f's friendship list. Her position in $Flist_{U_f}$ is subsequent to the prefix that already has been merged into U's friendship list.
- At all times, there is always only one friend U_{ff} originating from U_f's friendship list in PQ_U. Hence, U_f's friendship list is also only once referred to in PQ_U and, moreover, U_f can only appear in a single triple as the first element of an entry in PQ_U. However, the same user can be a friend of several different users and, thus, she may be known in several different friendship lists. Therefore, the same user can appear several times as the second element in different triples in PQ_U.

Maintaining a disk-resident version of PQ_U

In Section 9.4.1 we explained that our algorithm dynamically builds friendship lists in main memory by sequentially retrieving entries from a previous, disk-resident version of the list. Once, the processing of a query finishes, the updated list in main memory is written back to the storage backend.

By introducing PQ_U, we need to extend this procedure accordingly. For efficiency reasons, PQ_U has to be completely available in main memory as soon as the query processing needs to access U's priority queue since PQ_U is not suitable for being maintained by sequential accesses only. Hence, PQ_U is always associated to the friendship list that is dynamically built in main memory. There is no priority queue associated to the disk-resident friendship list whose entries, as before, are just sequentially read and inserted into the friendship list in main memory. For this reason, the priority queue PQ_U can be written back to disk at any time by simply replacing any earlier version on disk and loaded back into main memory whenever a query needs to access it.

Since the priority queue PQ_U can usually be expected to be much smaller than $Flist_U$ (depending on the value of *top-k*—not more than *top-k* users can be actually candidates for query results of size *top-k*) writing back and reloading PQ_U into main memory is usually not a big performance issue. Over time, however, when the friendship graph and the user's *top-k* friends change a lot, there might be old candidates in PQ_U due to queries to early graph versions that may never belong to the current best *top-k* friends. Hence the queue grows over time and, thus, such weak candidates should be removed from PQ_U by merging their lists completely into $Flist_U$. In our experiments presented in Section 9.12, the priority queues PQ_U for users U involved in queries, do not exhibit enormous sizes and their loading times—even though no caching techniques are applied but the loading improvements described in Section 9.10.6—do only a little penalise the average runtimes of our *LAP* approach compared to *fixed-size EAP*.

The maintenance of the in-memory friendship list is unchanged with regard to the description given in Section 9.4.1.

Next we describe the implementation details of the update and merge operations with the *LAP* approach.

9.10.2 $U.update()$

A call of $U.update()$ finds and removes all new or updated friends U_f in the operation map $OP[U]$ for U, computes the respective friendship strength $s = s_f(U, U_f)$ and adds U_f together with s to U's priority queue PQ_U. The pseudocode for an update operation is given in Listing 11.

As mentioned earlier, an entry in PQ_U is a triple $pq_t = (U_f, U_{ff}, s)$ and we set $U_{ff} = U_f$ for all friends U_f who have been newly found over a shortest path consisting of only a single edge starting at U. The priority queue ensures that the head of the queue is always the candidate with the highest friendship strength. If there is a friendship update $(U \rightarrow U_f)$ for U with respect to a user U_f and U_f or a user from U_f's friendship list is already a *"next best friend"*-candidate in PQ_U, it means, U_f was previously discovered either over an indirect path from U to U_f or due to a differently weighted direct edge. Hence, the following has to be accomplished for a friendship update on U with respect to U_f:

- If the weight of the new or updated edge $e = (U, U_f)$ is smaller than the weight of an already known path to U_f, then the friendship update can simply be discarded since the shortest path to U_f has not changed.
- Otherwise, if there is an entry (U_f, U_{ff}, s) in PQ_U, it has to be removed in order to update s to the new friendship strength of U_f and we set $U_{ff} = U_f$.

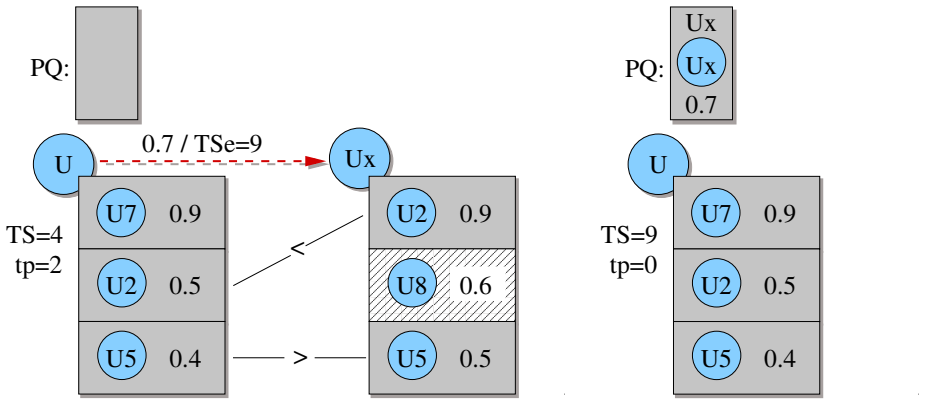
  The latter is done to process U_f's friendship list from the top again and is necessary because all friends of U found on a shortest path over U_f have become better friends due to the increase in the friendship strength of U_f.

  Finally, the entry (U_f, U_{ff}, s) is added to PQ_U again.
- If there is not yet an entry (U_f, U_{ff}, s) in PQ_U, we add $(U_f, U_{ff} = U_f, s)$ to PQ_U with s is equal to U_f's friendship strength.

Next, the timestamp TS_U and the timestamp validity pointer tp_U of U's friendship list $Flist_U$ can be changed. TS_U is set to the newest timestamp TS_e of all friendship

```
1:  U.update() {
      // find the timestamp for newest edge update
2:    maxTS_e = max{TS_e|(U_f, TS_e) ∈ OP[U]}
      // put all new friends in PQ_U
3:    FOREACH ( e = (U, U_f) with op_e = (U_f, TS_e) ∈ OP[U] ) {
4:      OP[U].remove(op_e)
        // compute friendship strength s for e = (U, U_f)
5:      s_new = strength(U, U_f)
        // if there is an entry in PQ_U which was found in U_f's friendship list ..
6:      IF (∃ pq_t = (U_f, U_ff, s) ∈ PQ_U) {
          // .. then U_f is already known. If U_f became a better friend now ..
7:        IF (s_new > s)
          // .. we remove pq_t, so we can also update all friends of U_f wrt. U
8:          PQ_U.remove(pq_t)
9:        ELSE
          // else, we can discard the update. There is a shorter path to U_f
10:         CONTINUE
11:     }
        // and put a new triple with U_ff = U_f in PQ_U
12:     PQ_U.add(U_f,U_f,s_new)
13:   }
      // U is now up-to-date, there are no more new edges
14:   TS_U = maxTS_e
      // after updating, U's best friend is either at Flist_U[0]
      // or the head element of PQ_U
15:   tp_U = 0
16: }
```

Listing 11: $U.update()$



(a) Example for a friendship update $(U \rightarrow U_x)$ with initial $Flist_U$ and empty PQ_U.

(b) Resulting $Flist_U$ and PQ_U after applying an update operation.

Figure 16: In (a) the same example for a friendship update $(U \rightarrow U_x)$ is given as shown with our first approach. (b) shows the resulting friendship list and priority queue for U after the update operation on U with our second approach.

updates because all changes to the social network with respect to user U and timestamp TS_e are eventually known to U. Hence, all friends of U up to this timestamp have been seen and are either in PQ_U or in $Flist_U$. That means, the very best friend of U is either still at the first position of U's friendship list $Flist_U[0]$ or can now be found in PQ_U. Finally, we safely can set the timestamp validity pointer tp_U to 0 because it is guaranteed to find at least the best friend of U with respect to the timestamp TS_U.

An example for an update operation on U is given in Figure 16. The setup follows the one given in Figure 15a for the *EAP* approach and is re-sketched with respect to *LAP* in Figure 16a. Actually, the only difference is that in addition an empty priority queue of *"next best friend"*-candidates is given. The initial friendship lists with friends and their friendship strengths of both users U and U_x, and the friendship update $(U \rightarrow U_x)$ at timestamp TS_e with weight 0.7 are unchanged.

However, as shown in Figure 16b, the resulting friendship list of U differs considerably from our first approach. In fact, U's friendship list has not changed at all. Instead, only the new or updated friend U_x has been added to U's priority queue. The timestamp TS_U of U's friendship list and the timestamp validity pointer tp_U are nevertheless set in the same way as in the *EAP* approach to $TS_U = 9$ and $tp_U = 0$, respectively.

A subsequent call to $U.getFriend(0)$ determines U's best friend by comparing the friendship strength of the first friend in U's friendship list with the one at the head of the priority queue. See Section 9.10.4 for a detailed explanation.

9.10.3 $U.merge(U_f)$

In contrast to the *EAP* approach, with *LAP*, we do not want to immediately propagate into U's friendship list all updates found in a friend's friendship list but only to identify that single user who is a *"next best friend"*-candidate for U. In U's priority queue PQ_U, we remember the new candidate U_{ff}, her friendship strength s with respect to U and the owner U_f of the friendship list in which the candidate was found,

```
1:  U.merge(U_f) {
      //if there is a user from U_f's friendship list in PQ_U → remove her
2:    IF (∃ pq_t = (U_f, U_ff, s) ∈ PQ_U)
3:      PQ_u.remove(pq_t)
      // get the very best friend of U_f
4:    U_ff=U_f.getFriend(0)
      // put U_f's best friend to PQ_U
5:    PQ_U.add((U_f, U_ff, s = strength(U, U_f) · strength(U_f, U_ff)))
      // we merged in all infos known at time of max. timestamp
6:    TS_U=max(TS_U,TS_U_f)
      // U's list can only have changed after position of U_f
7:    tp_U=pos_U(U_f)+1
8:  }
```

Listing 12: $U.merge(U_f)$

i.e. $(U_f, U_{ff}, s) \in PQ_U$.

The head of the queue is always the candidate with the highest friendship strength. A triple (U_f, U_{ff}, s) is permanently removed from PQ_U only, when all friends found in U_f's friendship list has been merged into U's friendship list. There is always only one triple in PQ_U with U_f as a first element, and therefore, at the same time, only one friend originating from U_f's friendship list can be a candidate for the next best friend of U.

A call of $U.merge(U_f)$ only happens for users U_f in U's friendship list and only, when U_f's friendship list has changed such that it potentially could contain a candidate for U's next best friend.

Observation A shortest path which leads over a friend U_f and ends at one of her friends can only be longer than a shortest path that ends at U_f herself. Therefore, all friends found in U_f's friendship list can only have got a lower friendship strength with respect to U than U_f herself. Hence, all friends of U_f who are merged into U's friendship list will be sorted below the position of U_f whose position and friendship strength itself remains unaffected.

However, U's next best friend after U_f may have changed due to updates in U_f's friendship list. Therefore, U_f's very best friend U_{ff} is added to PQ_U because she is a possible candidate for this position. According to Definition 9.7, the friendship strength for this candidate is the product of U_f's friendship strength with respect to U and that of U_{ff} with respect to U_f, i.e. $s_f(U, U_{ff}) = s_f(U, U_f) \cdot s_f(U_f, U_{ff})$.

After adding the very best friend of U_f to PQ_U, we know for sure that U's next best friend after U_f is either already the next entry in U's friendship list or can be found in PQ_U.

For the same reasons as for an update operation, we have to remove from PQ_U an already existing entry with U_f as friendship list owner before we add the new entry. Hence, in contrast to *EAP*, a merge operation is very similar to an update operation with our *LAP* approach.

The timestamp TS_U is set to the maximum of the timestamps of U's and U_f's friendship list because we have identified (and remembered in PQ_U) all information known from both lists with respect to the maximum timestamp. The timestamp validity pointer tp_U can safely be set to the next entry after U_f. The next best friend for this position is either already found at that position or can be found in PQ_U.

The pseudocode for a merge operation on U is shown in Listing 12.

9.10.4 $U.getFriend(i)$

As seen in the previous sections, there is almost no work done by an update or merge operation. When a need for an update or merge operation has been identified, only (a triple with) the new friend in case of an update or (a triple with) the friend of a friend in case of a merge operation is put into PQ_U.

While with the *EAP* approach of our algorithm both operations have to do all the work immediately, the operations for update and merge with *LAP* are almost identical and defer the actual work to the time when indeed the next best friend has to be identified.

That is also the reason why with *EAP* the method call $U.getFriend(i)$ did actually nothing but returning the friend at the *i-th* entry in U's friendship list, i.e. $Flist_U[i]$, while the same method with the *LAP* approach has to do almost the entire work: identifying the next best friend from all candidates and correcting U's friendship list if necessary.

A call of $U.getFriend(i)$ to U's friendship list is always done in a sequential manner from the top, starting with $i = 0$ towards the bottom with i set to the last entry for which a next best friend can be correctly identified either in $Flist_U$ or PQ_U and without the need for any further merge or update operations. The timestamp validity pointer tp_U is a marker for that entry. Therefore, when $U.getFriend(i)$ is called, $i \leq tp_U$ is always true.

For the case $i < tp_U$, the method simply returns the friend found at the i-*th* position of U's friendship list and nothing else has to be done. This is correct because of the characteristics of tp_U.

The more interesting case occurs when $i = tp_U$. In this case, we have to identify the candidate in PQ_U who is not yet one of the *top-i* best friends of U and check if this candidate is indeed a better friend than the user who is already listed at the *i-th* position in U's friendship list.

Note: For all candidates U_{pq} found in PQ_U, we can safely say that their friendship strengths with respect to U are smaller or equal to those of all friends at positions $i < tp_U$, i.e. $s(U, U_{pq}) \leq s(U, Flist[i'])$ with $i' < tp_U$. That is true, because otherwise the candidate had been found in a previous call of $U.getFriend(i')$ with $i' < i$.

However, identifying the actual *i-th* friend from all candidates in PQ_U is not trivial and can be expensive because of potentially many nested method calls.

The following tasks have to be done:

1. Remove from PQ_U the triple (U_f, U_{pq}, s) with the top *"next best friend"*-candidate U_{pq}. The user U_f is the owner of the friendship list in which U_{pq} has been found. The friendship strength s of candidate U_{pq} is the highest among all candidates in PQ_U.

2. If s is *not* higher than the friendship strength of the user at the i*-th* entry in U's friendship list, we have to put the current candidate back to PQ_U. The user at $Flist[i]$ is a better friend but U_{pq} is still a candidate for a later position in U's friendship list. At this point, the method call is done and returns the friend found at $Flist[i]$.

3. Otherwise, if s is the currently highest friendship strength, we have to identify U_f's next friend after U_{pq} and add her to U's priority queue PQ_U. This step actually causes a (nested) call to $U_f.getFriend(i')$ and is necessary because there could be more not yet known friends in U_f's friendship list that at a later point in time have to be merged in U's list.

   Remember: by using an *iterator* on each opened list, the argument i of the method $.getFriend(i)$ is actually not needed. It is used only to ease the discussion of our algorithm. Hence, the argument i' mentioned above is implicitly known during the execution.

4. Next, we have to check if the current candidate U_{pq} is already in U's friendship list at an already processed position $< i$. If so the candidate was already found earlier by a different path which is shorter. Hence, we have to go back to task 1 again.

5. We also have to check if U_{pq} is equal to U herself. If so, we have found a path with a cycle that leads back to U. That path can be ignored by going back to task 1 again.

6. Finally, if the friendship strength s of the top candidate U_{pq} is higher than the current friend at position i of U's friendship list, then U_{pq} is indeed the next best friend and we insert an entry (U_{pq}, s) at the i*-th* position in $Flist_U$. If the friendship strength of U_{pq} actually was increased over time, it means, that U_{pq} is moving up (see Section 9.4.1) from a later position than tp_U to $i = tp_U$.

7. The method call is finally done by returning the friend found at $Flist[i]$.

The pseudocode of $U.getFriend(i)$ is shown in Listing 13.

The method $U.queueNext(pq_t)$ adds the next friend of U's friend U_f to U's priority queue PQ_U and is actually only an auxiliary function to improve readability. The pseudocode is shown in the following Listing 14.

```
1:  U.getFriend(i) {
      // entries up to Flist_U[tp_U − 1] are always correct. Flist[tp_U] is
      // either correct or entry is found in PQ_U. It's always true: i ≤ tp_U
2:    IF (i == tp_U) {
        // get and remove from PQ_U the triple pq_t with top candidate U_pq,
3:            pq_t = (U_f, U_pq, s) = PQ_U.poll()
        // when U_pq is known or equals to U, put the next friend from U_f's
        // friendship list in PQ_U and fetch again the top candidate
4:      WHILE(0 ≤ pos_U(U_pq) < tp || U_pq == U ) {
        // note: here it is always true that s ≤ strength(U, U_PQ)
        // because otherwise U_pq would have been found earlier
5:        U.queueNext(pq_t)
          // get & remove the new top element
6:        pq_t = PQ_U.poll()
7:      }
        // pq_t==NULL: no more candidates in PQ_U, return U_f
8:      IF (pq_t==NULL) RETURN(U_f with U_f ∈ (U_f, s) = Flist_U[i])
        // IF: now, U_pq is either U's next best friend and has not yet been
        // known (pos_U(U_pq) < 0) or has been found at tp_U with a better s
9:      IF (Flist[i]==NULL || s > strength(U, Flist[i]) {
        // here: strength(U, Flist_U[i − 1]) ≥ s > strength(U, Flist_U[i])
          // insert or move up U_pq to i
10:       FList_U.set(U_pq, i)
11:       U.queueNext(pq_t)
        // ELSE: U_pq is not U' next best friend
12:     }ELSE {
          // we put the top user U_pq back into PQ_U because
          // the i-th entry in Flist_U is correct
13:       PQ_u.add(U_f, U_pq, s)
14:     }// here, i == tp_U, and the i-th entry in Flist_U is now correct
         // therefore, increase tp_U by one, so (i+1)-friend is found next
15:     tp_U++
16:   }
17:   RETURN Flist_U[i]
18: }
```

Listing 13: $U.getFriend(i)$

```
1: U.queueNext(pq_t = (U_f, U_pq, s)) {
     // check if there is a need to update U_pq's friendship list ..
2:   IF (U_pq.needs_update()) U_pq.update()
     // .. and if we need to merge any updates of U_pq in U_f's friendship list
3:   IF (U_f.needs_merge(U_pq)) U_f.merge(U_pq)
     // find the next friend from U_f's list ..
4:   U_pq=U_f.getFriend(pos_{U_f}(U_pq) + 1)
     // .. and put her into PQ_U
5:   PQ_U.add(U_f, U_pq, s = (strength(U, U_f) · strength(U_f, U_pq)))
6: }
```

Listing 14: $U.queueNext(pq_t = (U_f, U_{pq}, s))$

Note: When in U's priority queue PQ_U the currently processed candidate U_{pq} is equal to the friendship list owner U_f because U_f herself is the candidate for U's next best friend, then the candidate U_{pq} cannot be found in U_f's friendship list. Hence, $pos_{U_f}(U_{pq}) = -1$ and $U_f.getFriend(pos_{U_f}(U_{pq}) + 1) = U_f.getFriend(0)$, which means, we correctly queue U_f's very best friend in PQ_U.

9.10.5 Resolving Cyclic Friendship Connections

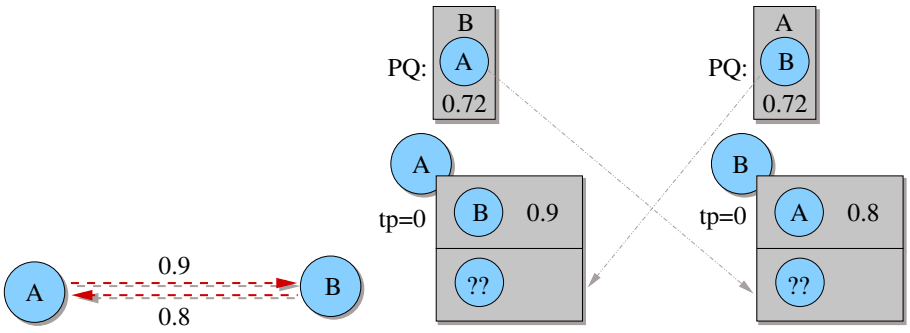
With *LAP*, we also have to pay attention to cycles in friendship graphs. The reason is, that in this approach, a merge operation does not merge entire lists but only considers a single entry from a list.

To illustrate the problem and complexity of resolving cycles in friendship lists, we present two showcases and also discard our assumption (see Section 9.4.2) that friendship lists are completely located in main memory. Instead, $Flist_U$ is constructed from an empty list by sequentially reading a previous version of U's friendship list and by examining U's priority queue of *"next best friend"*-candidates.

For a user U, let DB_U denote the latest known version of her friendship list which is correct with respect to its timestamp TS_U up to the position indicated by tp_U and let DB_U be located at a storage backend, e.g. a database (remember from Section 9.10.1: there is no priority queue associated with DB_U). Furthermore, we continue to restrict ourselves to only sequentially access entries $db_e = (U_f, s)$ in DB_U with U_f is a friend of U and s denotes her friendship strength. Therefore, at any time, we only can fetch the next entry in DB_U, that is, $DB_U[0]$ when accessing the friendship list DB_U for the first time, $DB_U[1]$ when accessing the list a second time, and so on until there is no more entry in DB_U.

For a given query, U's friendship list $Flist_U$ is then build in main memory by reading from DB_U and its associated priority queue PQ_U.

Note: It is possible during the query processing, that there is an entry in the in-memory prefix of U's friendship list which is located at a position $Flist[i]$ with $i > tp_U$. This eventually means that the list has been already processed to a position later than the current value of tp_U at a time earlier than indicated by the current value of TS_U. Hence, a user located at that *i-th* entry is maybe not the true *i-th* friend with respect to the current timestamp TS_U. However, as users in $Flist_U$ originate either from PQ_U



(a) Example for graph with a cycle (b) Resulting friendship lists and priority queues

Figure 17: In (a) two users A and B are given with friendship updates from one to each other. In (b), the resulting data structures are depicted after a *top-1* query on A. The grey, dashed arrows indicate the problem with retrieving the next best friend of A.

or DB_U, it means that each user found in U's currently processed friendship list $Flist_U$ must be already a better friend than anyone yet to be found in DB_U; candidates chosen from PQ_U are either not yet known friends and, thus, cannot be found in DB_U, or, with reference to the current timestamp, are now even better friends. Users from DB_U are fetched in descending order of their friendship strength and as soon as a user is chosen to be the next best friend in $Flist_U$, the list DB_U is traversed to its next entry, and therefore, the next user in DB_U cannot be a better friend than anyone who is already located in $Flist_U$.

Showcase I: Assume we have got two users A and B and both users are mutual best friends. Further assume that A is the querying user and for A and B, the very best friend is discovered during the query execution by processing in each case a friendship update $(A \rightarrow B)$ with weight 0.9 or $(B \rightarrow A)$ with weight 0.8, respectively. The example graph is depicted in Figure 17a.

When A issues a query, our algorithm first updates A with the new edge to B by adding $pq_t = (B, B, 0.9)$ to A's priority queue PQ_A due to a call of $A.update()$. As B is by assumption the new best friend of A, the entry will be at the head of PQ_A.

When our algorithm now tries to identify the very best friend of A by calling $A.getFriend(0)$, it finds B as top candidate in A's priority queue and inserts $(B, 0.9)$ as top entry in $Flist_A$. Furthermore, B's very best friend is queued next into PQ_A. Since A is by assumption the best friend of B, it means, A herself is regarded as candidate with a friendship strength of $0.72 = 0.9 * 0.8$. Hence, the triple $(B, A, 0.72)$ is added to PQ_A.

However, to identify that A is B's very best friend, for B, the same checks and method calls $B.update()$ and $B.getFriend(0)$ have to be accomplished. Indeed, this is the case with our second approach due to A's call of $A.queueNext((B, 0.9))$. In conclusion, first $(A, 0.8)$ is added to PQ_B because of the friendship update $(B \rightarrow A)$ with weight 0.8. Then, A is chosen to be at the top entry in $Flist_B$ because A is by assumption the very best friend of B and, finally, A's very best friend is queued next in PQ_B, which is again B herself with a friendship strength $0.72 = 0.8 * 0.9$. Hence, $(A, B, 0.72)$ is the next triple in PQ_B.

At this point in time, we have identified the first friend for a *top-k* query issued

by A and in the main loop of our algorithm, we increase i by one to get the next best friend. A snapshot of our current data structures is illustrated in Figure 17b.

The problem with cycles in friendship graphs becomes obvious when trying to find A's next best friend, i.e. the second best friend, in order to update A's friendship list at $Flist_A[1]$.

Assume, as depicted in the example graph in Figure 17a, there is no other friendship list involved, i.e. there is no further candidate in PQ_A and PQ_B and also there is no other friend found at the storage backend DB_A or DB_B.

Now, when trying to find A's next best friend, the candidate at the head of the priority queue PQ_A is fetched which is A as (B, A, s_{ba}) is the only entry in PQ_A. Since A is obviously not the right next best friend, and because there is no other friend in $FList_A$ or DB_A, our algorithm tries to queue B's next best friend as another candidate for A.

However, by trying to find B's next best friend, we will end up in the same situation as with A. The candidate for B's next best friend is A's next best friend. Hence, we are stuck in a cycle which is depicted in Figure 17b by the two crossing arrows pointing from the entries in each user's priority queue to the unknown entry of the other user's friendship list.

When we are able to identify a cycle, the solution for our first showcase is simple. As there is no other candidate for a best friend, we know, that each list ends at the current entry and there is only one friend for A and B.

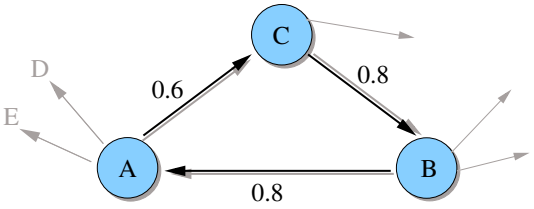
Showcase II: When there are more possible candidates involved in a search for a next best friend and a call to $U.getFriend(i)$ ends in a cycle, we have to take care of all potential sources where the true *i-th* friend could be found.

Assume the following plausible state of our algorithm as illustrated in Figure 18a where A is the querying user and B and C are already known friends of A. Moreover, there is a cycle from A over C to B and back to A, and B is the final user on the cycle before A is reached again. The corresponding data structures are depicted in Figure 18b.

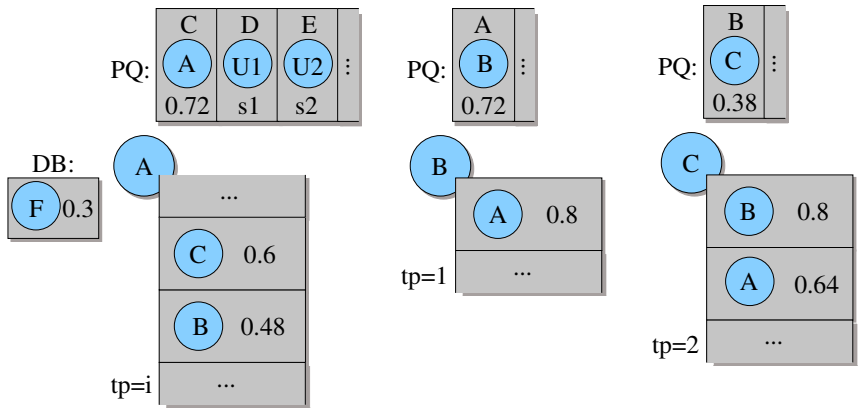
The friendship list of user A has already been processed up to its $(i-1)$*-th* entry and B is the corresponding friend at that position. The best candidate in PQ_A for the i*-th* best friend of A is found at the head of the priority queue and is currently A herself who previously was found in C's friendship list. Furthermore, B's very best friend is A and C's very best friend is B. Obviously, we have got a cycle from A over C to B and back to A.

Further assume we have identified the cycle at this point and also that the friendship strength of friend F, found at the top position i' of A's storage backend DB_A, is weaker than the weight 0.384 of the cyclic path to A over C. In this situation, we have not yet identified A's next best friend but instead have to find an adequate candidate in PQ_A.

Potentially, the following users could be A's next best friend: One of the next best friends found in C's friendship list, which also could be one of the friends in B's friendship list. However, A's next best friend could also be found among the remaining candidates in PQ_A, e.g. users found in the lists of D or E. However, if these users are all already known by A and, thus, were previously found over a shorter path, every other potential candidate in PQ_A is indeed a candidate for A's next best friend. Eventually, the *i-th* friend could also be retrieved from the storage backend DB_A or, in addition, the appropriate friend could already be in $Flist_A$ at the position indicated by A's timestamp validity pointer tp_A.



(a) Example graph with a cycle over 3 users



(b) A possible snapshot of the data structures for the users involved in a cycle

Figure 18: In (a) an example graph is given with a cycle over 3 users. A conceivable snapshot of each user's friendship list and priority queue is given in (b).

Nevertheless, the number of elements in PQ_A is limited by the number of already processed friends in $FList_A$. In the worst case, no two users in $FList_A$ are found in the same friendship list, i.e. each candidate is equal to the friendship list owner because of edge updates. If a friend U_{ff} of A is known due to a path over another friend U_f, then the path to U_f is shorter and, hence, U_f is a better friend in $Flist_U$. Since U_f is a friendship list owner in PQ_A and each friendship list can be referenced only once by a triple in PQ_A, the number of potential candidates is limited.

Furthermore, even in the worst case the i-*th* friend of A is found not later then at the i-*th* position in a friendship list of a friend as each user can only appear once in a friendship list and friends are sorted in descending order of their friendship strengths. All users at previous positions are already better friends of A.

In conclusion, we have to proceed over at most i users in the worst case to find the i-*th* friend. Each of these users themselves could also proceed only over i users to find their own i-*th* friend. After $O(i^2)$ users, we must have identified the i-*th* friend. Moreover, as shown in our experiments (see Section 9.12) on real friendship graphs of social networks, on average, the actual number of opened lists and accessed friendship list entries is much lower.

U.*resolveCycle*()

A cycle is only of concern when we want to find the *i-th* friend of a user U and $tp_U = i$. When $i < tp_U$ the *i-th* entry in $Flist_U$ is already correct and $i > tp_U$ is not possible because we can only sequentially process a friendship list and tp_U is never set to a

previously processed entry—only to the currently processed entry.

When having identified a cycle, the following tasks have to be done to find the true next best friend:

1. If the triple $pq_t = (U_f, U_{pq}, s)$ is the head in PQ_U and U_{pq} belongs to a cycle then remove pq_t from PQ_U and call $U.queueNext(pq_t)$ in order to proceed to the next friend in U_f's friendship list and add the new candidate to PQ_U. Repeat this step until we have got a candidate in the head position of PQ_U who does not belong to a cycle or there is no more candidate.

   If there is no more next friend in U_f's friendship list, discard the triple from PQ_U and repeat step 1 until we found a valid candidate or PQ_U is empty.

2. If there is no more candidate in PQ_U, the *i-th* friend is either found at $Flist[tp_U]$, or if there is no user at position tp_U in U's friendship list, the *i-th* friend is equal to the friend found at the top entry of the storage backend $DB_U[i']$. In this case, set $Flist[tp_U] = DB_U[i']$ and traverse DB_U to its *i'+1* entry.

   If there is no more friend at $DB_U[i']$, too, then there are also no more friends of U and we are done.

3. Otherwise, determine the *i-th* friend among U_{pq} and $Flist[tp_U]$ or among U_{pq} and $DB_U[i']$ if there is no entry at $Flist[tp_U]$. Set $Flist[tp_U]$ appropriately.

4. If U_{pq} is not the next best friend add pq_t back to PQ_U—she is still a candidate for a later best friend position. Otherwise, call $U.queueNext(pq_t)$ to queue U_f's next friend in PQ_U which is another candidate for the subsequent next best friend position.

As it can be seen, multiple calls to $U.queueNext(pq_t)$ can be involved.

Identifying Cycles in Friendship Graphs

Luckily, it is easy to identify a cycle; it can be done in different ways. One way is to remember for a query all users U who cause a call to the method $U.queueNext(pq_t)$ in a global data structure, e.g. a *HashSet*. When the method returns, U is removed from the data structure again. If the method is called for a user U and the next entry in the friendship list of a friend U_f shall be queued, i.e. $U.queueNext(pq_t)$ with $U_f \in pq_t = \{U_f, U_{pq}, s\}$, and that friend U_f can be found in the global data structure, we have got identified a cycle.

Obviously, we also have found a cycle as soon as a user U should be put into her own priority queue of *"next best friend"*-candidates. This could happen when U appears in a friendship list of her friend U_f. The cycle leads then from U over U_f back to U.

9.10.6 Further Improvements of the *LAP* Approach

Before the query execution for a user U can start, all necessary data structures must be appropriately loaded into main memory, including PQ_U as described in Section 9.10.1, in order to efficiently determine the user's next best friend. Initialising a priority queue with possibly many candidates means that many friendship lists have to be opened and sequentially processed until the correct candidate can be fetched and loaded into PQ_U. There is no way to avoid the processing of these lists until each candidate is put

into PQ_U because at some point in time, each candidate could become U's next best friend. Of course, to initialise a priority queue in this way is expensive, especially if there are many candidates that are located at late positions in a friendship list.

An option to speed up the initialisation of PQ_U is to defer also the loading of candidates to a point when we indeed need to identify the best candidate. For this, we do not remember in PQ_U a reference to a user anymore but the index of the corresponding entry in a friendship list where the candidate was found. Furthermore, instead of the candidate's friendship strength with respect to U, we remember the friendship strength of the owner of the friendship list where the candidate was found, and as well, the friendship strength of the candidate with respect to that list owner. In this way, we determine the actual friendship strength of a candidate with respect to U by multiplying the two remembered values.

The advantage of this modification is that we maybe do not need to open all friendship lists to identify the next best friend and her friendship strength. When the friendship strength of the friendship list owner or the multiplication of both remembered values is already too low to change the user at the head of PQ_U, there is no point in processing a list to find a candidate. Moreover, over time it is quite likely that such candidates in PQ_U exist who never will belong to the best *top-k* friends. All merge operations during a query are merging in at most the first *top-k* users from a friendship list owner in PQ_U and thus, the *top-k+1*-th user in the list will never be an actual candidate for the next best friend. However, if some later queries require to identify more than the previous *top-k* friends, our algorithm is still able to handle that case as the positions of all possible candidates in all associated lists are still included in PQ_U. Hence, friendship lists for fetching *"next best friend"*-candidates are only opened when there is a need for it.

Of course, as soon as the algorithm indeed needs to look into a friendship list in order to identify a candidate, a list will be sequentially processed to the position indicated by the remembered index of the priority queue. Yet this is only done when needed and at most once for a query and friendship list's owner in PQ_U. The latter is true because as soon as we have opened a list and processed it to a certain entry, we can keep the list open until the query has been finished.

Deferring the processing of friendship lists in PQ_U to only retrieve one candidate at a time requires some additional attention to the task of determining the next best friend of a querying user U.

Let's say, during the processing of a query from a user U, the friendship list of a user U_f in PQ_U has to be opened. When the $i-th$ entry in U_f's friendship list is U's best candidate, U_f's list is sequentially accessed until the $i-th$ entry can be retrieved. However, entries at positions prior to i can have changed since the index i was remembered in PQ_U. Therefore, when finally processing a friendship list from PQ_U, there have to be checks for update and merge conditions on each user in the list until the $i-th$ one has been identified. If some entry requires an update or merge operation, we execute that operation and chose as new candidate the user in U_f's friendship list that follows the position causing the update or merge operation. The newly chosen candidate is maybe not yet known to U because of the modifications in U_f's friendship list, hence, she is indeed a candidate for being U's next best friend.

Due to the postponed loading of all candidates in PQ_U, the identification of the proper next best candidate can again lead to several nested method calls to different lists and priority queues. However, there is no extra work involved but it is only postponed

to a point in time when it is needed to be done. Therefore, it is just an implementational challenge because of the need for additional checks of merge and update conditions and the corresponding update and merge operations that have to be done in order to finally determine the correct next candidate. When not all lists in PQ_U have to be opened, the additional effort is worthwhile because answering a query is sped up in this way.

9.11 Data Structure Extensions to *EAP* and *LAP*

In both approaches of our algorithm, there is potentially more than once a merge operation on a user with the same friend, even though it would not be necessary for a correct identification of the user's next best friend.

The following sections explain why such redundant merge operations happen and, in addition, gives suggestions for both approaches of our algorithm how to avoid them.

9.11.1 Missing Time Information

The merge condition given in the following Listing 15 can cause merge operations on users with friends although no merge operation is necessary.

```
1:  IF (i > tp_U && TS_{U_f} > 0)
2:      U.merge(U_f)
```

Listing 15: Merge condition causes redundant merge operations

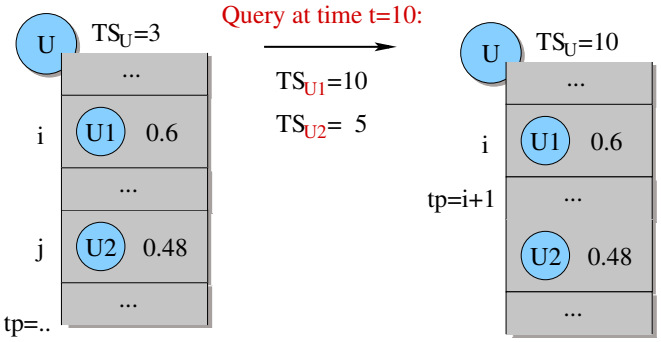
When U_f is a user found at a position later than the one indicated by tp_U in U's friendship list, and when the timestamp TS_{U_f} of U_f's own friendship list is greater than 0, then the only information we have got with respect to U_f is that there was at some point in time at least one update to U_f's friendship list. However, we do not know if U already knows about that update or not.

