Improving P2P keyword search by combining .torrent metadata and user preference in a semantic overlay

Niels Zeilemaker

Delft University of Technology

Improving P2P keyword search by combining .torrent metadata and user preference in a semantic overlay

Master thesis assignment Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Niels Zeilemaker
1315250
niels@zeilemaker.nl

12th February 2010

Preface

I would like to thank ir. M. Clements for suggesting possible directions and allowing me to think out loud with him. Discussions on why or how different similarity functions were performing allowed me to gain basic knowledge on similarity functions in a small amount of time. This was very useful at the start of this project.

I would also like to thank dr. ir. J.A. Pouwelse for his comments on both the content and layout of this document. His unlimited enthusiasm was a great motivator, which allowed me to stay focused during the whole project.

Niels Zeilemaker

Delft, The Netherlands
12th February 2010

Contents

Preface **v**

1 Introduction **1**
1.1 Overlay networks . . . 2
1.1.1 Structured overlays . . . 2
1.1.2 Unstructured overlays . . . 3
1.2 Keyword search . . . 3
1.2.1 Decentralized keyword search . . . 4

2 Problem description **9**
2.1 Research question . . . 9
2.2 Document layout . . . 12

3 Evaluation infrastructure **13**
3.1 Evaluation statistics . . . 14
3.1.1 Precision and recall . . . 14
3.1.2 F1 score . . . 15
3.1.3 NDCG . . . 15
3.1.4 Computational efficiency . . . 16
3.2 Significance . . . 17

4 Dataset **19**
4.1 Parsing . . . 21
4.2 Data statistics . . . 21
4.2.1 User activity . . . 21
4.2.2 Rankings . . . 23
4.2.3 Top-N user behavior . . . 26
4.2.4 Ground truth . . . 26
4.3 Tribler and Zipf's law . . . 27

5 Candidate solutions **29**
5.1 Known techniques . . . 29
5.1.1 Tribler . . . 30
5.1.2 Tribler optimized . . . 30

5.1.3 Cosine similarity 31
5.1.4 Similarity coefficients 32
5.1.5 Pseudo uservector 34
5.2 Advanced optimizations 38
5.2.1 Long profile boost 39
5.2.2 Balanced pseudo uservector 40
5.2.3 Levenshtein tuning 40
5.2.4 Pseudo uservector boost 43
5.2.5 Additional remarks 44
5.2.6 Resulting functions 44

6 Results 47
6.1 Search effectiveness 47
6.1.1 Precision and recall 47
6.1.2 F1 and NDCG performance 49
6.2 Computational efficiency 52
6.2.1 Runtime 52
6.2.2 Database impact 55
6.3 Best of both worlds 56

7 Discussion 59
7.1 Identifying the problems 59
7.2 Improved search effectiveness 60
7.3 Improved computation efficiency 61
7.4 Implementation details 61

8 Conclusions and Future Work 63
8.1 Conclusion 63
8.2 Future work 65

A Ground truth 71
A.1 TF-IDF 71
A.2 Categorization 72

B SQL statements 73
B.1 CosineBoost 73
B.2 ItemItem(Filename) 73
B.3 ItemItem(Levenshtein) 74

C Raw results 75

Chapter 1

Introduction

Peer-to-Peer (P2P) technology was introduced in 1979, when Usenet started to appear. Usenet is a network in which servers exchange information with each other in a non-centralized way. This approach deviates from the more traditional client-server type network, in which a central component has more responsibilities such as routing, processing etc. A P2P network only has peers (computers/devices connected to the network), which all basically have the same role. The peers together strive to complete a task, which results in a network without a centralized component. Best known variants of P2P are file sharing networks. But P2P technology has surfaced in other fields as well e.g. searching for extraterrestrial intelligence (SETI) or making phone calls possible between computers (Skype).

The lack of a central component makes P2P networks very scalable. When more clients connect to a traditional client-server network, the capacity of the central server must increase accordingly, in order to cope with the higher demand. If this central server provides search functionality, e.g. Google, and more clients start searching for content Google has to add more capacity to its servers or searching will slow down. In a P2P network this scaling problem does not occur because all peers together are responsible for the tasks, in this case search. More peers joining the network will cause more peers to be available for dividing the load and thus the network will scale relatively easy. Another benefit of not having a central component is the elimination of a single point of failure. When the server in a client-server network fails, the whole system fails. In a P2P network all peers are equal, thus the loss of one peer does not influence the performance of the network much.

But the lack of a centralized component causes some tasks to be harder to achieve. Without a central component, locating peers inside the P2P network suddenly becomes a difficult task, because there is no database with complete knowledge of the network. Locating peers is necessary during search and communication with other peers, but also when trying to join a P2P network.

Because of the dynamic nature of a P2P network and the lack of centralized components, it is unknown which peers are connected to the P2P network. And

thus which peer to contact in order to join the network (this is often referred to as the *bootstrapping* problem). This problem is usually solved by including a list of known peers into the installer, or using several peers which are always online and allow new peers to connect to the network.

1.1 Overlay networks

P2P networks are constructed as an additional layer on top of a preexisting network, this layer is called an overlay. P2P overlays can be organized into two classes, structured and unstructured, as discussed by Lua et al. [19].

1.1.1 Structured overlays

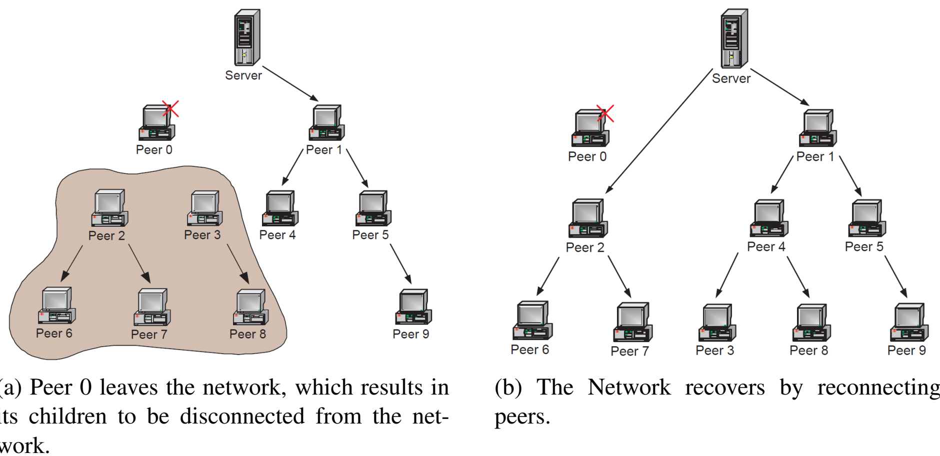
In a structured overlay all peers connect to the network in an organized way. A method for creating a structured overlay is a distributed hash table (DHT). A DHT uses a hash function to generate keys, which in turn can be used to address a peer or file.

A common example in which a DHT is used is a ring-based overlay. In such an overlay all peers have a unique id which is generated by the hash function (the ip-address is usually used as input). This id places a peer on a location in the ring. Items are also placed on the ring using the same hash function. Peers are responsible for a segment of the ring (which can contain several items), usually close to their own location on the ring.

Messages are routed using a routing table which contains id-ip mappings and then forwarded to the numerically closest peer until the destination is reached. An implementation of a ring based DHT is the Chord network as described by Lua et al.[19]. In Chord it can be shown that the number of hops (times a message is forwarded) is limited to $O\left(logN\right)$, this is due to the organization of the routing table which stores the ip-addresses of peers using a log scale (ever increasing steps further away from this peer).

Structured overlays could also be based on a tree. These overlays are particularly useful for live-streaming (as shown by Liu et al. [18]), because information can be poured in at the root and easily distributed along its children. An example of a live-streaming event would be watching a live soccer match on tv.

Still a structured overlay has one big problem and that is dealing with churn. Churn is caused by the continuous leaving and joining of new peers and the organizational complexity of dealing with this. A P2P network under churn usually has a lot of overhead as a result. In structured overlays this overhead becomes unacceptably large when a network increases in size. An example is given by Liu et al. [18] (which is shown in Figure 1.1), in which if a peer leaves the network a complete subtree (all of its children) need to be reconnected to the network.



(a) Peer 0 leaves the network, which results in its children to be disconnected from the network.

(b) The Network recovers by reconnecting peers.

Figure 1.1: Churn example as shown by Liu et al. [18]. After Peer 0 leaves, multiple messages need to be sent in order to "fix" the network. Peer 2 connects to the Server, Peer 3 and 8 are connected to Peer 4 to balance the tree.

1.1.2 Unstructured overlays

In an unstructured overlay peers are connected to a number of neighbors which are used to route messages. Neighbors are the peers to which a peer is directly connected, these peers are chosen based on a metric e.g. the responsiveness of peers. There is no correlation between the neighbors and their locations, other peers are discovered by gossiping. Gossiping is a technique in which peers exchange lists of known peers. This process is repeated at an interval and ensures that eventually almost all peers know each other.

Locating a peer has to be done using flooding (sending a message to all neighbors which is then again sent to all neighbors etc. until the peer has been found) because there is no correlation between a peer and its location. Compared to a structured overlay, this is much more problematic, but the unstructured nature of the overlay results in less overhead when dealing with churn [38]. Lua et al. compares several unstructured overlays [19].

1.2 Keyword search

The focus of this thesis is on keyword search in a P2P network. Different strategies have been developed for doing so and in the following paragraphs some well known examples are highlighted. Keyword search is a common operation in file sharing networks, because only after locating a file, a user can download it. Controversy surrounds these networks because they allow users to share files which often are illegal to distribute.