Imagine $TS_{U_f} > 0$ and there was already a merge operation on U with U_f during some previous query on U. When a later query on U modifies U's friendship list at a position prior to U_f, then the timestamp TS_U increases while the timestamp validity pointer tp_U decreases. Hence, if finally a subsequent query discovers U_f again in $Flist_U$, the check shown in Listing 15 will indicate a need for a merge operation on U with U_f—even if nothing has changed for U_f since the last merge operation on U with U_f.

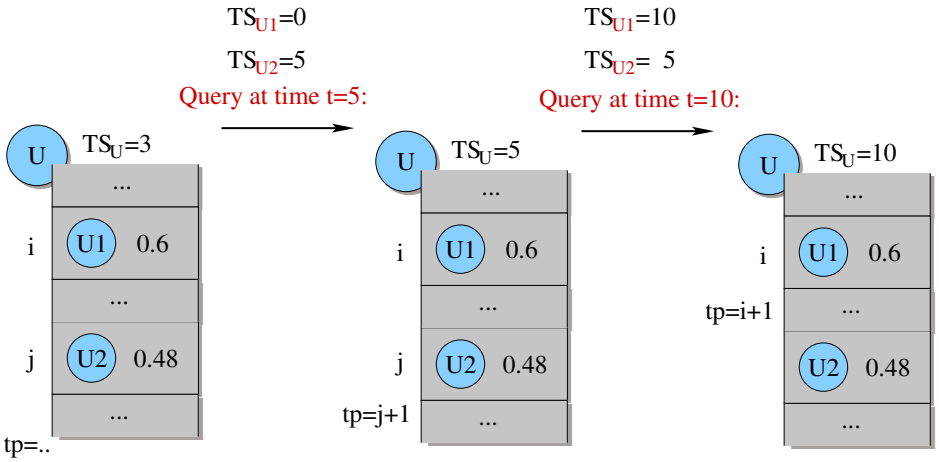
The problem exists because by decreasing the timestamp validity pointer tp_U, we lose the information about those users who have already been seen for an earlier timestamp at later positions in a friendship list. Nonetheless, we do have to decrease tp_U and increase TS_U when doing a merge operation with a friend at a position less than tp_U because otherwise we could miss updates for those users at subsequent positions who's friendship lists indeed changed at a time between the old and new value of TS_U. The following example clarifies this fact and is sketched in Figure 19.

Example: Assume there is a user U, her friendship list is timestamped with $TS_U = 3$ and there are two friends U_1 and U_2 in $Flist_U$, located at position i and j, respectively, with $i < j < tp_U$.

Furthermore, assume that at this stage, an update operation on the friend U_1 is applied for a friendship update with timestamp $TS_e = 10$ such that U_1's timestamp



(a) Initial and resulting friendship list of U after a query at time $t = 10$



(b) Initial and resulting friendship list of U after two subsequent queries at time $t = 5$ and $t = 10$

Figure 19: In (a) an example is given for a missed merge operation on U with $U2$ if tp were *not* set to $i + 1$. In (b) an example is given for a redundant merge operation on U with U_2 because of setting tp to $i + 1$.

is increased to $TS_{U1} = 10$. A subsequent query on U at time $t = 10$ increases the timestamp of U's friendship list to $TS_U = TS_{U1} = 10$, too, and decreases the timestamp validity pointer to $tp_U = i + 1$ because of a merge operation on U with U_1 as soon as U_1 is encountered in $Flist_U$. Assume additionally, there was prior to the query on U also an update operation on the friend U_2 of U for a friendship update with timestamp $TS_e = 5$. Hence, the update operation increased the timestamp of U_2's friendship list to $TS_{U2} = 5$. Moreover, no other change relevant to U exists at the time of the query on U. See Figure 19a for an illustration of the resulting friendship list of U and the timestamps of all involved parties.

Given this setup, when finally a query at time $t = 10$ passes the entry with U_2 in U's friendship list, the timestamp of $TS_{U2} = 5$ of $U2$'s friendship list is smaller then $TS_U = 10$. If the merge operation on U with U_1 didn't decrease the timestamp validity pointer tp_U to $i + 1$, our check shown in Listing 15 for a merge operation on U_2 would not indicate a need for merging U_2's friendship list with U's and, thus, the friendship update for U_2 at time 5 had been missed.

On the other hand, because of decreasing tp_U to $i+1$ at time $t = 10$, our algorithm

initiates under similar circumstances a redundant merge operation on U. To see this, let's again assume the same setup as described for our previous example. The initial friendship list of U is depicted in Figure 19b on the left hand side.

In contrast to the previous example, we assume a query on U at time 5 previous to the friendship update for U_1 but after the update operation on U_2. The query increases the timestamp of U's friendship list to $TS_U = 5$ and decreases the timestamp validity pointer tp_U to $tp_U = j + 1$ as soon as U_2 is passed in $Flist_U$ due to a merge operation on U with U_2. Since at this time there has not yet been a friendship update for U_1, nothing has changed in U_1's friendship list and while processing the query, U_1's entry in $Flist_U$ is correctly passed without indicating an update or merge operation. At this point, all relevant information from U_1's and U_2's friendship list up to time 5 are known to U and already available in her friendship list as depicted in the middle of Figure 19b.

Finally, when at time 10 a query on U is issued after the update operation on U_1 (which caused U_1's timestamp to be increased to $TS_{U_1} = 10 > TS_U$), our algorithm correctly indicates a need for a merge operation when passing position i of U_1 in $Flist_U$. Consequently, the timestamp of U's friendship list is set to $TS_U = TS_{U1} = 10$ and the timestamp validity pointer is decreased to $tp_U = i + 1$. The resulting friendship list of U up to U_2's position j and the timestamps of all involved parties TS_U, TS_{U1} and TS_{U2} are eventually equal as in the previous example and are shown in Figure 19b on the right hand side. The only difference to the previous example is that the merge operation on U with U_2 due to the friendship update for U_2 at time 5 has been already applied.

In this scenario, even if there's no more change in any friendship relation, the next query processing $Flist_U$ up to the entry with U_2 initiates an additional merge operation on U with U_2 due to the check given in Listing 15. Hence, a redundant merge operation on U is applied.

The reason for these kind of redundant merge operations is that there is no information about when the last merge operation with a friend has taken place. To avoid the overhead of redundant merge operations, we need to determine if a user's friendship list has changed since a previous merge operation.

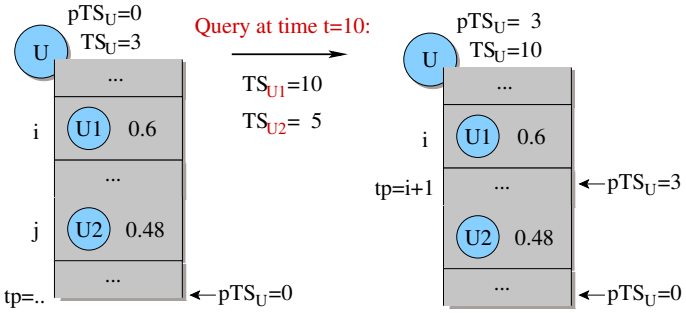
Timestamp Extensions

As an extension to the *EAP* and *LAP* approach of our algorithm, we add timestamps as additional bookkeeping information to each user's friendship list. These timestamps allow us to determine for a user U if all friendship updates for a friend U_f have already be seen or if new updates have been applied to U_f's friendship list since she was last encountered in $Flist_U$.

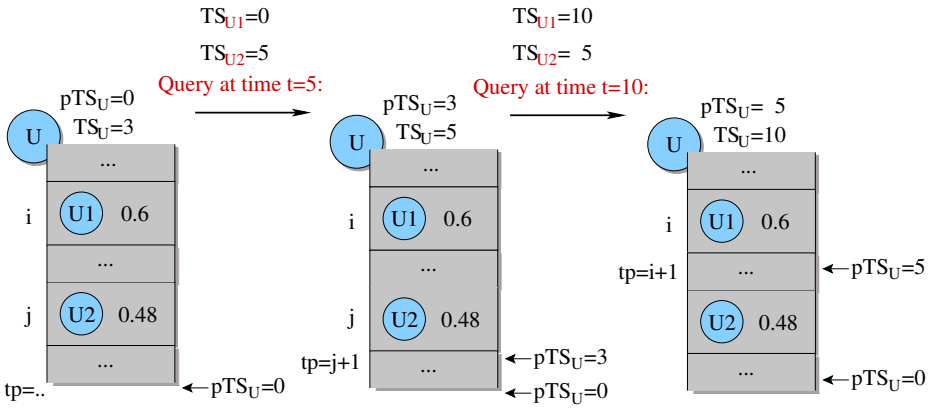
Therefore, for each user, we need a second, single timestamp pTS_U which is associated (globally) to her friendship list in order to remember a *previous* value of TS_U. The timestamp pTS_U is initially set to 0 and changed to the value of TS_U when the following two constraints apply:

1. Prior to a query on U, the timestamp TS_U is greater than 0.
2. The timestamp TS_U changes during a query for the first time due to a merge operation with a friend of U.

At the end of the query execution, the current value of pTS_U is additionally associated (locally) to a list entry. This works as follows:



(a) Initial and resulting friendship list of U with additional timestamp pTS_U after a query at time $t = 10$



(b) Initial and resulting friendship list of U with additional timestamp pTS_U after two subsequent queries at time $t = 5$ and $t = 10$

Figure 20: (a) shows that *no* merge operation on U with $U2$ is missed despite pTS_U. (b) shows that *no* redundant merge operation on U with U_2 is applied because of pTS_U.

The value of pTS_U is stored in $Flist_U$ next to the entry found at the position indicated by tp_U if and only if there is no such timestamp yet. A timestamp is removed from an entry in U's friendship list again as soon as the query processing encounters it in $Flist_U$. The removed timestamp replaces then the one previously stored in pTS_U.

In this way, all users found in U's friendship list at positions later than the one indicated by tp_U are either new friends with a friendship list with a timestamp newer than TS_U, or are already known friends of U that have been seen last at time pTS_U.

For clarification, the example previously used and depicted in Figure 19 is extended in Figure 20 to comply with the suggested timestamp extension.

Example: On the left hand side in Figure 20a, the friendship list of user U is shown before a query in her context is submitted at time $t = 10$. The friendship list's initial timestamps are $TS_U = 3$ and $pTS_U = 0$. The friendship lists of U's friends U_1 and U_2 are timestamped at query time with $TS_{U1} = 10$ and $TS_{U2} = 5$, respectively. U_1 is located at position i and U_2 at position j in U's friendship list with $i < j$. For simplicity, we assume that the last query on U proceeded deeper in $Flist_U$ than any other query before. Therefore, at the position $tp_U > j$, the entry is marked with the

initial value 0 of the timestamp pTS_U.

In the beginning, when the query on U at time $t = 10$ is issued, the value of pTS_U is unchanged. However, when U_1 at position i is encountered in $Flist_U$ during the query processing, a merge operation on U with U_1 is applied because of the higher timestamp of U_1's friendship list. As with the previous example, this causes the timestamp of U's friendship list to be changed to $TS_U = TS_{U1} = 10$ and the timestamp validity pointer to be set to $tp_U = i+1$, and therefore, in pTS_U the previous value 3 of TS_U is saved. If the query stopped at this point, the timestamp pTS_U would be stored next to the entry pointed to by $tp_U = i + 1$. The resulting data structures are sketched in Figure 20a on the right hand side.

The effect now is, that for the current query and all subsequent queries on U (as long as there is no other friendship update), all entries in U's friendship list starting from position $i + 1$ to the entry marked with the timestamp 0, are known to be correct at time $pTS_U = 3$. Nevertheless, since the timestamp of U_2's friendship list $TS_{U2} = 5 > 3$, we still know that a merge operation on U with U_2 has to be applied.

In the second scenario, given in Figure 20b, we now can see that we avoid redundant merge operations by taking the pTS_U value into account. Since there is prior to time $t = 10$ a query on U at time $t' = 5$, the timestamp of U's friendship list is set to $TS_U = 5$ by the merge operation on U with U_2. Hence, during the query at time 10, the timestamp pTS_U is set to 5 when the merge operation on U with friend U_1 at position i is applied. Consequently, all entries starting from entry $tp_U = i + 1$ to the entry marked with the initial timestamp 0 of pTS_U have already been seen at the time of the current value of $pTS_U = 5$.

When finally U_2 is encountered in $Flist_U$, it can be justified now from the value of $pTS_U = 5$ that no more need for a merge operation on U with U_2 is necessary since the timestamp $TS_{U2} = 5$ is not newer than the current value of pTS_U. Hence, no change to U_2's friendship list happened since U_2 was seen for the last time and the redundant merge operation can be avoided.

As already said, the value of pTS_U is adjusted whenever a timestamp is found next to an entry of U's friendship list. Obeying these rules ensures that the timestamp for an entry at the furthest position away which ever was discovered during any query execution is 0 and all remembered timestamps at entries in $Flist_U$ are always in descending order of their values.

Finally, by extending the bookkeeping data structures for each user U with an additional timestamp pTS_U for their friendship list, we can modify the merge condition shown in Listing 15 as follows: When during the processing of a query an entry with a friend U_f in $Flist_U$ is passed and if U_f is found at the position indicated by the timestamp validity pointer tp_U, then there is only a need for a merge operation when the timestamp of U_f's friendship list is greater than pTS_U. The pseudocode is given in the following Listing 16.

```
1:    IF (i > tp_U && TS_{U_f} > pTS_U)
2:        U.merge(U_f)
```

Listing 16: Merge condition with timestamp extension pTS_U.

If the timestamp TS_{U_f} is less than or equal to pTS_U, we have seen that entry already

at the time indicated by pTS_U and all corresponding merges have already been done. Hence, we do not need to do another merge operation.

Advantage The advantage of extending friendship lists with timestamps is obvious: We can avoid doing unnecessary merge operations. Although a merge operation is cheaper for the *LAP* approach of our algorithm than with *EAP*, especially when there is nothing that has to be merged for the first *top-k* elements, avoiding redundant merge operations is desirable. Moreover, the *EAP* approach could benefit a lot from this extension because even to find out that nothing has to be merged, the entire friendship lists of both involved users have to be processed.

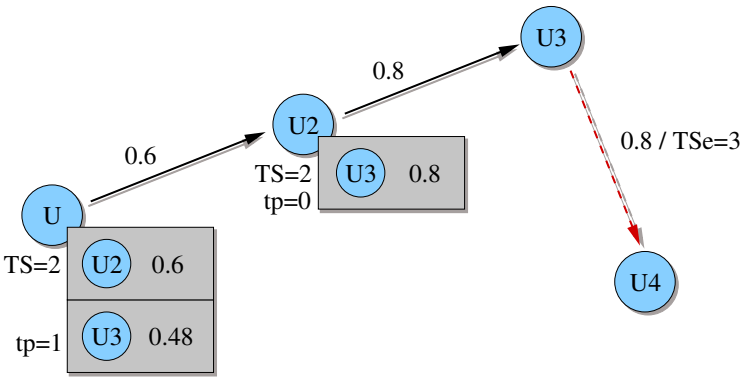
Disadvantage On the other hand, there is also a disadvantage involved that cannot be disregarded. The space requirements to store a timestamp at potentially every entry in each friendship list can be huge. Depending on the storage backend, it could mean, that for each entry in all friendship lists additional space has to be reserved such that the additional timestamp value can be placed there. A friendship list does not consist of entries of pairs anymore—the user ID and the value of the friendship strength—but of triples. In each entry a placeholder is needed for potentially additional values of the timestamps pTS_U for each user U.

9.11.2 Missing Path Information

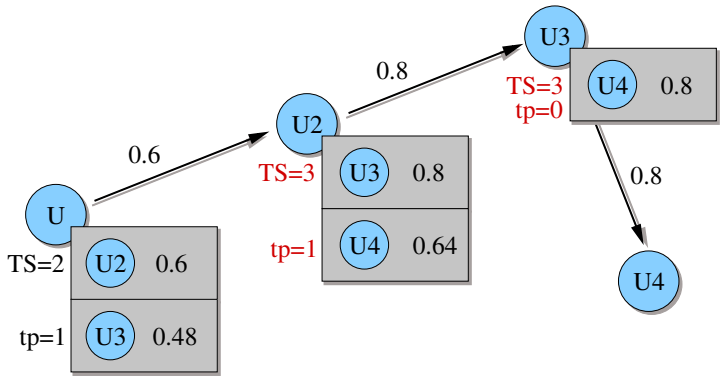
Another kind of redundant merge operation occurs because of missing path information for shortest distance values. With just the information available in a user's friendship list, it is not possible to know if the shortest path to a friend is a prefix of the shortest path to another friend.

When doing a merge operation with a friend who is found on a prefix of a path to some later friend and that merge operation includes already friendship updates from both friends, then there is no way to know about that and potentially a merge operation can happen with the later friend again even if there is no new information available.

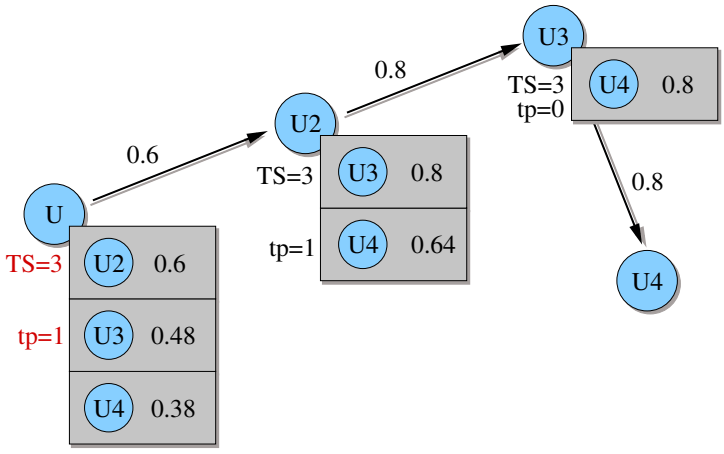
Example: Let U, U_2, and U_3 be users with U_2 is U's best friend, U_3 is U_2's best friend and no other friendship connections exist at time $t = 2$. Moreover, all user's friendship lists are up-to-date at that time. At time $t' = 3$, assume there is a friendship update $(U_3 \rightarrow U_4)$ for U_3 with respect to a user U_4. Figure 21a visualises this initial setup. Assume that first a query for U_2's *top-2* friends is issued. In this case, our algorithm first updates U_3's friendship list and finally propagates the information about the new friend U_4 also into U_2's friendship list. The timestamp for both users' friendship list is equal to $t' = 3$ afterwards. Since, neither a query in the context of U is issued at this point nor is U part of the shortest part from U_2 to U_4, the friendship list of U does not change. Figure 21b shows the updated setting for all users after the retrieval of U_2's two best friends. However, a subsequent query on U for her *top-3* friends first merges the changes from U_2's into U's friendship list and sets $tp_U = 1$ and TS_U to time $t' = 3$. At this point, U's friendship list is already up-to-date because the new information about U_4 has been found by the merge operation with U_2. Hence, the first entry in U's friendship list contains U_2, the second entry contains U_3 and the third entry contains U_4 and the friendship strengths of all friends are correct. Figure 21c depicts the current state of our example.



(a) Initial setup



(b) Intermediate step after *top-2* query on U_2



(c) Redundant merge operation on U caused when identifying U's 3rd best friend.

Figure 21: In (a) an initial example graph is given. A redundant merge operation on U occurs when first a *top-2* query on U_2 occurs, depicted in (b), and then a *top-3* query on U follows. Figure (c) shows the state of the graph right before the redundant merge operation.

In the final step of U's *top-3* query, for identifying U's third best friend, the information is lost that U_4 was already found in U_3's friendship list and propagated along the shortest path over U_2 back to U by the merge operations on U with U_2. Therefore, our algorithm indicates a need for a merge operation on U with U_3 when processing the third entry in U's friendship list due to the merge condition drafted in Listing 15. Even when using the modified version introduced in Section 9.11.1 and given in Listing 16, a need for a merge operation is still indicated because U_3 has been previously seen at time 2 as shown in Figure 21b, and hence, the introduced previous timestamp pTS_U is set to $pTS_U = 2 < TS_U = 3$. In summary, at the current stage of our example, the following condition indicates the redundant merge operation:

```
1:    IF (i > tp_U && TS_U3 > pTS_U)
2:        U.merge(U_3)
```

where $i = 2$, $tp_U = 1$ and the timestamps $TS_{U_3} = 3$ and $pTS_U = 2$.

Path Information Extension

To avoid this kind of redundant merge operation, we have to enrich each entry of a user's friendship list with another user identifier. There are two options:

1. We remember the owner of the friendship list where a shortest path distance value originates from:

   For each merge operation on U with a friend U_m, and each new or updated friend U_f who is merged into U's friendship list, we add to the associated entry in $Flist_U$ also the owner U_m of the friendship list where U_f was found.

2. We remember the first friend on a shortest path for the corresponding shortest distance value in a user's friendship list:

   Same as with option 1, but if U_m was previously found by a merge operation with a user U'_m, we remember U'_m also for the modified entry with U_f instead of U_m. In this way, the remembered user is always the first user different from U on the shortest path from U to U_f and, hence, a direct successor of U.

With both options, we call the user U_m the *intermediate user* on the path from U to U_f.

A friend U_f of U, who is not found by a merge operation, must be a direct successor of U, i.e. the shortest path from U to U_f is a direct edge. For those users, U_f is remembered twice at the corresponding entry in $Flist_U$, such that $U_m = U_f$.

The extended definition of $Flist_U$ and its entries is then the following:

Definition 9.26 (Extended Friendship List $Flist_U$)**.** *For each user U, $Flist_U$ is the inverted list of all transitive friends of U sorted in descending order of their friendship strengths. $Flist_U$ is called the extended friendship list of U.*

An entry in $Flist_U$ is a triple $fl_e = (U_f, s, U_m)$ with $s = s_t(U, U_f)$ is the friendship strength of U_f with respect to U and U_m is either

- ***Option 1:** the user with whom the merge operation on U was applied to find U_f*

  or

- ***Option 2:** the direct successor of U on the shortest path from U to U_f.* ∎

With these ingredients, we can again modify the check for our merge operation in order to avoid redundant merge operations.

Let be $(U_f, s, U_m) = Flist_U[tp_U]$ the entry in U's friendship list at position tp_U and assume, that a query on U is going to identify the $(tp_U + 1)$-*th* friend of U. If the timestamp TS_{U_f} of U_f's friendship list is greater than 0 (or greater than pTS_U when using the modification introduced in Section 9.11.1), we additionally check if U_f is located on a path that is already up-to-date. If so, there is no need for another merge operation.

Consider the following cases:

$TS_{U_m} < TS_{U_f}$: When the timestamp of U_m's friendship list is older than that of U_f's then clearly, U_f has been updated with information that U_m and also U cannot know yet. Hence, we need to do a merge operation on U with U_f.

$TS_{U_m} = TS_{U_f}$: When the timestamp of U_m's friendship list is equal to that of U_f and $U_m \neq U_f$, then we do not need to apply a merge operation on U with U_f. The reason is that timestamps of friendship updates are unique and the only case when the timestamps of two lists become equal is when those two lists are merged. The timestamp of the friendship list on which the merge operation is applied is set by definition to the maximum timestamp of both lists.

Therefore, TS_{U_m} can only be equal to TS_{U_f} when the timestamp of U_f's friendship list is newer than that of U_m's before a merge operation $U_m.merge(U_f)$ is applied. However, that merge operation must already have propagated into U_m's friendship list all new information found on the path over U_f. Furthermore, since U_m is a better friend of U than U_f, she is discovered prior to U_f during the query processing and then all relevant information from U_m is inserted into U's friendship list by an appropriate merge operation on U with U_m. Hence, no other $U.merge(U_m)$ is necessary anymore.

When $TS_{U_m} = TS_{U_f}$ and $U_m = U_f$, then U_f is a direct successor of U and no intermediate user on the path from U to U_f exists. Hence, no merge operation on that path has been applied so far and the merge operation still needs to be done.

$TS_{U_m} > TS_{U_f}$: This final case is slightly more difficult to check because we cannot know if changes to U_f have already been merged into U_m's friendship list without (roughly) knowing the position of U_f in $Flist_{U_m}$. When the position of U_f in U_m's friendship list is prior to tp_{U_m}, then we know by definition of tp_{U_m} and the correctness of our basic algorithm that U_f's friendship list is also up-to-date with respect to U_m's timestamp. Hence, we only need to apply a merge operation on U with U_f if $pos_{U_m}(U_f) \geq tp_{U_m}$.

In summary, we can further adjust our check for a merge operation to avoid redundant merge operations due to missing path information in the way, shown in Listing 17.

```
1:    IF (i > tp_U && TS_{U_f} > pTS_U)
2:      IF ( U_m = U_f || TS_{U_m} < TS_{U_f}
3:           || (TS_{U_m} > TS_{U_f} && tp_{U_m} ≤ pos_{U_m}(U_f)) )
4:        U.merge(U_f)
```

Listing 17: Merge condition with path information

Example: With reference to our example given above and sketched in Figure 21, the redundant merge operation on U with U_3 can now be avoided. To see this, we rewrite the merge condition to match the situation depicted in Figure 21c. User U_3 in U's friendship list has been found by a merge operation on U with U_2. Therefore, the intermediate user (U_m) for U_3 is U_2 (with both options 1 or 2) and the final check for a merge condition on U with U_3 looks in this state as follows:

```
1:    IF (i > tp_U && TS_{U_3} > pTS_U)
2:      IF ( U_2 = U_3 || TS_{U_2} < TS_{U_3}
3:           || (TS_{U_2} > TS_{U_3} && tp_{U_2} ≤ pos_{U_2}(U_3)) )
4:        U.merge(U_3)
```

The corresponding data structures have got the following values: $i = 2$, $TS_U = 3$, and $TS_{U_2} = 3$. Since $pos_{U_2}(U_3) = 0 < tp_{U_2} = 1$, there is no redundant merge operation on U.

Of course, in order to apply this modification, it is crucial for efficiency reasons to find out easily the position of U_f in U_m's friendship list without sequentially processing the list to U_f's position.

Since a merge operation between U and U_m has to touch the first tp_{U_m} entries in U_m's friendship list anyway, the idea is, to remember the corresponding friends in main memory during a query for each user U_m when a merge operation on U with U_m is applied. In this way, we can quickly check if the position of U_f is prior to tp_{U_m} in a friendship list for all intermediate friends U_m of U. If no entry for U_m has been remembered at all or if U_f is not among the first tp_{U_m} best friends of U_m, there was either no merge operation between U and U_m or no merge operation between U_m and U_f. Therefore, our check indicates the need for a merge operation between U and U_f and otherwise, avoids redundant merge operations.

As soon as the *top-k* friends for U has been identified, the memory occupied to remember the first tp_{U_m} friends of each user U_m that caused a merge operation on U can be released again. In the worst case, that are the first *top-k* friends for U's first *top-k* entries.

Differences of Option 1 and 2

Choosing option 1 or 2 for dealing with missing path information about shortest path distance values leads to different effects. If the friendship list of a direct successor of a querying user is rarely updated but there are frequent changes at subsequent friends, option 2 has got only a little effect.

However, option 1 is more difficult to maintain because we may want to change the identifier U_m to some U'_m for each entry with a user U_f in U's friendship list whenever there is a merge operation on U with U'_m and that user is part of a shortest path to U_f. We may want to do that even if the friendship strength of U_f does not change, just to reflect the most frequent updated user U_m on a shortest path to U_f in U's friendship list. In addition, such changes in friendship lists are expensive because at some point in time the modified entries of a friendship list have to be written back to the storage backend and, thus, they cause more I/O costs than option 2.

Furthermore, with both options, we cannot guarantee to avoid all possible redundant merge operations of the kind that occur because of missing path information. We only check for one user of a shortest path to some friend at a potentially far distance if a merge operation for the later friend on the same path is indeed needed. However, the greater the distance of a friend the more and better friends a querying user has already identified and, thus, the less redundant merge operations can be applied until a query finishes.

In order to avoid all redundant merge operations, we either have to traverse each shortest path and perform such additional checks not only at one intermediate user but at all users on a shortest path. With option 2, this could be done by traversing the path from one direct successor to the next which is expensive in terms of execution time and I/O costs, or with option 1, by storing not only one user, but all the users on a shortest path who caused a merge operation which is expensive in terms of storage – potentially all users of a shortest path have to be remembered at each entry in a friendship list – and also in terms of I/O costs because these information have to be properly maintained.

We assess both solutions as not viable. The I/O cost for storing or the effort for maintaining complete shortest path information in friendship list is way too high. However, as shown in our experiments (see Section 9.12), the performance of our algorithm is decent even in the presence of redundant merge operations.

9.12 Experiments

In this section, we introduce the setup for our runtime experiments that show the viability of our algorithm.

To evaluate our algorithm, we used two different datasets crawled from the two social network sites *LibraryThing.com* and *Twitter.com*.

1. For our experiments, we extended the dataset of *LibraryThing.com* used in Part A, Section 6.4 for evaluating our CONTEXTMERGE algorithm by yet another crawl of the *LibraryThing.com* social tagging network site. In this way, we obtained an extended dataset with $21,345$ users, $10,062,267$ books and in total $20,306,633$ tags. The connected friendship graph based on this extended dataset consists of $7,793$ distinct users with $28,853$ friendship connections. Though, only $5,813$ uses established outgoing friendship connections.

   The weight of a friendship edge corresponds to a measure for the overlap in tag usage between two directly connected friends. The weight was obtained by computing the dice coefficient of the tag sets used by two users, i.e.

   $$O(U_1, U_2) = \frac{2 \cdot |tagset(U_1 \wedge U_2)|}{|tagset(U_1)| + |tagset(U_2)|}$$

   Finally, the weight was normalised between $(0, 1)$ over all users.

2. In addition, we obtained a snapshot of an anonymised topology of the social network *Twitter.com* which was created in August 2009 by the authors of [89]. The dataset contains $41,652,164$ user accounts and $1,468,359,999$ social (follow) links. Edge weights were chosen uniformly at randomly from the interval $(0, 1)$.

We used for our experiments an Oracle 10 database as a storage backend on a Windows 2003 Server with 32GB of main memory and 2 dual-core AMD Opteron 2218 processors. The CPUs are running at a clock rate of around 2.4GHz. Although

the performance of the mentioned server is decent, its characteristics is far away from cutting-edge technology. However, for our experiments the performance is sufficient since only sequential accesses on the database are necessary. Friendship lists are represented by a table of the from `(user,friend,friendship strength)` with an index on all entries in descending order of the friendship strength.

All experiments were executed on a Linux desktop PC with 4GB main memory and a 4-core Intel i5 650 CPU running at a clock rate of 3.20GHz and were implemented in Java. In this setup, the main bottleneck, however, is the I/O access established via a remote TCP/IP connection to the database from the desktop PC. Hence, the I/O access dominates the overall runtime requirements and the CPU's speed of the desktop PC is not crucial. This also has been confirmed by our single threaded experiments which barely uses the computational power of only a single core.

For these reasons, we measured in addition to wall clock times also the number of performed merge operations, opened friendship lists and retrieved list entries to give a more complete picture about the actual runtime requirements of our algorithm.

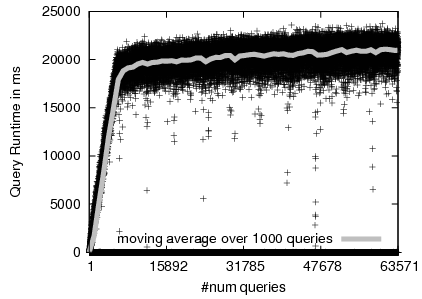
To evaluate our algorithm, we implemented both *EAP* and *LAP* as presented in the previous Section 9.9 and 9.10 including the introduced improvements for reducing the number of performed accesses to friendship lists caused by redundant merge operations. However, we disregarded the suggested extensions that add additional information to friendship lists, i.e. missing timestamp (see Section 9.11.1) and path information (see Section 9.11.2), in order to keep the inverted index list as simple as possible and the size of the storage backend low.

A series of experiments were done on the smaller friendship graph from *LibraryThing.com* to show the correctness of our implementations. For this, we precomputed all friendship lists by using an in-memory all pairs shortest path algorithm, executed on a Linux server with a decent amount of main memory and stored the results in our Oracle 10 database. Afterwards, we removed about $\frac{1}{3}$ of all edges from users with more than 2 friends and one edge from users with only 2 friends by randomly choosing the edge to remove. Additionally, for about half of all users with only 1 friend, we also removed the corresponding single outgoing edge. On the remaining subgraph, we again precomputed the inverted index lists and stored the lists of transitive friends in our database.

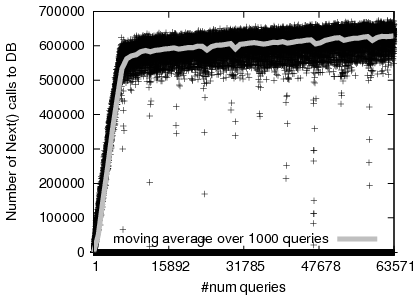
Finally, we ran our algorithm on the diminished graph of *LibraryThing.com* in the context of each user by querying for all or a certain fraction of all transitive friends while inserting the previously removed edges in the subgraph over time again. The order of querying users were chosen randomly. The resulting friendship lists retrieved by our algorithm were eventually compared to the precomputed one of the entire graph. We repeated these experiments several times with the *EAP* and *LAP* approach and at varying edge insertion rates. In all queries, the retrieved order of friends and the friendship strengths computed for each of them coincided for *EAP* and *LAP* with the values precomputed over the entire graph. Although, in general, this does not prove the correctness of our implementation, it gives a strong indication that it is at least correct for the *LibraryThing.com* dataset. The same experiments could not be done on the *Twitter.com* dataset due to its huge size.

Notwithstanding, a formal proof of the correctness of our basic algorithm is given in Chapter 10.

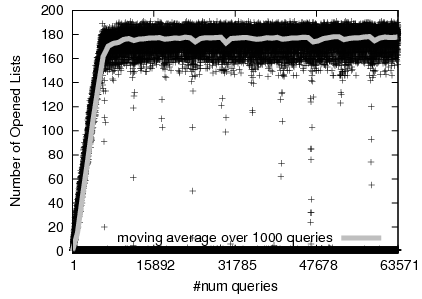
Next, we discuss how well our algorithm performs and how much the presented approaches differ with respect to their runtimes.



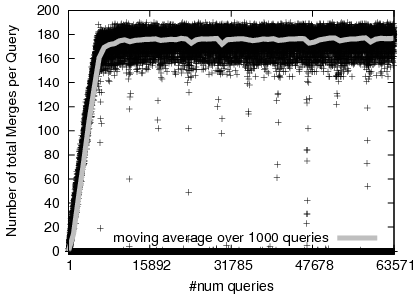
(a) RT, full merge, *top-k*=200, upd-ratio 1/100



(b) #SA, full merge, *top-k*=200, upd-ratio 1/100



(c) #OL, full merge, *top-k*=200, upd-ratio 1/100



(d) #MRG, full merge, *top-k*=200, upd-ratio 1/100

Figure 22: Experiments for *EAP* on *LibraryThing.com*, randomly selected querying users, 1 update per 100 queries, top-k=200, fully merged friendship lists for (a), (b), (c) and (d)

For the experimental evaluation, we measured the number of performed merge operations (#MRG), opened lists (#OL), the total number of list entries retrieved by sequential accesses (#SA) to the database and the wall clock runtimes (RT) for each query. The results are discussed in the following sections and reported by charts where the x-axis denotes the number of the query, and the y-axis denotes the monitored performance result for this query. Additionally, the charts contain the moving average over intervals of 1000 queries.

9.12.1 Results on *LibraryThing.com*

We performed a series on experiments on the dataset crawled from *LibraryThing.com*. About a third of all edges were removed from the graph and were re-inserted again during our experiments. The re-insertion of edges simulated the creation of new edges over time. Next, we randomly chose a user among the ones with at least one outgoing edge in the full dataset to query for her *top-k* best friends and measured wall clock times (RT) and counted the various number of I/O operations (i.e. #SA, #MRG and #OL) performed until the *top-k* results could be correctly found.

***EAP* Approach** With *EAP*, we chose *top-k* $= 200$ and queries were submitted sequentially by a randomly chosen user from *LibraryThing.com* who had at least one outgoing edge to a friend.

Figure 22a shows the wall-clock runtimes and Figure 22b the number of sequential accesses to the database measured by experiments with our algorithm implementing *EAP* which merges complete friendship lists. In total 63,572 queries were executed and every 100 queries one update was applied to the graph. With *EAP*, the runtimes are not affected that much from the number of merge operations (#MRG), or the number of opened friendship lists (#OL) but mainly from the total number of read operations on list entries which correspond to sequential accesses to the database (#SA). In the worst case, when each of the *top-k* friends are found in different lists by merge operations, 200 lists are opened with *top-k*=200. Indeed, it can be observed by the experiments, as depicted in Figure 22c, that for the *EAP* approach on *LibraryThing.com* on average around 180 lists have to be opened. This can be taken as evidence that the graph is tightly connected and a single update already influences many shortest path distances. Moreover, with *EAP*, #MRG=#OL, since friendship lists are opened only when merge operations are applied and then all entries are read such that complete lists are merged. For completeness, the Figures 22c and 22d visualise this property of our algorithm.

Finally, from the Figures 22a and 22b it easily can be seen that the runtimes are I/O driven and increase rapidly with the number of list entries fetched from the database. It takes over 20s to sequentially read around 60k entries from the involved friendship lists. The number of sequential reads is so large since the lists of transitive neighbours involved in a merge operation are long and, by the characteristics of the static algorithm, have to be read completely during a merge operation.

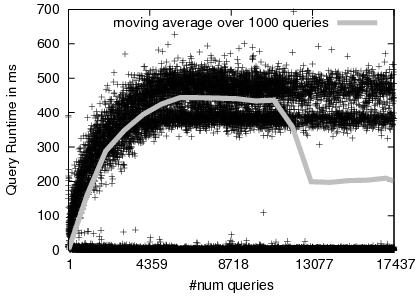
fixed-size EAP*: Restricting *EAP* to a fixed number of *top-k When always the same constant number of *top-k* friends are queried for users in some system, we can modify our *EAP* approach by restricting the number of list entries considered in merge operations to the value of *top-k*, denoting this restricted *EAP* approach as *fixed-size EAP* (see Section 9.9.6). *fixed-size EAP* makes sense since merging of friendship list with a huge number of transitive friends is expensive and merge operations only on prefixes of friendship lists still allow us to retrieve the correct *top-k* results as long as no query requires to fetch more than this fixed number of entries from any friendship list.

In the following experiment, we fixed the lists' lengths in merge operations to the number of retrieved *top-k* friends with *top-k*=200 and for all queries, always 200 friends were retrieved, if available.

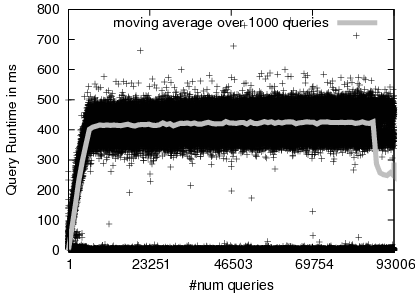
The results for RT are shown in the Figures 23a-23c and for #SA in the Figures 23e-23g. The update ratio varies in the corresponding experiments from 1 update per 1, 10 and 100 queries and, depending on the update ratio, $17,438$, $93,007$ or $218,910$ queries, respectively, were submitted in the context of a randomly chosen user.

It can be observed that the different update ratios have got only little effect on RT and #SA. For a query with an update ratio of 1/1, the maximum value for RT and #SA is $693ms$ or 2925 sequential accesses to the DB, respectively, and with an update ratio of 1/100, the maximum values are slightly over $723ms$ and 2130 sequential reads from friendship lists. The reason that RA is slightly higher in the latter case although #SA is lower is most likely caused by network latency issues during the experiments.

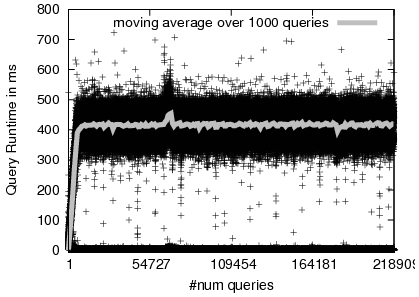
The charts additionally visualise the moving average over those (worst-case) queries that indeed received the total requested *top-k* friends within intervals of size 1000. Hence, the average RT and #SA is fairly similar for each update ratio, too. The drop in RT and #SA in the Figures 23a-23b and 23e-23f after 8,456 or 84,560 queries, respectively, happens because all available edge updates have been applied to the graph at that



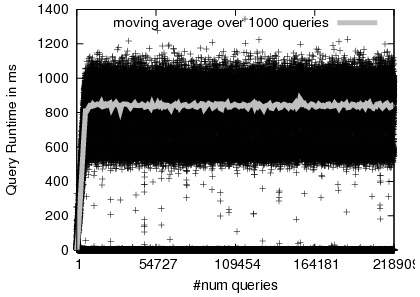
(a) RT, *max-k*=*top-k*, upd-ratio 1/1



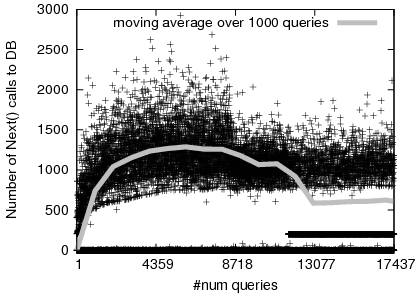
(b) RT, *max-k*=*top-k*, upd-ratio 1/10



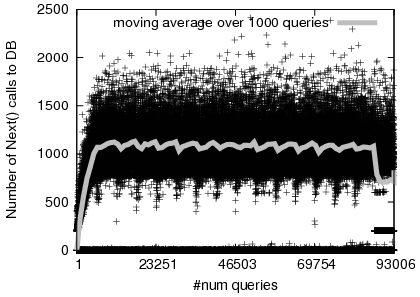
(c) RT, *max-k*=*top-k*, upd-ratio 1/100



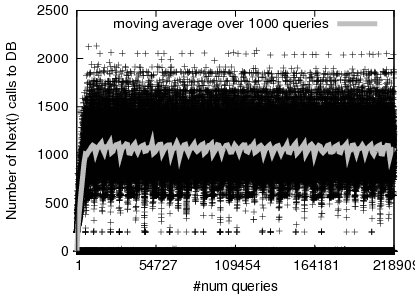
(d) RT, *max-k*=500, upd-ratio 1/100



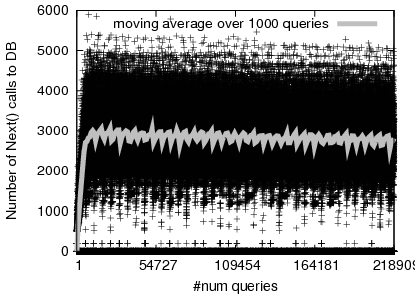
(e) #SA, *max-k*=*top-k*, upd-ratio 1/1



(f) SA, *max-k*=*top-k*, upd-ratio 1/10



(g) #SA, *max-k*=*top-k*, upd-ratio 1/100



(h) #SA, *max-k*=500, upd-ratio 1/100

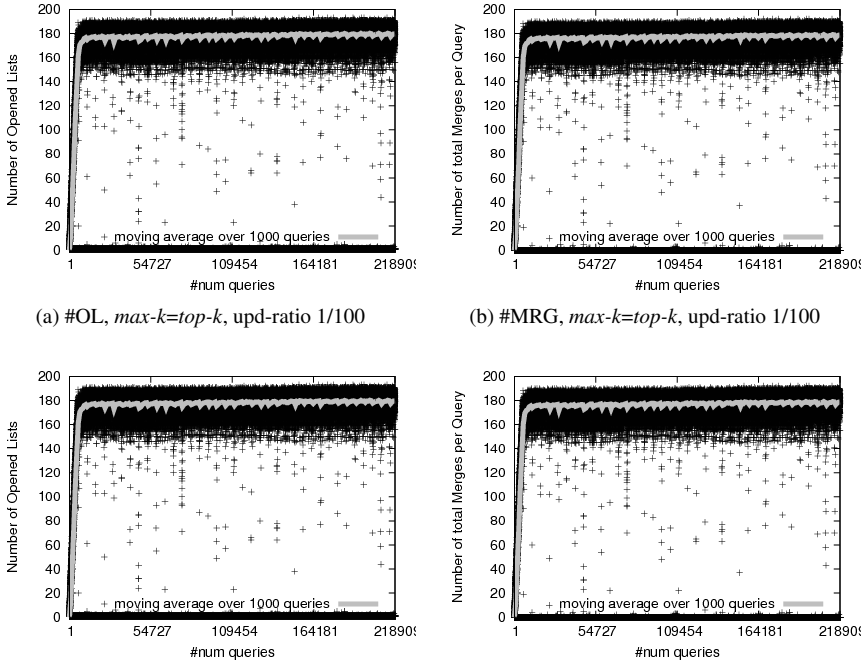
Figure 23: Runtime and #SA measurements of *EAP* on *LibraryThing.com*, randomly selected querying users, 1 update per varying number of queries: 1, 10 or 100, top-k=200, only top-k prefixes are merged. for (a), (b), (c), and (e), (f), (g). In (d) and (h) upd-ratio 1/100 and *top-k*=200 but merge operations on prefixes of size *max-k*=500.

time and, finally, the performance further improves when there are no more updates for a large number of queries. These experiments demonstrate that *EAP*, when restricting queries and merge operations to *top-k* list entries, can handle different update loads and benefits quickly from long periods without updates.

fixed-size EAP*: Restricting *EAP* to a maximum number *max-k To allow querying for a varied number of *top-k* friends, yet limiting the number of list entries involved in a merge operation, we introduces a constant *max-k* which specifies the maximum value allowed for *top-k*. In this way, a merge operation with *fixed-size EAP* merges always list prefixes of fixed-size *max-k* and the requested number of friends issued by a query can vary up to *max-k* while the correctness of the retrieved results still can be ensured by our algorithm.

The Figures 23d and 23h show the results for RT and #SA of our experiments with *max-k*=500 and 1 update per 100 queries. In order to compare the evaluation with the results shown in the Figures23c and 23g, the value for *top-k* has been kept equal to 200 for each query. As expected, RT and #SA increase compared to the modification which restricts queries and merge operations to a constant value of *top-k* entries because larger prefixes of lists need to be merged. However, the average #RA/#SA keeps fairly stable at around $850ms$ or 2800 sequential accesses to the DB, respectively.

For completeness, in Figure 24 the number of opened lists, #OL, and number of merge operations, #MRG, are shown for both versions of our *fixed-size EAP* approach.



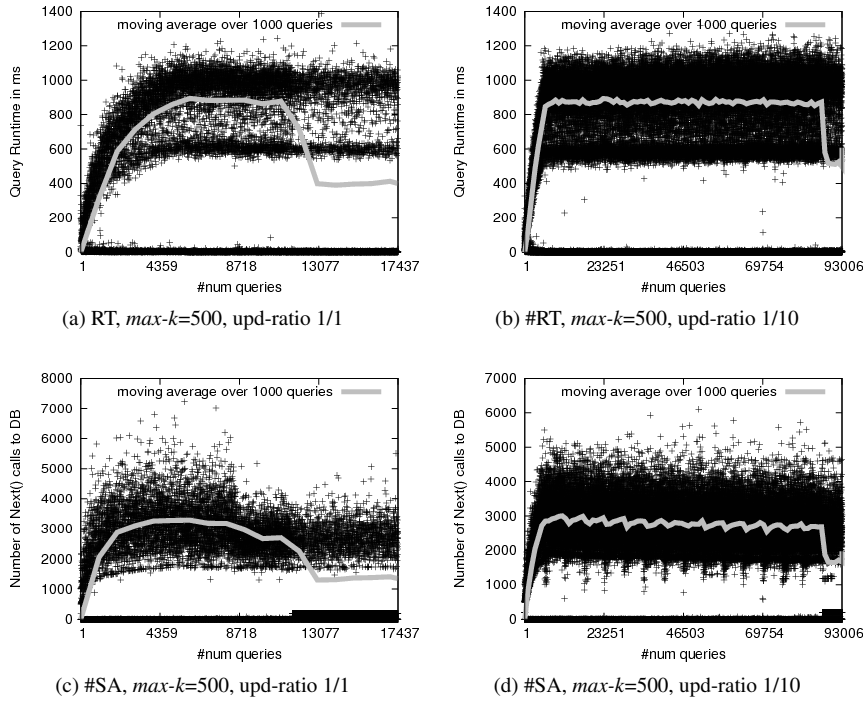
(a) #OL, *max-k*=*top-k*, upd-ratio 1/100

(b) #MRG, *max-k*=*top-k*, upd-ratio 1/100

(c) #OL, *max-k*=500, upd-ratio 1/100

(d) #MRG, *max-k*=500, upd-ratio 1/100

Figure 24: Experiments for *EAP* on *LibraryThing.com*, randomly selected querying users, 100 queries per update, top-k=200, merges only of top-k prefixes in (a), (b) of size 200 and in (c), (d) of size 500



(a) RT, *max-k*=500, upd-ratio 1/1

(b) #RT, *max-k*=500, upd-ratio 1/10

(c) #SA, *max-k*=500, upd-ratio 1/1

(d) #SA, *max-k*=500, upd-ratio 1/10

Figure 25: Experiments for *EAP* on *LibraryThing.com*, randomly selected querying users, *top-k*=200, and 1 update per 1 query in (a), (c), and 1 update per 10 queries in (b) and (d) with merges of size *max-k*=500.

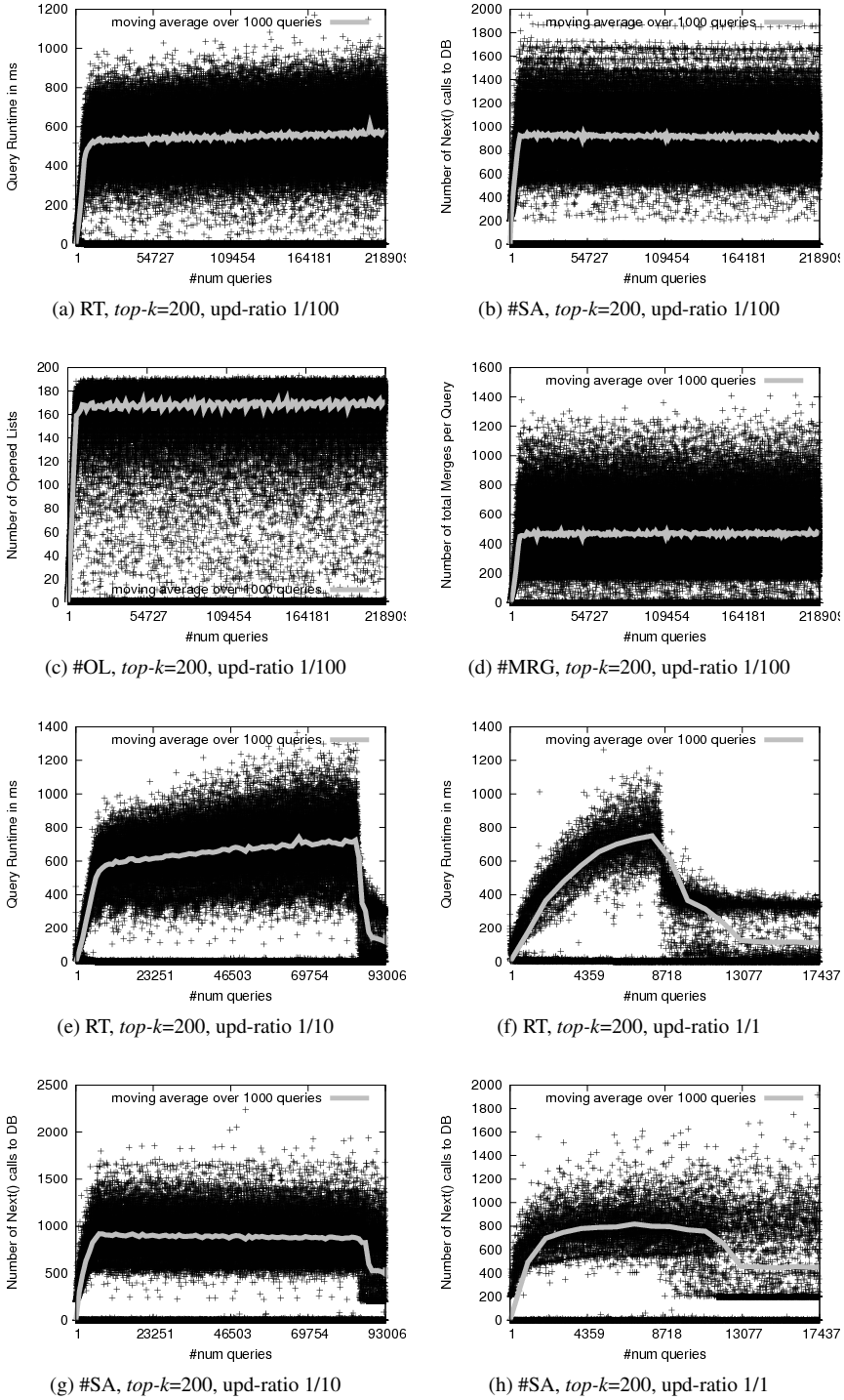
By the characteristics of *EAP*, it follows #OL=#MRG which is confirmed by our experiments shown in the Figures24a-24b and 24c-24d. Furthermore, since only larger list prefixes are involved in merge operations but *top-k* is also not varied in our experiments with *max-k*=500, of course, #OL and #MRG are exactly the same as in the previous experiments since the same merge calls have to be done for retrieving the correct *top-k* friends. Hence, the corresponding charts are equal.

Additionally, the Figures 25a, 25c and 25b, 25d show in comparison with Figure 23d and 23h again that at higher update ratios RT and #SA keeps fairly stable in our experiments with *max-k*=500, too. The drop in RT and #SA after $8,456$ or $84,560$ queries, respectively, occurs again because all edge updates have been applied to the graph at that time.

***LAP* Approach** For the experiments with the *LAP* approach of our algorithm, we set *top-k* $= 200$, too, in order to compare the results with *EAP*. Moreover, we varied the update ratio of new friendship edges in the same way as with *EAP*.

With *LAP* the runtimes, RT, and the number of sequential accesses to the database, #SA, depend on the number of opened lists, #OL, and the number of merge operations, #MRG, since only single list entries are merged on demand.

The Figures 26a-26d show the results for our experiments with the *LAP* approach of our algorithm. New friendship edges are inserted at an update ratio of 1 update each 100 queries. It can be observed that #SA on average is fairly low although #MRG



(a) RT, *top-k*=200, upd-ratio 1/100

(b) #SA, *top-k*=200, upd-ratio 1/100

(c) #OL, *top-k*=200, upd-ratio 1/100

(d) #MRG, *top-k*=200, upd-ratio 1/100

(e) RT, *top-k*=200, upd-ratio 1/10

(f) RT, *top-k*=200, upd-ratio 1/1

(g) #SA, *top-k*=200, upd-ratio 1/10

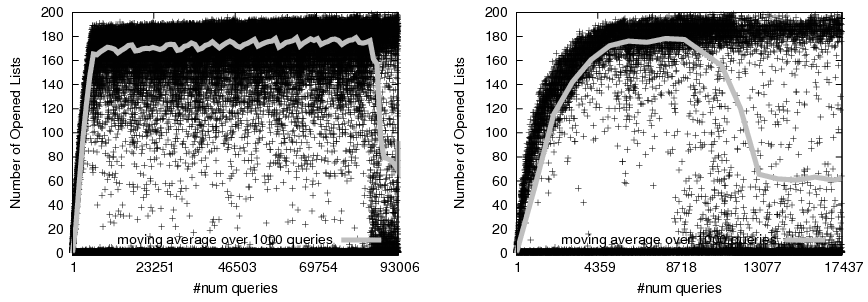
(h) #SA, *top-k*=200, upd-ratio 1/1

Figure 26: Experiments for *LAP* on *LibraryThing.com*, randomly selected querying users, top-k=200, 1 update per 100 queries in 26a-(d), and and 1 update per query for RT in (f) and #SA in (h), and 1 update per 10 queries for RT in (e) and #SA in (g).

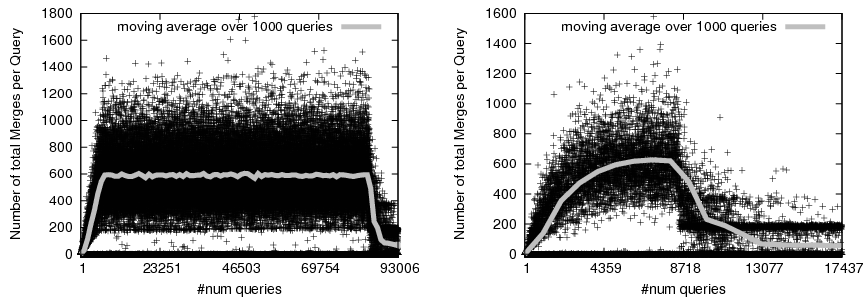
is much higher than with the *EAP* approach, even though the same number of lists are opened (#OL). The reason is that more merge operations have to be applied until all updates from other lists are merged since only a single entry is merged with each merge operation. In addition, for #SA is even lower as in the best case of *EAP* that restricts merge operations on only short prefixes of the size of some constant *top-k*, it also shows that indeed on average shorter prefixes of lists (< 200) have to be read until the *top-k* nearest friends can be correctly identified.

However, when comparing the average RT of *LAP* shown in 26a with the results of *fixed-size EAP* shown in 23c and 23d, we can see that RT is slightly worse than the *fixed-size EAP* with a constant *top-k* value of 200 but much better than with the one with *max-k*=500. However, in any case, #SA is better with the *LAP* approach. There are two reasons why RT is not better than both versions of *fixed-size EAP* although #SA is: first, maintaining the queue of candidates with *LAP* causes additional costs—again the dominating factor is I/O, not CPU time. Second, although the queue of candidates is small (no more than *top-k* friends can be candidates), it is initialised with each query by fetching each single candidate from DB. A more efficient maintenance that allows to retrieve the lists with only one access to the DB could additionally reduce the costs of maintenance.

From the Figures 26e-26h it easily can be observed that *LAP* is more sensitive to the update ratio. For higher update ratios, both RT and #SA increase. The higher sensitivity becomes especially visible in the Figures 26f and 26h after $8,456$ queries,



(a) #OL, *top-k*=200, upd-ratio 1/10 (b) #OL, *top-k*=200, upd-ratio 1/1



(c) #MRG, *top-k*=200, upd-ratio 1/10 (d) #MRG, *top-k*=200, upd-ratio 1/1

Figure 27: Experiments for *LAP* on *LibraryThing.com*, randomly selected querying users, *top-k*=200, 1 update per 10 queries for #OL in (a) and #MRG in (c), and 1 update per query for #OL in (b) and #MRG in (d).

and Figures 26e and 26g after $84,560$ queries, when all friendship updates have been applied. The drop in RA and #SA is much more abrupt and steep than with *EAP* as depicted in the Figures 23a-23b and 23e-23f.

For completeness, the Figures 27a-27d depict the number of opened lists and number of merge operations applied with *LAP* at an update ratio of 1 update per 10 queries and 1 update per query. As expected, in comparison with Figure 26d, it can be observed that with higher update ratios, the number of merge operations increases. However, the average number of opened lists stays the same since the final *top-k* friends, of course, are found in the same lists. Again, although the number of merge operation increases with higher update ratios, the number of sequential accesses to the database does not. This can be taken as another indication that the graph is tightly connected and shorter prefixes of involved lists (compared to *EAP*) have to be read until the *top-k* friends can be identified.

9.12.2 Results on *Twitter.com*

For *Twitter.com*, we were not able to use the same setup as for *LibraryThing.com* because of its huge size. We preprocessed the *Twitter.com* dataset in the same way as the one from *LibraryThing.com* by removing around $\frac{1}{3}$ of the edges from all *Twitter.com* users with more than 2 friends and for half of all users with only 2 edges, we removed one of them. In both cases we randomly chose which edge to remove. Additionally, we also randomly chose half of the users with only 1 edge and removed the only existing edge. All removed edges and the finally remaining subgraph were stored in the same Oracle 10 database as the *LibraryThing.com* dataset.

However, the size of the subgraph still exceeded by far the size of usual memory configurations of modern PCs and also of our database server that we used for our experiments. In addition, for being able to compare the results of our algorithm on *Twitter.com* with these on *LibraryThing.com*, we wanted to use exactly the same implementation of our algorithm. However, the runtime of our Java implementation of Dijkstra's APSP algorithm on such a huge graph like *Twitter.com* adds up to weeks instead of hours—even when all data completely is available in main memory. Therefore, we didn't precompute the shortest paths between all possible user combinations like we did on *LibraryThing.com*.

Before we are going into details about the final setup for *Twitter.com*, we want to give more precise information about the resource requirements needed for our straight forward implementation of Dijkstra's APSP algorithm in Java:

The graph has been implemented by using objects for nodes and edges. An object of type *Node* contains an variable ID of the primitive data type *int*, a variable $weight$ of the primitive data type *double* (which is needed for the APSP algorithm), and an array of *Edge*-objects, i.e. *Edges[]*, to maintain outgoing edges. An object of type *Edge* contains a variable ID and $weight$ of type *int* and *double*, respectively, too. Eventually, as commonly adequate in object-oriented programming, our Java implementation of *Edge*- and *Node*-objects has been completed by methods to set the $weight$, connect edges with nodes, etc.

Using this straight forward construction of a graph, in order to keep only the reduced subgraph of the *Twitter.com* dataset in main memory, an initial allocation of far over 120GB of heap space is required with Java. We then first tried to compute the shortest paths between all pairs of users in this subgraph by our (again straight forward) implementation of Dijkstra's APSP algorithm which was already used with the *LibraryThing.com* dataset. For this, we managed to get access to a powerful Linux

server with 256GB of main memory. Nevertheless, the computation could not be finished because the runtime simply was way too long and, hence, we stopped it.

Instead, we randomly chose a single user with 20 directly connected friends and applied a variant of Dijkstra's SSSP algorithm starting with this user and, subsequently, for each other user found during the execution of the algorithm. The search, however, was restricted to only the closest one thousand transitive friends of each user. After around 2 weeks, we stopped the execution and had managed to compute the shortest path distances for 2.427.523 of 41.659.253 users and their 1000 closest friends.

After these precomputation steps, we started the execution of our algorithm while re-inserting the former removed edges over time again. We randomly chose distinct users from the 2.4 millions in total and queried in the context of these users for their *top-200* friends while randomly adding friendship edges to the graph. Each applied update was always related to one of the users with precomputed friendship lists, too.

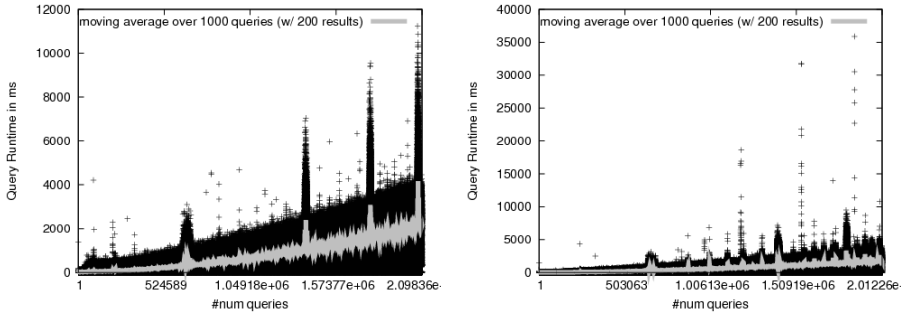
Again, as with *LibraryThing.com*, we measured the wall clock runtimes (RA) needed to retrieve the results, the number of sequential accesses to the database (#SA), the number opened friendship lists (#OL) and the number of merge operations (#MRG) needed to identify the correct *top-k* friends for each querying user.

Since the size of the reduced *Twitter.com* dataset used for our experiments is still huge, we only ran the most promising approaches of our algorithm as concluded from the experiments on the smaller *LibraryThing.com* graph. We additionally relinquished the verification of the correctness of the results on *Twitter.com* since the correctness of our implementation was already shown by the experiments on *LibraryThing.com* and because of the longish runtime of the straight forward implementation of Dijkstra's APSP algorithm in Java for such a huge dataset.

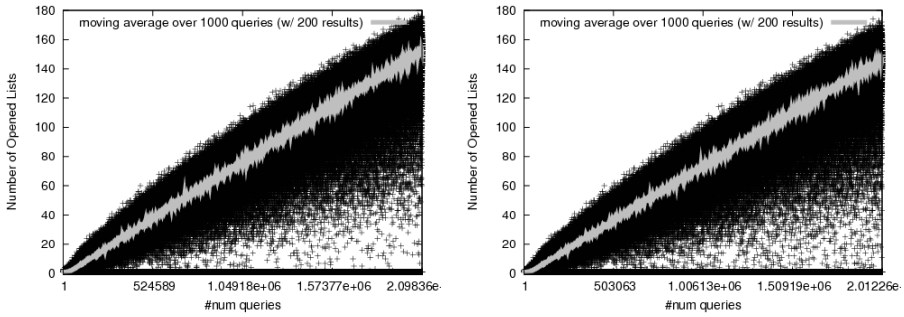
***fixed-size EAP* and *LAP* Approach** In our experiments on *Twitter.com* using the *fixed-size EAP* approach with *max-k*=*top-k* and also for the *LAP* approach of our algorithm, we chose *top-k*=200 and added each 100 queries a new edge to the precomputed graph.

For the number of users in the precomputed graph is quiet large, it takes a huge number of queries, as shown in our experiments, until the effects of update operations propagate through the graph and affect the runtimes. The Figures 28a-28d show the result for over 2 million queries on *Twitter.com* for RT, #SA, #MRG and #OL on both approaches *fixed-size EAP* and *LAP*. It can be observed that in general RT seem to increase linearly over time for *fixed-size EAP* and *LAP* but huge peaks happen here and there. However, no such peaks can be observed for the I/O related charts (see Figures 28b-28d) for #SA, #MRG and #OL. In the contrary, #SA even seem to increase slightly less the longer the algorithm runs.

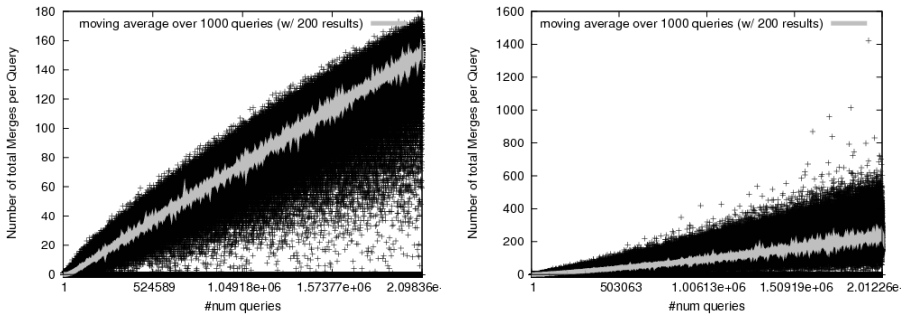
One explanation for this odd behaviour of our experiments on *Twitter.com* is that the experiments ran for weeks until over 2 million queries had been processed. Therefore, it is likely that these peaks in RT are network-related or are caused by problems on the database side. In addition, it turned out that the database needs to perform on average more than 3 accesses to the disk storage for each sequential read of a list entry. This can explain the relatively high runtimes on *Twitter.com*, even though #MRG, #OL and #SA have—even after 2 million queries—not yet reached the same level as shown in the experiments on the smaller dataset from *LibraryThing.com*. In fact, since the I/O load on *Twitter.com* in terms of #MRG, #OL and #SA is actually lower than on *LibraryThing.com*, the RT should be lower, too. However, apparently the huge amount of data in our current database setup with the attached disk storage or the remote access



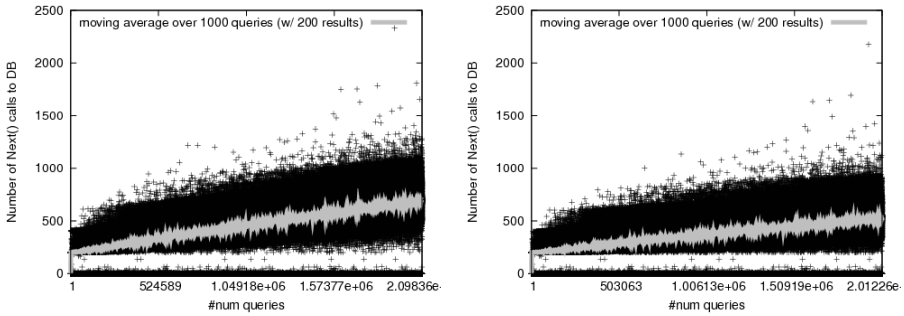
(a) RT: *fixed-size EAP* on the left, *LAP* on the right hand side



(b) #OL: *fixed-size EAP* on the left, *LAP* on the right hand side



(c) #MRG: *fixed-size EAP* on the left, *LAP* on the right hand side



(d) #SA: *fixed-size EAP* on the left, *LAP* on the right hand side

Figure 28: Experiments on *Twitter.com* for *fixed-size EAP* with *maxk*=*top-k* on the left and for *LAP* on the right hand side, randomly selected querying users, *top-k*=200, 1 update per 100 queries.

to the DB via Ethernet causes unexpected problems during long term experiments.

When comparing for *top-k*=200, *fixed-size EAP* using *max-k*=*top-k* with *LAP*, we can see that both approaches perform equally well on *Twitter.com*. Although, the absolute runtimes can hardly be compared due to the odd peaks in both charts, the moving average over intervals of 1000 queries show all about the same values (see Figure 28a). Not surprisingly, the same is true for #OL (see Figure 28b) since the same lists are involved in both approaches to find the *top-k* friends for a querying user. As expected, the number of merge operation (see Figure 28c) on *fixed-size EAP* is equal to the number of opened lists and much lower than in *LAP* since the *fixed-size EAP* approach merges full prefixes of lists while *LAP* only merges one single entry at a time. Figure 28b finally shows that with *LAP* also on *Twitter.com* the number of sequential accesses to the database can be slightly reduced compared to full prefix merge operations as with the *EAP* approach. That the difference in #SA is not huge shows that for *Twitter.com*, the *top-200* friends of users indeed are from many different friendship list retrieved. Hence, the users in our dataset are tightly connected in the friendship graph of *Twitter.com*.

9.12.3 Conclusion

The evaluation of our experiments show that our shortest path distances algorithm can dynamically handle insertion and increasing edge weights for large, disk-resident graphs up to millions of nodes. Both of our introduced approaches support a large variety of update ratios up to extreme situations where the number of updates and queries are similar. Moreover, the experiments have shown that both approaches *EAP* and *LAP* benefit quickly when updates occur less frequent or even do not happen for longer time periods. While the *fixed-size EAP* approach with an upper bound for the number of *top-k* results is even less sensitive for higher update ratios, the *LAP* approach not only quickly benefits from phases where no updates happen, it also allows to retrieve a flexible number of *top-k* results for different queries without introducing a larger performance penalty compared to *EAP*. As shown in our experiments, *EAP* and *LAP* are also applicable when the graph is tightly connected and a single update already influences large parts of the graph.

10 Proof of Correctness

To prove the correctness of our algorithm for maintaining dynamic shortest distances, we formalise the operations used in our *EAP* approach as described in Section 9.4 and show in a first step that by applying the two basic operations $U.update()$ and $U.merge(U_f)$, the data structures used for maintaining shortest distances are always well-defined and in a predictable state for all users and at any time. To do so, we introduce in Section 10.4 two invariants on friendship lists and prove they are not violated for all users in the graph by an update or a merge operation.

In a second step, we show in Section 10.8 with the help of these invariants that for each querying user our algorithm successively finds the user's correct next best friend and computes her correct friendship strength. Both steps combined prove that our algorithm always returns the requested *top-k* best friends in descending order or their friendship strength.

In Section 10.1, we introduce some notation and in Section 10.2, we give some properties on the defined friendship strengths of users in friendship graphs. In Section 10.3, we introduce the state description which is the basis for proving the correctness of our algorithm and formalise both operations $U.update()$ and $U.merge(U_f)$ in this regard. Afterwards, in Section 10.4, two important invariants are introduced and the proof that they hold for all users U and update or merge operations on U is given in Section 10.5 and Section 10.6, respectively. Section 10.7 extends the proof to all users in the friendship graph and, finally, with the help of the introduced invariants, in Section 10.8, a proof by induction is given that shows our algorithm always computes the correct results.

10.1 Notation

To ease readability, we briefly repeat in the following from Section 9.2 and Section 9.5 the most important notation used for the data structures of our algorithm:

- $Flist_{U,t}$: the dynamic friendship list (see Definition 9.11) of U containing all her transitive friends in G_t at time t of the dynamic friendship graph G (see Definition 9.4).
- $Flist_U$: the friendship list as maintained by our algorithm for user U with entries $e = (U_f, s)$ sorted in descending order of the friendship strength s of U's friends U_f.

  $Flist_U[i]$ refers to the *i-th* entry $e_i = (U_f, s)$ in U's friendship list.
- TS_U : timestamp of $Flist_U$
- tp_U : timestamp validity pointer of $Flist_U$
- $OP[U]$: operation map of pending friendship updates for U. OP contains sets of pairs $op_e = (U_f, TS_e)$ for each user U and pending friendship updates $e = (U \to U_f)$ with timestamp TS_e, i.e.

$$OP[U] = \begin{cases} \{op_e = (TS_e, U_f) \mid & \exists (U \to U_f) \in G_t \text{ with } t = TS_e \text{ and} \\ & s_{f,t}(U \to U_f) > s_{f,t-1}(U \to U_f)\} \\ \text{or } NULL \end{cases}$$

- $pos_U(U_f)$: auxiliary method that returns the position i of a friend U_f in $Flist_U$ where $(U_f, s) = Flist_U[i]$ or -1 otherwise.

To simplify the discussion, we introduce some additional notation in regard to the dynamic friendship graph defined in Section 9.2 and for conveniently referencing elements in friendship lists.

We define a shortcut for referring to a friend U_f or her friendship strength s_f at a given position in U's friendship list.

Definition 10.1 (Shortcut for U_f or s_f at position i in $Flist_U$)**.**

$$U_f \in Flist_U[i] \;\; or \;\; s_f \in Flist_U[i] \Leftrightarrow (U_f, s_f) = Flist_U[i]$$

∎

For convenience, we define U to be the owner of the friendship list $Flist_U$ who can be referenced by the negative index -1 in $Flist_U$ with friendship strength 0.

Definition 10.2 (Ownership of $Flist_U$)**.** *For referencing the owner U of a friendship list $Flist_U$, we define*

$$Flist_U[-1] = (U, 1)$$

∎

We denote with $bf_t(U)$ the *best friend* of U at time t

Definition 10.3 (Best Friend $bf_t(U)$)**.** *The best friend of a user U at time t is defined as*

$$bf_t(U) = U_f \; with \; (U_f, s_{f,t}) = Flist_{U,t}[0]$$

∎

We denote with $U_f \in Flist_{U,t}[i]$ the *i-th* best friend U_f of U in G_t and with $s_{f,t} \in Flist_{U,t}[i]$ her friendship strength at time t.