One of the first P2P networks which allowed users to share files was Napster, as

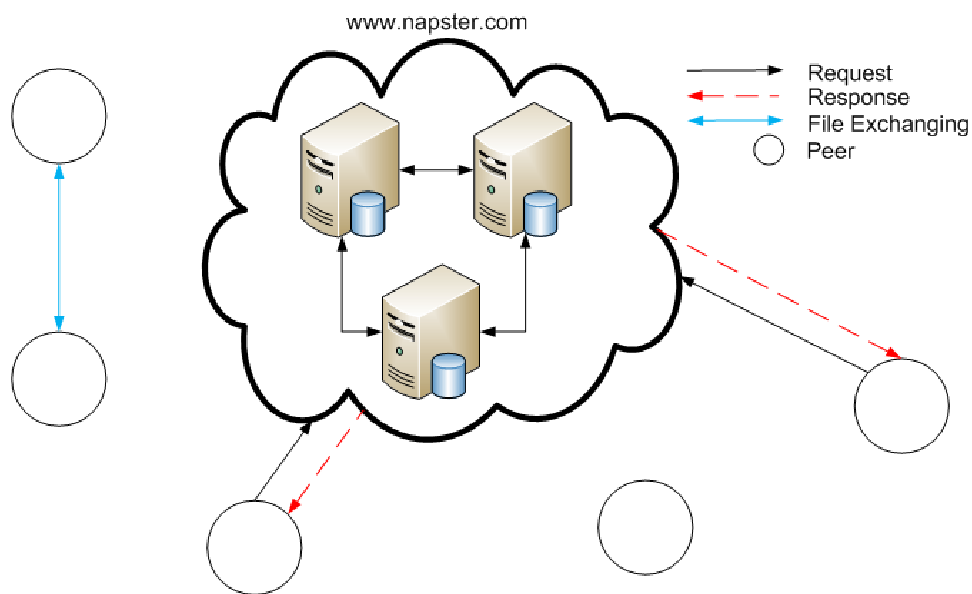


Figure 1.2: Napster search, using a central server to locate peers. Then downloading directly, as described by Saroiu et al. [31].

described by Saroiu et al. [31]. Released in 1999, it used a central server which allowed a peer to search for a file in the network, as shown in Figure 1.2. Using this central server, a peer requested a list of peers which are sharing items matching the query. After this list is returned by the server, a peer can decide if and which items to download. Items are downloaded in a direct fashion, thus no intermediate peers are used.

Due to the central component, Napster is not a true P2P network. A true P2P network being a network in which all peers are equal (the central server is not equal to all other peers). Additionally when faced with a lawsuit, which resulted in a $26 million settlement, Napster was quickly shutdown because of its central servers.

1.2.1 Decentralized keyword search

The central component inside the Napster network, the central search server, allowed for an easy shutdown. But having a central server for searches is very efficient, because due to the unstructured overlay Napster would have to provide search using a flooding-variant which is not very efficient.

But in the wake of the $26 million settlement, the decentralized keyword search problem had to be overcome and several different approaches were tried by different P2P networks.

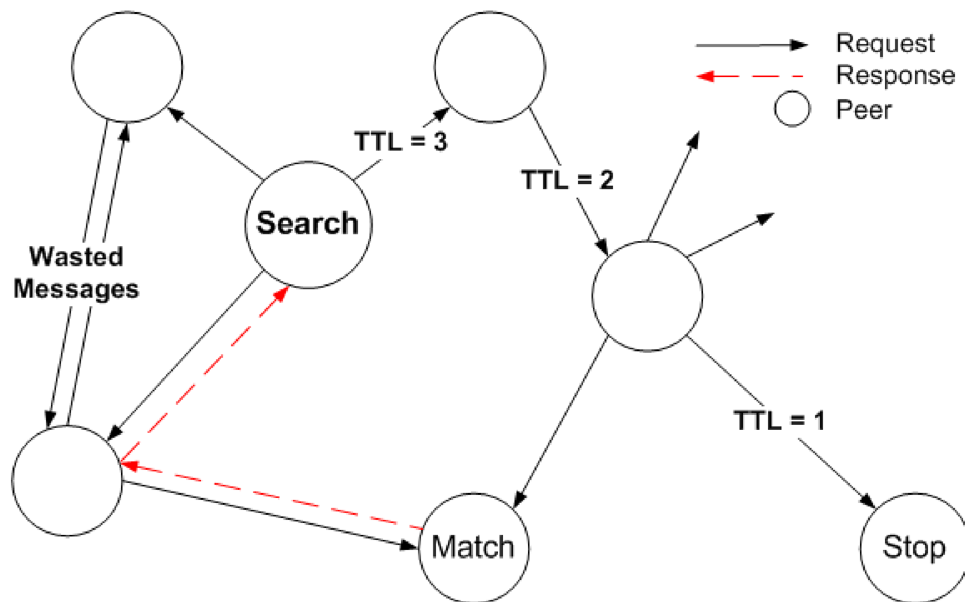


Figure 1.3: Gnutella search, including TTL, a loop and a QUERY response. The peer which initiates the search, sends a request to all neighbors. This message is forwarded, but if a match is found a response is sent back.

Gnutella

Gnutella was released in early 2000 and in January of 2006 approximately 3.4 million peers were connected to it, as shown by Rasti et al. [27]. In contrast to Napster, Gnutella does not rely on a central component to perform searches, but uses flooding.

Messages are called QUERY messages and contain an id, sender, Time-To-Live (TTL, number of resends allowed) and the specified search query. QUERY messages are broadcasted to all neighbors (Figure 1.3) until the TTL equals 0. In the first versions of the Gnutella network the TTL was 7, but eventually this number was reduced to 4. Upon receiving a QUERY message, the search string is matched against a set of local filenames. When a file was found a QUERY response is back-propagated, as described by Ripeanu et al. [29].

The search algorithm is a typical flooding-variant. It sends messages to all neighbors and thus floods the network with QUERY messages. Flooding the network is not efficient, Ripeanu et al.[29] showed that when using a TTL of 7, roughly 95% of all peers will receive the QUERY message. But not all peers receive a message once, because there is no knowledge on which peers have already received the message. This is illustrated in Figure 1.3, where the two peers on the left send each other messages which are unnecessary.

Risson et al. [30] measured that in 2000, when the Gnutella network had only 50,000 connected peers, QUERY messages alone consumed 1.7% of the total US

Internet backbone traffic. This is unacceptable and results in an unscalable network. Less capable peers will saturate their Internet connection with only QUERY messages, which in turn leads to a degradation of the search performance (because not all messages are forwarded by the saturated peers, resulting in parts of the network which are not reached).

Flooding does have one benefit and that is responsiveness, because the number of hops between the current peer and the peer with an item is at most the TTL, thus the speed of a search is relatively fast. The overall performance of the keyword search in Gnutella is low, which is supported by Ripeanu et al. [29] which reports that 92% of all messages in the network are QUERY messages.

Kazaa

Kazaa uses the FastTrack network. Introduced in 2001, it added the use of superpeers which help to improve efficiency in keyword search. These more powerful peers, which are elected by the program itself based on the number of available resources a peer has, are used to index the network. Ordinary peers are connected to several superpeers, which index which items these ordinary peers have. The indexes are used to produce search results. When an ordinary peer searches the network, it sends a message to the superpeers it is connected to. This message is flooded to all other superpeers (of which there are less), the indexes are checked and results are returned. More recent versions of Gnutella also implemented the use of superpeers to improve efficiency of its protocol [38].

The use of superpeers is a temporary solution, because eventually when the network increases in size the original problem of saturated peers will occur again. Only this time at the less capable superpeers, this is suggested by Sarshar et al. [32].

Another fact Kazaa is famous for, is its settlement of $100 million with 4 big record companies[1]. In December of 2003 the FastTrack network peaked at 4.8 million concurrent users, as described by Liang et al. [16].

eMule

In 2002 eMule was released. It connected to two different networks, eDonkey and Kad. We will only describe the Kad network, because it is based on a structured overlay. Which differs from Gnutella and Kazaa, because they use an unstructured overlay.

The Kad network is based on a DHT, which is a structured overlay. In the Kad network keyword search is then translated to locating the peer corresponding to the first keyword (the keyword is translated to a location on the ring, as described in the introduction). Then after finding the responsible peer the complete keyword list is send to it. This peer then looks up all the results for the search and returns

[1] Taken from: `http://en.wikipedia.org/wiki/Kazaa`

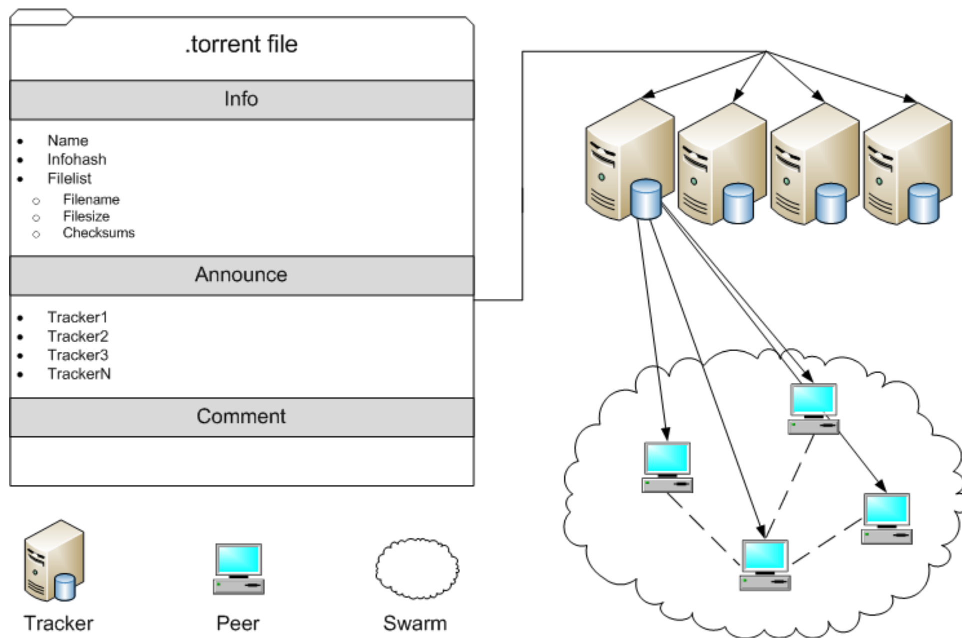


Figure 1.4: A Simple Overview of the BitTorrent network. After a .torrent file is downloaded from a website, it is used to connect to Trackers which know which Peers are present in the swarm.

the peers which should have files matching the keywords, described by Stuztbach et al. [37].

This works because when a peer publishes a file (or shares it) a publish message is created. This message is sent to all peers corresponding to a keyword of the file using the hash-function (and a couple of its neighbors, to reduce load of popular keywords). After 24 hours this process is repeated, as described by Steiner et al. [36].

This scheme allows for easy searches, but publishing is more complex. According to Steiner et al. 90% of all messages in the Kad network are publish messages [36]. Instead of an improvement over the 92%, which is achieved by Gnutella, this only changes the problem. Additionally, a DHT has more problems regarding churn making us prefer a solution using an unstructured overlay.

BitTorrent

Finally the BitTorrent network, developed by Bram Cohen[5], will be described. BitTorrent is the most popular P2P network currently used, according to an Internet survey conducted by Ipoque [34] between 2008 and 2009. In Germany 37% of all data-traffic is caused by BitTorrent.

BitTorrent started operating in 2001 and uses websites to perform searches. These websites provide a way to download a .torrent file, an example of which

is shown in Figure 1.4. A .torrent file contains a filelist which describes a file or collection of files which can be downloaded, for consistency we will call this collection of files the item. All peers which are currently downloading an item are located in a swarm. A separate swarm is created for each item. Inside a swarm peers are exchanging pieces of an item, using a mechanism which prevents free-riding (downloading an item without uploading). The rarest pieces of an item, pieces which only a couple of peers have, are downloaded first. This ensures that pieces which have a higher chance of not being able to be downloaded at a later stage will be downloaded now. Included in the .torrent file are features for error checking, which are used to verify the received pieces.