Definition 10.4 (*i-th* Best Friend in $Flist_{U,t}$)**.** *For the user U_f with the* i-th *best friendship strength $s_{f,t}$ with respect to U in G_t, we define:*

$$U_f \in Flist_{U,t}[i] \Leftrightarrow (U_f, s) = Flist_{U,t}[i] \; and \; s = s_{f,t}(U, U_f)$$

and

$$s_{f,t} \in Flist_{U,t}[i] \Leftrightarrow (U_f, s) = Flist_{U,t}[i] \; and \; s = s_{f,t}(U, U_f)$$

∎

10.2 Properties

In this section we declare some properties of a friendship graph G_t and involved data structures which will ease the understanding of the discussions in subsequent sections.

A property of the friendship strength $s_{f,t}(U, U_f)$ (see Definition 9.9) is that it is a multiplication of minimal number of edges weights which maximizes its values along the shortest path from U to U_f.

Property 10.1.

$$s_{f,t}(U, U_f) = \max(\{s_{f,t}(U \to U_f)\} \quad \cup \\ \{s_{f,t}(U \to \hat{U}) \cdot s_{f,t}(\hat{U}, U_f) \mid \textit{for all edges } U \to \hat{U}\})$$

■

Property 10.2. *Monotonicity of friendship strengths*

$$t < t' : \Pi_t(U, U_f) \subseteq \Pi_{t'}(U, U_f) \textit{ and} \\ : \forall \pi \in \Pi_t(U, U_f) : s_{f,t}(\pi) \leq s_{f,t'}(\pi)$$

■

The monotonicity of friendship strengths follows from the requirement that only friendship updates are considered which reduces the length (maximises the weight) of shortest paths to friends.

Property 10.3. *$bf_t(U)$ is a direct successor of U in G_t* ■

Property 10.3 follows from Property 10.1 and can trivially be proved by counter example: Assume that U_1 is the best friend of U in G_t and U_1 is not a direct successor. Then, there is a path $\pi = (U \to U_2 \to \ldots \to U_1)$ from U to U_1 over a direct successor U_2 of U. In that case, $s_{f,t}(U \to U_2)$ must be greater than $s_{f,t}(U, U_1)$ which is a contradiction to our assumption.

Property 10.4. *For all users U, when at time t* no *friendship update appears for U in G_t, her best friend does not change in G_t, i.e. $bf_{t-1}(U) = bf_t(U)$.* ■

Property 10.4 immediately follows from Property 10.3.

Property 10.5. *If a friendship update $(U \to U_f)$ appears in G_t, then*

$$bf_t(U) = \begin{cases} U_f & \textit{if } s_{f,t}(U \to U_f) \geq s_{f,t}(U, bf_{t-1}(U)) \\ bf_{t-1}(U) & \textit{otherwise.} \end{cases}$$

Property 10.5 immediately follows from Property 10.3 and Property 10.4.

Property 10.6. *When applying a new friendship update to G_t, the shortest path between all users in G_t who are connected over a path including the new friendship update, potentially change.*

Let be $(U \to U_f)$ the friendship update that is applied at time t to G_t, and $s_{f,t}(\pi) = s_{f,t}(U_1, U) \cdot s_{f,t}(U \to U_f) \cdot s_{f,t}(U_f, U_2)$ is the maximum weight for a path π from U_1 to U_2 which includes the friendship update $(U \to U_f)$. Then for all nodes U_1, U_2:

$$s_{f,t}(U_1, U_2) = \begin{cases} s_{f,t-1}(U_1, U_2) & \textit{if } s_{f,t-1}(U_1, U_2) > s_{f,t}(\pi) \\ s_{f,t}(\pi) & \textit{otherwise.} \end{cases}$$

■

From the Properties 10.2, 10.3 and 10.4 follows: $s_{f,t-1}(U_1, U) = s_{f,t}(U_1, U)$ and $s_{f,t-1}(U_f, U_2) = s_{f,t}(U_f, U_2)$. Hence, also Property 10.6 must hold.

10.3 Mode of Operation

In this section we give an overview of our correctness proof. For this, we introduce a state description on friendship graphs and define the update and merge operation in this regard.

Query-Dependent Maximal Timestamp TS_{max}

The value of TS_{max} (see Definition 9.17) corresponds to the timestamp of the latest friendship update in the dynamic friendship graph G (see Definition 9.4 in Section 9.2) and increases whenever there is an additional friendship update for some user in G, extending the sequence of friendship graphs G_t defined by G.

A query $\mathcal{Q}_U$ submitted in the context of a user U is supposed to retrieve the *top-k* best friends of U with respect to query time $t(\mathcal{Q}_U) = TS_{max}$ (see Definition 9.13). However, as discussed in Section 9.6, we may assume that there is no friendship update while a query is processed and, thus, that the value of TS_{max} does not changes during a query.

For our correctness proof, we formalise this assumption by the following definition.

Definition 10.5 (Query-Dependent Fixed Timestamp TS_{max})**.** *For each query $\mathcal{Q}_U$, the value of the maximal timestamp TS_{max} does not change but stays equal to the query time $t(\mathcal{Q}_U)$ as long as $\mathcal{Q}_U$ is not completely processed.*

In other words: There are no friendship updates while a query from a user U is processed. ∎

Point of Interest $x(U)$

To discuss the changes in friendship lists for all users in the friendship graph of a social network due to update and merge operations on an arbitrary user, we introduce an important definition.

We define the point of interest in a user's friendship list to be the first position in the list that requires our attention to adjust the next best friend or her friendship strength found at this point.

Definition 10.6 (Point of Interest $x(U)$)**.** *We define the* Point of Interest $x(U)$ *with respect to a friendship list $Flist_U$ of a user U in the following way: Let be,*

$$firstUpdate(U) = \min\{i \mid U_f.needs\_update() \text{ with } U_f \in Flist_U[i]\}$$
$$firstMerge(U) = \min\{i \mid TS_{U_f} > TS_U \text{ with } U_f \in Flist_U[i]\}$$

Then,

$$x(U) = \min\{tp_U,\ firstUpdate(U),\ firstMerge(U)\}$$

∎

Note: With new friendship updates in G, $firstUpdate(U)$ or $firstMerge(U)$ might change without the need of a query or an update/merge operation on $Flist_U$. According to Definition 10.5 the value of TS_{max} does not change during a query but may change between two subsequent queries. Thus, when there is no query on U, but there are friendship updates related to U, the value of $x(U)$ can change over time.

10.3.1 State Description

For proving the correctness of our algorithm, we first assume that all the data structures representing a friendship graph G_t are in a valid state at a given time t. Then, we discuss the resulting state of all data structures after an update or a merge operation is applied to any user in the friendship graph.

Definition 10.7 (Initial State σ_U)**.** *For all users U, we define the state of all user-specific data structures as*

$$\sigma_U = (Flist_U, tp_U, TS_U, tsmax_U) \text{ and } x(U)$$

with $tsmax_U$ is equal to TS_{max} at the time of the last or current update or merge operation on $Flist_U$. ∎

The initial state σ_U contains all user-related data structures that might change over time due to any operations of our algorithm. The timestamp TS_{max} implicitly changes for all users when new friendship updates arrive but is actually tied to the user-specific query time $t(Q_U)$ when U issues a query. Although, by Definition 10.5, TS_{max} is fixed during a query, it can change between two subsequent queries. Thus, we define the additonal data structure $tsmax_U$ to be part of the initial state σ_U (and of the resulting state, see Definition 10.8) to reflect possible changes of TS_{max} between two subsequent queries in the friendship graph. As a consequence, $tsmax_U$ denotes the timestamp of the last known friendship update at the time of the transition to state σ_U. Moreover, due to Definition 10.5, $tsmax_U$ corresponds to the query time $t(\mathcal{Q}_U)$ of the last query on U.

The point of interest $x(U)$ is not a user-specific data structure but indicates the position in a friendship list that needs attention—hence, it is related to a user-specific data structure. Since that position might change when applying update or merge operations, we have to take care of it together with σ_U. In contrast to $tsmax_U$ (or TS_{max}), it is not obvious when $x(U)$ changes for a user U but needs to be discussed for each user and update or merge operation.

After applying an update or a merge operation on some user U_j in G_t, the resulting state for each user is denoted as follows.

Definition 10.8 (Resulting State σ'_U)**.** *For all users U in G_t at time t, we define the resulting state σ'_U after applying an update or merge operation on any user U_j due to a friendship update transforming G_t into $G_{t'}$ as:*

$$\sigma'_U = (Flist'_U, tp'_U, TS'_U, tsmax'_U) \text{ and } x'(U)$$

where $Flist'_U, tp'_U, TS'_U, tsmax'_U$ correspond to the data structures of user U right after the applied operation; $x'(U)$ is the resulting point of interest in $Flist'_U$. ∎

Although our algorithm shown in Listing 5 of Section 9.4 ensures that the following is true (since the method $U_f.needs\_update()$ is called before the call of the method $U.needs\_merge(U_f)$), we define an order on operations on users to clarify which operation is applied first.

Definition 10.9 (Priority of Operations)**.** *In case, while processing a query, there is both a need for an update operation on a user U or a user U_f* and *a merge operation on U with U_f then the update operation on user U or U_f is always applied first.* ∎

With these ingredients, we can derive invariants on the state of all users in the friendship graph (see Section 10.4) and prove they hold for update and merge operations (see Section 10.5, 10.6 and 10.7) by discussing the resulting state of σ'_U for all users U as defined by our algorithm and comparing the resulting friendship list $Flist_U$ with the dynamic friendship list $Flist_{U,t'}$ at time t' for all users U in the friendship graph, where t' is equal to the friendship list's resulting timestamp TS'_U.

To this end, we formalise the semantic of an update operation and a merge operation on U in G_t by deriving the resulting state σ'_U from an initial state σ_U.

10.3.2 State Transition for U.$update()$

For a user U, we assume an initial state σ_U and define an update operation on U by describing the resulting state σ'_U of U.

Definition 10.10 (Update Operation on U)**.** *For a friendship update $(U \to U_f)$ at time $t' > t = TS_U$ with associated friendship strength $s_{f,t'}(U \to U_f)$, an update operation on U transforms σ_U into U's resulting state*

$$\sigma'_U = (Flist'_U, tp'_U, TS'_U, tsmax'_U)$$

where

$$\sigma'_U = \sigma_U \quad \text{if} \quad s_{f,t'}(U \to U_f) < s = s_{f,t}(U, U_f) \text{ and } (U_f, s) \in Flist_U$$

or otherwise,

- $Flist'_U$ *is the result from merging into* $Flist_U$ *:*
  1. *the new or now better friend U_f of U with the new friendship strength $s = s_{f,t'}(U \to U_f)$*
  2. *the friends of U_f merged in by a merge operation on U with U_f, i.e. U.$merge(U_f)$.*
- *U's timestamp validity pointer is set to* 0*, i.e.*
$$tp'_U = 0$$
- *the timestamp of U's friendship list is set to the time of the the latest friendship update t', i.e.*
$$TS'_U = t'$$
- *the maximal timestamp does not change during an update operation, i.e.*
$$tsmax'_U = tsmax_U$$

■

Note: If there is more than one friendship update for a user U, the update operation on U is called for each friendship update in ascending order of their timestamps TS_e until all friendship updates for U have been inserted in $Flist_U$.

The pseudocode implementing this state transition of the update operation U.$update()$ on U is shown in Listing 8 of Section 9.4.

10.3.3 State Transition for $U.merge(U_f)$

Before a merge operation on some user U with some friend U_f can be applied, first all update operations on U_f are applied (see Definition 10.9) by calling $U_f.update()$ if necessary (i.e. when there is at least one pending friendship update for U_f).

For a user U, we again assume an initial state σ_U and define a merge operation on U with some user U_f by describing the resulting state σ'_U of U.

The resulting state of U when a merge operation on U with U_f is applied is defined as follows:

Definition 10.11 (Merge Operation on U with U_f)**.** *For a merge operation on a user U with a user U_f, first all update operations on U_f are applied for pending friendship updates, then the merge operation on U results in U's following state:*

$$\sigma'_U = (Flist'_U, tp'_U, TS'_U, tsmax'_U)$$

where

$$\sigma'_U = \sigma_U \quad \text{if} \quad U_f \notin Flist_U$$

and otherwise,

let be $s = s_{f,t}(U, U_f)$ the friendship strength of U_f wrt. U. Then,

- *$Flist'_U$ is the result from merging into $Flist_U$ from $Flist_{U_f}$:*
  1. *all users who are not yet friends of U at time t.*
  2. *all users who have become better friends to U because of a shorter path over U_f, i.e.*

     $$\forall (U_{ff}, s_f) \in Flist_{U_f} \wedge\ s_{f,t'}(U, U_{ff}) = s \cdot s_f > s_{f,t}(U, U_{ff}) :$$
     merge $(U_{ff}, s \cdot s_f)$ *in* $Flist_U$

     Note*: to "merge" means an entry with U_{ff} is either replaced or inserted in $Flist_U$.*

- *the timestamp TS_U and timestamp validity pointer tp_U of U's friendship list is set in the following way:*

  If $pos_U(U_f) \leq tp_U$

  $$\begin{aligned} TS'_U &= max\{TS_U, TS_{U_f}\} \\ tp'_U &= pos_U(U_f) + 1 \end{aligned}$$

  otherwise

  $$\begin{aligned} TS'_U &= TS_U \\ tp'_U &= tp_U \end{aligned}$$

- *the maximal timestamp does not change during a merge operation, i.e.*

  $$tsmax'_U = tsmax_U$$

■

Note: According to this definition of a merge operation, it immediately follows that $pos'_U(U_f) = pos_U(U_f)$ and $Flist'_U[0:i] = Flist_U[0:i]$ with $i \leq pos_U(U_f)$.

The pseudocode implementing the defined state transition for the merge operation $U.merge(U_f)$ is shown in Listing 9 of Section 9.4.

10.3.4 State Transition for $\forall U_i \neq U : U.update(), U.merge(U_f)$

When applying an update or merge operation on U, then, for all users $U_i \neq U$, the initial state of U_i does not change.

Definition 10.12 ($U_i \neq U$: Update or Merge operation on U)**.** *An update or merge operation on U does not change the state of any user $U_i \neq U$. Hence, $\sigma'_{U_i} = \sigma_{U_i}$:*

- $Flist'_{U_i} = Flist_{U_i}$,
- $TS'_{U_i} = TS_{U_i}$, *and*
- $tp'_{U_i} = tp_{U_i}$.
- $tsmax'_{U_i} = tsmax_{U_i}$ ∎

Note: $tsmax_{U_i}$ in σ_{U_i} is part of the state description for U_i. Hence, it is equal to the last known friendship update in G at the time the data structures related to U_i were set into state σ_{U_i}. $\forall U_i \neq U$, it is not necessarily equal to the timestamp of the latest friendship update in G (in contrast to $tsmax_U$ in σ_U for any operation on U due to Definition 10.5).

Next, we define the processing of friendship lists and the meaning of tp_U in this respect.

Definition 10.13 (Sequential List Access)**.** *Each friendship list $Flist_U$ is processed sequentially. Update and merge operations are applied when needed. tp_U marks the last entry in $Flist_U$ that was processed with reference to TS_U. For this, tp_U is increased by one if no update operation on U_f or merge operation on U with U_f is applied for a user $U_f \in Flist_U[i]$ with $i = tp_U + 1$.*

The need for an update or merge operation with reference to TS_U is defined as in Definition 9.21 and Definition 9.22, respectively. ∎

In particular, Definition 10.13 implies the following lemma:

Lemma 10.1. $\forall U$: *If $TS_{max} = tsmax_U$ it follows $x(U) = tp_U$.* ∎

Proof. From Definition 10.13 and with $tsmax_U = TS_{max}$, it immediately follows that there cannot be any pending friendship update for U or a user $U_f \in Flist_U[i]$ with $i < tp_U$, and the timestamps TS_{U_f} of those users' friendship lists cannot be greater than TS_U by definition of an update and merge operation. Hence, with reference to Definition 10.6, it follows $x(U) = tp_U$. □

10.4 Invariants

In this section we introduce the two invariants on friendship lists that hold for each update or merge operation and all users U in a social friendship graph. In addition we introduce two lemmas that help to prove the correctness of the invariants.

10.4.1 Intuition

Before formalising the two invariants on friendship lists, we want to sketch the intuition that leads to their formulation.

Invariant 1

For all users U, the first invariant applies to all entries in a user's friendship list up to and including its *point of interest* $x(U)$. The intuition of Invariant 1 is fairly simple: At all times, the prefix of a friendship list specified by its point of interest $x(U)$ is correct, i.e. no friend is missing, the order of friends is correct and also their friendship strengths with respect to U correspond correctly to the weights of the respective shortest paths starting at U.

Invariant 2

Invariant 2 concerns the entry next to the point of interest $x(U)$ at position $x(U)+1$ in a friendship list of a user U.

The intuition behind Invariant 2 is not so easy to see. Therefore, we sketch it with the help of short examples for an update operation on a single user U. Though, the invariant holds, of course, for all users at any time and for all operations.

Note: For the time being, take the logical clauses in captions of the following example figures (like in Figure 29b: $\neg A \wedge D1$) just as discriminative identifiers. Their true intentions will become clear as soon as we formalise the intuition given in this section.

In Figure 29a, a graph G_2 with 3 users is visualised and an update operation on U has been applied for the friendship update $(U \rightarrow U_X)$ at time $t = 2$. Hence, $TS_{max} = 2$ as there is no other friendship update. After the update operation on U has been applied, the timestamp TS_U of U's friendship list is equal to 2 and the timestamp validity pointer tp_U is equal to 0. Hence, according to its definition, the point of interest $x(U)$ is equal to $tp_U = 0$ since neither there is a friendship update for the user U_X at position 0 nor the timestamp TS_{U_X} of her friendship list is greater than TS_U and no other user previous to U_X meets these requirements. What we can observe from Figure 29a is,

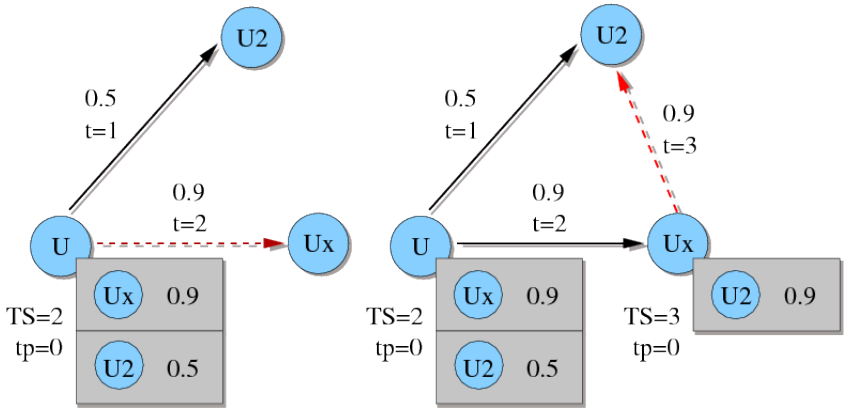
1. the shortest path to U_2 at position $x(U) + 1$ does *not* lead over U_X in G_2 with $TS_{max} = 2$.

2. the timestamp TS_U of U's friendship list is newer than TS_{U_X} of U_X's friendship list.

3. the entry $(U_2, 0.5)$ at position $x(U) + 1$ in U's friendship list is correct wrt. to the maximal timestamp $TS_{max} = 2$.

Hence, the intuition sketched in Figure 29a is the following:

A: *If the shortest path does not lead over the user found at position $x(U)$ in U's friendship list, then the timestamp TS_U is greater than that of the user at position $x(U)$ and the entry at $x(U) + 1$ is correct.*



(a) Example graph for A

(b) Example graph for $\neg A \wedge D1$

Figure 29: Example graph for σ'_U with friendship update $(U \rightarrow U_X)$. In (a) the update operation on U happened while $TS_{max} = 2$. In (b) the update operation on U happened while $TS_{max} = 3$ and U_X was previously updated.

In Figure 29b we see the same setup as in Figure 29a but after the update operation on U, a new friendship update $(U_X \rightarrow U_2)$ for U_X occurred. Hence, $TS_{max} = 3$ and by applying an update operation on U_X, the graph G_2 is transformed into G_3. Since the shortest path from U to U_2 has changed in G_3, the entry at position $x(U)+1$ in U's friendship list is not correct anymore. However, U_X is still correct and, according to its definition, $x(U) = pos_U(U_X)$ since the timestamp TS_{U_X} of U_X's friendship list is greater than the one of U's.

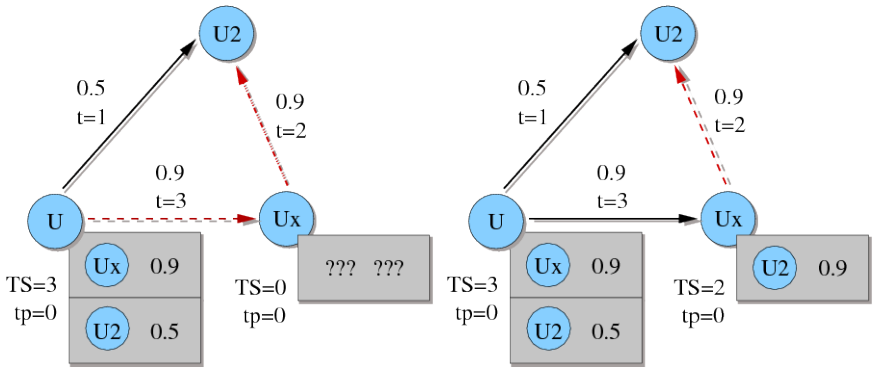
What we finally can observe from Figure 29b with respect to U's friendship list is,

1. the shortest path to U_2 at position $x(U)+1$ *leads* over U_X in G_3 with $TS_{max} = 3$
2. the timestamp $TS_{U_X} = 3$ of U_X's friendship list is greater than the timestamp $TS_U = 2$ of U's friendship list

Moreover, a merge operation on U with U_X would correct the entry with U_2 in U's friendship list. Hence, the intuition sketched in Figure 29b is the following:

$\neg$ **A** $\wedge$ **D1:** *If the shortest path* leads *over the user at position $x(U)$ in U's friendship list and the timestamp of her friendship list is greater than the one of U's, it contains updated information about shortest paths which needs to be merged in U's friendship list.*

The graph G_3 with $TS_{max} = 3$ depicted in Figure 30a shows the same users and friendship connections as previously but now, the order of the friendship updates has been changed. The friendship update $(U \rightarrow U_X)$ occurs at time $t = 3$ and is applied on U before the friendship update $(U_X \rightarrow U_2)$ which was inserted in G at time $t = 2$. Therefore, the timestamp TS_U of U's friendship list is equal to 3, $tp_U = 0$ and, according to its definition, $x(U) = pos_U(U_X)$ since U_X is the first friend in U's friendship list with a pending friendship update. Furthermore, the entry with U_2 at position $x(U) + 1$ is not correct with respect to G_3 and $TS_{max} = 3$.



(a) Example graph for $\neg A \wedge C$ (b) Example graph for $\neg A \wedge D2$

Figure 30: Example graph for σ'_U with friendship update $(U \rightarrow U_X)$. The update operation on U happened while $TS_{max} = 3$. However, in (a) U_X is still in a need for an update operation while in (b) the update for U_X was previously done.

What we now can observe from Figure 30a with respect to U's friendship list is,

1. the shortest path to U_2 at position $x(U)+1$ *leads* over U_X in G_3 with $TS_{max} = 3$
2. there is a pending friendship update for U_X

Moreover, an update operation on U_X and a subsequent merge operation on U with U_X would correct the entry with U_2. Hence, the intuition sketched in Figure 30a is:

$\neg$ **A** $\wedge$ **C:** *If the shortest path* leads *over the user with a pending friendship update at position $x(U)$ in U's friendship list, the shortest path information in U's friendship list for the $x(U) + 1$-th entry is possibly not yet correct.*

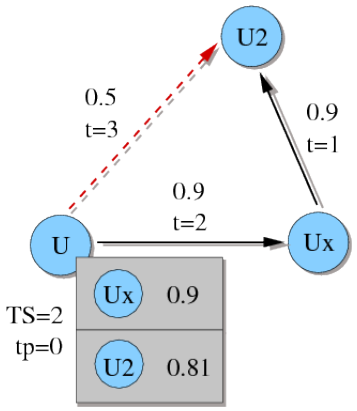
The Figure 30b shows the same setup as in Figure 30a. However, here the friendship update for U_2 that happened at time $t = 2$ is finally applied on U_2 in G_3 with $TS_{max} = 3$.

What we now can observe from Figure 30b with respect to U's friendship list is,

1. the shortest path to U_2 at position $x(U)+1$ *leads* over U_X in G_3 with $TS_{max} = 3$
2. the timestamp $TS_{U_X} = 2$ of U_X's friendship list is not newer than $TS_U = 3$ of U's friendship list
3. $x(U) = tp_U$ and the timestamp of the user's friendship list at position $x(U)$ is greater than 0, i.e. $TS_{U_X} > 0$

Moreover, a merge operation on U with U_X would again correct the entry with U_2. Hence, the intuition sketched in Figure 30b is the following:

$\neg$ **A** $\wedge$ **D2:** *When $x(U) = tp_U$ and the friendship list's timestamp of the user at position $x(U)$ is greater than 0, there could be shortest path information in that friend's friendship list which is not yet merged into U's friendship list.*



(a) Example graph for $\neg A \wedge E$

Figure 31: Example graph for σ'_U with a friendship update $(U \to U_2)$ and $TS_{max} = 3$. The best friend of U has already been found while $TS_{max} = 2$ and all friendship updates on U_X have already been applied.

The last example graph G_3 with $TS_{max} = 3$ is given in Figure 31a. Here the friendship update $(U \to U_2)$ at time $t = 3$ has got no effect on U's friendship list as the shortest path over U_X to U_2 is already known.

What we now can observe from Figure 31a with respect to U's friendship list is,

1. the shortest path to U_2 at position $x(U)+1$ *leads* over U_X in G_3 with $TS_{max} = 3$
2. the entry at $x(U) + 1$ in U's friendship list is already correct wrt. G_3 and $TS_{max} = 3$.

Hence, the intuition sketched in Figure 31a is the following:

$\neg$ **A** $\wedge$ **E:** *If none of the cases described in the previous example graphs are true, then all shortest path information is available in U's friendship list and the entry at position $x(U) + 1$ is correct.*

In summary, the main point of Invariant 2 is actually that for all operations done by our algorithm, the friendship lists for maintaining shortest path distances are always well-defined and in one of the described states depicted by the example graphs above—and no other state exists.

Next, we formalise these observations.

10.4.2 Formalisation

Invariant 1. *For all users $U \in G_{TS_U}$:*

$$\begin{aligned} Flist_U[0 : x(U)] &= \textit{ is a prefix of } Flist_{U,TS_U} \\ &= Flist_{U,TS_U}[0 : x(U)] \end{aligned}$$

■

Invariant 2. *For all users U :*

***A**: If the path with maximum weight to $U_f \in Flist_{U,TS_U}[x(U)+1]$ does* not *include $U_X \in Flist_{U,TS_U}[x(U)]$, then*
***B**: the entry $Flist_U[x(U)+1]$ is unchanged and correct:*
$Flist_U[x(U)+1] = Flist_{U,TS_U}[x(U)+1] = Flist_{U,TS'_U}(U)[x(U)+1]$

*¬**A**: Otherwise, for $U_X \in Flist_U[x(U)]$ the following or-cases hold:*

***C**: U_X needs an update and is older than U:*
$U_X.needs\_update() == True$ *and* $TS_{U_X} < TS_U$

***D**: There is a need for a merge operation on U with U_X, since*

***D1**: U_X's friendship list is newer than U's:*
$TS_{U_X} > TS_U$

***D2**: U_X is the last friend who is known to be the next best friend in $Flist_U$ but her friendship list was meanwhile updated:*
$x(U) = tp_U \wedge TS_{U_X} > 0$

***E**: entry $Flist_U[x(U)+1]$ is already correct:*
$Flist_U[x(U)+1] = Flist_{U,TS'_U}[x(U)+1]$

That means:
$(\mathbf{A} \Rightarrow \mathbf{B}) \vee (\neg\mathbf{A} \Rightarrow (\mathbf{C} \vee \mathbf{D} \vee \mathbf{E}))$ *with* $D = (D1 \vee D2)$ ∎

The statements being inferred from both invariants are informally spoken the following:

I1: At all times, all entries in a users friendship list up to $x(U)$ are correct.

I2: If the entry at $x(U)+1$ is not correct, at any time, the list is in a state such that update and merge operations will fix it. Afterwards, I1+I2 is still true.

I1+I2: The friendship lists can be incrementally maintained.

Before we prove that both invariants hold for update and merge operations for all users in a friendship graph and finally, (with the help of Invariant 1 and 2) that by incrementally processing friendship lists our algorithm always finds the next best friend, we first introduce and prove in the following two lemmas.

Lemma 10.2. *$\forall U_i \neq U$: If there is a need for an update or merge operation on U, the accomplishment of that operation does not affect $x(U_i)$, i.e. $x'(U_i) = x(U_i)$.* ∎

Proof. Before U is updated with a new friendship edge or before a merge operation on U is applied, either $U.needs\_update()$ must be $True$ or, for some friend U_f of U, $U.needs\_merge(U_f)$ must be $True$, respectively.

Therefore, if U is a friend of $U_i \neq U$ then no user in U_i's friendship list at later positions than U can match the definition of $x(U_i)$ (see Definition 10.6). That means,

$$\forall U_i \neq U \wedge pos_{U_i}(U) \geq 0 : x(U_i) \leq pos_{U_i}(U)$$

Hence, if

- $x(U_i) < pos_{U_i}(U)$ or $pos_{U_i}(U) = -1$.

  Then,
  $$x'(U_i) = x(U_i)$$
  because no change to U can affect $x(U_i)$. Operations on U can either only affect entries at positions subsequent to U in $Flist_{U_i,t}$—a path from U_i over U to friends of U is always longer than to U herself—or at no entries at all: if there is no path to U then there is also no path to U's friends over U.

  Hence, a new or a known friend of U whose friendship strength changed cannot reduce the friendship strength of any user that precedes U in $Flist_{U_i,t}$.

- $x(U_i) = pos_{U_i}(U) = tp_{U_i}$

  Then it is also true that
  $$x'(U_i) = x(U_i)$$
  since again, up to $pos_{U_i}(U)$ nothing on which $x(U_i)$ depends can change by an update or merge operation on U and, by Definition 10.12, tp_U does not change. Hence, $x'(U) = tp_U = x(U)$.

- $x(U_i) = pos_{U_i}(U) < tp_{U_i}$

  In this case, $x(U_i)$ can only be equal to the definition of (a) $U_i.firstUpdate()$ or (b) $U_i.firstMerge()$ (see Definition 10.6) and by assumption, $x(U_i)$ is equal to $pos_{U_i}(U)$. However, then it must be true again, that
  $$x'(U_i) = x(U_i)$$
  because the timestamp TS_U can only increase and, thus, either (a) is still true, or case (b) still is or becomes true since according to Lemma 10.4 the timestamp TS'_U increases to a value greater than TS_{U_i} by applying an update operation on U. Hence, in any case, $TS_U > TS_{U_i}$ or U is in the need of another edge update and $x'(U) = pos_{U_i} = x(U)$.

□

Lemma 10.3. *If due to an update operation on U or a merge operation on U with $U_X \in Flist_{U,t}[x(U)]$ the best new or updated friend U_f has been placed at a position greater than $x'(U) + 1$ in U's friendship list, then the following is true:*

If the path of maximal weight *to user $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ does not lead over $U'_X \in Flist_{U,t'}[x'(U)]$ then there is also no such path before the insertion of U_f.*

■

Proof. The update or merge operation can only affect entries at positions subsequent to U_f in $Flist_{U,t}$ because U_f is the user with the highest friendship strength among all users newly inserted in U's friendship list and a path from U over U_f to any other user can only be longer than to U_f herself.

Then,

- $x'(U) = 0 \leq x(U) \leq tp_U$

  in case of an update operation on U since $tp'_U = 0$ by definition of $U.update()$ (see Definition 10.10), or

- $x'(U) = x(U) + 1 \leq tp_U + 1$

  in case of a merge operation on U with $U_X \in Flist_{U,t}[x(U)]$ because either

  1. $x(U) = firstMerge(U) < tp_U$ or
  2. $x(U) = tp_U$
     (and $TS_{U_X} > 0$ or $x(U) = firstMerge(U) = tp_U$).

  according to the definition of $x(U)$ (see Definition 10.6) and U_X.

  Moreover, by definition of $U.merge(U_X)$ (see Definition 10.11) all edge updates for U_X must have been already applied and $tp'_U = x(U)+1$ and $TS'_U \geq TS'_{U_X}$. Hence, $x'(U) = tp'_U = x(U) + 1$.

In the case of a merge operation, by assumption, the position of U_f in $Flist_{U,t'}$ is greater than $x(U) + 2 = x'(U) + 1$ and therefore, the entry in $Flist_{U,t}$ at position $x(U)+1 = x'(U)$ cannot have changed due to that merge operation and is correct at time t' according to Invariant 1.

It follows, the entry was already correct at time t, too, because when

1. If $x(U) = firstMerge(U) < tp_U$, then, obviously $x'(U) = tp'_U \leq tp_U$.

   From Lemma 10.1 and from Invariant 1, it follows all entries in $Flist_U$ up to position tp_U are correct with respect to G_{TS_U}. Thus, at time $t = TS_U$, the entry at position $x'(U)$ is correct.

2. If $x(U) = tp_U$, then $x'(U) = tp_U + 1$ and due to our assumption that U_f is the best new or updated friend of U, inserted at a position later than $x'(U) + 1$ in U's friendship list, the friendship strength of U'_X at position $x'(U)$ cannot have change between time t and t'.

   Otherwise there must be either a user previous to $x'(U)$ in $Flist_{U,t}$ who is on a path to U'_X and whose friendship strength changed in between $(t, t']$ or a new direct edge from U to U'_X with a timestamp greater than t must still exist for U. However, that contradicts the definition of $x'(U)$.

   Hence, $Flist_{U,t}[x'(U) = x(U) + 1]$ must have been already correct at time t.

In any case, it is true for an update and merge operation that

$$\begin{aligned} Flist_U[0 : x(U) + 1] &= Flist'_U[0 : x'(U)] && \text{// due to x(U)+1<}pos_U(U_f) \\ &= Flist_{U,t'}[0 : x'(U)] && \text{// due to x'(U)<}pos_U(U_f) \\ &= Flist_{U,t}[0 : x(U) + 1] && \text{// as discussed above} \end{aligned}$$

Note: in case of an update operation: $x'(U) = tp'_U = 0$

Let be U'_{X+1} the user in $Flist_{U,t'}$ at position $x'(U)+1$ and at time t'. Furthermore, let be U_{X+1} the user in $Flist_{U,t}$ at the same position $x'(U) + 1$ but at the previous time t. Due to our assumption that at time t' the path from U to U'_{X+1} with maximal weight does not lead over $U'_X \in Flist_{U,t'}[x'(U)] = Flist_{U,t}[x'(U)]$, the following must be true at time t:

- Either there is a direct edge $U \to U'_{X+1}$, and then, $U_{X+1} = U'_{X+1}$,

because by assumption, in case of an update operation, $U \rightarrow U_f$ is the first new update for U after time t or, in case of a merge operation, U_f is the best user that is merged into U's friendship list. In both cases, $U'_{X+1} \neq U_f$.

However, then no previous user to U'_X suddenly can be in a need for an update or merge operation in the time interval $(t, t']$ because of Lemma 10.2 and by definition of $x(U)$.

- Or there is a direct edge from a user $U_j \rightarrow U'_{X+1}$ with $pos_U(U_j) < x'(U)$, and then, $U_{X+1} = U'_{X+1}$, too,

  because the best indirect path from U to U'_{X+1} can only lead over users at positions prior to $x'(U)$ — each user on a path with maximal weight to U'_{X+1} must be a better friend of U than U'_{X+1} — and there cannot be any new edge update for any such user U_j with $pos_U(U_j) < x'(U)$ in the time interval $(t : t']$ because of Lemma 10.2 and the definition of $x(U)$.

In any case, $U'_{X+1} = U_{X+1}$ for an update or merge operation and it immediately follows either

$$\begin{aligned} s_{f,t}(U, U_{X+1}) &= s_{f,t}(U \rightarrow U_{X+1}) \text{ // for a direct edge } U \rightarrow U_{X+1} \\ &= s_{f,t'}(U \rightarrow U'_{X+1}) \text{// the edge cannot have changed} \\ &= s_{f,t'}(U, U'_{X+1}) \\ &\textit{or} \\ s_{f,t}(U, U_{X+1}) &= s_{f,t}(U \rightarrow .. \rightarrow U_j \rightarrow U_{X+1}) \text{ // no direct edge from } U \\ &= s_{f,t'}(U \rightarrow .. \rightarrow U_j \rightarrow U'_{X+1}) \text{ // path cannot have changed} \\ &= s_{f,t'}(U, U'_{X+1}) \end{aligned}$$

Finally. it is true, that

$$\begin{aligned} Flist_U[0 : x'(U) + 1] &= Flist_{U,t}[0 : x'(U) + 1] \text{ // because } x'(U) + 1 < pos_U(U_f) \\ &= Flist'_U[0 : x'(U) + 1] \\ &= Flist_{U,t'}[0 : x'(U) + 1] \end{aligned}$$

Hence, the path with maximal weight from user U to user U'_{X+1} does also not lead over the user U'_X at time t because there is not such path at time t' and $Flist_U$ cannot have changed up to position of U'_{X+1} in the interval $(t, t']$. □

10.5 Update Operation – $U.update()$

In this section, we show that both invariants introduced in Section 10.4 holds for a user U when there is an update operation on U:

$$\forall U : Flist'_U[0 : x'(U)] = Flist'_{U,t'}[0 : x'(U)]$$

For this, we first introduce the following lemma and prove its correctness.

Lemma 10.4. *When there is a need for an update operation on $U_i \in Flist_{U,t}[i]$ with $i < tp_U$ then $TS_{U_i} > TS_U$ after the update operation on U_i.* ∎

Proof. Let denote TS'_{U_i}, TS'_U and tp'_U the timestamps of U_i and U, and U's timestamp validity pointer after the update of U_i, respectively. According to the definition of an update operation on U_i for $U \neq U_i$ (see Definition 10.12) we know, $tp'_U = tp_U$ and $TS'_U = TS_U$ for all users $U \neq U_i$.

Proof by contradiction: assume the lemma is wrong, i.e. $TS'_{U_i} <= TS'_U$.

In any way, there must be some pending friendship update $(U_i \rightarrow U_j)$ with a timestamp $t' > TS_{U_i}$ or otherwise there is no need to update U_i. After the update $TS'_{U_i} = t'$ by Definition 10.10 for that update operation.

From our assumption $TS'_{U_i} <= TS'_U$ it follows $t' <= TS'_U = TS_U$ and, thus, the friendship update for U_i already existed at time TS_U.

However, according to Definition 10.13, for users at position $i < tp_U$, thus including U_i, all friendship updates up to time TS_U have been checked (with calls to $U_i.needs\_update()$) and applied (by $U_i.update()$).

Therefore, either there is no pending friendship update for U_i or its timestamp must be greater than TS_U, i.e. $t' > TS_U$, which is a contradiction to our assumption. □

Theorem 10.1. *Invariant 1 holds for U and an update operation on U.* ■

Proof. According to Definition 10.10 given in Section 10.3.2 for the state transition σ'_U of an update operation, $tp'_U = 0$. It follows $x'(U) = 0$, too, because by Definition 10.6 $x'(U)$ cannot be smaller than 0 or greater than tp'_U. Therefore, we just have to show $Flist'_U[0] = bf_{t'}(U)$ (see Definition 10.3). We also may assume that Invariant 1 holds at time t for U and her friendship list $Flist_U$. Hence, all friendship updates for U up to time t have already been correctly inserted in $Flist_U$ or otherwise her best friend $bf_t(U)$ at time t could not be guaranteed to be correct. Since $x(U) \geq 0$ by definition and Invariant 1 says all entries up to $x(U)$ are correct at all times, $bf_t(U)$ must be correct, too.

By the definition of an update operation, all friendship updates will be removed from $OP[U]$ in the order of their timestamp and inserted one after the other in U's friendship list $Flist_[U]$ at their correct positions.

Let be U' with timestamp t' and friendship strength s' the first update inserted in $Flist_U$ due to an update operation.

Since by assumption there is no new or updated friendship edge for U in the time interval $[t, t')$, there is no change within this interval with respect to U's very best friend. Hence, $bf_{t'-1}(U) = bf_t(U)$. Therefore, after the insertion of the first friendship update, U's best friend at time t' can be either the previously known best friend $bf_t(U)$ or the user U' with friendship strength s'.

$$
\begin{aligned}
bf_{t'}(U) &= \begin{cases} U' & \text{if } s' \geq s_{f,t}(U \rightarrow bf_t(U)) \\ bf_t(U) & \text{else} \end{cases} \\
&= \begin{cases} U' & \text{if } s' \geq s_{f,t}(U \rightarrow bf_t(U)) \\ U_f \in Flist_U[0] & \text{else} \end{cases} \\
&= Flist_{U,t'}[0] \\
&= Flist'_U[0]
\end{aligned}
$$

Hence,

$$
\begin{aligned}
Flist'_U[0 : x'(U)] &= Flist'_U[0] \\
&= Flist'_{U,t'}[0] \\
&= Flist'_{U,t'}[0 : x'(U)]
\end{aligned}
$$

□

Theorem 10.2. *Invariant 2 holds for U and an update operation on U.* ∎

To prove that Invariant 2 still holds for U after an update operation has been applied on U, we may assume that both invariants hold for U's friendship list at time t and further have to show:

I: **A'** $\implies$ **B'**. If the path with maximal weight from U to her second best friend does not lead over her best friend $bf'_t(U) \in Flist_{U,t'}[0]$, then the entry at $Flist'_U[1]$ is correct with respect to time t'.

II: $\neg$**A'** $\implies$ **C'** $\vee$ **D1'** $\vee$ **D2'** $\vee$ **E'**. If the path with maximal weight from U to her second best friend leads over her best friend $bf_{t'}(U) \in Flist_{U,t'}[0]$, then there is either the need to update $bf_{t'}(U)$ or a merge condition is true or the entry at $Flist'_U[1]$ is correct with respect to time t'.

Alternatively, with $\neg$**D** := $\neg$(**D1** $\vee$ **D2**):

$\neg$**A'** $\wedge$ $\neg$**C'** $\wedge$ $\neg$**D'** $\implies$ **E'**. If the path with maximal weight from U to her second best friend leads over her best friend $bf_{t'}(U) \in Flist_{U,t'}[0]$, and there is neither the need to update $bf_{t'}(U)$ nor a merge condition is true, then the entry at $Flist'_U[1]$ is correct with respect to time t'.

Proof. According to Definition 10.10 of an update operation, $tp'_U = 0$, and therefore it follows that $x'(U) = 0$.

When a new or updated friendship edge $(U \to U_f)$ with weight s has been inserted in U's friendship list due to an update operation, there are three relevant cases where U_f can be located in $Flist'_U$ afterwards:

Case 1: $U_f \in Flist'_U[0]$ or

Case 2: $U_f \in Flist'_U[1]$ or

Case 3: $U_f \in Flist'_U[i]$ with $i > 1$

We have to show that in all three cases Invariant 2 is true.

- Case 1: $Flist'_U[x'(U)] = Flist'_U[0] = (U_f, s)$

  I: **A'**. The path with maximal weight to $Flist_{U,t'}[1]$ does *not* lead over $Flist_{U,t'}[0] = (U_f, s)$

  In this case, U's best friend at time t is U's second best friend at time t' and her friendship strength is unchanged, i.e.

  $$Flist_{U,t'}[1] = Flist_{U,t}[0].$$

  This is true because:

  1. Right before the update operation, U's best friend is also a direct successor of U due to Property 10.3 of the friendship graph G_t (see Section 10.2).

2. Since $Flist_{U,t}[0]$ is correct at timestamp t and each updated edge is applied in the order of the corresponding timestamps, there cannot be any other edge update in the time interval $[t, t')$ and, thus, there is no change to $Flist_{U,t}[0]$ until time t'.
3. Due to the assumption that U_f is the new best friend and that the second best friend is found over a direct edge, too, the best friend at time t has to be U's second best friend at time t'.

By definition, an update operation that inserts a user U_f into U's friendship list also merges the information about the shortest paths to U_f's friends into U's friendship list if adequate. However, in Case 1.I, we do not have to consider this additional merge operation since all friends found on a shortest path over U_f are inserted below the second best friend due to the assumption that the second best friend is a user found over a direct edge of U.

Hence,

$$\begin{aligned} Flist'_U[0] &= Flist_{U,t'}[0] \text{ (due to Invariant 1)} \\ Flist'_U[1] &= Flist_{U,t}[0] \text{ (due to correct insert/merge)} \\ &= Flist_{U,t'}[1] \end{aligned}$$

and **A'** $\implies$ **B'**:

II: ¬**A'**. The path to $Flist_{U,t'}[1]$ with maximal weight leads over $Flist_{U,t'}[0] = (U_f, s)$

Let us assume ¬**C'**, i.e. there is no new edge update for U_f in the time interval $[t, t']$ and ¬**D'**, i.e. $(\neg D1' \wedge \neg D2')$: the timestamp TS_{U_f} of U_f's friendship list is equal to 0, (—although, the latter assumption $\neg D'$ is not even needed as shown below).

We know (from being in Case 1.II) that U's second best friend is a successor of U's newly updated best friend U_f and, therefore, U's second best friend must be a user found over a direct edge starting at U_f. If there were some intermediate users, the first user on that path starting at U_f would be a direct successor and due to the monotonicity of friendship strengths (see Property 10.2 in Section 10.2) an even better friend for U_f and eventually also for U. Thus, if D or $\neg D$ is true can be ignored for this part of the proof since D is about the need for a merge operations which propagates indirect friends to friendship lists.

So, the second best friend of U at time t' is either U_f's very best friend or, in case of a friendship graph cycle such that $U \in Flist_{U_f}[0]$, U_f's second best friend.

Since an update operation on a user U always merges the friendship list of an updated friend after inserting her into $Flist_U$, also all direct friends of U_f in $Flist_{U_f}$ are merged into U's friendship list. In addition, no direct friend of U_f with respect to time t' is missing in U_f's friendship list because of our assumption ¬**C'**.

Hence,

$$Flist'_U[0] = Flist_{U,t'}[0] \text{ (due to Invariant 1)}$$
$$Flist'_U[1] = \begin{cases} Flist_{U_f,TS'_{U_f}}[0] & \text{if } bf_{t'}(U_f) \neq U \\ Flist_{U_f,TS'_{U_f}}[1] & \text{otherwise} \end{cases}$$
$$//\text{due to the definition of } \textit{update}$$
$$= Flist_{U,t'}[1] \text{ (due to } \neg\textbf{C'} \text{ and 1.II)}$$

Therefore, $\neg$ **A'** $\wedge\neg$ **C'** ($\wedge\neg$ **D'**) $\implies$ **E'**

- Case 2: $Flist'_U[x'(U)+1] = Flist'_U[1] = (U_f, s)$

  It follows that
  $$Flist'_U[x'(U)] = Flist'_U[0] = Flist_U[0]$$
  because with Case 2, U's newly updated friend U_f is sorted into $Flist_U$ at the position 1 and all other changes due to the update operation to $Flist_U$ can only affect entries at positions i, with $i > 1$, as there is no new or updated edge with a timestamp smaller than t' (the edge $(U \rightarrow U_f)$ is the first one for U with timestamp t') and a path to all friends of U_f that lead over U_f herself can only be longer than a direct edge to U_f.

  I: **A'**. The path with maximal weight to $Flist_{U,t'}[1] = (U_f, s)$ does *not* lead over $U'_0 \in Flist_{U,t'}[0]$

  Obviously, $Flist_{U,t}[0]$ does not change due to the update operation at time t' and U_f is a direct successor of U who, consequently, is inserted in $Flist_U$ with her correct friendship strength.

  It immediately follows:

  $$Flist'_U[0] = Flist_{U,t'}[0] \text{ (due to Invariant 1)}$$
  $$Flist'_U[1] = Flist_{U,t'}[1] = (U_f, s) \text{ (due to 2.I)}$$

  and **A'** $\implies$ **B'**

  II: $\neg$**A'**. The path to $Flist_{U,t'}[1] = (U_f, s)$ with the maximal weight leads over $U'_0 \in Flist_{U,t'}[0]$

  Again, there is no change to the first position in U's friendship list by applying the update operation. Furthermore, we may assume $\neg$**C'** $\wedge$ $\neg$**D'**.
  Consequently, when there is no update or change in the time interval $[t, t']$ with respect to U's very best friend U_0 and, by assumption 2.II, the shortest path to the second best friend U_f leads over U_0 at time t', then the same shortest path must have existed already at time t'. Furthermore, the new or updated direct friendship edge (U, U_f) cannot form a shorter path and, thus, has got no influence on the second entry in U's friendship list.

Finally, when Invariant 2 holds at time t and there is neither caused a change to $Flist_{U,t}[0]$ nor to $Flist_{U,t}[1]$ by the update operation, and in addition, when there is also no pending update operation for U's best friend U_0 (due to $\neg C'$) and no new information available in U_0's friendship list (due to $\neg D'$), then Invariant 2 must still hold at t'.

It follows:

$$\begin{aligned} Flist'_U[0] &= Flist_{U,t'}[0] \text{ (Invariant 1)} \\ Flist'_U[1] &= Flist_{U,t}[1] = (U_f, s) \text{ (2.II)} \\ &= Flist_{U,t'}[1] \ (\neg C' \text{ and } \neg D') \end{aligned}$$

and $\neg$ **A'** $\wedge \neg$ **C'** $\wedge \neg$ **D'** $\implies$ **E'**

- Case 3: $Flist'_U[x'(U)+i] = Flist'_U[i] = (U_f, s)$ with $i > 1$

  I: **A'**. The path with maximal weight to $U'_1 \in Flist_{U,t'}[1]$ does *not* lead over $U'_0 \in Flist_{U,t'}[0]$

  In this case, $Flist_{U,t'}[1]$ is a direct successor of U, too, because the best indirect successor can only lead over $Flist_{U,t'}[0]$ which contradicts 3.I.

  Since $(U \to U_f)$ is the first new friendship edge for U in the time interval $(t, t']$ and by assumption 3.I, the friendship strength $s = s_{f,t'}(U \to U_f)$ is smaller than the ones of $s_i \in Flist_U[i]$ for all $0 \leq i \leq x'(U)+1$, which means, the update has got no influence on any entry in U's friendship list up to $Flist_U[x'(U)+1]$, the best and second best friend of U cannot have changed between time t and t'. Otherwise U_f would have appeared in $Flist_{U,t'}[0:1]$ as both best friends in U's friendship list are found over direct edges at time t'. It follows:

  $$Flist_{U,t'}[0 : x'(U)+1] = Flist_{U,t}[0 : x'(U)+1]$$

  Moreover, as a result of Lemma 10.3, the path with maximal weight to $Flist_t[1]$ cannot have lead over $Flist_{U,t}[0]$ at time t.
  Since $x'(U) = 0 \leq x(U)$ by Definition 10.6 of $x(U)$ and Definition 10.10 of an update operation, and since by assumption Invariant 1 and 2 hold for $Flist_U$ at time t, both best friends in $Flist_U$ must be also correct at time t'. If $x'(U) < x(U)$ then this is true due to Invariant 1 and if $x'(U) = x(U)$ it is true due to Invariant 2.

  It follows:

  $$\begin{aligned} Flist'_U[0 : x'(U)+1] &= Flist_U[0 : x'(U)+1] \text{ (due to correct insert)} \\ &= Flist_{U,t}[0 : x'(U)+1] \\ &= Flist_{U,t'}[0 : x'(U)+1] \\ &\text{(due to } x'(U) \leq x(U), \text{ 3.I and} \\ &\text{both invariants hold at time } t) \end{aligned}$$

and **A'** $\implies$ **B'**

II: ¬**A'**. The path to $U_1' \in Flist_{U,t'}[1]$ with the maximal weight leads over $U_0' \in Flist_{U,t'}[0]$

We may again assume that the edge $(U \to U_f)$ at time t' is the first new edge for U since time t and from 3.II, we know that its associated weight is less then $s_0 \in Flist_U[0]$. Therefore, the best friend of U has not changed between t and t', i.e.

$$\begin{aligned} Flist_U[0] &= Flist_{U,t}[0] \\ &= Flist_{U,t'}[0] \\ &= Flist'_U[0] \end{aligned}$$

Since $Flist'_U[i] = (U_f, s)$ with $i > 1$, the edge (U, U_f) has got no influence on $Flist'_U[0:1]$ and, for this reason, $Flist_U[0] = Flist'_U[0]$ and $Flist[1] = Flist'_U[1]$.

However, to show that Invariant 2 still holds for $Flist'_U$, we have to distinguish 2 cases at time t:

(1) **A**. At time t: The path with maximal weight to $U_1 \in Flist_{U,t}[1]$ does not lead over $U_0 \in Flist_{U,t}[0]$

According to Invariant 2, $Flist_{U,t}[1]$ is correct at time t, but it is not at time t' because by assumption 3.II, at time t' there is a new path with maximal weight from U over $Flist_{U,t'}[0]$ to $Flist_{U,t'}[1]$.
Though, as there is no change in the time interval $(t, t']$ to the first entry of U's friendship list, and that entry is correct at time t, it is also correct at time t', i.e. $Flist_{U,t}[0] = Flist_{U,t'}[0]$.

According to (1) there is no path with maximal weight from $Flist_{U,t}[0]$ to $Flist_{U,t}[1]$ at time t but according to 3.II, there is such a path at time t'. It follows there must exist an edge update for $U_0 \in Flist_U[0]$—with a timestamp $t_x > 0$ by the definition of edge updates—such that the following situation is true: **A'** but ¬**B'**.

Consequently, at time $t'-1$, one of the following conditions must have been true:

a) $U_0.needs\_update() == True$

$\implies$ **C'**: As $U_0.needs\_update()$ is $true$ right before the update operation on U, it is still $true$ afterwards. Otherwise, condition b) or c) is true.

b) $TS_{U_0} > TS_U$: Then U_0 was updated before U has been updated and therefore, the timestamp of the friendship list of U's best friend U_0 is greater than U's, i.e. $t_x > t$.

If $t_x > t'$: $\implies$ **D1'**: After the update operation on U the timestamp $TS'_{U_0} = t_x$ is still greater than $TS'_U = t'$.

If $t_x < t'$: $\implies$ **D2'**: After the update operation on U the timestamp $TS'_{U_0} = t_x$ is smaller than $TS'_U = t'$ but $tp_U = 0$ by the definition of an update operation and $t_x > 0$.

c) $TS_{U_0} < TS_U$: Then U_0's friendship list was updated with a new edge at a time later than t but the timestamp t_x of the new edge is even smaller then t. However, $t_x > 0$.

$\implies$ **D2'**: As U_0's timestamp was greater than 0 before the update operation on U, it still must be greater afterwards, thus, $TS'_{U_0} = t_x > 0$ and $tp'_U = 0$ due to the definition of an update operation.

Hence, Invariant 2 holds: $\neg$ **A'** $\implies$ **C'** $\vee$ **D1'** $\vee$ **D2'**

(2) $\neg$**A**. At time t, the path to $U_1 \in Flist_{U,t}[1]$ with maximal weight leads over $U_0 \in Flist_{U,t}[0]$

Again, $Flist_U[0:1]$ is not changed by the update on U and according to Invariant 1, $Flist_U[0]$ is also correct at time t'. By assumption (2), we know that there was already a shortest path from the very best friend to the second best friend at time t. However, we have to verify the possibilities that could have occurred in the interval between time t and t' to U_0 in order to know if Invariant 2 holds for U_1 at time t'.

From the definition of an update operation, we again may assume that the updated edge (U, U_f) with timestamp t' is the first update for U since time t. Therefore, it is sufficient to concern about possible conditions at time $t' - 1$:

a) $U_0.needs\_update() == true$

$\implies$ **C'**: If there is a need for an update operation on U_0 at time $t' - 1$, the same is true after the update on U at time t'. Otherwise, condition b) or c) is true.

b) $TS_{U_0} > TS_U$: The timestamp of U_0's friendship list is newer than the one of U's,

$\implies$ **D1'** $\vee$ **D2'**: The timestamp of U_0's friendship list can only increase. Hence, after the update operation on U, either it is still true that $TS_{U_0} > TS_U$ or, at least, $TS_{U_0} > 0$ and, by definition of an update operation, $tp_U = 0$.

c) $0 < TS_{U_0} < TS_U$: The timestamp of U_0's friendship list is smaller than the one of U's,

$\implies$ **D2'**: Since the timestamp TS_{U_0} cannot decrease, it is still greater than 0 at time t', i.e. $TS_{U_0} > 0$ and, by definition of an update operation, $tp_U = 0$.

d) There is no update for $U_0 \in Flist_{U,t}[0]$ and $TS_{U_0} = 0$.
$\implies$ **E'**: By assumption, Invariant 2 holds at time t and due to d), i.e. $\neg$**C** $\wedge$ $\neg$**D**, it follows $Flist_U[1]$ is correct at time t. Since there are also no changes for U_0 up to time t', it follows that $Flist'_{U,t}[1] = Flist'_{U,t'}[1]$.
Therefore,

$$\begin{aligned} Flist'_U[1] &= Flist_U[1] \text{ (due to 3.II)} \\ &= Flist_{U,t}[1] \text{ (due to Inv 2 holds at } t \text{ and d)} \\ &= Flist_{U,t'}[1] \text{ (due to d))} \end{aligned}$$

and $Flist'_U[1]$ is already correct at timestamp t'.

Hence, Invariant 2 holds at time t' due to a) to d):
$\neg$ **A'** $\implies$ **C'** $\vee$ **D1'** $\vee$ **D2'** $\vee$ **E'**

Hence, Invariant 2 holds for all users U and $U.update()$:
(**A'** $\implies$ **B'**) $\vee$ ($\neg$**A'** $\implies$ **C'** $\vee$ **D'** $\vee$ **E'**)

□

10.6 Merge Operation – $U.merge(U_f)$

Next we show that both invariants introduced in Section 10.4 hold for U when there is a merge operation on a user U with a user U_f. For this, we may assume both invariants hold on $Flist_U$ right before the applied merge operation. Moreover, we introduce in the following three lemmas and prove their correctness.

Lemma 10.5. *When a merge operation $U.merge(U_f)$ inserts a user U_{ff} into $Flist_U$ with U_{ff} is a direct successor of U_f and no indirect path from U_f to U_{ff} exists with a higher weight, then the friendship strength of U_{ff} in $Flist_U$ is correct with respect to U and time t'.* ∎

Proof. When there is no better indirect path over another friend of U_f, a path to a direct successor of U_f must be correct with respect to time t' because by definition a merge operation on U with U_f is only applied when U_f has already been updated with all new or better edges. Hence, all direct edges are present in $Flist_{U_f}$ and up-to-date with respect to timestamp t'.

There also cannot be a shorter path from U to U_{ff} at time t' that does not lead over U_f, or U_{ff} had been merged earlier into $Flist_U$ since $x(U) = pos_U(U_f)$ or there is no merge operation on U with U_f. However, in this case, there is no friendship update to any user prior to U_f in $Flist_U$ in the time interval $[t, t']$ and all entries up to position $x(U)$ are correct.

Since the path with maximal weight to U_{ff} from U_f leads over a direct edge with a correctly computed edge weight and the path from U to U_f is a subpath of the one with

maximum weight to U_{ff}. the friendship strength of U_{ff} must be correct in $Flist_U$, too.

□

Lemma 10.6. *For a merge operation $U.merge(U_f)$, the friend U_{ff} with the highest friendship strength who is merged into $Flist_U$ is either a direct successors of U_f or of a weaker friend of U who is on the shortest path from U_f to U_{ff}.* ■

Proof. We may assume according to the definition of tp_U that $Flist_U$ is correct wrt. to G_{TS_U} up to tp_U before the merge operation is applied and, thus, all entries prior to U_f in $Flist_U$ are correct since $x(U) \leq tp_U$ by definition and there is only a merge operation when $pos_U(U_f) == x(U)$.

The shortest path from U to U_{ff} leads over U_f since U_{ff} is inserted in $FList_U$ due to the merge operation on U with U_f and no shorter path to U_{ff} that does not lead over U_f can exist. Otherwise, a user prior to U_f in $Flist_U$ must be a direct predecessor of U_{ff}. However, all direct friends of such a user must be already in $Flist_U$ because of the Definition 10.6 of $x(U)$, Definition 10.13 and our assumption $x(U) = pos_U(U_f)$.

Furthermore, if U_{ff} were not a direct successor of U_f or of a user at a position subsequent to U_f in $Flist_U$ then there must be an intermediate user on the shortest path to U_{ff} who is a direct predecessor of U_{ff} and either a better friend to U_f than U_{ff} and located prior to U_{ff} in $Flist_{U_f}$ or even a better friend to U than U_f is to U.

However, all users prior to U_{ff} in U_f's friendship list are already known to U and located in $Flist_U$ prior to U_{ff}. Otherwise, such a user would need to be merged into $Flist_U$ by the merge operation on U with U_f, too, but that is a contradiction to our assumption about U_{ff}.

Furthermore, all direct friends of users at positions prior to $x(U)$ must have already been merged into $Flist_U$ since otherwise there would be a need for an update or merge operation on such a user which again contradicts the definition of $x(U)$.

Hence, U_{ff} is either a direct successor of U_f or of a user subsequent to U_f in U's friendship list who is located on the shortest path from U_f to U_{ff}.

□

Lemma 10.7. *If U_{ff} is that friend of U_f with the highest friendship strength among all users merged into $Flist_U$ by a merge operation $U.merge(U_f)$ and if U_{ff} is positioned next to U_f in $Flist_U$ or if the shortest path to U_{ff} does* not *lead over users subsequent to U_f in $Flist_U$, then the entry with U_{ff} in $Flist'_U$ is correct with respect to time t'.* ■

Proof. Immediately follows from Lemma 10.6 and 10.5. □

Theorem 10.3. *Invariant 1 holds for U and a merge operation on U.* ■

Proof. To show that Invariant 1 holds, we have to distinguish the possible locations of U_f in $Flist_U$ relative to $x(U)$ for the initial state σ_U when a merge operation on U with U_f is applied (see Definition 10.11).

- $Flist_U[x(U)] = U_f$

  By definition $x(U) \leq tp_U$ and according to the definition of a merge operation, it follows $x'(U) \leq tp'_U = pos_U(U_f)+1 = x(U)+1 \leq tp_U+1$. In addition, $x'(U)$ cannot be smaller than $x(U)$ because a merge operation does not change any entries in $Flist_U[i]$ with $i \leq pos_U(U_f) = x(U)$ and $Flist_U[x(U)] = U_f$ by assumption.

Furthermore, there cannot be the need for an update operation with respect to time $t' \leq TS_{max}$ for any entry in $Flist_U[i]$ with $i < pos_U(U_f) = x(U)$ because the need for that update operation would have existed already before the current merge operation and, thus, is a contradiction to the definition of $x(U)$. For the same reason, no such entry can have a timestamp greater than t.

Note: $x(U)$ and merge operations are query or state dependent atomic operations on σ_U. During a query execution several merge operation on different friendship lists could run in parallel, such that $x(U)$ could even decrease before a current merge operation is executed. However, this only could happen, when TS_{max} increases because otherwise, we had identified the need for an update operation when passing such an entry. Therefore, $x(U)$ is always bound to a certain state σ_U with a fixed maximal timestamp TS_{max} as defined by Definition 10.5. Hence, query results are computed with respect to the given timestamp $tsmax_U$, corresponding to the maximal timestamp $t(\mathcal{Q}_U) = TS_{max}$ when the query is issued as defined by Definition 9.14.

It follows $x'(U) = x(U) + 1$ and

$$\begin{aligned} Flist'_U[0 : x'(U) - 1] &= Flist_U[0 : x(U)] \\ &= Flist_{U,t}[0 : x(U)] \\ &= Flist_{U,t'}[0 : x(U)] \\ &= Flist_{U,t'}[0 : x'(U) - 1] \end{aligned}$$

The user at position $x'(U) = x(U) + 1$ is either the best direct friend merged in from U_f's friendship list (see Lemma 10.7) if the path with maximum weight to U's next best friend leads over U_f, or the user at that position in U's friendship list is unchanged by the merge operation and correct at time t', too. The latter is true because in this case the shortest path from U to the user at position $x'(U)$ leads only over U's friends at positions prior to U_f in $Flist_U$ and no such friend can have changed in the time interval $[t, t']$ since it would contradict the definition of $x(U)$ which by assumption is equal to U_f's position in U's friendship list.

With respect to user U, let be s the friendship strength of the friend U_f, i.e. $s_{f,t'}(U, U_f) = s_{f,t}(U, U_f)$, and be s_{ff} the friendship strength of U_f's best friend U_{ff} that was merged into $Flist_U$, i.e. $s_{ff} = s \cdot s_{f,t}(U_f \rightarrow U_{ff})$, and be s_{X+1} the friendship strength of the user found at position $x(U)+1$ in $Flist_U$, i.e. $s_{X+1} \in Flist_{U,t}[x(U) + 1]$. Then,

$$Flist_{U,t'}[x'(U)] = \begin{cases} (U_{ff}, s_{ff}) & \text{if } s_{ff} > s_{X+1} \\ Flist_{U,t}[x(U) + 1] & \text{otherwise.} \end{cases}$$

Finally, it is true, that

$$Flist'_U[x'(U)] = Flist'_{U,t'}[x'(U)]$$

because either there is no change for $Flist_U$ caused at position $x(U) + 1$ by the merge operation or U_{ff} is correctly merged into $Flist_U$ by the definition of the merge operation.

- $Flist_U[x(U)] \neq U_f$ and $pos_U(U_f) < x(U)$

  In this case, according to Definition 10.6 of $x(U)$, there is neither a need to update U_f nor is the timestamp of her friendship list greater than the one of U and by assumption $pos_U(U_f) < x(U) \leq tp_U$. Therefore, there is no need for a merge operation on U with U_f. If we did a merge operation anyway, $Flist_U$ would stay unchanged because nothing not yet known to U can be merged from U_f.

  It follows, $x'(U) = x(U)$, $tp'_U = tp_U$, $TS_U = t = t' = TS'_U$ and

  $$\begin{aligned} Flist'_U[0 : x'(U)] &= Flist_U[0 : x(U)] \\ &= Flist_{U,t}[0 : x(U)] \\ &= Flist_{U,t'}[0 : x'(U)] \end{aligned}$$

- $Flist_U[x(U)] \neq U_f$ and $pos_U(U_f) > x(U)$

  Due to Definition 10.13 this case is not possible. However, if we did a merge for $pos_U(U_f) > tp_U = x(U)$, according to the Definition 10.11 follows:

  $$Flist'_U[0 : i] = Flist_U[0 : i]$$

  with $i \leq pos_U(U_f)$, and also $TS_U = t = t' = TS'_U$, $x'(U) = x(U)$. Hence,

  $$Flist'_U[0 : x'(U)] = Flist_{U,t'}[0 : x'(U)]$$

  because the merge operation has got no influence on any entry prior to $pos_U(U_f)$ in $Flist_U$ and actually does nothing by definition.

  A merge operation, however, for a user U_f selected by a random access at position $x(U) < pos_U(U_f) \leq tp_U$ must be strictly prohibited as it could increase TS_U and, thus, cause a missing merge operation for a user U_i at position i with $x(i) \leq i < pos_U(U_f)$. Hence, this case shows that sequentially processing friendship lists (as defined in Definition 10.13) is a fundamental requirement for our algorithm.

□

Theorem 10.4. *Invariant 2 holds for U and a merge operation on U.* ∎

Proof. For this proof, we again examine the possible changes in the transition from the initial state σ_U to the final state σ'_U (see Definition 10.7 and 10.8) for all users U when a merge operation (see Definition 10.11) on a user U or $U_i \neq U$ is applied.

When merging the friendship list of U_f into U's friendship list $Flist_U$, no entry in $Flist_U$ at positions prior to or equal to the one with U_f can change according to Definition 10.11 of a merge operation. All friends of U_f that are newly sorted into $Flist_U$ or have become better friends with respect to U will be merged subsequent to U_f's position into $Flist_U$. Furthermore, a friend of U_f which is not merged into $Flist_U$ must have previously been found on a path with a higher weight than the one over U_f, and hence, must be already a better friend of U.

According to Lemma 10.7, among all friends of U_f who are newly merged into $Flist_U$, the best new friend U_{ff} for U is a direct friend of U_f if placed next to U_f in $Flist_U$ or no user in between U_f and U_{ff} in $Flist_U$ is part of the shortest path from U to U_{ff}.

For the rest of this proof, we apply the following notation:

- U_{ff} denotes the best friend for U inserted in $Flist_U$ by a merge operation on U with U_f.
- $s_f \in Flist_{U_f}[pos_{U_f}(U_{ff})]$
- $s = s_{f,t}(U, U_f)$

Then, all users $U_i \in Flist_{U_f}[i]$ with $i < pos_{U_f}(U_{ff})$ are already friends of U and found at entries in $Flist_U$ prior to U_f. The friendship strength of U_{ff} with respect to U is $s_{ff} = s \cdot s_f$ according to the definition of a merge operation.

For a merge operation $U.merge(U_f)$ with $pos_U(U_f) < x(U)$ (although, the check for a merge operation would never indicate a need for it according to Definition 9.22), it follows from Definition 10.13, the Definition 10.6 of $x(U)$ and Invariant 1 that all friends in U_f's friendship list are already known to U and correctly inserted in $Flist_U$ and nothing changes for U. By definition 10.11 of a merge operation, it also follows that in this case, $x'(U) = tp'_U = pos_U(U_f) + 1 \leq x(U)$ and $TS'_U = TS_U$ is unchanged as by definition of $x(U)$ the timestamp of U_f's friendship list cannot be greater than TS_U.

When Invariant 2 holds for $Flist_U$ at time t even up to position $x(U)$, it obviously holds at the same time $t' = t$ also for any position $x'(U) \leq x(U)$.

$$\begin{aligned} Flist'_U[0 : x'(U) + 1] &= Flist_U[0 : x'(U) + 1] \\ &= Flist_{U,t}[0 : x'(U) + 1] \\ &= Flist_{U,t'}[0 : x'(U) + 1] \end{aligned}$$

For the rest of this proof, we can assume, a merge operation is only applied on U with a friend U_f, when $x(U) = pos_U(U_f)$ as by Definition 10.13, our algorithm processes friendship lists only sequentially. If $pos_U(U_f) > x(U)$ for some friend U_f then first a merge operation on U with $U_X \in Flist_U[x(U)]$ is applied before a merge operation on U_f can be applied. After the merge operation on U_X, by Definition 10.11, $tp'_U = x(U) + 1$ which is also the earliest possible position of U_f in $Flist_U$. If U_f is still at a later position, either tp_U is increased by one due to another merge operation or due to our main algorithm (as defined by Definition 10.13) proceeds to the next position in U's friendship list. In both cases it is always true that $x(U) = tp_U$. Hence, when initially $pos_U(U_f) > x(U)$ it follows that $x(U) = pos_U(U_f)$ when U_f is found in $Flist_U$.

From this it follows that for proving Invariant 2 holds for a merge operation on U with U_f, we have to discuss both parts of Invariant 2, i.e. **A'** and **¬A'**, only with respect to the possible position of the best friend U_{ff} merged into $Flist'_U$ from U_f's friendship list by a merge operation $U.merge(U_f)$.

- $Flist'_U[x'(U)] = (U_{ff}, s_{ff})$

  By Definition 10.11 of a merge operation, it follows $x'(U) = tp'_U = x(U) + 1$ and as discussed above, we may further assume, $x(U) = pos_U(U_f)$.

  I: **A'**. The path with maximal weight to $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ does not lead over $U'_X \in Flist_{U,t'}[x'(U)]$

  From Invariant 1, we may assume $Flist'_U$ is correct with respect to time t' and $x'(U)$. Therefore, it follows $U'_X = U_{ff}$. Next we discuss the possible

cases for U'_{X+1}.

1. The Path to $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ does not lead over U_f, too, but over U_i with $pos_U(U_i) = i$ and $i < pos_U(U_f)$.

   To make a valid conclusion about U'_{X+1}, we additionally have to discuss the possible changes of U's friendship list in the interval $[t, t']$. Remember, the friendship strength between two users can only increase over time. Therefore, we only have to distinguish the following two cases:

   (a) At time t the user U_{ff} is *not* in $Flist_{U,t}$ or is located at a position $pos_U(U_{ff}) > x(U)+1$.

   Then, U_{ff} is newly inserted into $Flist_U$ by the merger operation at position $x'(U) = x(U)+1$ and as a consequence, the user $U_{X+1} \in Flist_{U,t}[x(U)+1]$ must be located at the position next to U_{ff} at time t'. This is true because by assumption the path with maximal weight to U'_{X+1} does neither lead over U_f nor over U_{ff} and, thus, the entry at position $x'(U)+1$ is not set by the merge operation. Furthermore, there also cannot be another new or updated friendship edge at some time t_x with $t < t_x < t'$ such that there is a better friend of U who should be positioned at $x'(U)+1$ in $Flist_U$ instead. Such a new or updated edge would need to originate from a user prior to U_f in U's friendship list but that is a contradiction to our definition of $x(U) = pos_U(U_f)$.

   It follows, $U'_{X+1} = U_{X+1}$ and since there is no path with maximum weight over U_f to U'_{X+1} at time t' and as shown above, the shortest path to U_{X+1} did no change in $[t, t']$, there also cannot have been such a path over U_f at time t.

   Hence, the entry in $Flist_U$ with user U_{X+1} is correct at time t due to the assumption that Invariant 2 is true at time t and because of $U'_{X+1} = U_{X+1}$ and the shortest path from U to U_{X+1} has not changed since time t. Thus, the entry must still be correct at time t' and the following true:

$$\begin{aligned}
Flist'_U[0 : x'(U)-1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U)-1] \\
Flist'_U[x'(U)] &= (U_{ff}, s_{ff}) \text{ // (correct insertion)} \\
&= Flist_{U,t'}[x'(U)] \\
Flist'_U[x'(U)+1] &= Flist_U[x(U)+1] \text{ // (correct insertion)} \\
&= Flist_{U,t}[x(U)+1] \\
&= Flist_{U,t'}[x'(U)+1]
\end{aligned}$$

   (b) At time t it is already true that $pos_U(U_{ff}) = x(U)+1$

In this case, the position of the user U_{ff} in $Flist_{U,t}$ does not change due to the merge operation but the friendship strength does. That means, the user found at position next to U_f is the same at time t and t'. Again, because of $x(U) = pos_U(U_f)$ no other user can have become a better friend than U_{ff} in the time interval $[t, t')$.
Furthermore, due to our assumption that the path with the highest weight to $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ does neither lead over U_f nor over U_{ff}, the entry in $Flist_U$ at position $x'(U)+1$ cannot have changed since time t or otherwise, there would be a need for an update or merge operation to a user prior to U_f which again contradicts the definition of $x(U) = pos_U(U_f)$.

Accordingly, $x'(U)+1 == x(U)+2$ and, thus, it follows that $U'_{X+1} \in Flist_{U,t'}[x'(U)+1] == U_{X+2} \in Flist_{U,t}[x(U)+2]$.

Although, we cannot immediately follow by assumption that U_{X+2} was correct at time t, we can conclude the correctness in the current case.