Additionally, in a swarm some peers have already completely downloaded the item. These peers are called seeders and help others to complete their download. The name or title of an item is often referred to as the swarmname and specified in the .torrent file.

Several servers, which are called trackers, are used for bootstrapping purposes. Using the list of trackers in the .torrent file and the infohash (the unique identifier) a list of peers which are currently inside the swarm can be retrieved. A connection is then made with these peers and due to optimistic unchoking some pieces are received for "free". This allows peers which do not have pieces to exchange (peers who have just started downloading), to receive pieces without uploading.

Unchoking is a term used to describe that a peer is sending a requested piece to another peer. Peers are unchocked if a peer is one of the best uploaders (it sends many pieces) or if we are interested in a piece that another peer has. Finally peers are optimistically unchocked, which is used to find better peers (a peer that is better than the best uploaders).

Since version 4.2.0 of the BitTorrent protocol some steps are made to move to a trackerless BitTorrent and thus removing one of the two centralized components. But a non-centralized solution for search has not been found jet. BitTorrent thus uses a combination of two centralized components (websites hosting the .torrent file, trackers maintaining lists of peers), but with no correlation to the original authors. As with Napster recent lawsuits against the central components in the network have led to the closure of some of them. Most notable Mininova (a website hosting .torrent files) and the The Pirate Bay tracker [2].

[2] http://torrentfreak.com/top-tier-bittorrent-sites-suffer-pain-in-2009-091229/

Chapter 2

Problem description

Given the current state as described in the introduction, a different approach is needed to find an efficient method to locate files in a P2P network without the use of a centralized component. In this chapter the semantic overlay is introduced, which is a possible solution.

2.1 Research question

The overhead of not having a centralized component, wasting 90% of all messages on search, creates an unscalable network. A requirement of a P2P network is scalability, thus this behavior is undesired.

But what causes this overhead? Voulgaris et al. [39] suggests that this is caused by the different requirements of search and data exchange. An overlay which is efficient in data exchange focuses on transmission speed whereas an overlay which is focused on search requires peers which the same interests to be close.

Voulgaris et al. [39] suggests creating an additional semantic overlay, thus using two overlays instead of one. Then this additional overlay is used only for searching and separates data from meta-data. Resulting in two overlays which can be designed to fit their requirements.

Semantic overlay

A semantic overlay is an overlay in which peers are clustered by taste. An example of such an overlay is shown in Figure 2.1, where Peer 1 and 3 and Peer 2 and 3 are connected. But no connection exists between Peer 1 and 2, because of not being similar enough (Peer 1 is interested in animals, Peer 2 in programming).

Different approaches exist for creating a semantic overlay. Crespo et al. [6] suggest creating a separate overlay for each group of peers. This approach is deployed on a P2P network which allows peers to share music. Groups are defined as a genre, all peers which are interested in "Rock" are put in an overlay. Peers interested in "Rap" are put in another overlay, but peers are allowed to join mul-

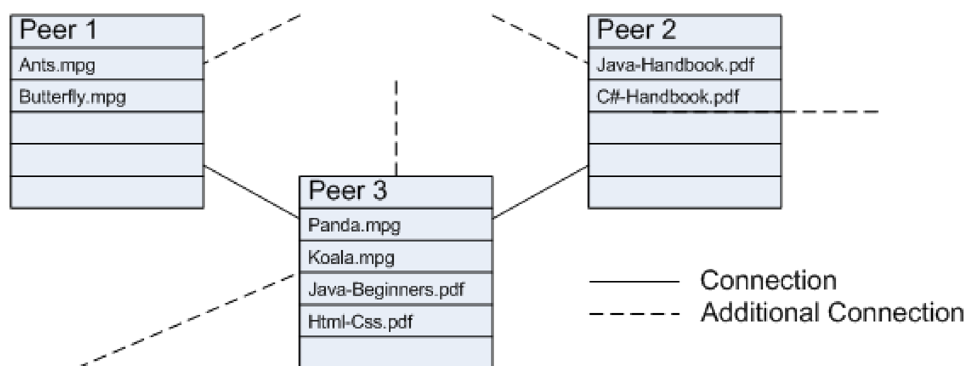


Figure 2.1: An Example of a Semantic Overlay, as Suggested by Voulgaris et al. [39]. Peer 1 and 3 are connected, because they are interested in Animals. Peer 2 and 3, because of Programming. No connection between Peer 1 and 2, because they do not have overlapping interests.

tiple overlays. Additionally, some groups have sub-groups further specifying the genre, e.g. “Soft Rock”. Which overlays to join is determined using a document classifier.

When searching for items, the search query is first analyzed to find the associated groups and the appropriate overlays are searched. This resulted in a decrease in the number of messages sent (461 while Gnutella needed 1731). Crespo et al. [6] used an acyclic network, which in practice would not occur thus the actual performance would be different. Additionally creating a document classifier for all datatypes (instead of only music) is difficult. Crespo et al. used an online database, which allowed items to be classified. Such a database does not exists for all datatypes.

Instead of creating multiple semantic overlays, Voulgaris et al. [39] suggest creating one additional overlay. In this semantic overlay, peers send each other information regarding their download history and other peers they know using a gossip protocol. This allows for efficient peer discovery and by comparing the download history, similar peers can be found. The benefit of gossiping is the control over the amount of data that is received/sent by adjusting the rate of gossiping. Additionally, a gossip protocol ensures that eventually almost all information will be exchanged between peers.

Peers in the semantic overlay connect to 10 neighbors/peers which are most similar. Similar peers are found by comparing the download history. Searching is done using only these neighbors, no flooding is required thus only 10 messages are sent. In experiments a hit ratio of 36% is achieved (hit ratio, the percentage of files downloaded by a peer found at its neighbors).

Voulgaris et al. [39] also discusses the bandwidth requirements needed for such an additional overlay. These range from 213 to 640 bytes per second, which is, as suggested by Voulgaris, small if not negligible compared to the bandwidth consumed by downloading files.

Tribler

Tribler [1, 25] is a BitTorrent client which implements a semantic overlay as described by Voulgaris et al. [39]. Peers, using the semantic overlay, exchange information regarding their download history using a protocol called BuddyCast. Information on peers, which is received using BuddyCast, is stored locally in a database (up to 5000 items). Which is then used by Tribler to find the most similar peers to create a neighborhood (the 10 most similar are used, and are called taste buddies), as suggested by Voulgaris et al. [39]. Buddycast messages are sent every 4 hours, to known peers. This rule and others are implemented as "rate control" and limit the amount of bandwidth spent on BuddyCast (no specific amount is specified).

After finding peers with similar interest, search is then much more efficient (as suggested by Voulgaris[39]). Combining the local database and those of the 10 connected neighbors, peers can search an effective 55,000 items as stated in the TriblerSpec[3]. Which eliminates the need for a flooding during keyword search.

When searching for an item, first the local database is searched. Then a QUERY message is sent to all 10 neighbors. This message contains an id and a query. The neighbors respond with a QUERY-REPLY message, which contains a list of items which match the query. And per item the name, size, number of leechers, number of seeders and category is described. This list does not contain the actual .torrent files yet, because this would increase the size of the reply by a large factor.

Upon receiving a QUERY-REPLY message, the list is added to the search results. First only local results of a query are shown, but after receiving a QUERY-REPLY message this list is updated. If a user clicks on a remote result, the corresponding item is retrieved which contains all information needed for downloading (this item is the .torrent file as described in Section 1.2.1). After the .torrent file is retrieved, Tribler starts downloading it similar to a "normal" BitTorrent client.

The use of taste buddies is not the only innovative idea being currently implemented in Tribler. It also integrates some additional social features to improve the P2P experience. An example of this is the notion of friends. These peers can be used to improve download speed by solving the asymmetric part of a common ADSL line (having more download speed than upload speed). The BitTorrent protocol prevent free-riding by restricting the amount of data uploaded to a peer to be more or less equivalent to the amount received from that peer. This causes peers with asymmetric upload speed to never saturate their connection. Friends will help a peer to improve download speed by also downloading the file and sending parts of it to the peer for free.

Similarity function

A semantic overlay is in theory an efficient method to do keyword search, because only 10 request messages are sent for each search query (one message per neighbor). But the quality of the results depend on how well peers are matched. Similar peers are found by applying a similarity function on the download history of known

peers. Similarity functions can determine the similarity between items or objects, but in this case peers. And because better matched peers result in an improved overlay, a lot depends on the performance of this similarity function.

Which is where Tribler hits a snag. The current similarity function is not performing that well (both in terms of matching users and computational efficiency). Combining this results in the following research question:

> Find a user similarity function from which an efficient P2P topology for keyword search emerges.

2.2 Document layout

The rest of this document is structured as follows: Chapter 3 will describe the Evaluation Infrastructure, which introduces the various methods by which performance of a similarity function can be determined. Section 4 will discuss the dataset, which was especially created for this research and was used to perform the evaluations on. Chapter 5 will introduces the different similarity functions which have been tried and tested. And the various optimizations which modified the functions to be used in a P2P environment. Chapter 6 presents the results of all evaluations of similarity functions, the evaluation is divided into two parts. One describing the matching performance of the similarity functions, the other the computational efficiency. Chapter 7 will discuss all results as described in Chapter 6 and finally in Chapter 8 we will conclude with the conclusion and future recommendations.

Chapter 3

Evaluation infrastructure

In order to evaluate the performance of the different similarity functions, an evaluation infrastructure was created. This setup consists of two parts, one being the dataset which will be described in Chapter 4. The other part, the evaluation methods which express the performance of the similarity functions into scores. This allows the similarity functions to be compared on both Effectiveness and Efficiency.

Manually evaluating the performance of the similarity functions is not possible, thus a technique used in Collaborative Filtering was used. When calculating similarity based upon ratings, or in the case of Tribler if a user has downloaded a file yes/no, then parts of this data can be used to check the performance. Using 80% of all data available on a user, the train set, we calculate the similarities between other users. The remaining 20% allows us the check the performance (which is referred to as the testset). In the case of Tribler we are thus predicting which items a user will download based on 80% of its history and then checking the number of correct recommendations on the remaining 20%. Dividing the data in a train and testset is common practice in Collaborative Filtering [7, 8, 10].

But the similarity functions calculate a similarity between the active user and the other users. These similarities need to be converted to a list of recommended items

	Item 1	Item 2	Item 3	Item 4	Similarity
Active User	1	0	0	0	
User 1	1	0	1	0	0.5
User 2	1	1	1	0	0.4
⋮					
User 10	1	1	1	1	0.3
Total		0.7	1.2	0.3	

Table 3.1: Converting User Similarity to Item Recommendations. If a User has an Item not downloaded by the Active User its Similarity is used as an value. The totals determine which Items are recommended.

which can then be compared to the trainset.