At time t all entries in $Flist_{U,t}$ up to position $x(U)$ are correct by assumption. In Addition, the friendship strength of U_{ff} at position $x(U)+1$ becomes stronger only at time t'. However, the maximum path to U's subsequent friend U_{X+2} still does not lead over U_f or U_{ff} but over a user prior to U_f by assumption. Therefore, that path cannot have changed since time t or there would have been a need for an update or merge operation on one of the users located on this path which again is a contradiction to the definition of $x(U)$.

Hence, also at time t the path with maximum weight to U_{X+2} did not lead over U_f or U_{ff} and, thus, $Flist_{U,t'}[x(U)+2]$ must be correct. Assuming otherwise, then there is at time t a direct friendship edge with weight w from some user $(U_i, s_i) \in Flist_{U,t}[i]$ with $i < x(U)$ to U_{X+2} such that $s_i \cdot w > s_{X+2} \in Flist_{U,t}[x+2]$. However, either that edge was not yet updated into U_i's friendship list at time t or there is a need for U to merge this new information from U_i into U's friendship list. In both cases, this is a contradiction to the definition of $x(U)$. It follows:

$$
\begin{aligned}
Flist'_U[0 : x'(U)-1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U)-1] \\
Flist'_U[x'(U)] &= (U_{ff}, s'_{ff}) \text{ // (correct change of } s_{ff}) \\
&= Flist_{U,t'}[x'(U)] \\
Flist'_U[x'(U)+1] &= Flist_U[x(U)+2] \text{ // (correctly unchanged)} \\
&= Flist_{U,t}[x(U)+2] \\
&= Flist_{U,t'}[x'(U)+1]
\end{aligned}
$$

Hence, in both cases (a) and (b) Invariant 2 holds: **A'** $\implies$ **B'**

2. Path to $Flist_{U,t'}[x'(U)+1]$ leads over U_f

   Again, we have to discuss the possible states of $Flist_U$ at time t.

   (a) At time t, both users U_{ff} and $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ are either *not* yet friends of U, i.e. $\notin Flist_{U,t}$, or not that good friends, i.e. $pos_U(U_{ff}) > x(U)$ or $pos_U(U'_{X+1}) > x(U)$, respectively.

   In this case, since by assumption the shortest path to U'_{X+1} leads over U_f but does not lead over U_{ff}, it follows U'_{X+1} is a direct successors of U_f, too. Moreover, U_{ff} at position $x'(U) = x(U)+1$ and U'_{X+1} at $x'(U)+1 = x(U)+2$ have been newly merged into $Flist_{U,t'}$ with their correct friendship strengths according to the definition of a merge operation. It immediately follows $Flist_{U,t'}[x'(U)+1]$ is correct:

$$\begin{aligned} Flist'_U[0:x'(U)-1] &= Flist_U[0:x(U)] \\ &= Flist_{U,t}[0:x(U)] \\ &= Flist_{U,t'}[0:x'(U)-1] \\ Flist'_U[x'(U)] &= (U_{ff}, s_{ff}) \text{ // (correctly inserted)} \\ &= Flist_{U,t'}[x'(U)] \\ Flist'_U[x'(U)+1] &= (U'_{X+1}, s'_{X+1}) \text{ // (correctly inserted)} \\ &= Flist_{U,t'}[x'(U)+1] \end{aligned}$$

   (b) At time t, only U_{ff} is either *not* yet a friend of U, i.e. $\notin Flist_{U,t}$, or a not that good friend, i.e. $pos_U(U_{ff}) > x(U)$, but the user $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ is already a friend at time t.

   To summarise, by assumption the following is true: U_{ff} is the best friend of U_f that is merged into $Flist_U$ at position $x'(U)$ but the friend U'_{X+1} at the position subsequent to U_{ff} is not merged into U's friendship list by the merge operation with U_f.
   That means, the merge operation on U pushes U_{X+1} downwards by one position in order to insert U_{ff} in $Flist_U$. Thus, it follows $U'_{X+1} \in Flist_{U,t'}[x'(U)]$ with $U'_{X+1} = U_{X+1}$.
   Again, there cannot be another user who became a better friend of U than $U'X+1$ in the time interval $[t,t']$ or that user must be found on a shortest path that does not lead over U_f but over friends prior to U_f in $Flist_U$. However, that is again a contradiction to Invariant 1 and the definition of $x(U)$.
   Furthermore, since the path with maximal weight to U'_{X+1} leads over U_f, the merge operation correctly adjusts U'_{X+1} friendship strengths, if necessary—if there was no change to the friendship strength, i.e. the shortest path is a unchanged direct edge to U'_{X+1} from U_f, there is also no change for the entry with U'_{X+1} in $Flist_U$. Hence, the entry with U'_{X+1} must be correct at time t' because either it is correctly adjusted or it was already correct at time t.

It follows:

$$\begin{aligned}
Flist'_U[0 : x'(U) - 1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U) - 1] \\
Flist'_U[x'(U)] &= (U_{ff}, s_{ff}) \text{ // (correctly inserted)} \\
&= Flist_{U,t'}[x'(U)] \\
Flist'_U[x'(U) + 1] &= \begin{cases} Flist_U[x(U) + 1] & \text{if correct at } t \\ (U'_{X+1}, s'_{X+1}) & \text{if } s_{X+1} \text{ increased} \end{cases} \\
&= Flist_{U,t'}[x'(U) + 1]
\end{aligned}$$

(c) At time t, user U_{ff} is already located at position $x'(U)$, but U'_{X+1} is not yet a friend, i.e. $\notin Flist_{U,t}$, or not a that good friend, i.e. $pos_U(U'_{X+1}) > x(U)$.

In this case, the merge operation on U_f correctly adjusts U_{ff}'s friendship strength at position $x'(U) = x(U) + 1$ and correctly inserts U'_{X+1} into $Flist_U$ at position $x'(U) + 1 = x(U) + 2$. It immediately follows:

$$\begin{aligned}
Flist'_U[0 : x'(U) - 1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U) - 1] \\
Flist'_U[x'(U)] &= (U_{ff}, s'_{ff}) \text{ // (correct merge)} \\
&= Flist_{U,t'}[x'(U)] \\
Flist'_U[x'(U) + 1] &= (U'_{X+1}, s'_{X+1}) \text{ // (correctly inserted)} \\
&= Flist_{U,t'}[x'(U) + 1]
\end{aligned}$$

(d) At time t, both users U_{ff} and $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ are already at position $x'(U)$ or $x'(U) + 1$, respectively, and thus, $pos_U(U_{ff}) = x(U)$ and $pos_U(U'_{X+1}) = x'(U) + 1$.

In this case, both users at the positions $x'(U) = x(U) + 1$ and $x'(U) + 1 = x(U) + 2$ do not change in $Flist_{U,t'}$ compared to time t but the friendship strength of at least U_{ff} does. In any case, the merge operation correctly adjusts both values if necessary. Again, it immediately follows:

$$\begin{aligned}
Flist'_U[0 : x'(U) - 1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U) - 1] \\
Flist'_U[x'(U)] &= (U_{ff}, s'_{ff}) \text{ (correct merge)} \\
&= Flist_{U,t'}[x'(U)] \\
Flist'_U[x'(U) + 1] &= (U'_{X+1}, s'_{X+1}) \text{ (correct merge)} \\
&= Flist_{U,t'}[x'(U) + 1]
\end{aligned}$$

Hence, in all four cases (a), (b), (c) and (d) Invariant 2 holds: **A'** $\implies$ **B'**

II: $\neg$ **A'** . The path to $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ with the maximal weight leads over $U_{ff} \in Flist_{U,t'}[x'(U)]$

From Invariant 1, we know $Flist_U[0 : x'(U)]$ is correct. Now, we again have to distinguish the possible positions of U_{ff} at time t. Since U_{ff} is found on a shortest path over $U_f \in Flist_U[x(U)]$, it follows, at time t, U_{ff} cannot be located prior to U_f in U's friendship list. Therefore, we only have to distinguish the following two cases:

1. At time t the user U_{ff} is *not* in $Flist_{U,t}$ or $pos_U(U_{ff}) > x(U)+1$.

   We may assume, U_{ff} does not need to be updated and there is no need for a merge operation on U_{ff}, i.e. $\neg$**C'** $\wedge$ $\neg$**D'**.

   By assumption, at time t' the user U'_{X+1} is the next best friend of U after U_{ff} and additionally, is found on a shortest path over U_{ff}. In this case, U'_{X+1} must be a direct successor of U_{ff} or there would be a better friend of U_{ff} on the shortest path to U'_{X+1}. However, such a user cannot exist: (a) if such a user existed and were an even better friend with respect to U than U_{ff}, the shortest path to U'_{X+1} could not lead over U_{ff} which is a contradiction to our assumption and (b) if such a user existed and were not a better friend with respect to U than U_{ff}, she would at least be a better friend than U'_{X+1} because of the shorter path over U_{ff}. Again, this is a contradiction to our assumption that U'_{X+1} is U's next best friend after U_{ff}. Furthermore, U'_{X+1} must be a already in U_{ff}'s friendship list at time t due to $\neg$**C'** $\wedge$ $\neg$**D'**.

   Consequently, the entry with U'_{X+1} in U_{ff}'s friendship list exists and is correctly merged into U's friendship list.
   It immediately follows:

$$\begin{aligned}
Flist'_U[0 : x'(U)-1] &= Flist_U[0 : x(U)] \\
&= Flist_{U,t}[0 : x(U)] \\
&= Flist_{U,t'}[0 : x'(U)-1] \text{ (due to Inv. 1)} \\
Flist'_U[x'(U)] &= (U_{ff}, s_{ff}) \text{ (due to correct merge)} \\
&= Flist_{U,t'}[x'(U)] \text{ (due to II.1)} \\
Flist'_U[x'(U)+1] &= (U'_{X+1}, s'_{X+1}) \text{ (due to correct merge)} \\
&= Flist_{U,t'}[x'(U)+1] \text{ (due to II.1)}
\end{aligned}$$

2. At time t it is already true that $pos_U(U_{ff}) = x(U)+1$

   As a consequence, only the friendship strength s_{ff} of the user $U_{ff} \in Flist_{U,t}[x(U)+1]$ can have changed but U_{ff} is still the same best friend at position $x'(U) = x(U)+1$ and time t'.

   We again may assume, U_{ff} does not need to be updated and there is

no need for a merge operation on U_{ff}, i.e. ¬**C'** ∧ ¬**D'**.

Since in addition, the shortest path to $U_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ leads over U_{ff} by assumption, the entry in $Flist'_U$ must be correct for the same reasons as in the previous case II.1. It immediately follows:

$$\begin{aligned} Flist'_U[0 : x'(U) - 1] &= Flist_U[0 : x(U)] \\ &= Flist_{U,t}[0 : x(U)] \\ &= Flist_{U,t'}[0 : x'(U) - 1] \text{ (due to Inv. 1)} \\ Flist'_U[x'(U)] &= (U_{ff}, s_{ff}) \text{ (due to correct merge)} \\ &= Flist_{U,t'}[x'(U)] \text{ (due to II.2)} \\ Flist'_U[x'(U) + 1] &= (U'_{X+1}, s'_{X+1}) \text{ (due to correct merge)} \\ &= Flist_{U,t'}[x'(U) + 1] \text{ (due to II.2)} \end{aligned}$$

Hence, in both cases Invariant 2 holds: ¬ **A'** ∧¬ **C'** ∧¬ **D'** $\implies$ **E'**

- $Flist'_U[x'(U) + 1] = (U_{ff}, s_{ff})$

  From Invariant 1, we may assume that at $Flist_U[0 : x'(U)]$ is correct at time t' and Invariant 2 holds at time t. Next we show that Invariant 2 also holds at time t'.

  I: **A'** . The path with maximal weight to $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ does *not* lead over $U'_X \in Flist_{U,t'}[x'(U)]$

  By assumption, $x(U) = pos_U(U_f)$ at time t or there would be no merge operation. Furthermore, at time t' the shortest path to U_{ff} leads over U_f. Otherwise, either U_{ff} would not be merged into U's friendship list if there was already a shorter path known to U at time t or in the time interval $(t, t']$ a friendship update must have happened for U or one of her friends with a stronger friendship strength than U_f. However, the first is a contradiction to our assumption about U_{ff} and the second contradicts the definition of $x(U)$.
  Since by assumption U_{ff} is merged at position $x'(U) + 1$ and the shortest path to U'_{X+1} does not lead over U'_X, it immediately follows from Invariant 1 that the entry in $Flist_U$ with U'_X is correct and according to Lemma 10.7 that U_{ff} is a direct successor of U_f and the corresponding friendship strength in $Flist_U$ is correct, too. Therefore,

  $$\begin{aligned} Flist'_U[0 : x'(U)] &= Flist_U[0 : x(U) + 1] \\ &= Flist_{U,t'}[0 : x'(U)] \text{ (due to Inv. 1)} \\ Flist'_U[x'(U) + 1] &= (U'_{X+1}, s'_{X+1}) \text{ (due to correct merge)} \\ &= Flist_{U,t'}[x'(U) + 1] \text{ (due to Lemma 10.7)} \end{aligned}$$

  Hence, Invariant 2 holds: **A'** $\implies$ **B'**

  II: ¬ **A'** . The path to $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ with the maximal weight leads over $U'_X \in Flist_{U,t'}[x'(U)]$

In this case, the path with maximal weight to U_{ff} leads over U_f, too, or U_{ff} would not be merged into U's friendship list. Hence, the path with maximal weight from U to U'_X leads also over U_f or the weight of the shortest path to U_{ff} could not be maximal over U_f which contradicts our assumption ¬**A'**.

In addition, $U'_X = U_{X+1} \in Flist_U U, t[x(U)+1]$ because the merge operation can only change entries with users subsequent to U_{ff} and there also could not be a friendship update at a time $t_x \in (t, t')$ such that the entry at $Flist_{U,t}[x(U)+1]$ changed because of the definition of $x(U)$, our assumption $x(U) = pos_U U_f$, and eventually because of the correctness of Invariant 1.

Moreover, U_{ff} must be a direct successor of U'_X and U'_X must be a direct successor of U_f. Assuming otherwise, then there exists an intermediate user U_i on the path with maximal weight from U_f to U'_X or to U_{ff}, respectively. That means, U_i is a better friend to U_f than U_f's friends U'_X or U_{ff}, respectively. Furthermore, since U_i was not merged into $Flist_U$ (or her merge position in $Flist_U$ would have been located prior to U_{ff} which contradicts our assumption that U_{ff} is the best friend of U_f who is merged into $Flist_U$), the user U_i must be a even better friend to U than U_f. But then, the path from U over U_i to U'_X or U_{ff} that does not lead over U_f would be shorter and its weight higher. However, this is a contradiction to our assumption ¬ **A'**.

Again, we may assume that neither a condition for an update on U'_X nor a merge operation on U with U'_X is true, i.e. ¬**C'** ∧ ¬**D'**.

According to Invariant 1, U'_X is correct with respect to time t' and $Flist'_U$. Furthermore, as there is no need for an update operation, all direct edges of U'_X and their weights must be correct and found in U'_X's friendship list. Since $TS_{U'_X} = 0$, there is no need for a merge operation on U with U'_X and all direct friends of U'_X are already correctly known to U at time t, too. Consequently, U'_{X+1} in $Flist'_U$ must be correct as well.

Note: At time t, there must have already existed a path with maximal weight over a direct edge from $U'_X == U_{X+1}$ to U'_{X+1}. Otherwise, at some time $t_x \in (t, t']$ there is a friendship update to U'_X such that a new path with maximum weight to U_{ff} exists. However, in that case, at time t' there would be still a need for an update operation on U'_X or a merge operation which is a contradiction to our assumption.

It follows,

$$\begin{aligned} Flist'_U[0 : x'(U)] &= Flist_U[0 : x(U)+1] \\ &= Flist_{U,t'}[0 : x'(U)] \text{ (due to Inv. 1)} \\ Flist'_U[x'(U)+1] &= (U'_{X+1}, s'_{X+1}) \text{ (due to correct merge)} \\ &= Flist_{U,t'}[x'(U)+1] \\ &\text{(due to ¬C' ∧ ¬D' and } x(U) = pos_U(U_f)) \end{aligned}$$

Hence, Invariant 2 holds: ¬ **A'** ∧¬ **C'** ∧¬ **D'** $\implies$ **E'**

- $Flist'_U[x'(U)+i] = (U_{ff}, s_{ff})$ with $i > 1$

  I: **A'**. The path with maximal weight to $U'_{X+1} \in Flist_{U,t'}[x'(U)+1]$ does not lead over $U'_X \in Flist_{U,t'}[x'(U)]$

  According to Invariant 1 all entries up to U'_X in $Flist'_U$ are correct with respect to time t'. It follows there is no need for an update or a merge operation on any user $U_i \in Flist_{U,t'}[i]$ with $i < x'(U) = x(U)+1$. Hence, all direct edges from such a user U_i must have already been updated and the timestamps of corresponding friendships list must be $\leq TS_U$. Assuming otherwise would contradict the definition of $x'(U)$.

  By assumption, the merge operation does not change the entry with the user $U'_{X+1} = U_{X+2}$ and, thus, that entry cannot have changed since time t. By assumption, at time t' there is no shortest path from U over U'_X to U'_{X+1} but over some user $U_i \in Flist_{U,t'}[i]$ with $i < x'(U)$. Hence, only the following two options are possible at time t':

  (a) At time t', there is a shortest path from U to U'_{X+1} over a direct edge from some user U_i to U'_{X+1} with $U_i \in Flist_{U,t'}[i]$ and $i < pos_U(U_f)$.

  Assuming such a shortest path did not exist at time t, then within the interval $(t, t']$ a friendship update must have happened to a friend $U_i \in Flist_{U,t'}[i]$ with $i < pos_U(U_f)$ such that at time $t_x \leq t'$ there is a shorter path from U to U'_{X+1} which leads over a direct edge from U_i to U_{X+1}. However, according to the definition of $x(U)$, it means, $x(U)$ is equal to $pos_U(U_i)$ at time t which contradicts our assumption $x(U) = pos_U(U_f)$.

  Hence the shortest path from user U over U_i to user U'_{X+1} existed already at time t and it immediately follows that the entry with the user $U_{X+2} \in Flist_{U,t}[x(U)+2]$ is equal to the one with U'_{X+1} at time t'. Moreover, that entry was already correct at time t because all direct edges of users prior the one at position $x(U)$ in $Flist_U$ are up-to-date with respect to time t and correct. Hence, the entry is unchanged in between time t and t' and it follows

  $$\begin{aligned} Flist'_U[0:x'(U)] &= Flist_U[0:x(U)+1] \\ &= Flist_{U,t'}[0:x'(U)] \text{ (due to Inv. 1)} \\ Flist'_U[x'(U)+1] &= Flist_U[x(U)+2] \text{ (due to correct merge)} \\ &= Flist_{U,t}[x(U)+2] \text{ (due to } U_i \rightarrow U'_{X+1}) \\ &= Flist_{U,t'}[x'(U)+1] \\ &\text{(due to } U'_{X+1} = U_{X+2} \text{ and (a))} \end{aligned}$$

  (b) At time t', there is a shortest path from U to U'_{X+1} leading over U_f.

  Consequently, there is a direct edge from U_f to U'_{X+1} at time t', which will be proved by contradiction.

Assuming (a) is not true and also that there is no direct friendship edge $(U_f \rightarrow U'_{X+1})$ but an indirect shortest path from U_f to U'_{X+1} means that there must be direct successor of U_f who is a better friend of U_f than U'_{X+1}.
From Definition 10.11 follows that all direct successor of U_f must be correctly known in $Flist_{U_f}$ before the merge operation is applied since pending friendship updates are applied to U_f first. Furthermore, any user unknown to U or any friend of U in $Flist_U$ with a lower friendship strength than U_f (i.e. users located subsequently to U_f in $Flist_U$), but who are successors of U_f on the shortest path to U'_{X+1}, have to be located prior to U'_{X+1} in $Flist_{U_f,t'}$ and would be merged in U's friendship list at a position prior to U'_{X+1}, too, according to the Definition 10.11 of a merge operation.
Since all direct edges of U_f are correctly known in $Flist_{U_f}$ and appropriately merged into $Flist_U$, all direct friends of U_f who are better friends than U'_{X+1} have to be located in $Flist_U$ between U_f and U'_{X+1}, including the direct friend of U_f who is part of the shortest path to U_{X+1}.
At time t', there is only one user U'_X at position $x'(U)$ in $Flist_U$ who potentially is a predecessor of U'_{X+1} of a shortest path from U leading over U_f. However, from Invariant 1, we know the entry with U'_X in $Flist_U$ is correct with respect to time t' and, thus, is indeed U's next best friend after U_f. Since the path to U'_{X+1} does not lead over U'_X by assumption, it follows from the reasoning above, that U_{X+1} is a direct successor of U_f.

Finally, since U_{ff} is the best user from U_f's friendship list that is inserted into $Flist_U$ at a position later than U'_{X+1} by the merge operation, the (better) direct friend U'_{X+1} of U_f known to U already at time t. Hence, the corresponding entry in $Flist_U$ must have been already correct at time t or the merge operation would have modified it by definition.
Therefore,

$$
\begin{aligned}
Flist'_U[0 : x'(U)] &= Flist_U[0 : x(U) + 1] \\
&= Flist_{U,t'}[0 : x'(U)] \text{ (due to Inv. 1)} \\
Flist'_U[x'(U) + 1] &= Flist_U[x(U) + 2] \text{ (due to correct merge)} \\
&= Flist_{U,t}[x(U) + 2] \\
&\text{(due to } e = (U_f \rightarrow U_{X+2})) \text{ and no update for } e \\
&= Flist_{U,t'}[x'(U) + 1] \\
&\text{(due to } U'_{X+1} = U_{X+2} \text{ and } pos_U(U_{ff})
\end{aligned}
$$

Hence, Invariant 2 holds: **A'** $\implies$ **B'**

II: $\neg$ **A'** . The path to $U'_{X+1} \in Flist_{U,t'}[x'(U) + 1]$ with the maximal weight leads over $U'_X \in Flist_{U,t'}[x'(U)]$

We may assume again there is no need for an update operation on $U'_{X'}$ or a merge operation on U with $U'_X = U_{X+1}$, i.e. ¬**C'** ∧ ¬**D'**.

In this case, the entry with $U_{X+1} \in Flist_U$ must have already been correct at time t because by definition of a merge operation $U'_X = U_{X+1}$ and by Invariant 1, U'_X is correct at time t'. In addition, no friendship update can have happened to any user prior to U_X in $Flist_U$ in the time interval $[t, t')$ due to $x(U) = pos_U(U_f)$ and by assumption, the merge operation only affects entries at later positions than even $x'(U)+1$. Therefore, there cannot have been a change for the entry at position $x'(U)$ since time t.

From the assumption ¬ **A'** it follows that the shortest path from U to U'_{X+1} leads over a direct edge from U'_X. Assuming otherwise, then there exists a direct successor of U'_X on the shortest path from U'_X to U'_{X+1} which is a better friend of U than U'_{X+1}. However, the shortest path from U to such an intermediate user must also lead over U'_X or the path to any subsequent user on such a path would be shorter than the one over U'_X, too, which is a contradiction to our assumption ¬**A'**. Therefore, no such intermediate user can exist and U'_{X+1} is a direct friend of U'_X.

From ¬**C'** it follows that there is no pending friendship update for U'_X and all entries of direct successors of U'_X are correct in $Flist_{U'_X}$. Since at time t', by definition of the merge operation, $tp'_U = x'(U) = pos_U(U'_X)$, it follows from our assumption ¬**D'**, that the timestamp of U'_X's friendship list $TS_{U'_X}$ is equal to 0. It follows there never was a friendship update for U'_X. Hence, all direct successors of U'_X must have been already known by U at time t and, thus, all direct friends of U_{X+1} must be correctly located in $Flist_U$.

Since $U'_X = U_{X+1}$ and the shortest path from U to U'_{X+1} leads over a direct edge from U'_X at time t' that already existed at time t, it follows $U'_{X+1} = U_{X+2}$. Moreover, as U'_X is correct and all its direct friends are correctly known in $Flist_U$ at time t, also $U'_{X+1} = U_{X+2}$ must have been correct at time t. Finally, it follows

$$
\begin{aligned}
Flist'_U[0 : x'(U)] &= Flist_U[0 : x(U) + 1] \\
&= Flist_{U,t'}[0 : x'(U)] \text{ (due to Inv. 1)} \\
Flist'_U[x'(U) + 1] &= Flist_U[x(U) + 2] \text{ (due to correct merge)} \\
&= Flist_{U,t}[x(U) + 2] \\
&\text{(due to } e = (U'_X \to U'_{X+1})) \\
&= Flist_{U,t'}[x'(U) + 1] \\
&\text{(due to } U'_{X+1} = U_{X+2} \text{ and } pos_U(U_{ff})
\end{aligned}
$$

Hence, Invariant 2 holds: ¬ **A'** ∧¬ **C'** ∧¬ **D'** $\implies$ **E'**

Hence, Invariant 2 holds for all users U and $U.merge(U_f)$:

(**A** $\implies$ **B**) ∨ (¬**A** ∧ ¬**C** ∧ ¬**D** $\implies$ **E**)

□

10.7 Update, Merge – $\forall U_i \neq U : U.update(), U.merge(U_f)$

Finally, we show that both invariants introduced in Section 10.4 also hold for each user $U_i \neq U$ for either an update operation or a merge operation on U.

Theorem 10.5. *Invariant 1 holds $\forall U_i \neq U$ and for an update or a merge operation on U.* ∎

Proof. For any operation on U, Definition 10.12 defines $\forall U_i \neq U$: $Flist'_{U_i} = Flist_{U_i}$, $tp'_{U_i} = tp_{U_i}$, $TS'_{U_i} = TS_{U_i}$.

According to Lemma 10.2, $x'(U_i) = x(U_i)$ and according to Definition 10.6, $x(U_i) \leq pos_{U_i}(U)$ if $U \in Flist_{U_i}$. From this, it immediate follows that no entry prior to $x(U_i)$ can have changed and because of Definition 10.10 and 10.11 for update and merge operations the entry with U itself also cannot have changed by any operation on U. A shortest path from U_i to U is always shorter than a path to U's friends which includes U. The same is obviously true, if U is not a friend of U_i at all, i.e. $U \notin Flist_{U_i}$.

As a consequence, $Flist'_{U_i}$ is unchanged and correct up to position $x'(U_i)$ as $x'(U) = x(U)$, $TS'_{U_i} = TS_U$, $Flist'_{U_i} = Flist_{U_i}$ and since we may assume Invariant 1 holds for U_i right before any operation on U starts.

It follows,

$$
\begin{aligned}
Flist'_{U_i}[0 : x'(U_i)] &= Flist_{U_i}[0 : x(U_i)] \\
&\quad \text{(due to } x'(U_i) = x(U_i) \text{ and by definition)} \\
&= Flist_{U_i, TS_{U_i}}[0 : x(U_i)] \text{ (correct by assumption)} \\
&= Flist_{U_i,t}[0 : x(U_i)] \text{ (due to } t = TS_{U_i}) \\
&= Flist'_{U_i,t'}[0 : x'(U)] \text{ (due to } t' = t \text{ and } x'(U_i) = x(U_i))
\end{aligned}
$$

□

Theorem 10.6. *Invariant 2 holds $\forall U_i \neq U$ and for an update or a merge operation on U.* ∎

Proof. According to Invariant 1, $\forall U_i \neq U : Flist'_{U_i}$ is correct up to position $x'(U_i)$. For the discussion about the entry at the position $x'(U_i)+1$, we have to distinguish the possible cases of U's position relative to $x(U_i)$ in a user's friendship list $Flist_{U_i}$.

Remember: According to Lemma 10.2, $\forall U_i \neq U : x'(U_i) = x(U_i)$ and from the definition of $x(U_i)$, it follows, $x(U_i) \leq pos_{U_i}(U)$.

Moreover, $Flist'_{U_i} = Flist_{U_i}$, $TS'_{U_i} = TS_{U_i}$ and $tp'_{U_i} = tp_{U_i}$ by Definition 10.10 and Definition 10.11 of an update or merge operation on U for all user $U_i \neq U$. Furthermore, we may assume at time $t = TS_{U_i}$ that both invariants hold.

- Case 1: $x(U_i) < pos_{U_i}(U, s)$

  I: **A'**. The path with maximal weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i) + 1]$ does *not* lead over $U'_X \in Flist_{U_i,t'}[x'(U_i)]$

  Due to $x'(U_i) = x(U_i)$ and Lemma 10.3, it follows: $A' \implies A$, i.e. the shortest path does not lead over a edge $(U_X \to U_{X+1})$ at time t. Hence, at

time t, the entry at position $x(U)+1$ was already correct by assumption that Invariant 2 is correct at time t.
Since a merge operation on U can only affect entries in $Flist_{U_i,t}$ at later positions than $pos_{U_i}(U) \geq x(U_i)+1$, and $t' = t$, $x'(U_i) = x(U_i)$, it follows $Flist'_{U_i}[x'(U_i)+1]$ must still be correct.
Thus, **A'** $\implies$ **B'**.

II: ¬**A'**. The path with maximal weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i)+1]$ leads over $U'_X \in Flist_{U_i,t'}[x'(U_i)]$

Here, the same argumentation is true as before. The operation on U cannot change anything for $Flist_{U_i,t}$ up to $pos_{U_i}(U) \geq x(U_i)+1$. Hence, the entries up to $x'(U_i)+1 = x(U)+1$ cannot have changed for $t' = t$ due to the update or merge operation on U.

Since $t' = t$ and $x'(U) = x(U)$ and we may assume that Invariant 2 holds at $Flist_{U_i}$ right before an operation on U, then Invariant 2 must still hold for $Flist'_{U_i}$ because $Flist'_{U_i} = Flist_{U_i}$.

- Case 2: $x(U_i) = pos_{U_i}(U,s) < tp_{U_i}$

Note: This case can only occur when TS_{max} has changed since the last time a query on U_i discovered U, i.e. $tsmax_U < TS_{max}$ and at least one friendship update related to U occurred in G. Otherwise, there would be no need for an operation on U because of $pos_{U_i}(U,s) < tp_{U_i}$ and Definition 10.13. Hence, the reason why $tsmax_U$ is part of the state description σ_U (see Definition 10.7).

I: **A'**. The path with maximal weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i)+1]$ does not lead over $U == U'_X \in Flist_{U_i,t'}[x'(U_i)]$
Due to our assumption that there is no direct edge from $U == U'_X$ to U'_{X+1}, there cannot have been such edge before the update operation on U because of $x'(U_i) = x(U_i)$ and according to Lemma 10.3.
Hence,

$$\begin{aligned} Flist'_{U_i}[x'(U_i)+1] &= Flist_{U_i}[x(U_i)+1] \text{ (due to Case 1, by def.)} \\ &= Flist_{U_i,t}[x(U_i)+1] \\ &\text{(due to A' and Lemma 10.3)} \\ &= Flist_{U_i,t'}[x'(U_i)+1] \text{ (due to x'(U)=x(U), } t' = t) \end{aligned}$$

Furthermore, $Flist_{U_i,t}[x(U_i)+1]$ was correct at time t and so it is at $t' = t$.
It follows: **A'** $\implies$ **B'**

II: ¬**A'**. The path with max. weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i)+1]$ leads over $U == U'_X \in Flist_{U_i,t'}[x'(U_i)]$
This is an obvious case. According to Lemma 10.4, at time t', we know, $TS'_U > TS'_{U_i}$.
Hence, ¬ **A'** $\implies$ **D'**

- Case 3: $x(U_i) = pos_{U_i}(U, s) = tp_{U_i}$

  A' : The path with maximal weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i) + 1]$ does not lead over $U == U'_X \in Flist_{U_i,t'}[x'(U_i)]$

  Again, because of Lemma 10.3 and $x'(U_i) == x(U_i)$, it follows that there is also no such shortest path from $U_X \in Flist_{U_i,t}[x(U_i)]$ to the user $U_{X+1} \in Flist_{U_i,t}[x(U_i) + 1]$ at time t.
  As $x(U_i) = tp_{U_i}$ and there is no direct edge from $U_X = U'_X$ to $U_{X+1} = U'_{X+1}$, it follows that at time t the path with maximum weight to U_{X+1} leads over a user at $Flist_U[j]$ with $j < x(U_i)$. This path must have been already correct at time t or otherwise it would contradict the definition of $x(U_i)$.
  Since the entry at $Flist_U[x(U_i) + 1]$ was already correct at time t and due to **A'** the operation on U cannot have changed it, and with $t' = t$, the same entry still must be correct after the operation on U.
  It follows,

  $$\begin{aligned} Flist'_{U_i}[x(U_i) + 1] &= Flist_{U_i}[x(U_i) + 1] \text{(due to Case 3)} \\ &= Flist_{U_i,t}[x(U_i) + 1] \text{ (due to A' and Lemma 10.3)} \\ &= Flist_{U_i,t'}[x'(U_i) + 1] \text{ (due to } t' = t, x'(U_i) = x(U_i)) \end{aligned}$$

  A' $\implies$ **B'**

  $\neg$**A'** : The path with max. weight to user $U'_{X+1} \in Flist_{U_i,t'}[x'(U_i) + 1]$ leads over $U == U'_X \in Flist_{U_i,t'}[x'(U_i)]$

  Again, this is an obvious case, because $TS'_U > 0$ and $x'(U_i) = tp'_{U_i}$ by assumption, it follows, $\neg$ **A'** $\implies$ **D2'**

Finally, in all possible cases, we could show, that Invariant 2 holds for an update or merge operation on U and all users $U_i \neq U$. □

10.8 U.get_Friends()

The final step to prove the correctness of our algorithm for finding always the true *top-k* friends of some user U is done by induction over the position i of the last identified next best friend. *Note:* As our algorithm traverses a friendship list always sequentially by starting from the first entry and increases tp_U to i whenever i exceeds tp_U, the current *i-th* best friend for U is always located at a position $\leq tp_U$ in $Flist_U$.

Theorem 10.7. *Our Algorithm U.get_Friends() presented in Listing 5 works correctly and retrieves for each user U her* top-k *best friends.* ∎

Proof. **Induction Basis.** $i = 0$: When $i = 0$, we want to identify the best friend of U. According to Property 10.3, the best friend is a direct successor of U. At query time, all new edges for U are updated and new friends and their friendship strengths are correctly inserted into U's friendship list. Hence, the very first entry $Flist_U[0]$ is correct according to the correctness of Invariant 1 with respect to U.update().

Hence, our algorithm correctly identifies with $U.get\_Friend(0)$ the best friend of user U.

Induction Step. $i \rightarrow i+1$: Assuming our algorithm has already correctly identified the first i friends, then, we have to show that the algorithm correctly identifies the next best friend. In other words, after having identified the *i-th* best friend of U, we next want to correctly identify the *(i+1)-th* friend.

We have to distinguish the following cases:

- $i < tp_U$:

  From Invariant 1 and by applying the induction hypothesis, we are allowed to assume that all entries in $Flist_U$ up to position i are correct and hence, all necessary update or merge operations have been already applied on users found at these entries. Therefore, $x(U)$ can only be equal to or greater than i by definition of $x(U)$.

  When $i < x(U) \leq tp_U$, then there is no need for an update or merge operation on $U_i \in Flist_U[i]$ and $i+1 \leq x(U) \leq tp_U$. Hence, according to Invariant 1 the *(i+1)-th* entry is correct and the user at that position is indeed U's next best friend.

  When $i = x(U) < tp_U$, then there is either (a) a need for an update operation on U_i or (b) a need for a merge operation on U with U_i according to the definition of $x(U)$. However, the checks for $U.needs\_update()$ or $U.needs\_merge(U_i)$ defined by our algorithm exactly match the definition of $x(U)$. Therefore, before identifying the next best friend, our algorithm applies the operation $U_i.update()$ or $U.merge(U_i)$, respectively. In addition, an update operation on a friend U_i of U causes always a merge operation on U with U_i.

  By definition of $U.merge(U_i)$, the timestamp validity pointer tp_U is set to $i+1$ which correctly corresponds to the position of the next best friend of U according to Invariant 1.

  Hence, in any case, our algorithm correctly identifies the *i+1*-th friend of U.

- $i = tp_U$:

  In this case, by definition of $x(U)$ and by applying the induction hypothesis that all entries up to position i are correct, with all necessary merge and update operation have been applied, it follows, $x(U) = tp_U$.

  Moreover, when $i == x(U)$, our algorithm identifies as a next step the friend at position $x(U)+1$. The path with maximal weight to this next best friend leads either (a) over a user in $Flist_U$ prior to the *i-th* position and not over the *i-th* friend or (b) the *(i+1)*-th best friend of U is a direct successor of U's *i-th* friend.

  When there is no need for an update or merge operation on the *i-th* friend, then according to Invariant 2, the *(i+1)*-th entry in $Flist_U$ is already correct in both cases (a) and (b). Hence, our algorithm can safely increase tp_U to $i+1$ and return the *i+1* best friend of U.

  When (b) is true but there is a need for an update operation or a merge operation on the *i-th* user $U_i \in Flist_U[i]$ as defined by $x(U)$, then, the check for $U_i.needs\_update()$ or $U.needs\_merge(U_i)$ is true as these methods exactly

match the definition of $x(U)$ and, thus, the corresponding operation is applied. Furthermore, an update operation on U_i always causes a merge operation on U with U_i.

Hence, in any case, after the merge operation on U, the next best friend can be correctly identified because then tp'_U is equal to $x'(U) = i + 1$ and according to Invariant 1 all entries up to $x'(U)$ are correct for any update or merge operation.

Finally, Invariant 2 guarantees to identify a need for a merge operation on U with U_i or the need for an update operation on U_i, and therefore, our algorithm correctly identifies the $i + 1$-th friend of U.

Both, induction basis and induction step have been proved. Hence, it has been proved that our algorithm correctly works and always identifies for any i the *i-th* best friends of a querying user U.

□

Part C
Peer-To-Peer User Networks

11 Introduction

For technological as well as business reasons, today's social tagging networks are typically implemented in a centralised way on servers or server farms belonging to single companies. However, Peer-to-Peer (P2P) Systems naturally provide a suitable environment for hosting social tagging networks with users and their data being mapped to the peers in the P2P system. Such a distributed network of users introduces the additional challenge of how to counter misbehaviour like cheating of users without reverting to any centralised controlling mechanisms.

In this chapter, we provide a distributed algorithm which allows to compute authority scores over social networks in a Peer-to-Peer environment and which correctly works even in the presence of cheating users.

11.1 Motivation

In Web 2.0 applications like social tagging networks, users are not only content consumers but also content providers. As described in Part A, Section 3.2, e.g. users publish own photos in *Flickr.com*, links to web pages (bookmarks) in *Delicious.com*, and information about books they own or have read in *LibraryThing.com*. In addition, users tag, comment or rate the items in the network, adding valuable attributes to them and enabling social recommendations and social search strategies based on this user-provided feedback—as shown by our socially enhanced search and exploration framework *SENSE* presented in Part A of this work.

Since users' own contents are an intrinsic feature of social tagging networks, naturally, this motivates the idea that contents should reside on the users' own computers rather than being given away and stored at a centralised service on the web. The approach of keeping the users' data on the users' own computers is also advantageous in terms of lower vulnerability to privacy breaches, and other forms of attacks, censorship, or manipulation and even to performance bottlenecks because of the joint power of potentially hundreds of millions of computers. Due to its scalability and the autonomy of peers, a peer-to-peer system exactly features such an approach.

Social tagging networks can be cast into P2P systems by mapping the users of the network to individual peers. In such a scenario, user-specific contents like their photos in *Flickr.com*, bookmarks in *Delicious.com* or books in *LibraryThing.com*, etc., or the users' lists of friends, the information about private interactions, personal recommendations, tags and subjective ratings reside on the users' private computers—each acting as a peer in the corresponding P2P system. Hence, the peers in the network are collaborating and forming a distributed search engine to enable searching for and browsing of other users' contents.

Of course, in such a distributed setting where users have to collaborate to identify authoritative information, the risk of misbehaving or cheating users arises.

The notion of authority is not only bound to user graphs of social tagging networks and differs with respect to the graph and the defined semantic of the associated node relations. However, it typically refers to a combination of factors like trustworthiness [60], reputation [78], importance or prestige of entities [87, 71], or the level of attention. Hence, misbehaving or cheating users in social networks may try to influence authority computations to promote own preferences or demote those of others. Therefore, especially in a distributed setting like in a P2P system, authority computations over social networks in the presence of users trying to influence the results are a challenging task.

Before introducing our algorithm for countering cheating in such a P2P environment, we discuss related work.

11.2 Related Work

Link analysis for authority scoring is a well studied topic with rich literature. One of the best known models is PageRank, introduced by Brin and Page [31], and further explored in many other works. A good survey can be found at [90].

With the advent of the Web 2.0, there has been much research on adapting existing retrieval and mining techniques from web search to online communities. [14] discusses the challenges of searching and ranking in social communities. Many methods for ranking based on the analysis of social links have been developed. [70] proposes *FolkRank* for identifying important users, data items, and tags. [133] compares different methods for identifying authoritative users with high expertise. [19] introduces *SocialPageRank*, to measure page authority based on its annotations, and *SocialSimRank* for the similarity of tags, and [111] considers the link structure defined by the relationship among users to improve the retrieval quality. Another interesting work is [93] where the authority scores are computed on the graph defined by the users' browsing experience, and [107] where the authors devise an efficient algorithm to estimate PageRank in large graphs.

In the context of P2P networks [120], work has been devoted to techniques for distributed link analysis. In [128], Wang and DeWitt presented a framework, in which the authority score of each page is computed by performing the PageRank algorithm at the web server that is the responsible host for the page, based only on the intra-server links. They also assign authority scores to each server in the network, based on the inter-server links, and then approximate global PageRank values by combining local page authority scores and server authority values. Wu and Aberer [129] pursue a similar approach based on a layered Markov model. A fundamental approach to distributed spectral decomposition of graphs is given by Kempe and McSherry [82], where distributed link analysis would be a special case of the presented mathematical and algorithmic framework.

In [106], Sankaralingam et al. presented a P2P algorithm in which the PageRank computation is performed at the network level, with peers constantly updating the scores of their local pages and sending these updated values through the network. Shi et al. [116] also compute PageRank at the network level, but they reduce the communication among peers by distributing the pages among the peers according to some load-sharing function. The JXP algorithm, presented in [101] performs local PageRank score computations on the peers' local graph fragments, where each local graph is augmented by a *world node* that represents the locally unknown part of the global graph. Meetings among peers are used for mutual exchange of information about their local graph fragments and to continuously improve each peer's knowledge about its world node.

Several other works have looked at social networks from a decentralised point of view, also in the context of P2P networks. Some of these approaches exploit social links to propose new strategies for content searching [102]. Aspects of user communities have also been considered for P2P search, most notably, for establishing "social ties" between peers and routing queries based on corresponding similarity measures (e.g., similarities of queries issued by different peers). [25] has studied "social" query routing strategies based on explicit friendship relationships and behavioural affinity. [103] has developed an architecture and methods for "social" overlay networks

that connect "taste buddies" with each other. [97] has proposed a community-enhanced web search engine that takes into account prior clicks by community members. [42] has proposed the notion of Peer-Sensitive ObjectRank, where peers receive resources from their friends and rank them using peer-specific trust values.

All the distributed approaches mentioned above require that every site participating in the computation is trustful and the values reported are not manipulated: a strong and rather unrealistic assumption in a P2P environment, given that high authority/reputation scores are desired by users, with potential benefits like higher attention, access traffic, or even income from ads and sales.

There has been much work on establishing reputation systems that would help to assess the quality and trustworthiness of peers. In [95], the authors present a complete overview of the issues related to the design of a decentralised reputation system. EigenTrust [79] is one of the first methods introduced to assign a global trust value to each peers, computed as the stationary distribution of the Markov chain defined by the normalised local trust matrix C where c_{ij} is the local trust value that a peer j assign to a peer j. Extensions towards distributed and non-manipulable EigenTrust computations are presented in [10]. Another framework for reputation-based trust management is presented in [131], where peers give feedback about other peers' good or bad behaviour and various forms of network-wide trust measures can be derived in a P2P-style distributed computation. A general framework for different types of trust and distrust propagation in a graph of web pages, sites, or other entities is introduced in [59].

Instead of trying to identify malicious peers to eliminate their impact on the distributed computation, our algorithm instead is able to correctly compute the PageRank values without the need to identify the cheating sites, as long as the fraction of cheating peers is only a minority. Our only assumption is that peer identities are unforgeable, which can be guaranteed by standard methods of cryptographic security [94]. There is also work on using the link structure to identify spam pages [23] and computing authority scores based only on trusted sites [63, 62]. Our work, however, focuses on malicious behaviour of peers, regardless of the content of the entities they possess. Deciding whether a certain entity contains spam or not is orthogonal to our work, and could as well be incorporated into our framework.

12 Countering Cheating in P2P Authority Computations over Social Networks

In this section, we address the problem of cheating peers in a decentralised computation of authority values, providing a solution being surprisingly simple, i.e. not difficult to implement, and fairly general, i.e. making very few and weak assumptions about peers and the network.

12.1 Overview

Ranking entities, e.g. web pages, users, photos, etc., in social networks, web graphs, and other relational structures is important in many applications such as web search or Web 2.0 applications. A widely used family of measures to analyse authority, trust, or reputation is by computing the principal eigenvector of a matrix derived from the underlying relation (e.g. a weighted adjacency matrix for web pages or a weighted friendship/acquaintance matrix for the users of a social network). The standard algorithm for

computing such authority measures is the Jacobi power-iteration method [90], the most prominent use case being Google's PageRank algorithm. On large graphs, these computations are expensive, especially in terms of memory requirements, so that distributed algorithms for eigenvector analysis are an attractive option. Moreover, in some applications like social-network analysis, users are the owners of contents and may care about autonomy and privacy, so that they would ideally keep their parts of the overall contents on their own computers, including e.g. their friendship lists. For instance, in a setting like the *LibraryThing.com* social tagging network, every user may manage her own book libraries, friendship lists, and friends information on her own computer or at least by means of a personalised agent running in the provider's data centre and which is governed by the user's individual policy, e.g. for visibility by other parties and for revoking information. This aspect calls for a peer-to-peer (P2P) approach with decentralised computation spread across largely autonomous, asynchronously communicating peers.

Distributed and P2P algorithms for power iteration and other spectral analyses have received significant attention in prior research, most notably for but not limited to web-graph link analysis, providing *good solutions* such as [45, 82, 101, 106, 116, 128, 129]. A common principle among these distributed analysis algorithms is that peers perform local computations on their, relatively small, local fragments of the underlying graph, and then communicate these local results among each other. For example, approximate authority values for the entities owned by or known to one peer may be propagated along outgoing edges to neighbouring peers [82]. Alternatively, peers could communicate in a batched manner, exchanging locally computed vectors of estimated PageRank or other authority or prestige values to improve other peers' estimates in an iterative manner [101], [102].

However, regardless of the details of such distributed computations, this opens up opportunities for *bad guys*: dishonest peers ignoring the rules of the algorithm and playing their own games. For example, a user may want to artificially boost the importance of a web page or book that she likes, or improve her own social prestige in an undeserved, manipulative manner. The forms of misbehaviour may range from egoistic behaviour and cheating all the way to being malicious and attempting to sabotage the entire social network.

The ways in which peers cheat and misbehave cannot be easily specified, e.g. in the form of an attacker model, and their effects cannot be controlled at all if an unbounded number of peers ignores the rules. However, we can realistically assume that it is indeed only a small minority, or some bounded fraction of all peers that attempts cheating. Even under this assumption, all the previously proposed algorithms could possibly arrive at largely distorted authority and reputation scores that may arbitrarily deviate from the true values that an unmanipulated computation would yield. These algorithms have not got any built-in countermeasure to cheating, and thus, are facing a severe problem in real world P2P user network.

Next, we formally define the problem that has to be solved in P2P authority computations when there are peers in the network which ignore the computational guidelines.

12.2 Problem Definition

We start with briefly summarise the PageRank (PR) algorithm as it is the main tool we use in order to evaluate authority scores.

The basic idea of PR is that if an entity e_l has got a link to an entity e_i then e_l is implicitly endorsing e_i, i.e. giving some importance to entity e_i. How much e_l contributes to the importance of e_i is proportional to the importance of e_l itself.

This recursive definition of importance is captured by the stationary distribution of a Markov chain that describes a random walk over the graph where we start at an arbitrary entity and at each step choose a random outgoing edge from the current entity. To ensure the ergodicity of this Markov chain (i.e. the existence of stationary entity-visit probabilities), additional random jumps to uniformly chosen target entities are allowed with small probability $(1 - \alpha)$. Formally, PageRank is defined for a given directed graph G (representing an ergodic Markov chain) as follows:

Definition 12.1 (PageRank $\pi_G(e_i)$)**.** *For an entity e_i in a directed graph G, the PageRank $\pi_G(e_i)$ of e_i is defined as:*

$$\pi_G(e_i) = \frac{(1-\alpha)}{m} + \alpha \sum_{e_l | e_l \to e_i} \frac{\pi_G(e_l)}{\delta_l}$$

where m is the total number of entities in the link graph, $\pi_G(e_l)$ is the PR score of the entity e_l, δ_l is the outdegree of e_l and $(1 - \alpha)$ is the random jump probability, with $0 < \alpha < 1$. ∎

The sum for calculating the PR of entity e_i ranges over all link predecessors of e_i, and α is usually set to a value like 0.85. In what follows, we shall omit the subscript G in π_G wherever this is not crucial.

12.2.1 P2P Authority Computations & Malicious Peers

PR values are usually computed by initialising a PR vector with uniform values $\frac{1}{m}$, and then applying a power iteration method, with the previous iteration's values substituted in the right-hand side of the above equation for evaluating the left-hand side. This iteration step is repeated until sufficient convergence, i.e. until the PR scores of the high-authority entities of interest exhibit only minor changes.

For referring to the score computed at step t in the power iteration, we introduce the following definition:

Definition 12.2 ($t-$refined Score)**.** *The score computed at iteration step t in the power iteration method is denoted as t-refined score.* ∎

Informally, we can define the problem of computing PageRank scores using the power iteration method in a P2P network (without malicious peers) as follows: A directed graph G is divided into (possibly overlapping) subgraphs, each subgraph G_i being stored at a peer p_i. Initially, peers know only their own subgraphs, yet they wish to compute their entities' PageRank scores with respect to the global graph G.

In the presence of malicious peers, we should be more careful in defining our problem, as such peers might lie not only about the computed score values for own entities but also about the existence of edges as well as nodes in their local graphs (in the spirit of link spammers, see e.g. [62, 34]). In this case the global graph G would no longer be uniquely defined; so the peers can impossibly rebuild the true G, not even in a centralised manner, and there is no way of computing the real scores. To escape this ill-defined problem, we essentially *define* G to be the union (merging) of the peers' local graphs. In the cases when two peers disagree about whether an edge exists or not, we use a simple majority scheme. This motivates the following formal definition of the merged graph $M_{\mathcal{G}}$:

Definition 12.3 (Merged Graph $M_{\mathcal{G}}$)**.** *Let $\mathcal{G} = G_1, \ldots, G_n$ be a collection of directed graphs where $G_i = (V_i, E_i)$. V_i denotes the set of vertices in G_i and E_i denotes the set of directed edges in G_i for all $i = 1, \ldots, n$.*

For each edge $e = u \to v$ let n_e^+ be the number of graphs in the collection $\mathcal{G}$ containing $u \to v$ and let n_e^- be the number of graphs containing both u and v but not $u \to v$.

Then, the graph $M_{\mathcal{G}} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, obtained by merging the graphs in $\mathcal{G}$ is defined as follows:

- $V_{\mathcal{G}} = \bigcup_i V_i$;
- $e = u \to v \in E_{\mathcal{G}}$ *iff* $n_e^+ > n_e^-$. ∎

The definition is also motivated by the fact that if a peer knows the entities u and v it knows also whether there is an edge between them. This is especially true in the case when entities are web pages or users of a social tagging network.

If there is an entity contained only at one peer there is no way to avoid in the formal definition of the problem the implicit assumption that this peer informs correctly about this entity. Hence, we are also implicitly assuming that malicious peers do not change their local graph G_i by adding additional edges and entities, or if they do, that there is a stable state where the majority of peers knows if there's such an edge or entity or not. Computing scores for the "true" global graph would have to simultaneously address the link-spam-detection problem for the graph construction and the peers' cheating behaviour in the distributed power iteration. Therefore, we focus in this work on the well-defined problem of computing the PageRank scores in $M_{\mathcal{G}}$.

12.2.2 Main Issues

In this section, we discuss the problems caused by coalitions of malicious peers and by applying the power iteration method in a fully asynchronous P2P network.

Distributed algorithms for computing authority scores in a P2P setting obey the common principle that peers

- perform computations only on their locally stored subgraph
- communicate their results to other peers

Consequently, for locally computing correct PageRank scores, a peer has to meet other peers in the network to find out about predecessors of the peer's local entities and the predecessors' score contributions.

Coalitions of Malicious Peers

A malicious peer might communicate wrong results to other peers in the network in order to manipulate the authority score computation for certain entities. In the case that more than one peer can communicate a score value for the same entity, the problem becomes even harder since malicious peers might form a coalition such that they all report the same wrong value for an entity without even knowing its real score value.

The problem caused by malicious peers forming a coalition to influence the authority computation of entities is illustrated in Figure 32. It shows a peer p_i with her local subgraph, containing the entities A, B, C and D, and which meets other peers in the

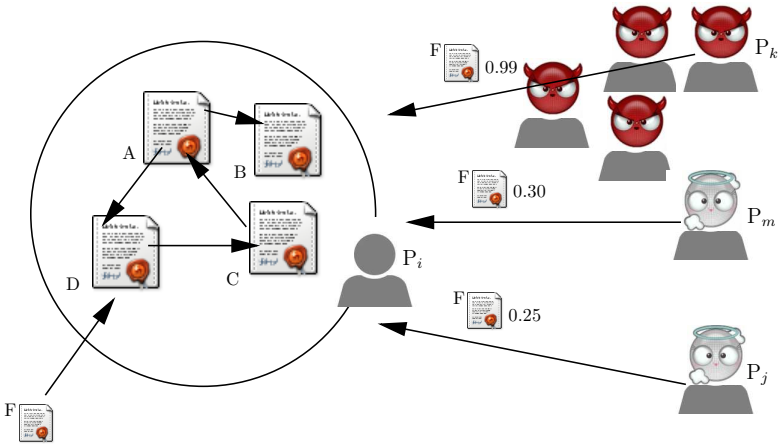


Figure 32: Example for a coalition of malicious peers in an asynchronous P2P network

network in order to find out the score contribution of D's predecessor F. In this example, the malicious peer p_k forms a coalition with 3 other malicious peers to wrongly communicate a too high score value for entity F. In this way, the malicious peers outnumber the two remaining honest peers p_m and p_j reporting also score values for the same entity F.

Asynchrony

As indicated in Figure 32, even when peers are honest, it can happen in a fully asynchronous P2P network that peers do not agree on the same score value for certain entities. In the given example, peer p_j might have not yet finished all necessary iteration steps for entity F or is still missing some information about predecessors of F that peer p_m already knows. Hence, even if peers are honest, their reported score values might not be true in the beginning due to asynchronous computations steps.

12.2.3 Assumptions on the P2P Network

Our algorithm only requires very weak and realistic assumptions on the P2P network. In this section, we introduce the P2P setup for computing authority scores in the presence of malicious peers.

Each peer p_i in the network initially knows only a local subgraph G_i of the network's underlying global graph structure but can increase its knowledge about the global graph $M_{\mathcal{G}}$ by exchanging information with other peers selected at random. The storage capacity of each peer is bounded and typically much smaller than the capacity necessary to store the whole graph G.

A fraction f of the peers in the P2P network is malicious, that is, they may not execute the algorithm as it is presented in the subsequent Section 12.3. However, we make the following assumptions:

1. The majority of peers is not malicious, i.e. $f < 1/2$.

2. There is an unforgeable mechanism for peers' identities.

These assumptions appear to be natural and realistic for a P2P network. In literature there are many studies on how to provide peers with an unforgeable mechanism for their identities. Some of them rely on the authority of a trusted agency (see [51] for a survey), some others tackle this in a different way (see e.g. [22]). Having a small fraction of malicious peers also reflects typical scenarios, besides being necessary for the problem to be solvable.

Additionally, we assume that there is an underlying mechanism which may be invoked by any peer to pick another peer uniformly at random. This abstraction allows us to devise a distributed algorithm for all kinds of P2P networks (see [120] for a survey) where the details of the diverse protocols used, might just blur the essence of the problem. We are considering an asynchronous model where no mechanism is available for global synchronisation among peers.

12.2.4 Problem Statement

The goal of each peer p_i is to compute the stationary distribution $\pi_{M_\mathcal{G}}(j)$ for each entity e_j in G_i. The total cost of the distributed computation is measured in terms of number of meetings performed by all peers in order to have got sufficiently accurate values for the PageRank scores. Formally, the problem is defined as follows:

Definition 12.4 (Problem Statement)**.** *Given a set of n peers containing $f \cdot n$ malicious peers where $0 < f < 1/2$, a set of m entities, a collection of directed graphs $\mathcal{G} = G_1, \ldots, G_n$ over the set of entities with each G_i being stored at peer p_i, a real value $0 < \alpha < 1$, the goal for each peer is to compute $\pi_{M_\mathcal{G}}(e_j)$ for each e_j in G_i.* ∎

As defined in Definition 12.1, the PageRank score of an entity is a linear function of its predecessors scores. In a distributed environment, an entity and its predecessors might be located in different peers allowing malicious peers to spread wrong scores in the network as well as to form coalitions to fool other peers.

12.3 Distributed Algorithm

In addition to provide an asynchronous and distributed implementation of the PageRank scores computation, we also face the difficulty of dealing with malicious peers who may mislead the computation of such scores to turn it to their advantage. We present a distributed algorithm which computes a good approximation of the scores, even when a large part of the network is compromised. Our result does not assume any limitation on the freedom of malicious peers who can lie about the score of any page and even make coalitions with other malicious peers.

12.3.1 Properties

Our algorithm is based on two key principles. First, we introduce a certain amount of redundancy into the network by replicating the entities of interest (i.e. web pages, books, friends) in a randomised manner. Second, whenever two peers exchange information about the authority of an entity, they provide a version history of the entity's previous values, thus enabling the receiving peers to compare values from different peers or at

different times in a meaningful manner. The algorithm works under the realistic assumptions that the fraction of cheating peers is bounded and only a minority, and that peer identities are unforgeable using standard methods of cryptographic security.

The salient properties of our distributed algorithm are the following:

- The algorithm works in a perfectly decentralised and asynchronous manner, can handle arbitrary distributions of the data across peers, makes only weak and reasonable assumptions about the form of how peers cheat or behave maliciously, yet computes non-distorted, correct authority scores.
- The algorithm is practically viable in terms of the necessary degree of replication, convergence speed, and communication overhead. We show this by providing experimental evidence, based on an excerpt of the social-tagging network *LibraryThing.com*.

12.3.2 Design Principles

Any distributed algorithm for our problem must tackle the following two main difficulties:

- Information provided by malicious peers is potentially manipulated and needs to be filtered out; this is complicated by the fact that malicious peers may form a coalition (see Section 12.2.2).
- There is no mechanism for global synchronisation. For example, we cannot and do not want to run a distributed power iteration in a lock-step manner, with every peer making one step for each of its entities and waiting for all peers to complete this step before entering the next round of such steps.

One approach to the first problem is to replicate each entity on the majority of peers. In this way, the exact score of an entity may be retrieved by asking all peers and then computing the majority. However, this approach is not practically viable since it causes very high storage costs by the massive replication and also requires a huge amount of messages to be sent.

Our solution is to make sure that each entity is replicated in only a (sufficiently large) random sample of peers. The main idea is to use randomisation to guarantee that the majority of peers in the random sample is honest with high probability; then the true score could be obtained by computing the majority in the sample. Unfortunately, this solution presents the issue that malicious peers may claim to possess a replica even if this is not the case. This allows them to form coalitions to outnumber the honest peers which actually possess the entity. To overcome this problem, our algorithm makes sure that every peer is responsible (i.e. can "vote") only for a subset of entities. An entity's score computed by a peer is considered only if the peer is responsible for that entity, otherwise it is ignored. The key point is that such an assignment of responsibilities is verifiable by any peer in the system due to unforgeable peer identifiers and a globally known family of hash functions for the assignment. The details are given in Section 12.3.3.

Asynchronous Mode of Operation

We now discuss the main issues of replicating entities to multiple peers caused by the lack of global synchronisation.

Our algorithm is a distributed implementation of the power iteration method where in iteration t, each peer needs to collect a set of $(t-1)$-refined scores in order to compute a t-refined score. For a given entity that the peer is responsible for, the relevant set of scores to obtain from other peers are the ones for the entity's in-link neighbours, i.e. the predecessors in the global graph. In an asynchronous computation, these scores may not be received in the right order as the following example illustrates (and as already indicated in Section 12.2.2, albeit without reference to replicas) :

Example: Consider an entity e at a peer with in-link neighbours a and b where each of a and b are replicated across three peers. Ideally, the peer that wants to compute the t-refined score for e should receive 6 messages with the $(t-1)$-refined scores for the replicas of a and b. But in an asynchronous system, some peers may be slower than others so that the receiving peer may de facto have got scores for a as of iterations $t-1$, $t-2$, and $t-9$ and for b as of iterations $t-3$, $t-4$, and $t-5$.

To solve this issue, each score that peers send to other peers need to be extended with an integer t indicating the step in the power iteration method that the sender has last performed. Now, a naive algorithm could be to let each peer wait until all $(t-1)$-refined scores have arrived before computing its t-refined scores. However, this would lead us back towards a mostly synchronous system.

In order to support full asynchrony and also to speed up the entire computation, we let peers to periodically send for each entity a short history of (improving estimations of) t-refined scores for $t = 0, \ldots, T-1$, where T is in the order of a few tens (e.g. 50). This value of T corresponds to the maximum number of power iteration steps that a centralised link-analysis algorithm needs to compute sufficiently accurate scores. With these history vectors of scores, which are also maintained at each peer for each in-link neighbour of any entity the peer is responsible for, it is now easy to perform meaningful t-refined scores. The procedure is the following:

- The peer computing a t-refined score of an entity e executes one step of the power iteration method by considering the set of available $t-1$-refined scores of the in-link neighbours of e as input.

  Initially, the scores of some in-link neighbours might be missing but eventually all the necessary scores will be collected and a good estimation of e's score will be computed.

- To effectively remove the distorting effects of malicious peers, instead of considering one single t-refined score of a given entity e, the most frequent score among the ones received for e is used in each step of the power iteration method.

  Again, initially the majority of scores received could be manipulated by malicious peers but since the honest peers responsible for e outnumber the malicious ones, eventually, the votes for the true scores will outnumber the manipulated ones, too.

If a peer is responsible for multiple entities, this method is applied separately for each different entity at the peer.

The bottom line of this approach is that it allows for arbitrary asynchrony as caused, for example, by temporal and spatial load variation in the network. While this leads a more complicated solution than simply synchronising the iterations of the power

iteration method, it can easily cope with transient failures and it actually leads to faster convergence of the authority scores.

Our solution entails bookkeeping overhead at each peer and also increases the network bandwidth consumption by sending history vectors rather than single scores. But the storage cost at each peer is very modest: a few hundred bytes for each entity at the peer. And the number of messages to be sent does not increase at all; the extra payload is simply piggybacked on the messages that are sent anyway. The cost difference between sending a message of 100 bytes and a message of say 500 bytes is negligible in practise.

12.3.3 Cheating-Resistant P2P Algorithm

In this section we present in detail our distributed P2P algorithm which correctly computes authority scores even in the presence of malicious or cheating peers.

Notation

The following notation is used with our algorithm.

- **Peer p_i :** Each peer is associated with a unique ID p_i where $i = 1, \ldots, n$ and n denotes the total number of peers in the network.
- **Item e_j :** Each item is associated with a unique ID e_j where $j = 1, \ldots, m$ and m denotes the total number of items in the network.

Each peer is responsible for computing authority scores only for a certain subset of entities in the network.

Definition 12.5 (Set of Responsibility $\mathcal{R}_i$)**.** *The set $\mathcal{R}_i$ denotes the subset of all entities the peer p_i is responsible for.* ∎

Next we define when an entity e_j is said to be compromised.

Definition 12.6 (Compromised Entity)**.** *An entity e_j is said to be* compromised *if the majority of peers who is responsible for e_j is malicious.* ∎

We define that the fraction f of malicious peers is a minority in the network.

Definition 12.7 (Fraction f of Malicious Peers)**.**

$$f \leq \frac{1-\epsilon}{2} \text{ with } \epsilon \in (0,1)$$

∎

For each of the entities in the network, our algorithm requires that it is replicated randomly over all peers. We define the degree of replication as follows:

Definition 12.8 (Degree of Replication r)**.** *Let m be the number of entities in the network. For each of the m entities at least r randomly chosen peers are responsible for it. We denote r as the degree of replication.* ∎

In the Section 12.4, we study how different values of the replication degree r affect the performances of our algorithm.

Once chosen, the degree of replication is obtained by simulating in a distributed environment the following centralised procedure. For each item e_j, $j = 1, \ldots, m$, the set of r peers responsible for such items is determined by drawing without replacement r integer numbers in the range $[1, n]$. This ensures that exactly r replicas are generated. The set of responsibility $\mathcal{R}_i$ for each peer p_i is then computed accordingly.

To simulate this procedure in a distributed environment, we assume peers to be provided with a common random number generator. For each item e_j, $j = 1, \ldots, m$, each peers generates (without replacement) a set of r random numbers in the range $[1, n]$ corresponding to the set of peers who will be responsible for item e_j. Such a procedure is iterated for all items. In order to achieve among all peers the $\mathcal{R}_i$'s to be consistent, we let each peer use the same seed (for instance the start date of the algorithm). In this way, each set $\mathcal{R}_i$ can be verified by every peer simply by re-executing the generating procedure (recall that the ID p_i of a peer can be trusted). By standard assumptions, the distributed procedure gives a good approximation of the centralised one.

Our algorithm implicitly assumes that the total number of entities in the network is known or can be estimated with decent accuracy. This is not a critical assumption, since there are efficient techniques for distributed counting with duplicate elimination (e.g. [74, 81]).

Data structures and Bookkeeping

For computing correct authority scores, our algorithm makes use of several data structures that are defined in the following.

Peers keep a list of predecessors to entities they are responsible for. Formally, we define the list of predecessor as follows:

Definition 12.9 (List of Predecessors $P_i(e_j)$)**.** *For peer p_i and an entity e_j which peer p_i is responsible for, the list $P_i(e_j)$ denotes all predecessors of entity e_j that peer p_i is aware of.* ∎

Moreover, for each predecessor e_l of an entity e_j that a peer p_i is responsible for, the peer p_i remembers all $t-$refined score values of e_l reported in meetings with other peers. Additionally, a peer remembers which peer reported which value. In this way, the corresponding value can be adjusted when the same peer is met again, reporting a corrected score value.

Definition 12.10 (Set of $t-$refined Scores $S_i^t(e_l)$)**.** *Given a peer p_i, a predecessor e_l of an entity e_j that peer p_i is responsible for, $S_i^t(e_l)$ denotes the set of $t-$refined scores collected by peer i during meetings with other peers.*

An entry in $S_i^t(e_l)$ is a pair (p_k, s_k) with p_k is the ID of peer p_k which sent peer p_i the t-refined score s_k of e_l. ∎

From all remembered score values in $S_i^t(e_l)$, each peer p_i determines the most frequent value. It corresponds to the score value reported by the majority of peers.

Definition 12.11 (Majority $t-$refined Score $\pi_i^t(e_l)$)**.** *Let e_l be an predecessor of an entity e_j that peer p_i is responsible for. Then, we denote with $\pi_i^t(e_l)$ the most frequent value of a $t-$refind score found in $S_i^t(e_l)$.* ∎

After having defined the required data structures, we can now outline the information every peer p_i maintains for each entity e_j it is responsible for and its corresponding value at initialisation time.

Each peer p_i maintains the following information for $e_j \in \mathcal{R}_i$:

- The list of predecessors $P_i(e_j)$.

  Initially, $P_i(e_j)$ is an empty list.

- A collection of sets $S_i^1(e_l), \ldots, S_i^T(e_l)$ for each predecessor $e_l \in P_i(e_j)$.

  Initially, each $S_i^t(e_l)$ for $t = 0, \ldots, T-1$ is an empty set.

- The majority score $\pi_i^t(e_l) \in S_i^t(e_l)$ for each predecessor $e_l \in P_i(e_j)$.

  Initially, the scores $\pi_i^1(e_j)$ for all entities e_j are set to $\frac{1}{m}$.

Both, the list of predecessors $P_i(e_j)$ and the sets of $t-$refined scores $S_i^t(e_l)$ will be updated as new predecessors or adjusted score values of predecessors are discovered during peer meetings.

Peer Meetings and Score Computations

Peers in the network meet uniformly at random. When a peer p_i meets another randomly selected peer p_k, the following activities take place:

- First, peer p_i learns from peer p_k about previously unknown predecessors e_l of entities $e_j \in \mathcal{R}_i$ that p_i is responsible for.

- Next, peer p_i receives from peer p_k the scores $\pi_k^1(e_l), \ldots, \pi_k^T(e_l)$ for each entity e_l that peer k is responsible for and which in addition is a predecessor of an entity e_j that peer p_i is responsible for, i.e.

$$\text{peer } p_k \text{ sends to } p_i : \pi_k^1(e_l), \ldots, \pi_k^T(e_l) \;\Leftrightarrow\; e_l \in \mathcal{R}_k \wedge e_l \in P_i(e_j)$$

- Finally, peer p_i in turn sends to peer p_k the scores of the predecessors of entities peer p_k is responsible for, i.e.

$$\text{peer } p_i \text{ sends to } p_k : \pi_i^1(e_l), \ldots, \pi_i^T(e_l) \;\Leftrightarrow\; e_l \in \mathcal{R}_i \wedge e_l \in P_k(e_j)$$

Of course, in a setting with malicious peers, the scores for predecessors reported by a peer p_k might be manipulated and that peer could also report scores for entities it is not responsible for. However, since the peer ID p_k is unforgable by assumption and because peer p_i can verify if $e_l \in \mathcal{R}_k$ and $e_l \in P_i(e_j)$, score values are only accepted when the conditions stated above are true.

Definition 12.12 (Acceptance of $\pi_k^t(e_j)$)**.** *When two peers p_i and p_k meet, we define the following rule for accepting $t-$refined score values of an entity e_l:*

$$\textit{peer } p_i \textit{ accepts form } p_k : \pi_k^1(e_l), \ldots, \pi_k^T(e_l) \;\Leftrightarrow\; e_l \in \mathcal{R}_k \wedge e_l \in P_i(e_j)$$

where e_j is an entity that peer p_i is responsible for. ∎

After the meeting, peer p_i (and analogously peer p_k) updates its scores $\pi_i^t(e_l)$ for each $e_l \in P_i(e_j)$ and $e_j \in \mathcal{R}_i$ if necessary by determining the most frequent value in the corresponding set $S_i^t(e_l)$. The scores of entity e_j are then refined by including in the summation of the power iteration method the new score $\pi_i^t(e_l)$ of all predecessors $e_l \in P_i(e_j)$ at each iteration step t.

Formally, for each $t = 1, \ldots, T-1$, we apply the following definition to compute the $t-$refined scores for an entity e_j.

Definition 12.13 ($t-$refined Score Computation)**.** *Let $P_i(e_j)$ and $\pi_i^t(e_l)$ be defined as in Definition 12.9 and 12.11, respectively. A peer p_i which is responsible for an entity e_j computes the $t-$refined score as follows:*

$$\pi_i^t(e_j) = \frac{1-\alpha}{m} + \sum_{e_l \in P_i(e_j)} \pi_i^{t-1}(e_l)\frac{\alpha}{\delta_l}$$

where $(1-\alpha)$ is the random jump probability, m the number of entities in the network and δ_l the outdegree of entity e_l. ∎

This $t-$refined score computation may be seen as a stepwise approximation of an ideal, global power iteration method, where every peer obtains increasingly better knowledge of the global graph. The new set of $t-$refined scores computed this way, will be communicated to the other peers during the next meetings and will in turn be used there to refine their own scores.

To demonstrate how the main issues presented in Section 12.2.2 are resolved by this algorithm, we pick up the same example scenario already depicted in Figure 32.

Example. Figure 33 re-sketches the example scenario previously given in Figure 32, and illustrates how the attempt of cheating by a coalition of malicious peers is countered and how the issue caused by asynchrony is tackled. To this end, Figure 33 additionally depicts the data structures introduced with our algorithm

Setup : As in the example scenario previously given in Section 12.2.2, peer p_i owns a local subgraph with entities A, B, C and D. However, peer p_i does not compute authority scores for the entities in this local subgraph but only for the entities in the set $\mathcal{R}_i$ that peer p_i is responsible for and which were randomly selected at initialisation time. The information about the local subgraph is used only to inform other peers about predecessors of entities.

In our example scenario, the coalition around the malicious peer p_k wants to manipulate p_i's authority computation on entity D by reporting a way too high score for D's predecessor F. The two honest peers p_m and p_j also report score values for entity F and they are outnumbered by the coalition of malicious peers.

Countering Coalitions of Malicious Peers : Since p_i only accepts information about entities from peers who are responsible for these entities and entities are distributed uniformly at random, the coalition of malicious peers hardly succeed with their goal to manipulate the score computation. It's unlikely that the majority of peers being responsible for the same entity F is malicious (and maybe in addition is responsible for entities which malicious peers find worthwhile to manipulate). In our example, let's say the malicious peer p_k is indeed responsible for entity F and, thus, can communicate a wrong value to peer p_i. By remembering all received score values for F and determining the most frequent one among them, finally, the true score reported by the honest

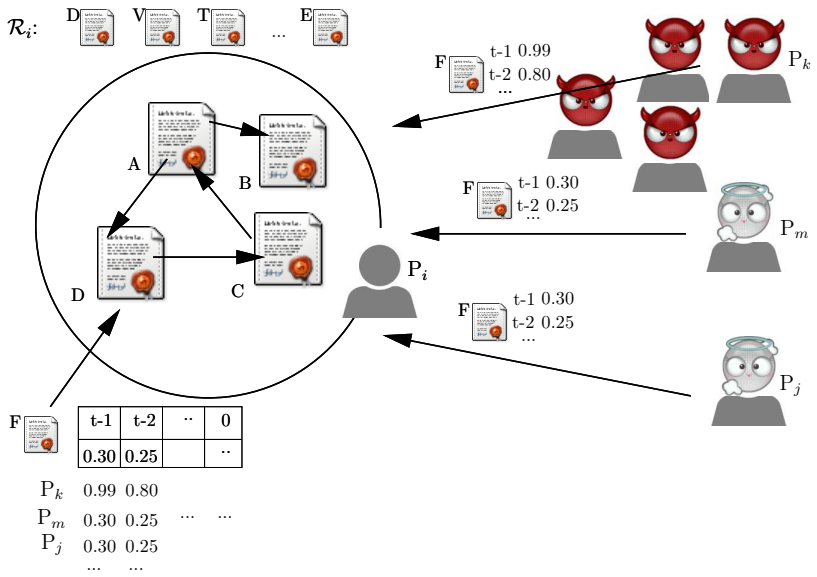


Figure 33: Example for countering a coalition of malicious Peers in a asynchroneous P2P network

peers outnumber the wrong one. Hence, our algorithm successfully counters cheating of coalitions of malicious peers.