This list is constructed as shown in Table 3.1. First a similarity between the active user and each other user is calculated. Then the top-10 most similar users are selected, which is similar to the number of neighbors Tribler uses in the semantic overlay. And using these most similar users a recommendation is calculated for each item the current user has not downloaded. This recommendation is simply the sum of all user similarities which have downloaded this item. Thus as shown in Table 3.1, using the similarities of the other users Item 2 gets a recommendation value of 0.7, Item 3 of 1.2 etc.

The list is sorted and the top-N most recommended items are used for evaluation. In the example the recommended list would be Item 3, Item 2 and finally Item 4. If one of these items were to be present in the testset of the active user, then the similarity function found 10 users which are at least somewhat similar.

3.1 Evaluation statistics

Using only the number of correctly recommended items does not provide an accurate statistic. Thus some more sophisticated statistics will be introduced in the following paragraphs.

3.1.1 Precision and recall

When comparing a list of recommended items to the test set, precision and recall can be calculated. This is a common statistic used to evaluate the performance of recommendations.

$$precision = \frac{nr_correct}{nr_recommended_items} \tag{3.1}$$

$$recall = \frac{nr_correct}{max_correct} \tag{3.2}$$

Precision (Function 3.1) states the percentage of correct items in the list of recommended items. And recall (Function 3.2) is the percentage of relevant items which are predicted.

Then by changing the number of items that are recommended, the performance can be visualized. Usually when more items are recommended, recall increases and precision decreases. This is evident after looking at the formulas, because when more items are recommended the chance to recommend a correct item also increases, thus recall increases. But the number of incorrect items that are recommended also increases, thus reducing the precision.

In our research the recall could be compared to the hitrate, as introduced by Voulgaris et al. [39]. Only more strict, instead of looking at the number of items downloaded by this peer which are present at its neighbors, the combined top-N items of those neighbors are used, with the similarities of those peers as a measure of importance.

3.1.2 F1 score

The Precision and Recall statistics can be converted into a grade. In collaborative filtering a common method is the F1 score, as introduced by Rijsbergen [28]:

$$F1\ score = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{3.3}$$

The F1 score is the harmonic mean of the precision and recall numbers, which can be used to easily compare the performance of the similarity functions on both statistics simultaneously.

3.1.3 NDCG

Another method to evaluate the performance of the recommendations is the use of Normalized Discounted Cumulative Gain (NDCG [13]). NDCG compares the list of recommended items to the optimal list. When an item is recommended but at a position that is not optimal (e.g. position 10 instead of 1), the score will be lower. A higher NDCG score is thus indicative on the quality of the order of the recommended items list.

NDCG is best explained by using a running example. During the evaluation the items in the test set of a user are known. For an ideal recommendation these items are expected to be at the start of the recommended list. Additionally, a relevancy of 1 is assigned, because these items are actually downloaded by the user (relevancies are further explained in Section 3.1.3). For example, if a user has 5 items in its testset and 10 items would be recommended, the ideal list would be:

$$Ideal = \{1, 1, 1, 1, 1, 0, 0, 0, 0, 0\}$$

This is referred as the ideal gain vector. In real world experiments, the chance that all items are recommended in this order will be very small. Thus an ordinary gain vector if 10 items are recommended could look like:

$$Ordinary = \{1, 0, 0, 0, 1, 1, 1, 0, 0, 1\}$$

Converting into a score

To express this vector into a score the Cumulative Gain (CG) is introduced:

$$CG_p = \sum_{i=1}^{p} G_i \tag{3.4}$$

CG is a summation of the first p items and their relevancies. In both the ideal and real world gain vector, the CG_{10} (thus a summation of the first 10 relevancies) would be the 5. But there is a difference in quality between the two vectors. If this list would be used in a search-engine, having the first 5 results to be relevant is better than having those relevant items scattered over multiple pages. To emphasis

the importance of the positions of the relevant items CG is extended to Discounted Cumulative Gain (DCG).

$$DCG_p = G_1 + \sum_{i=2}^{p} \frac{G_i}{\log_2 i} \tag{3.5}$$

DCG uses a log scale to discount the relevant items at lower positions in the list. For the two examples this would result in:

p	1	2	3	4	5	6	7	8	9	10
IDCG	1.00	2.00	2.63	3.13	3.56	3.56	3.56	3.56	3.56	3.56
DCG	1.00	1.00	1.00	1.00	1.43	1.82	2.18	2.18	2.18	2.48

Comparing both DCG_{10} values shows that the ideal list has a higher value, which is desired. But before this statistic can be used as a score, it needs to be normalized. Because now the DCG value will continue to increase if the number of recommended items is increased. This is done by comparing the differences in performance between the IDCG and DCG at each step and converting this to an average performance.

$$NDCG_p = \frac{\sum_{i=1}^{p} \frac{DCG_i}{IDCG_i}}{p} \tag{3.6}$$

In the running example the NDCG score would be:

p	1	2	3	4	5	6	7	8	9	10
NDCG	1.00	0.75	0.63	0.55	0.52	0.52	0.53	0.54	0.55	0.56

The resulting $NDCG_{10}$ score would be 0.56.

Relevancies

One of the benefits of the NDCG is the notion of graded relevancies, which state that some items can be more relevant than others. This allows us to use categories to extend the ideal gain vector such that it consists of the items in the test set followed by items matching the categories of those items. A relevancy of 0.5 is set for items which have the same category. Resulting in an ideal gain vector of:

$$Ideal = \{1, 1, 1, 1, 1, 0.5, 0.5, 0.5, 0.5, 0.5\}$$

How this relates to the Tribler data will be described in Section 4.2.4.

3.1.4 Computational efficiency

The performance of a user similarity function is determined by two factors. The obvious one is how well it is in matching similar users, which can be determined

with the methods as introduced above. But additionally computational efficiency is a factor. There are 3 scenarios in which similarity has to be calculated:

- A new user is discovered.
- The profile of a known user is updated.
- The profile of the active user is updated.

When a new user is discovered or when its profile is updated a Single Update needs to be run. This calculates the similarity between the active user and the updated one. Because no other users are changed there is no change in the similarity between them. A Single Update is a common operation, thus needs to be fast. When a user downloads a new item, then similarity between it and all other known users could change, thus a Full Update needs to be calculated. This operation will take longer to complete, because more similarities have to be calculated, but still needs to be fast.

A similarity function needs to make a trade-off between raw computational efficiency and matching performance. Building a hugely complex similarity function usually improves the matching of peers, but at a computational cost and the other way around. During the evaluation the time it takes to do a Full and Single Update is measured. The Single Update performance is the absolute worst case, measuring how long it will take to compute similarity between this user and the user which has downloaded the most items. More items will cause more overlap, thus this is the worst case.

3.2 Significance

After running the experiments and applying the statistics as described above, the results will be presented in graphs and tables. These will give an indication of the performance differences between the similarity functions, but no actual conclusions can be extracted from them. To prove that one result is better or worse than another significance has to be proven. This is a term in statistics which is used if a difference between two values is "unlikely to have occurred by chance". Having a significant difference between the results thereby will say something on the reliability of this difference.

Significance will be calculated by using the raw output of the F1-Score and NDCG statistics for each user. These values are then written to a file and processed using SPSS[1]. SPSS is a complex piece of software and can be used to perform all kinds of statistical analysis on data.

Significance is calculated using an ANOVA with a Tukey HSD post hoc. This method can calculate the significance between multiple groups, or in this case similarity function scores, in one run. The benefit calculating the significance in one

[1] http://www.spss.com

run, is that no human errors between runs can occur, as described by Park [24]. Park actually advises the ANOVA, if significance has to be calculated between more than two groups. The significance level is kept at the default value of 0.05.

To ensure the correctness of the significance testing, an expert[2] was consulted which has verified the results.

[2] Thierry Worch, `http://www.logic8.nl`

Chapter 4

Dataset

Another important part of the evaluation infrastructure is the dataset. A common mistake is to create or use a dataset which does not represent the actual situation. A similarity function which needs a large amount of knowledge on a user, but then performs well. Could when compared to a similarity function which does not need this amount of knowledge, but has an average performance, perform much worse if used in a situation when less knowledge is known.

The dataset used to evaluate the performance of the similarity functions should thus accurately match the structure of the Tribler network. Density of a dataset is an important indicator if a dataset is difficult to compute similarity matching's on. A dataset which is relatively dense (each user has downloaded a high number of items) has more knowledge on its users, resulting in different performing similarity functions.

$$Sparsity = 1 - \frac{NrOfDownloads}{NrOfUsers \cdot NrOfItems} \tag{4.1}$$

Sparsity is the reverse of density and calculated as shown in Equation 4.1. To create a dataset which accurately represents the Tribler network, data from the Tribler network was used which was extracted from the logs of the superpeers.

The Tribler network currently consists of 8 superpeers, which are used for bootstrapping purposes. Using a static list included in the Tribler software the superpeers are contacted when a peer has just installed the software and does not know any other peers. But after the initial bootstrapping, peers continue to respond to buddycast messages from the superpeers. Resulting in the superpeer logs (all responses are written to logfiles), files which contain the preferences or items which a user has downloaded until then.

Collecting, parsing and extracting data from the superpeer logs took a month to complete. Logs were available from 10/2006 until 06/2009.

From this dataset a subset was used during evaluation. This subset consists of the 50,000 users which have downloaded the most items. Only items for which the .torrent file was present were used, because these contain the meta data which will be used during the experiments.

	Users	Items	Number of downloads	Sparsity
Complete dataset	424,485	819,732	2,318,587	0.9999993
Subset	50,000	343,541	1,173,072	0.99993
MovieLens [21]	6,040	3,900	1,000,000	0.9745
MovieLens [14]	943	1,682	100,000	0.9369

Table 4.1: Datasets compared. Highlighting different ways to construct subsets. MovieLens is limited in Number of downloads, our subset is only limited by users. Sparsity indicated the difficulty of a dataset, a higher number is a more empty dataset.

Table 4.1 shows the complete dataset, the subset and two different versions of the commonly used MovieLens dataset [14, 21]. Only comparing the sparsity levels, shows that the two MovieLens datasets are much more dense. Additionally it can be seen, that these datasets are created by limiting the number of downloads/rankings. This results in a dataset which is restricted both by only selecting the top-X most downloaded items and the top-X users which have downloaded the most items.

By creating a dataset which is only limits the number of users used, we think that the dataset is a more fair representation of the Tribler network. And except for removing the items for which no .torrent file was present, the subset is not modified.

Additionally, it can thus be said that if a user has downloaded enough items to be part of this top-50,000 subset, then the similarity function resulting from this research would be the optimal choice for this user. A user is part of the subset if he downloaded 7 or more items, but only 80% of these items are used in the evaluation or 5 items. The resulting similarity function would thus be the optimal choice for 92,089 users, which all have downloaded 5 or more items. In the smaller variant of the MovieLens database all users have rated at least 20 items.

Simply using a variant of the MovieLens dataset could have caused different behavior of the similarity functions. Not only is the density higher, the sheer number of unique items which occur in the Tribler network causes unique problems. Any conclusions resulting from experiments run on the MovieLens datasets would have been useless.

It has to be noted that when implemented in Tribler, the megacache (the local database each peer has) will not be as complete as the subset which is constructed in this chapter. Using BuddyCast, information is exchanged between peers which eventually will cause each peer to have a database in which 5,000 items and the peers which have downloaded those items are stored. This database will evolve to a database consisting of items which are most relevant to the peer, and should result in a database containing all information which is used by all similarity functions.

4.1 Parsing

The superpeer logs were parsed using python scripts, first these files were reduced to a file only consisting of buddycast messages.

Buddycast messages contain: Timestamp, Type of message, Ip address, Port, Permid, Overlay version, Message Content (consisting of preferences, taste buddies and random-peers). Then from these reduced files, information was extracted and stored in 25 separate files describing:

- Which items an user has downloaded
- How many items an user has downloaded
- How many times an item has been downloaded
- When an user was connected to the network

Additionally, files were created mapping the original hashes of an user or file to an integer, and specific files were created to generate plots.

4.2 Data statistics

Using the parsed superpeer logs and gnuplot[1], graphs were created to give insight into the Tribler network. This was the intended use of the superpeer logs, but until now not previously done. Additionally, a modified version of the parse script is now run daily to further understand the network. In the following subsections, different aspects of peer behavior in the network is shown.

4.2.1 User activity

Figure 4.1 shows how many users were online each day, specified by the version they were using. Some users were not contacted directly by a superpeer, but discovered indirectly using another user. These random peers or taste buddies are shown in the graphs, but if not previously seen without a version because this information not specified for random peers and taste buddies. Important events such as the release of a new Tribler version is shown in the time line. A server crash resulted in the missing data between 17 November 2006 and 21 may 2007.

A peak in usage was achieved after both BBC-News[2] and Slashdot[3] posted articles on Tribler. As can be seen in Figure 4.2, since then a steady decline in usage can be observed, but at the end of the analyzed logs still roughly 5000 users use Tribler daily. Additionally, version 3.3.4 is still the most used version of Tribler.

[1] `http://www.gnuplot.info/`

[2] `http://news.bbc.co.uk/2/hi/technology/6971904.stm`

[3] `http://it.slashdot.org/story/07/08/29/1924237/Internet-Bandwidth-to-Become-a-Global-Currency`

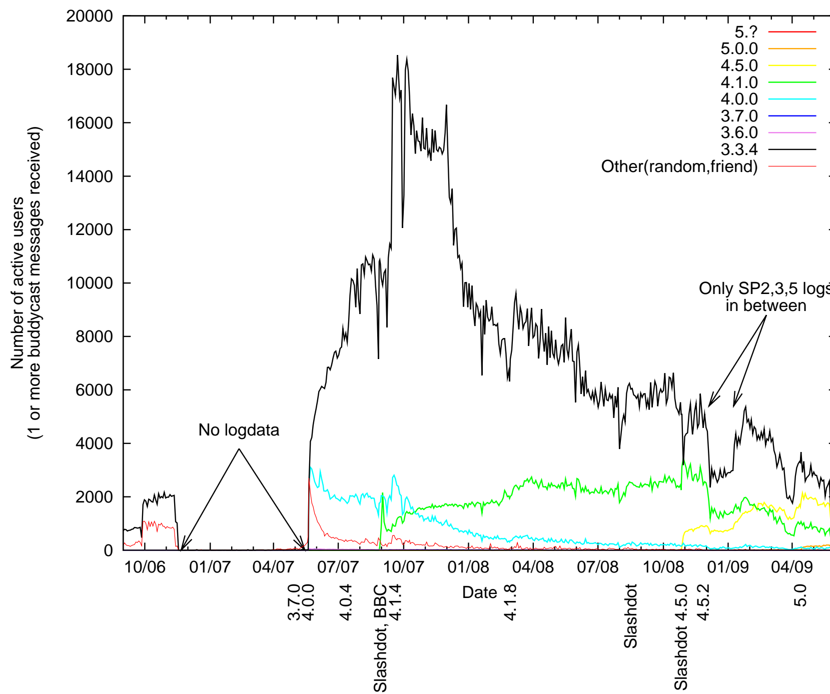


Figure 4.1: User activity by day, shown for each version. Version 3.3.4 is still the most popular, but its use is decreasing steadily.

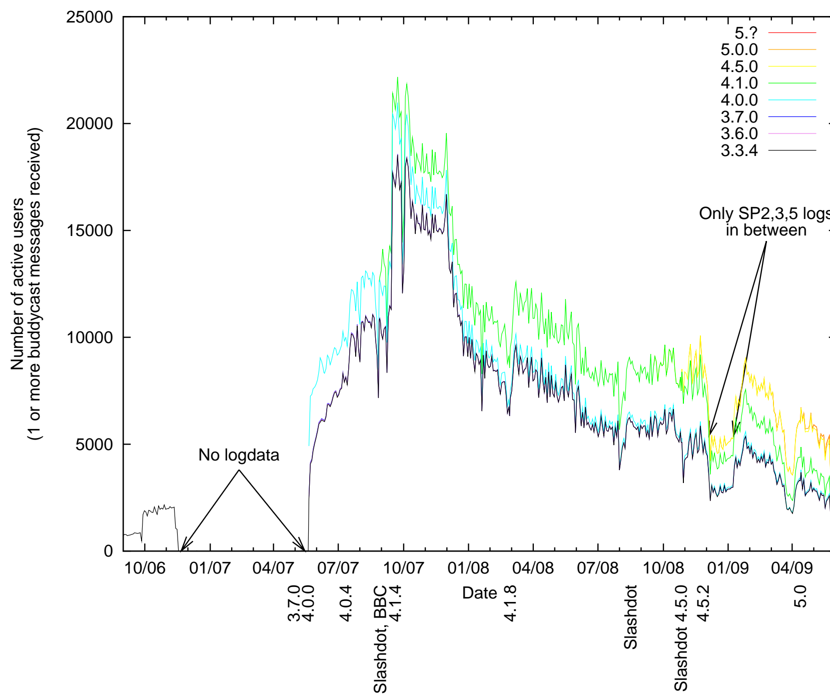


Figure 4.2: User activity by day, cumulative usage. Overall usage is decreasing since the news posted on Slashdot and the BBC, but still roughly 5000 users use Tribler daily.

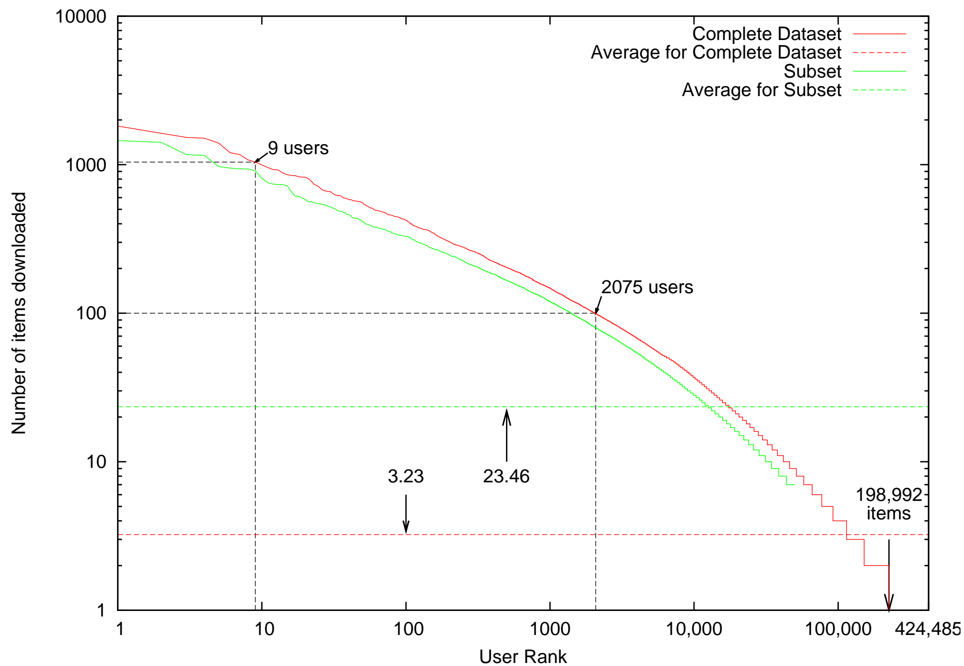


Figure 4.3: User distribution, 9 Users have downloaded more than 1000 items. On average 3.23 items have been downloaded by 424,485 users. For the Subset, 50,000 users have downloaded 23.46 items on average.

Two other articles posted on Slashdot did not show an increase in usage. The number of comments on these articles is higher, which is strange because one would expect more popular articles to have a higher impact on usage. The only difference is the BBC not posting an article and the latter articles having a slightly less controversial subject.

Another particularly interesting observation, that can be seen in Figure 4.1, is when a new version is released the version it replaces also increases in popularity. This is probably due to server overloads and alternative mirrors which serve the old version.

4.2.2 Rankings

The following section ranks users and items in different categories. The difference between the complete dataset and the subset is plotted.

Figure 4.3 shows the distribution of the length of user profile (the number of items a user has downloaded). 9 users have downloaded more than 1000 items, 2075 have downloaded more than 100. But more important, 198,992 users have only downloaded one item. The average number of items a user has downloaded is 3.23.