Tackling the Issue of Asynchrony : Since in a meeting the peers not only submit a single score value but all $t-$refined scores starting from $t = 0$ to the latest iteration step a peer managed to compute, the peer p_i in our example can map the received score values to the iteration steps they belong to and select the most frequent value in each step to adjust its own authority computations. In Figure 33 this is shown for entity F, depicted as a table of score values received for iteration step 0 to $t - 1$ from the peers p_k, p_m and p_j. Hence, asynchrony has got no negative effect but with more and more meetings, all scores in each iteration step can be correctly identified.

We observe that initially, the scores computed by peers are not guaranteed to be a good estimation of the real scores, as they may not be trusted (since not enough scores may have been collected); moreover, in the beginning, peers may be aware of only a proper subset of predecessors of their own entities, therefore having only incomplete information. However, as more knowledge of the global graph is gained, these two problems gradually disappear and the wrong scores will be replaced by the correct ones.

12.4 Experiments

The experimental evaluation of our P2P algorithm for authority computations in the presence of malicious peers was performed on a simulated P2P network using data obtained from the social tagging network *LibraryThing.com* (see Part A, Section 3.2 for an introduction to *LibraryThing.com*). To this end, each peer in the P2P network represents a user and contains the user's collection of books. The simulation was done by running all peers on the same Linux server.

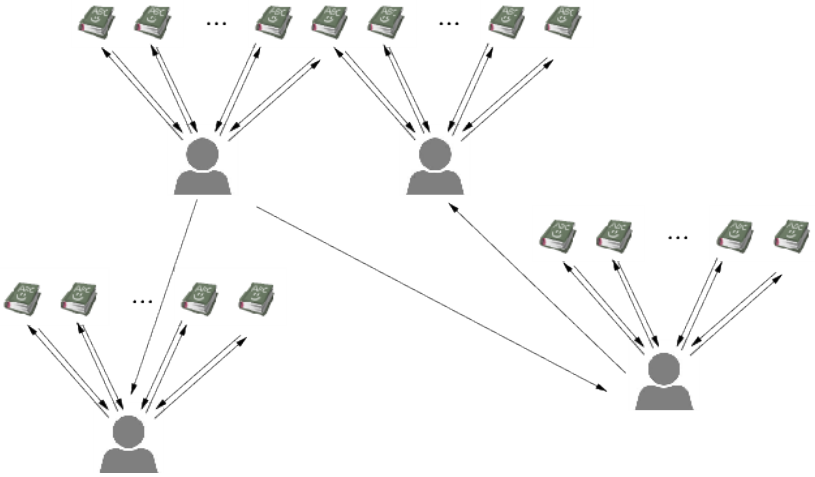


Figure 34: Construction of the global graph with data from *LibraryThing.com*

12.4.1 Setup

Figure 34 indicates the construction of the graph used in our experimental evaluation of our algorithm. The graph is created as follows:

1. The users and their books in *LibraryThing.com* correspond to the entities in G.
2. A directed edge is created from each user to the books she owns and from each book to all users who own it.
3. The "friends" and "interesting libraries" relations in *LibraryThing.com* define additional directed edges between users.

Consequently, we define the global graph, for whose entities we want to compute meaningful authority scores, according to this construction.

Definition 12.14 (Global Graph)**.** *The union of all users, books, and the edges among them define the* global graph*.*

In our experimental setup we have got 243 peers which together store a graph of $15,242$ entities (i.e. users and books).

As defined by our P2P algorithm (see Section 12.3) for countering cheating in authority computations over social networks, the entities in the global graph are replicated and distributed uniformly at random. In the context of our chosen social network setup, this means, that books as well as users are assigned to the peers' set of responsibility $\mathcal{R}$ (see Definition 12.5). Figure 35 shows an example of how the entities of the global graph can be distributed among peers. The books and users are arbitrarily distributed to honest and malicious peers without, of course, being known which peer is honest or malicious. Thus, any peer can be responsible to compute authority scores for books and users, only for books or—even though unlikely because of the ratio of number of books to numbers of users—only for users.

For our experimental evaluation, peers in the network are also chosen randomly to be malicious. The percentage of malicious peers in the network and the number

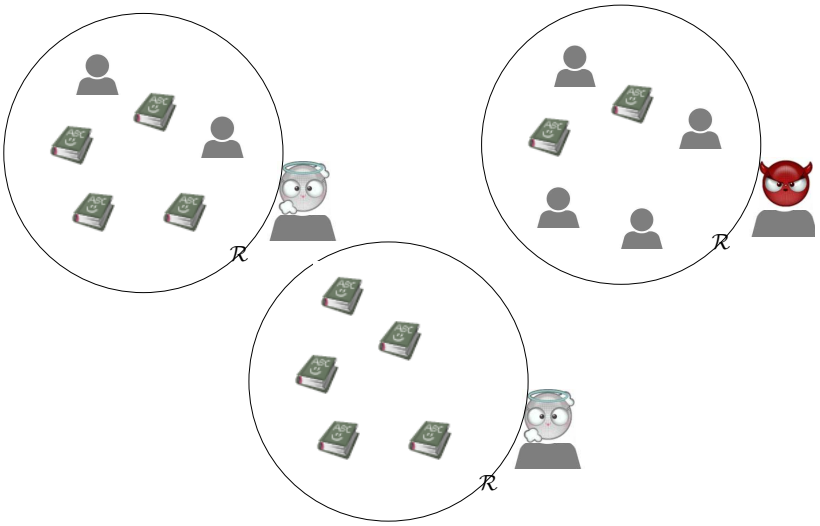


Figure 35: Books and users from *LibraryThing.com* are distributed uniformly at random to the peers' set of responsibility $\mathcal{R}$

of replicas per entity vary in our experiments and, thus, are given in Section 12.4.3 together with the achieved results for each setting.

12.4.2 Evaluation Methods

From the construction of the global graph as described in Section 12.4.1, it is easy to see that users occupy the top positions in the entities rank since users possess many books. Hence, entities corresponding to users have got many incoming edges but books have got only a handful of owners. Moreover, since every book contributes only with a small fraction to the users' scores, a wrong score provided by a malicious peer for a book will not have a big impact in the users' rank. Scores of books, on the other hand, are highly sensitive to the scores of their incoming neighbours, making them more susceptible to malicious attacks.

For this reason, given the construction of the graph as described in Section 12.4.1, we consider in our experimental evaluation only the ranking and scores of books. Before proceeding with describing the way of how to evaluate our P2P algorithm, we first introduce the following definitions:

Definition 12.15 (Global Score Vector)**.** *The vector which contains the true authority scores of all books and represents their ranking in the global graph is denoted as global score vector.* ∎

We create the global score vector by computing the scores of all entities in the global graph by a centralised algorithm and include only the scores for books in that vector. The global score vector is the ground truth in all of our experiments.

Definition 12.16 (Local Peer Vector)**.** *The local peer vector represents the ranking and authority scores of the books in the set of responsibility of a local peer as computed by our algorithm.* ∎

The local peer vector is created at each peer p_i by computing the authority scores for all entities in $\mathcal{R}_i$ as presented in Section 12.3 with our P2P algorithm, and keeping only those entities that are books.

With the help of these definitions, we now can describe how we do the score comparison of the values as computed by all peers in the network applying our distributed algorithm and the true authority scores as computed by a centralised algorithm on the global graph. We apply the following practice:

- We compute the authority scores of every entity in the graph, but for evaluation purposes, we compare only the authority scores of books returned by our distributed algorithm against the true authority scores of the books in the complete graph.
- For comparing against the true authority scores, given that in our approach the entities are distributed among the peers, we first merge the authority scores in all *local peer vectors* from all honest peers and, then, compare the resulting vector with the *global score vector*.

  Note: This is done for the experimental evaluation only. In a real P2P network knowing which peers are honest and merging their knowledge would neither be needed nor desired.
- Since entities are replicated and no synchronisation is required, it can happen that a particular entity has got different scores at different peers. In this case, the entity's score for the ranking over all peers is considered to be the average over its scores in each local peer vector.

For completeness, we define the vector achieved by merging all local peer vectors of honest peers as follows:

Definition 12.17 (Merged Peer Vector)**.** *The resulting vector obtained by merging all local peer vectors from honest peers, while considering the average over different scores for the same entity at honest peers, is denoted as merged peer vector.*

Finally, the merged peer vector as computed by our P2P algorithm is evaluated against the global score vector by comparing ranking and scores using three different measurement methods.

To this end, we compute the *top-k Kendall's Tau distance*. It corresponds to the number of pair-wise disagreements for the entities in the *top-k* positions and is defined as follows:

Definition 12.18 (*top-k* Kendall's Tau Distance $K(\tau_1, \tau_2)$)**.** *For two score vectors τ_1, τ_2 and a given value of* top-k*, the Kendall's Tau distance for the* top-k *entities is defined as follows:*

$$\begin{aligned} K(\tau_1,\tau_2) = & |\{(e_i,e_j) : e_i,e_j \in \tau_1[0:k-1], e_i \neq e_j, \\ & (\tau_1(e_i) < \tau_1(e_j) \wedge \tau_2(e_i) > \tau_2(e_j)) \vee \\ & (\tau_1(e_i) > \tau_1(e_j) \wedge \tau_2(e_i) < \tau_2(e_j))\}| \ / \ (k(k-1)/2) \end{aligned}$$

where $\tau_1[0:k-1]$ denotes the set of entities at the first top-k *positions in τ_1, and $\tau_1(e_i)$ and $\tau_2(e_i)$ is the rank position of entity e_i in the corresponding vector τ_1 and τ_2, respectively.* ∎

Kendall's Tau is a standard measure for comparing rankings; it is normalised between 0 and 1 where 1 means that there is not a single pair-wise ordering of τ_1 preserved in τ_2 and 0 indicates that the two rankings are identical.

In addition, we calculate the *top-k recall* and the *top-k statistical distance* measure which both are introduced next.

Let τ_G be the global authority vector representing the correct scores and ranking of all books in the defined global graph, and let τ_M be the merged peer vector representing the ranking of books according to our P2P algorithm in presence of malicious peers. Moreover, let $\tau_G[0 : \textit{top-k} - 1]$ and $\tau_M[0 : \textit{top-k} - 1]$ denote the set of books in τ_G or τ_M, respectively, with the *top-k* highest authority scores.

In our setting, the *top-k* recall defines the percentage of the books in the *top-k* positions of the ground truth which are also found in the *top-k* positions of the merged peer vector.

Definition 12.19 (*top-k* Recall)**.** *The* top-k *recall of a score vector τ_M with ground truth τ_G is defined as:*

$$recall_k = \frac{|\{i | i \in \tau_G[0 : k-1] \wedge i \in \tau_M[0 : k-1]\}|}{k}$$

■

The *top-k* statistical distance is defined in our setting as the sum over the differences of the score values of the *top-k* entities in the global score vector and the score values of those entities in the merged peer vector.

Definition 12.20 (*top-k* Statistical Distance)**.** *The* top-k *statistical distance for a vector τ_M with respect to the ground truth vector τ_G is defined as:*

$$dist_k = \sum_{e_i \in \tau_G[0:k-1]} |\tau_G(e_i) - \tau_M(e_i)|$$

where $\tau_G[0 : k-1]$ denotes the top-k *elements in τ_G, $\tau_G(e_i)$ and $\tau_M(e_i)$ denotes the score of entity e_i in τ_G and τ_M, respectively.* ■

Consistent Lying Behaviour

In our experiments, we consider the worst case scenario where all malicious peers form a coalition by consistently lying about score values of entities. This means, whenever any two malicious peers are responsible for the same entity, they both report the same wrong value such that they can compromise (see Definition 12.6) as much entities as possible.

It works as follows: first, we inform the malicious peers by an artificial oracle about the ground truth scores. The bad peers aim at promoting low ranked entities and demoting high ranked ones, and at the same time permuting the scores of entities in the middle of the rank. The consistent lying behaviour of malicious peers with respect to the authority score of an entity e_i is then defined as follows:

Definition 12.21 (Consistent Lying Behaviour)**.** *Let τ_G be the global score vector with entities e_i where $0 \leq i < m$ denotes the entity's rank and 0 is the rank with the highest authority score.*

For a given top-k*, we define the consistent lying behaviour of all malicious peers in the following way: If a malicious peer is responsible for entity e_i, it reports the following wrong score value:*

$$score(e_i) = \begin{cases} \frac{(1-\alpha)}{m} & \text{if } i < k \\ trueScore(0) & \text{if } i \geq m-k \\ trueScore(m-i+1) & \text{otherwise} \end{cases}$$

where $(1-\alpha)$ is the random jump probability, m the number of all entities in the global graph, and $trueScore(i)$ is the true score of the entity at rank i in τ_G. ∎

The value $\frac{(1-\alpha)}{m}$ corresponds to the score contribution by a random jump, i.e. the lowest score an entity can have.

12.4.3 Results

We evaluate our algorithm for countering cheating in authority computations over social networks by two sets of experiments. In the first set, we vary the fraction of malicious peers and in the second set, we vary the degree of replication.

Varying fraction of malicious peers

In our first set of experiments, we examine the influence of a coalition of malicious peers on our P2P authority computations by varying the percentage of malicious peers in the network while keeping the number of replicas for each entity constant. To this end, we compute the *top-100* Kendall's Tau distance (see Definition 12.18), the *top-100* recall (see Definition 12.19) and the *top-100* statistical distance (see Definition 12.20) in regard to the global score vector (see Definition 12.15) and the merged peer vector (see Definition 12.17).

Peers in the network are randomly chosen to behave maliciously by applying the consistent lying model (see Definition 12.21) and entities are replicated as defined by our algorithm according to the description given in Section 12.3.3. Hence, in this set of experiments, we fix the degree of replication r (see Definition 12.8) to 5 replicas per entity in the network and vary the percentage of malicious peers with each experiment by choosing the fraction f of malicious peers (see Definition 12.7) in the network to be 0%, 5% and 20%.

Figure 36, 37 and 38 show the results for the *top-100* Kendall's Tau distance, *top-100* recall, and *top-100* statistical distance, respectively.

We can observe from Figure 36, visualising the Kendall's Tau distance, that when no peer behaves maliciously, the ranking as given in the merged peer vector and computed by our distributed algorithm matches with increasing numbers of meetings the ranking of the global score vector defining the ground truth. The same is true for the computed authority score values of the *top-k* entities as shown by the statistical distance in Figure 38, finally converging to 0.

However, when the percentage of malicious peers is increased, the gap between the computed scores of our algorithm and the ground truth gets larger; yet we are able to obtain results. In particular, note from Figure 37 that 10.000 meetings suffice to obtain a *top-100* recall of 90%. Though, as indicated by the results for the Kendall's Tau distance and statistical distance, already with 5% of the peers being dishonest, only 5 replicas are not enough to guarantee convergence.

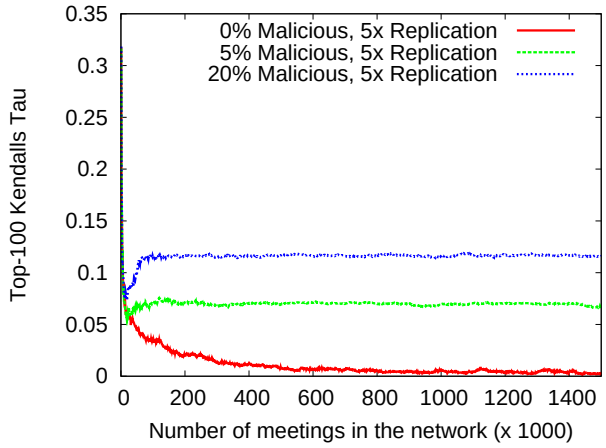


Figure 36: Top-100 Kendall's Tau: 5 replicas per entity, varying % of malicious peers

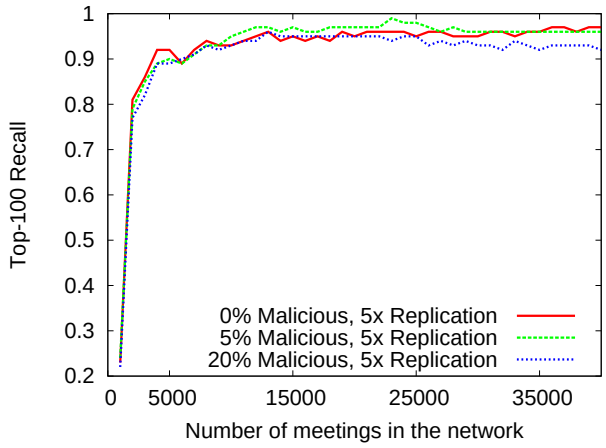


Figure 37: Top-100 Recall: 5 replicas per entity, varying % of malicious peers

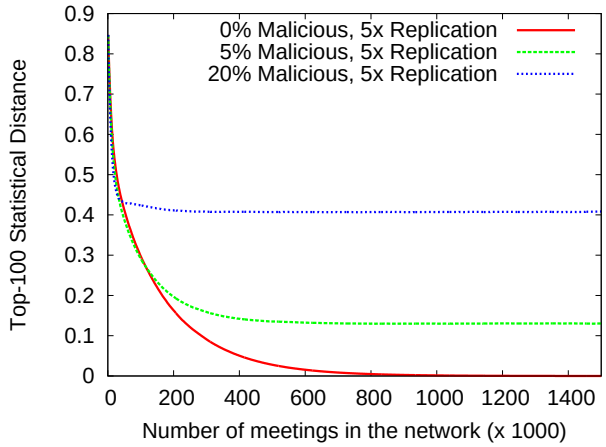


Figure 38: Statistical Distance: 5 replicas per entity, varying % of malicious peers

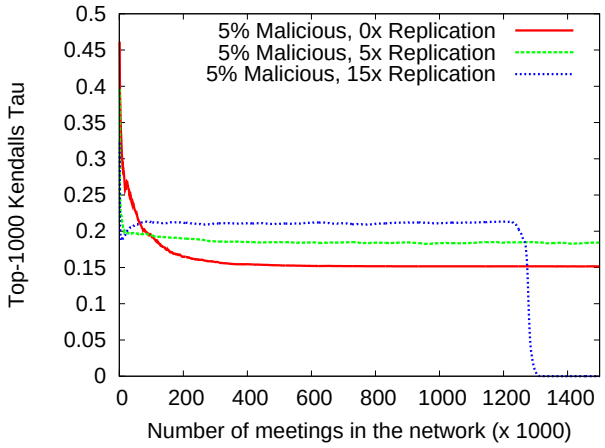


Figure 39: Top-1000 Kendall's Tau: 5% malicious peers, varying the number of replicas

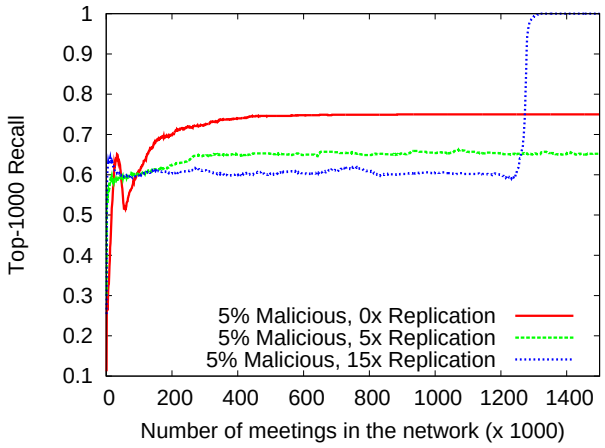


Figure 40: Top-1000 Recall: 5% malicious peers, varying the number of replicas

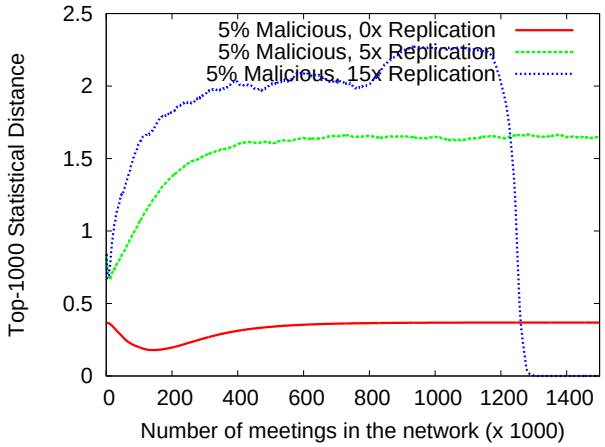


Figure 41: Statistical Distance: 5% malicious peers, varying the number of replicas

Varying degree of replication

In our second set of experiments, we change the setting in order to determine if indeed an increased number of replicas per entity breaks the influence of a given coalition with malicious peers.

For this, we fix the fraction f of malicious peers in the network to 5% and instead, vary in our experiments the degree or replication, evaluating our algorithm with 0, 5 and 15 replicas for each entity in the network. Again, malicious peers are chosen randomly among all peers and all malicious peers lie consistently about the score values of entities as defined in Definition 12.21, and replicas are distributed uniformly at random to all peers, too.

We compute in this set of experiments the Kendall's Tau distance, the recall and the statistical distance for the *top-1000* entities in our merged peer vector.

Figure 39 shows the results for the *top-1000* Kendall's Tau distance, Figure 40 for the *top-1000* recall and Figure 41 for the *top-1000* statistical distance.

At first glance, the results for this second set of experiments appear to be counterintuitive: In all three charts it can be observed that adding more replicas causes the results to initially become worse. The pair-wise disagreements in the ranking as shown by the Kendall's Tau distance, as well as the recall for the *top-1000* rankings are worse when more replicas per entity are used with our algorithm. The effect is even more visible for the score values of the *top-1000* entities. With more replicas, the statistical distance shows that initially, with an increasing number of peer meetings, the authority scores computed by our algorithm veer away from the true authority scores. After more and more meetings and a sufficient number of replicas, however, the ranking and scores almost instantly converge to the correct values. This oddity becomes evident by considering the following explanation.

Since each entity is replicated more times, each peer (both honest and dishonest ones) is responsible for more entities. In this way, malicious peers are able to report false scores for a larger fraction of the entities. Moreover, more honest peers are responsible for the same entities, too. Hence, in total, more meetings are necessary until all honest peers have learnt about all predecessors of the same entity and can agree on the same score value. Until they don't agree on the same scores, a consistently lying coalition of malicious peers might have the majority in all equal votes for an entity's authority score (and if there is no distinct majority yet, only an arbitrarily value can be chosen).

Nevertheless, from the three charts shown in Figure 39, 40 and 41, we can see, too, that with 15 replicas and an increasing number of meetings, every honest peer finally is able to obtain the correct ranking and scores by computing the majority for each entity, with scores correctly converging to the ground truth.

This second set of experiments give valuable insights on the behaviour of our algorithm. When the number of malicious peers is small, introducing more replicas might initially affect the quality of the results adversely, but guarantees convergence. One interesting direction for future research is to derive the optimal number of replicas as a function of the number of malicious peers.

Also it is worth noticing, when comparing the results from both sets of experiments for the setup with a fraction of 5% malicious peers and 5 replicas, the results for the *top-100* scores seem to be more accurate than those for the *top-1000* ones. This might depend on the heavy-tailed distribution of the scores which social networks frequently

exhibit. In this case, low-rank scores lie within a short range, making the task of retrieving the correct *top-k* scores harder for higher values of *top-k*.

12.5 Conclusion

We believe that decentralised algorithms for eigenvector-oriented computations of authority and social prestige will gain importance for P2P-based physically distributed settings or for cases where user-specific autonomous agents interact on a server(-farm)-based platform. These settings are inevitably vulnerable to malicious behaviour by bad peers.

Our algorithm is the very first one that we are aware of, that can counter the effects of misbehaviour in a systematic manner, and our computational model makes only very weak and realistic assumptions about the system.

Interesting directions for future work are a formal analysis of the algorithm's properties, most notably, its convergence speed, and also to research optimisations that aim to minimise the degree of replication that is necessary to counter a certain fraction of bad peers.

References

[1] Screen shot and feature overview of delicious 2.0 preview. http://techcrunch.com/2007/09/06/exclusive-screen-shots-and-feature-overview-of-delicious-20-preview/, Sept. 2007.

[2] Delicious.com. http://delicious.com, 2011.

[3] Flickr. http://advertising.yahoo.com/article/flickr.html/, Aug. 2011.

[4] Flickr boasts 6 billion photo uploads. http://news.softpedia.com/newsImage/Flickr-Boasts-6-Billion-Photo-Uploads-2.jpg/, Aug. 2011.

[5] Flickr.com. http://flickr.com, 2011.

[6] Librarything.com. http://librarything.com, 2011.

[7] A new flavor...still delicious. http://blog.delicious.com/2011/09/a-new-flavor...still-delicious/, Sept. 2011.

[8] Wikipedia entry: Delicious.com. http://en.wikipedia.org/wiki/Delicious.com, Jan. 2012.

[9] Zeitgeist overview. http://www.librarything.com/zeitgeist, Jan. 2012.

[10] Z. Abrams, R. McGrew, and S. Plotkin. A non-manipulable trust system based on eigentrust. *SIGecom Exch.*, 5(4):21–30, 2005.

[11] G. Adomavicius and A. Tuzhilin. 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.

[12] Y.-Y. Ahn et al. Analysis of topological characteristics of huge online social networking services. In *WWW*, 2007.

[13] S. Amer-Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1):710–721, 2008.

[14] S. Amer-Yahia et al. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.

[15] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, 2006.

[16] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *In 6th Inx'l Semantic Web Conference, Busan, Korea*, pages 11–15. Springer, 2007.

[17] R. A. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. In *KDD*, 2007.

[18] N. Bansal and N. Koudas. Searching the blogosphere. In *WebDB*, 2007.

[19] S. Bao, G.-R. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *WWW*, pages 501–510, 2007.

[20] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[21] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. In *STOC*, pages 117–123, 2002.

[22] R. A. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 312–320, New York, NY, USA, 2005. ACM.

[23] L. Becchetti, C. Castillo, D. Donato, R. Baeza-Yates, and S. Leonardi. Link analysis for web spam detection. *ACM Trans. Web*, 2(1):1–42, February 2008.

[24] M. Bender, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, R. Schenkel, and G. Weikum. Exploiting social relations for query expansion and result ranking. In *Data Engineering for Blogs, Social Media, and Web 2.0, ICDE 2008 Workshops*, pages 501–506, Cancun, Mexico, 2008. IEEE Computer Society.

[25] M. Bender, T. Crecelius, M. Kacimi, S. Michel, J. X. Parreira, and G. Weikum. Peer-to-peer information search: Semantic, social, or spiritual? *IEEE Data(base) Engineering Bulletin*, 30(2):51–60, 2007.

[26] M. Bender, T. Crecelius, S. Michel, and J. X. Parreira. P2p web search: Make it light, make it fly (demo). In *CIDR*, pages 164–168, 2007.

[27] B. Billerbeck and J. Zobel. Questioning query expansion: An examination of behaviour and parameters. In *ADC*, 2004.

[28] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis. Evaluating reachability queries over path collections. In *SSDBM*, pages 398–416, 2009.

[29] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *IEEE Trans. Knowl. Data Eng.*, 22(5):682–698, 2010.

[30] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30(1–7):107–117, 1998.

[31] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. volume 30, pages 107–117, 1998.

[32] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[33] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'El, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *CIKM*, pages 1227–1236, 2009.

[34] C. Castillo, D. Donato, A. Gionis, V. Murdock, and F. Silvestri. Know your neighbors: web spam detection using the web topology. In *SIGIR*, pages 423–430, 2007.

[35] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[36] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[37] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[38] T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, R. Schenkel, and G. Weikum. Making sense: Socially enhanced search and exploration. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB 2008)*, volume 1, pages 1480–1483, Auckland, New Zealand, 2008. ACM.

[39] T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, R. Schenkel, and G. Weikum. Social recommenations at work (demo). In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research & Development on Information Retrieval (SIGIR 2008)*, pages 884–884, Singapore, Singapore, July 2008. ACM.

[40] T. Crecelius and R. Schenkel. Evaluating network-aware retrieval in social networks. In S. Geva, J. Kamps, C. Peters, T. Sakai, A. Trotman, and E. Voorhees, editors, *SIGIR 2009 Workshop on the Future of IR Evaluation*, pages 17–18, Boston, USA, 2009. IR Publications.

[41] S. Cronen-Townsend, Y. Zhou, and W. B. Croft. Predicting query performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 299–306, 2002.

[42] A. Damian, W. Nejdl, and R. Paiu. Peer-sensitive objectrank - valuing contextual information in social networks. In A. H. H. Ngu, M. Kitsuregawa, E. J. Neuhold, J.-Y. Chung, and Q. Z. Sheng, editors, *WISE*, volume 3806 of *Lecture Notes in Computer Science*, pages 512–519. Springer, 2005.

[43] A. Das et al. Google news personalization: scalable online collaborative filtering. In *WWW*, 2007.

[44] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[45] J. V. Davis and I. S. Dhillon. Estimating the global pagerank of web communities. In *KDD*, pages 116–125, New York, NY, USA, 2006. ACM Press.

[46] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[47] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.

[48] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[49] P. A. Dmitriev, N. Eiron, M. Fontoura, and E. J. Shekita. Using annotations in enterprise search. In *WWW*, pages 811–817, 2006.

[50] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining transitive closure of graphs in sql. *Int. Journal on Information Technology*, 5:1–23, 1999.

[51] J. Douceur. The sybil attack. In *IPTPS*, pages 251–260, 2002.

[52] M. Dubinko, R. Kumar, J. Magnani, J. Novak, P. Raghavan, and A. Tomkins. Visualizing tags over time. *ACM Transactions on the Web*, 1(2), 2007.

[53] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[54] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[55] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *J. Algorithms*, 49(1):86–113, 2003.

[56] S. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2):198–208, April 2006.

[57] J. Graupmann, R. Schenkel, and G. Weikum. The spheresearch engine for unified ranked retrieval of heterogeneous xml and web documents. In *VLDB*, pages 529–540, 2005.

[58] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large networks. In *CIKM*, 2010.

[59] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of trust and distrust. In *WWW*, pages 403–412, New York, NY, USA, 2004. ACM Press.

[60] R. V. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of trust and distrust. In *WWW*, pages 403–412, 2004.

[61] U. Güntzer, W. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.

[62] Z. Gyöngyi, P. Berkhin, H. Garcia-Molina, and J. O. Pedersen. Link spam detection based on mass estimation. In *VLDB*, pages 439–450, 2006.

[63] Z. Gyöngyi, H. G. Molina, and J. Pedersen. Combating web spam with trustrank. In *VLDB*, pages 576–587, 2004.

[64] H. Halpin et al. The complex dynamics of collaborative tagging. In *WWW*, 2007.

[65] T. H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[66] D. Heckerman et al. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1:49–75, 2000.

[67] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22(1), 2004.

[68] P. Heymann et al. Can social bookmarking improve web search? In *WSDM*, 2008.

[69] P. Heymann and H. Garcia-Molina. Collaborative creation of communal hierarchical taxonomies in social tagging systems. Technical Report 2006-10, Stanford University, April 2006.

[70] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In Y. Sure and J. Domingue, editors, *The Semantic Web: Research and Applications*, volume 4011 of *LNAI*, pages 411–426, Heidelberg, June 2006. Springer.

[71] H. Hwang, V. Hristidis, and Y. Papakonstantinou. Objectrank: a system for authority-based search on databases. In *SIGMOD Conference*, pages 796–798, 2006.

[72] K. Järvelin and J. Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *SIGIR*, pages 41–48, 2000.

[73] G. Jeh and J. Widom. Scaling personalized web search. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 271–279, New York, NY, USA, 2003. ACM Press.

[74] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, (3), 2007.

[75] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[76] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[77] K. S. Jones. Idf term weighting and ir research lessons. *Journal of Documentation*, 60:521–523, 2004.

[78] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW*, pages 640–651, 2003.

[79] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW*, pages 640–651, New York, NY, USA, 2003. ACM Press.

[80] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[81] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS*, pages 482–491, 2003.

[82] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *STOC*, pages 561–568, 2004.

[83] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *FOCS*, pages 81–91, 1999.

[84] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In J. Wang, editor, *COCOON*, volume 2108 of *Lecture Notes in Computer Science*, pages 268–277. Springer, 2001.

[85] J. A. Konstan, S. M. McNee, C.-N. Ziegler, R. Torres, N. Kapoor, and J. Riedl. Lessons on applying automated recommender systems to information-seeking tasks. In *AAAI*, 2006.

[86] K. Kremerskothen. Flickr blog: The 6 billionth photo upload. `http://blog.flickr.net/en/2011/08/04/6000000000/`, Aug. 2011.

[87] V. Krikos, S. Stamou, P. Kokosis, A. Ntoulas, and D. Christodoulakis. Directoryrank: ordering pages in web directories. In *WIDM*, pages 17–22, 2005.

[88] R. Kumar et al. Structure and evolution of online social networks. In *KDD*, 2006.

[89] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[90] A. N. Langville and C. D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.

[91] C. Li, K. Chang, I. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[92] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1), 2003.

[93] Y.-T. Liu, B. Gao, T.-Y. Liu, Y. Zhang, Z. Ma, S. He, and H. Li. Browserank: letting web users vote for page importance. In *SIGIR*, pages 451–458, 2008.

[94] S. Marti and H. Garcia-Molina. Identity crisis: Anonymity vs. reputation in p2p systems. In *Peer-to-Peer Computing*, page 134, Washington, DC, USA, 2003. IEEE Computer Society.

[95] S. Marti and H. Garcia-Molina. Taxonomy of trust: Categorizing p2p reputation systems. *Computer Networks*, 50(4):472–484, 2006.

[96] U. Meyer. On dynamic breadth-first search in external-memory. In S. Albers and P. Weil, editors, *STACS*, volume 1 of *LIPIcs*, pages 551–560. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008.

[97] A. Mislove et al. Exploiting social networks for internet search. In *HotNets*, 2006.

[98] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[99] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[100] C. Pang, G. Dong, and K. Ramamohanarao. Incremental maintenance of shortest distance and transitive closure in first-order logic and sql. *ACM Trans. Database Syst.*, 30(3):698–721, 2005.

[101] J. X. Parreira, C. Castillo, D. Donato, S. Michel, and G. Weikum. The juxtaposed approximate pagerank method for robust pagerank approximation in a peer-to-peer web search network. *VLDB J.*, 17(2):291–313, 2008.

[102] J. X. Parreira, S. Michel, M. Bender, T. Crecelius, and G. Weikum. P2P authority analysis for social communities. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *33rd International Conference on Very Large Data Bases (VLDB 2007)*, pages 1398–1401, Vienna, Austria, 2007. ACM.

[103] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. volume 20, pages 127–138, Chichester, UK, UK, 2008. John Wiley and Sons Ltd.

[104] G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[105] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, 1994.

[106] K. Sankaralingam, M. Yalamanchi, S. Sethumadhavan, and J. C. Browne. Pagerank computation and keyword search on distributed systems and p2p networks. *J. Grid Comput.*, 1(3):291–307, 2003.

[107] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *PODS*, pages 69–78, 2008.

[108] B. M. Sarwar et al. Item-based collaborative filtering recommendation algorithms. In *WWW*, 2001.

[109] J. Savoy. Statistical inference in retrieval effectiveness evaluation. *Inf. Process. Manage.*, 33(4):495–512, 1997.

[110] J. B. Schafer et al. Collaborative filtering recommender systems. In *The Adaptive Web*, 2007.

[111] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR*, pages 523–530, 2008.

[112] R. Schenkel, T. Crecelius, M. Kacimi, T. Neumann, J. X. Parreira, M. Spaniol, and G. Weikum. Social wisdom for search and recommendation. *IEEE Data Eng. Bull.*, 31(2):40–49, 2008.

[113] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[114] C. Schmitz et al. Mining association rules in folksonomies. In *Data Science and Classification*. Springer, 2006.

[115] S. Sen et al. Tagging, communities, vocabulary, evolution. In *CSCW*, 2006.

[116] S. Shi, J. Yu, G. Yang, and D. Wang. Distributed page ranking in structured p2p networks. In *ICPP*, pages 179–186, 2003.

[117] M. Sozio, T. Crecelius, J. X. Parreira, and G. Weikum. Good guys vs. bad guys: Countering cheating in peer-to-peer authority computations over social networks. In *WebDB*, 2008.

[118] Special section on social media and search. *IEEE Internet Computing*, 11(6), 2007.

[119] Special issue on data management issues in social sciences. *IEEE Data Engineering Bulletin*, 30(2), 2007.

[120] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[121] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.

[122] J. Surowiecki. *The Wisdom of Crowds*. New York, 2004.

[123] C. Tantipathananandh et al. A framework for community identification in dynamic social networks. In *KDD*, 2007.

[124] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, pages 242–249, 2005.

[125] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[126] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

[127] T. Vander Wal. Folksonomy. `http://vanderwal.net/folksonomy.html`, Feb. 2007.

[128] Y. Wang and D. J. DeWitt. Computing pagerank in a distributed internet search engine system. In *VLDB*, pages 420–431, 2004.

[129] J. Wu and K. Aberer. Using a Layered Markov Model for Distributed Web Ranking Computation. In *ICDCS*, pages 533–542, 2005.

[130] Y. Xiao, W. Wu, J. Pei, W. W. 0009, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.

[131] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Knowl. Data Eng.*, 16(7):843–857, 2004.

[132] S. Xu et al. Using social annotations to improve language model for information retrieval. In *CIKM*, 2007.

[133] J. Zhang, M. S. Ackerman, and L. A. Adamic. Expertise networks in online communities: structure and algorithms. In *WWW*, pages 221–230, 2007.

List of Figures

List of Tables

List of Listings

Index