Because of removing items for which the .torrents files are not present the users

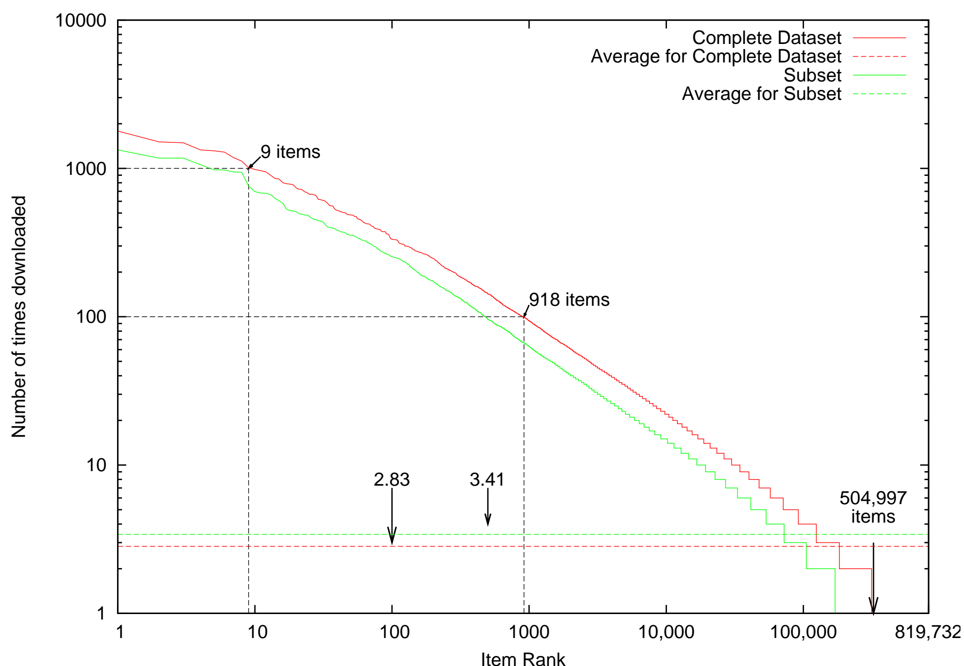


Figure 4.4: Item distribution, 9 items have been downloaded more than 1000 times. On average an item has been downloaded 2.83 times. But 504,997 items have only been downloaded once.

in the subset have less items in their profiles. But besides this small difference the two datasets behave the same.

Figure 4.4 shows the distribution of items, 9 items have been downloaded more than 1000 times. 918 items have been downloaded more than 100 times, but more important, 504,997 items have only been downloaded once. This leads to a very sparse dataset, because more than 61% of the total number of items (819,732) have been downloaded once. On average an item is downloaded 2.83 times.

The huge number of items is caused by the unstructured nature of the dataset. In P2P every user can create and upload his version of an item, which results in a large number of duplicates. Liang et al. determined that in the FastTrack network a single popular song had 48,613 different versions [15]. In contrast a "normal" dataset used for recommender research is structured, an example would be the Amazon.com dataset, this means that every book will only occur once and is assigned to predefined categories.

The small difference in the complete dataset and the subset indicates that this subset is still accurately representing the original dataset. By removing the users which have not made it into the top 50,000 all items have now been downloaded less, which is as expected. In the subset there are 170,435 items which have been downloaded once out of 343,541, which translates to 49.61%. On average an item has been downloaded 3.41 times.

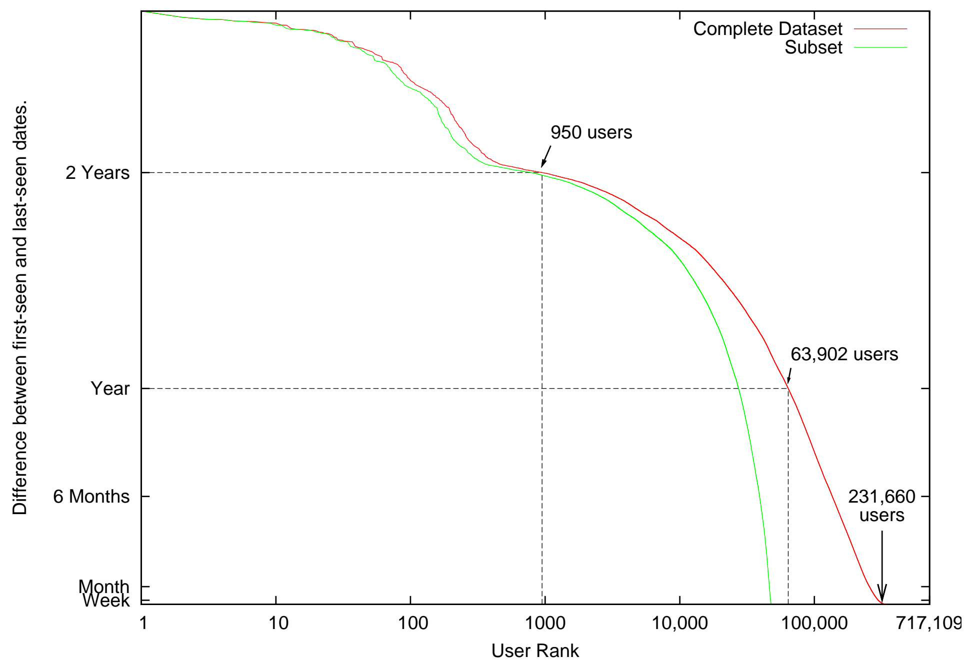


Figure 4.5: User loyalty, 950 Users have been using Tribler for more the 2 years. The dip is explained by the missing data. As can be seen, some users which have been using Tribler for a short period are included in the Subset.

Using the first and lastseen dates of a user, user loyalty was plotted. A loyal user (one that keeps using Tribler), has a large difference between those two dates. Figure 4.5 plots this information. As can be seen, a large number (63,902) users are/have been using Tribler for more than one year. The dip in users that are using Tribler just more than 2 years is probably due to the missing data between 17 November 2006 and 21 may 2007 (Shown in Figure 4.1). Users that started using Tribler during this period, will register as starting to use Tribler at the 21th of may 2007.

Because the data has originates from different superpeers (8 in total), the first and lastseen dates are not all that reliable. No information is known on time-synchronization between the servers. But because every peer is contacted every 4 hours we have an accuracy of 4 hours. In total 346,813 users have used Tribler for more than 4 hours, the others (370,296 users) have used Tribler for a shorter period and could be considered at one-minute users.

Another interesting observation is the difference between the subset and the complete dataset. In the subset only 27,288 users have used Tribler for more than 1 year. One would expect that the 50,000 users which have downloaded most items would also be using Tribler the longest. This is apparently not the case, with one user being in the subset which has used Tribler for only 65 seconds or less than 4 hours.

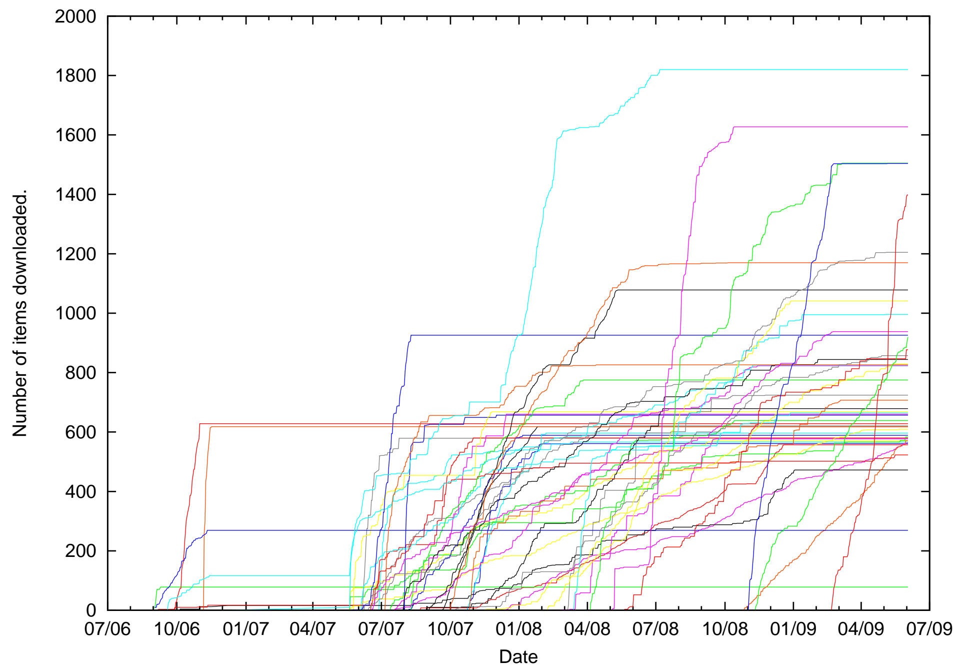


Figure 4.6: User download statistics. A pattern appears in which users stop using Tribler after downloading 600-800 items.

4.2.3 Top-N user behavior

Selecting only a top number of users, allows us to get more insight into the behavior of these users. Figure 4.6 shows the cumulative download behavior of the top-50 most active users (users which have downloaded the most items). Using this cumulative plot, it can be seen that a large number of users stop using the program after downloading 600-800 items. Users that stop using the Tribler software also frequently do this in a on/off matter. Meaning there is no gradual decent in the number of items downloaded, but a sudden stop.

This could be due to a reinstall, or a problem in the software which causes it to become unworkable after downloading N files. A problem which is further investigated upon in Section 6.2.1.

4.2.4 Ground truth

The NDCG (Section 3.1.3) statistic uses the notion of relevancies for items. For items which fall into the same category the relevancies are set at 0.5. The ground truth set specifies these categories, such that if an item is recommended which is similar it will not be threated as an error. Allowing us to identify near misses, which are not the same as incorrect predictions. E.g. when a user downloads a backup-copy of his favorite movie (which he/she already bought on dvd), containing dutch

subtitles and is recommended part 2 of that movie only containing Spanish subtitles. This should not be treated as an error. Some similarity functions might have a tendency to recommend those near misses instead of the actual items, due to using more meta-data.

During the construction of the ground truth set, the tf-idf function was applied to determine important words in the swarmnames.

$$td - idf\ (i, j) = \frac{N_{i,j}}{\sum_k N_{k,j}} \cdot \log\left(\frac{N}{N_i}\right) \tag{4.2}$$

By counting the number of times a word occurs in a swarmname, normalizing this number by the overall size of the swarmname. And multiplying this by the log of the total number of items divided by the number of items containing this word.

The extracted words were written to a file and then further processed to calculate word-co-occurrence. From these two lists words were selected by hand and items containing these words were selected and written to a category file. These category files per word(s), were then manually checked to filter out errors. The result of this process was the categorization of 40,684 items or almost 12% of the subset into 34 categories. These categories consisted of popular TV-Series, Applications and Movies and are further specified in Appendix A.

4.3 Tribler and Zipf's law

The plots as shown in Figure 4.3, 4.4 and 4.5 are called rank/frequency plots, as described by Newman et al. [23]. Instead of a frequency, different quantities can be used, such as number of files shared etc. An interesting observation is that if such plots are created with a log-log scale, often an almost straight line emerges.

One of the first to observe this behavior was Zipf [41], since then this distribution is referred to as Zipf's law or power law. This behavior occurs in a wide variety of quantities. Newman et al. [23] gives 12 examples ranging from word frequencies in English, to crater diameters.

More recently Zipf's law was shown to exists in websites, as described by Adamic et al.[2]. Many websites only have a few pages, but only some have millions of pages. Additionally, most websites only receive a couple of visitors, while some have millions.

Zipf's law was shown to exists in the number of files shared in Kazaa, as shown by Iamnitchi et al. [12], and eDonkey, as shown by Handurukande et al. [11]. The number of items downloaded by users in the Tribler network (as shown in Figure 4.3), equally behaves according to Zipf's law, which indicates that similar user dynamics underlie these networks.

The combination of a semantic overlay, with an optimized similarity function would thus additionally provide a method to improve search in Kazaa and eDonkey. Which are networks with magnitudes more users than Tribler (Kazaa had 4.8 million concurrent users, as described in Section 1.2.1).

Chapter 5

Candidate solutions

In this chapter all implemented similarity functions will be introduced. But first some terminology has to be introduced.

User preference data can be represented in a User×Item matrix, as shown in Table 5.1. In this matrix each row is called a userprofile or uservector. In this userprofile only the items which have a rating are added, or in the case of Tribler it consists of the items which a user has downloaded. The length of a userprofile is then the number of items a user has downloaded.

In the functions below, I_u represents the profile of User u. L_j is the set containing all users which have downloaded item j. And finally $v_{u,j}$ is a boolean value indicating whether User u has downloaded Item j (0 not downloaded, 1 downloaded).

	Item 1	Item 2	Item 3	Item 4
User 1	1			1
User 2	1	1		
User 3		1	1	
User 4		1		1

Table 5.1: The User×Item matrix. If a cell contains a 1 then this item was downloaded by this User.

5.1 Known techniques

During the literature research, some well-known similarity functions which are used in the collaborative filtering field were identified. From these functions a selection was made consisting of similarity functions which were applicable to the Tribler network.

The download history of a user only consists of boolean values (if a user has downloaded an item yes/no), thus some similarity functions could not be used. An example of such a function is the well-known Pearson's correlation coefficient, as

described to Breese et al. [4]. The Pearson's correlation coefficient includes an average rating, which in Tribler will always be equal to 1. This average rating is then compared to the rating for the current item, which also equals to 1. Resulting in a similarity function which will compare 1 to 1 and strange behavior.

The functions which are applicable will be described in the next paragraphs.

5.1.1 Tribler

The first function is the currently implemented similarity function in Tribler, TriblerOld:

$$TriblerOld\left(u_1, u_2\right) = \frac{k}{|I_{u_1}|} \cdot \sum_{j \in I_{u_1}} \frac{v_{u_2,j} + \mu \cdot \frac{|I_{u_2}|}{|I|}}{|L_j| + \mu} \tag{5.1}$$

The function basically is a summation over all items of a user and checking if the other user also has these items. Additionally a normalization scheme is included which compares the length of the other user's profile and the popularity of this item. k is a scaling factor (which is set to $100,000$ in Tribler), μ is another scaling factor (and is set to 1 in Tribler).

This function is based on Bayesian Smoothing using Dirichlet Priors, as described by Zhai et al. [40].

$$p\left(w|d\right) = \frac{c\left(w, d\right) + \mu \cdot p\left(w|C\right)}{|d| + \mu} \tag{5.2}$$

This function was proposed to estimate the probability of a document matching a query. d is the document, w is a word occurring in the query, $c\left(w, d\right)$ is the number of times this word exists in this document and $p\left(w|C\right)$ is the popularity of this word given all seen words.

Then by treating w as the list of items of User 1 and d as the list of items of User 2 both functions somewhat overlap. $c\left(w, d\right)$ then equals to $v_{u_2,j}$, but the other two components differ. $\frac{|I_{u_2}|}{|I|}$ prefers users with a larger profile, instead $p\left(w|C\right)$ represent the popularity of this item compared to all items, thus $\frac{|I_j|}{|I|}$ would be expected.

Next, $|L_j|$ does not equal to $|d|$, or the sum of the number of times each item of User 2 is downloaded is not equal to the number of items User 2 has downloaded.

Why these changes were made in the function is not documented, but the changes causes users with larger profiles and items which have not been downloaded much to be preferred. Which seems reasonable, still why divide k by the length of a user his profile and multiply it with the calculated similarity? This seems like an implementation specific boost in order to cope with rounding errors.

5.1.2 Tribler optimized

The computational performance problems regarding the current TriblerOld algorithm were already known before this research, thus an optimized version was

created. This function behaves like the original Tribler function (which will be empirically proven in Section 6.1), but reduces the computational cost of the function.

TriblerOpt is a rewrite of the TriblerOld function, which allows for some optimizations.

$$TriblerOld\,(u_1, u_2) = \frac{k}{|I_{u_1}|} \cdot \sum_{j \in I_{u_1}} \frac{v_{u_2,j} + \mu \cdot \frac{|I_{u_2}|}{|I|}}{|L_j| + \mu} \tag{5.3}$$

$$= \frac{k}{|I_{u_1}|} \cdot \left(\sum_{j \in I_{u_1}} \frac{v_{u_2,j}}{|L_j| + \mu} + \sum_{j \in I_{u_1}} \frac{\mu \cdot \frac{|I_{u_2}|}{|I|}}{|L_j| + \mu} \right) \tag{5.4}$$

$$= \frac{k}{|I_{u_1}|} \cdot \left(\sum_{j \in I_{u_1 \wedge u_2}} \frac{1}{|L_j| + \mu} + \sum_{j \in I_{u_1}} \frac{\mu \cdot \frac{|I_{u_2}|}{|I|}}{|L_j| + \mu} \right) \tag{5.5}$$

$$TriblerOpt\,(u_1, u_2) = \frac{k}{|I_{u_1}|} \cdot \left(\sum_{j \in I_{u_1 \wedge u_2}} \frac{1}{|L_j| + \mu} + \mu \cdot \frac{|I_{u_2}|}{|I|} \cdot \sum_{j \in I_{u_1}} \frac{1}{|L_j| + \mu} \right) \tag{5.6}$$

In Equation 5.4 the first enumeration can be rewritten because $v_{u_{2,j}}$ equals 1 if both users have downloaded this file. Thus the enumeration only needs to consider items downloaded by both users, resulting in Equation 5.5.

In the resulting TriblerOpt function (as shown in Equation 5.6), a Full Update (calculating similarities between all other users, see Chapter 3.1.4), can reuse the first division and the second enumeration to reduce complexity. By calculating this value only once and reusing the value, because they only depend on User 1.

5.1.3 Cosine similarity

When searching for alternatives for the TriblerOld function, the Cosine function is a logical choice. It is often used in the literature and as shown by Breese et al. [4] performs well.

$$Cosine\,(u_1, u_2) = \sum_{j \in I} \frac{v_{u_1,j}}{\sqrt{\sum_{k \in I_{u_1}} v_{u_1,k}^2}} \frac{v_{u_2,j}}{\sqrt{\sum_{k \in I_{u_2}} v_{u_2,k}^2}} \tag{5.7}$$

In Tribler the user profiles consist of boolean values, this causes $v_{u_1,k}^2$ to be equal to 1. Additionally, if one user did not download item j the multiplication will be 0, due to $v_{u_1,j}$. This allows us to simplify the function into:

$$Cosine\,(u_1, u_2) = \sum_{j \in I_{u_1} \wedge I_{u_2}} \frac{1}{\sqrt{|I_{u_1}|}} \frac{1}{\sqrt{|I_{u_2}|}} \tag{5.8}$$

M_{10}	M_{11}	M_{01}	M_{00}	
■	■			User 1
	■	■		User 2

Table 5.2: Similarity Coefficients. The gray area represents a part of the items which a User has downloaded. M_{10} are the items which User 1 has downloaded but User 2 did not. M_{11} are the items which both Users have downloaded.

Or a multiplication of the number of items both users have downloaded, with a value determined by the number of files downloaded by both users.

Breese et al. [4] suggested another normalization techniques for the Cosine function, which was tried. This different normalization just uses the length of a userprofile and does not apply a root to it. After rewriting it to use boolean values, it results in:

$$CosineMod\left(u_1, u_2\right) = \sum_{j \in I_{u_1} \wedge I_{u_2}} \frac{1}{\left|I_{u_1}\right|} \frac{1}{\left|I_{u_2}\right|} \tag{5.9}$$

5.1.4 Similarity coefficients

The following functions are similarity coefficients. These functions are used primarily to group items, based upon boolean data. Grouping items allows for optimizations in for example the design of a cellular manufacturing system [9]. A matrix would be used describing for each cell-phone which components were needed in order to manufacture them. Then cell-phones which needed the same components could be identified and built sharing some manufacturing components.

The functions all exploit the same basic knowledge on items to calculate similarity. To improve readability these are defined in Table 5.2 and basically they represent the number of items a combination of the two users have downloaded. E.g. M_{11} = the number of items which both users have downloaded, M_{10} = the number of items which only the first user has downloaded etc.

Over time a large number of coefficients have been created, mainly due to the application of these functions in different fields (which all reinvented the wheel). The functions use different combinations of M_{11},M_{01} etc to calculate similarity between two users/items.

The implemented coefficients are:

$$Jaccard = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \tag{5.10}$$

$$Anderberg = \frac{M_{11}}{M_{01} + 2 \cdot (M_{10} + M_{11})} \tag{5.11}$$

$$SorensenDice = \frac{2 \cdot M_{11}}{(2 \cdot M_{11}) + M_{10} + M_{01}} \tag{5.12}$$

$$SimpleMatching = \frac{M_{00} + M_{11}}{M_{01} + M_{10} + M_{11} + M_{00}} \tag{5.13}$$

If a large overlap between two users exists (M_{11} is large, M_{01} and M_{10} are small), then Jaccard, SorensenDice and SimpleMatching will return a similarity value close to 1. When the values of M_{01} or M_{10} increase in size the similarity decreases, which is intuitive.

An interesting observation is that Anderberg will not return a value close to 1, but instead a value close to 0.5. Which is not so much of a problem, because it is only a scaling difference. Additionally, it will prefer users with a larger profile. Users with a larger profile will have higher values for M_{01}, because of the larger number of items this active user did not download yet. This however is less of a problem, when compared to a user which causes a higher value for M_{10}. Then similarity would be much less, because of the multiplication. Or having items another user does not have is twice as bad for similarity compared to the other user having item this user does not have.

Jaccard(u_1,u_2)	$\frac{10}{10+0+10} = 0.50$	
Jaccard(u_1,u_3)	$\frac{10}{90+0+10} = 0.10$	5 times lower.
Anderberg(u_1,u_2)	$\frac{10}{10+2\cdot(0+10)} = 0.33$	
Anderberg(u_1,u_3)	$\frac{10}{90+2\cdot(0+10)} = 0.09$	3.6 times lower, fractionally less overlap to other user less of a problem.

Table 5.3: Comparison of Behavior of Jaccard and Anderberg. Anderberg has a preference to users with larger profiles as shown by switching the input users.

An example is shown in Table 5.3, where the differences in behavior between Jaccard and Anderberg is shown. In this example we have three users, User 1 with a profile-length of 10, User 2 with a profile-length of 20 and User 3 which has an profile-length of 100. We fix M_{11} to 10, thus all the items of User 1 are also downloaded by User 2 and User 3.

User 3 with the same overlap (M_{11}), but a larger profile (M_{01}) results in a 5 times lower score in the Jaccard function. But Anderberg prefers users with larger profiles, and the larger profile of User 3 is less of a problem.

Rewriting Cosine

When applying the Cosine function on boolean ratings (In the case of Tribler, downloaded yes/no), these functions can be rewritten to resemble the similarity coefficients. Rewriting the functions allows us to compare the subtle differences

between them and to the other similarity coefficients.

$$Cosine = \frac{M_{11}}{\sqrt{(M_{11} + M_{10}) \cdot (M_{11} + M_{01})}} \tag{5.14}$$

$$CosineMod = \frac{M_{11}}{(M_{11} + M_{10}) \cdot (M_{11} + M_{01})} \tag{5.15}$$

The different normalization scheme of the CosineMod function, the removal of the root, causes it the have an higher preference for users with a larger fractional overlap. If we use the example as previously introduced to show the differences between Jaccard and Anderberg.

Cosine(u_1,u_2)	$\frac{10}{\sqrt{(10+0)\cdot(10+10)}} = 0.71$	
Cosine(u_1,u_3)	$\frac{10}{\sqrt{(10+0)\cdot(10+90)}} = 0.32$	2.2 times lower.
CosineMod(u_1,u_2)	$\frac{10}{(10+0)\cdot(10+10)} = 0.05$	
CosineMod(u_1,u_3)	$\frac{10}{(10+0)\cdot(10+90)} = 0.01$	5 times lower.

Table 5.4: Comparison of Behavior of Cosine and CosineMod. CosineMod has a higher preference for users with a larger overlap.

CosineMod has a much higher preference for the user (User 3) which has more overlap, as can be seen in Table 5.4. The chance that a user has a large overlap combined with a small M_{01} is higher for users with a small profile. Thus CosineMod will probably have a higher preference for users with a small profile compared to Cosine.

5.1.5 Pseudo uservector

One of the largest problems associated with recommendation is sparsity. Sparsity refers to the large number unknown values in the dataset, as introduced in Chapter 4. A possible improvement could be the use of a pseudo uservector as suggested by Melville et al.[20]. The idea behind this is to "fill in the gaps" in the profile of a user, using available meta-data. Melville et al. used it to improve movie recommendations, by using content based information such as title, cast, etc. a rating could be computed for each movie which was not rated. Users which have rated movies of a specific director high, would probably like a new movie directed by the same director.

As shown in the User×Item matrix in Table 5.5, we could use a content boosted technique to give items 2 and 3 for the active user a pseudo-rating (shown in italic). The Item-Item similarity is calculated between all items which have been downloaded and those downloaded in co-occurrence, this is done as a performance optimization.

	Item 1	Item 2	Item 3	Item 4	...	Item 343,514
Active User	1	*0.2*	*0*	1		0
User 1	1	1				
User 2		1	1			1
User 3		1		1		

Table 5.5: The pseudo User×Item matrix. Using Item-Item similarities (shown in italic), overlap is created between the active user and user 2. Additionally, user 3 is now more similar due to the additional overlap of item 2.

Once similarity between items is calculated they need to be aggregated, because more than one item could "recommend" an item (For the active user, item 1 recommends item 2 but so does item 4). Several options are available for this aggregation. Sum, average and maximum have been tried and sum was chosen after it was empirically shown to be the best performing.

The resulting pseudo uservector (containing both actual downloads and calculated Item-Item similarities) is then used as an input for the Cosine similarity function, in order to calculate User-User similarities. The Cosine similarity function was chosen because this function can handle both boolean (actual downloads) and calculated similarities. Additionally the Cosine similarity function does not incorporate features such as average rating which although commonly used in similarity functions would not work here.

Item-Item similarity

Calculating Item-Item similarities is a common operation and is being done on a very large scale. A well known example of Item-Item similarities are the book recommendations made by Amazon.com as shown in Figure 5.1.

Using the history of users and comparing two items, similarity can be calculated between them, Amazon then selects the items with the highest similarity [17]. Using these items the Amazon.com homepage is then tailor made for each user.

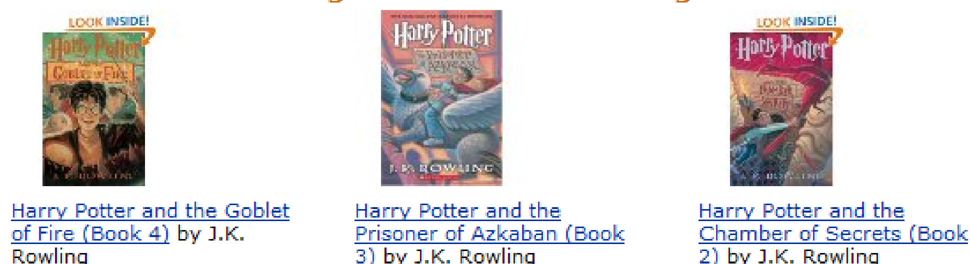


Figure 5.1: Recommendations made by Amazon.com. Using the buying history of other Users, similar books can be found and are shown on the website.

The CEO of Amazon.com has said:

> "If I have 3 million customers on the Web, I should have 3 million stores on the Web."[33].

Because if very similar users bought products, which the active user has not. Then these products should be very interesting to him and should result in an increase in revenue.

Cosine similarity

A relatively simple approach would be to apply one of the previously introduced User-User similarity function to calculate Item-Item similarity. An example would be the Cosine function as already described in Equation 5.7. Calculating the similarity between two items is almost the same as between two users. Instead of dividing by the number of items two users have in common, you divide by the number of users that have these two items in common.

A large number of approaches exists for calculating Item-Item similarities, but not all are applicable to this situation. As with User-User similarity functions they incorporate rating information which is not present in Tribler. Additionally it was shown by Demiriz [7] that Cosine based algorithms are superior to correlation based algorithms on boolean data.

A more complex approach would be to include the swarm and filenames of the items. These are available and could possibly reveal an accurate source of Item-Item matching.

Implemented and tested in were two variants:

- Filename matching
- Swarmname matching

Filename matching

Filenames can provide a way to efficiently match different versions of an item, these versions are also called duplicates. As introduced in Figure 1.4 a .torrent file describes an item, which could consist of multiple files. The largest file in an item was used as an identifier, if this file was larger than 5MB. Additionally several common files were excluded, i.e. .vob files, which are found on dvd's and always have the same filename. If the largest file was an audio file, then the file size limit was ignored. If no file was selected as the identifier, due to the restrictions as described, then filename matching was not possible.

The file identifier was then matched to all other file identifiers. Two different file matching heuristics were tried, one which made sure that the files had the exact same filename and size. The second heuristic dropped the filesize requirement, thus only checking filenames.

Swarmname matching

One problem associated with filename matching is that it does not work all that often as will be shown in Figure 5.4, because not all duplicate items have the exact same filename. Swarmnames, which are the titles of the items, are another possible source and could provide more information.

Matching swarmnames can be done using various techniques, which can match two strings and return the edit distance. Edit distance is the number of character operations which are needed to convert one string into the other. A simple technique could compare the two strings character for character, but as shown in this example taken from Soukoreff et al. [35]:

quick brown fox
quixck brwn fox

This technique would result in an edit distance of 6, where the minimal edit distance is 2 (inserting the x and removing the o).

A more complex technique is the Levenshtein edit distance. This function uses three operations, insertion, deletion and substitution, to determine the minimal edit distance. The edit distance is calculated by incrementally updating a matrix using the operations as shown in Table 5.6.

Insertion	Same Row, Previous Cell + penalty
Deletion	Previous Row, Same Cell + penalty
Substitution	Previous Row, Previous Cell + penalty
Correct	Previous Row, Previous Cell

Table 5.6: Levenshtein operations. Using these operations the minimal edit distance can be calculated.

After the computation, the edit distance can be read from the matrix in the lower right corner. Using only the first word introduced by the example in the previous paragraph, the execution is shown in Table 5.7.

		q	u	i	c	k
	0	0	1	2	3	4
q	1	**0**	1	2	3	4
u	2	1	**0**	1	2	3
i	3	2	1	**0**	1	2
x	4	3	2	**1**	1	2
c	5	4	3	2	**1**	2
k	6	5	4	3	2	**1**

Table 5.7: Levenshtein execution, minimal edit distance shown in bold. In each cell, the cheapest operation specified in Table 5.6 is chosen. The resulting minimal edit distance can be read from the lower right corner in the matrix.

In the example the two words are equal until the fourth character. Then the cheapest operation is chosen, which could be deletion (deleting the x). Substituting the x by an c would be equally cheap, but then in the next step another operation would be required resulting in a higher edit distance.

The Levenshtein edit distance is calculated between each item downloaded by the active user and its Top-X most co-occurring items. The list of co-occurring items is first sorted by the number of times each item is downloaded in co-occurrence with this specific item and then by the number of times an item has been downloaded. This causes the most similar (because of the high co-occurrence, which is an indicator of similarity) and most popular (because of the number of times an item is downloaded, which causes more overlap with other users) to be at the start of the list.

Additionally, a dynamic penalty is used, which results in the first few characters having a higher penalty than the last. Usually the most important word is located at the beginning of the swarmname. This word is very descriptive, which means that if the first few characters are equal, then the subject is likely to be the same. An example of this can be seen in Appendix A Table A.2, where “SUSE Linux 10.1” and “suse-10.1-boxed.set.dvd.iso” match for the first 4 characters (after converting to lowercase) and those first 4 characters actually describe the items well.

As a last step the edit distance between two items is normalized to return a value between 1 and 0, 1 meaning a complete match. Different normalization schemes were tried, but $min(0.5/editdist, 1.0)$ was chosen after empirically trying different schemes.

To improve the runtime of the Levenshtein function only the first 10 characters are matched. This is due to the dynamic penalty, which decreases to such a low number that all other characters do not have a large impact. Additionally a max edit distance is specified, which is compared to the minimal value of a row in the matrix after it is calculated. If this minimal value of a row is higher than the max edit distance, then the actual edit distance would also be higher. This is due to the incremental calculation of the edit distance in the Levenshtein function. And the process could be stopped. The matrix as used in the calculations is also reused instead of re-created when compared two strings.

The resulting algorithm is shown in Algorithm 1. One improvement max_additional_items shown in the code is explained in Section 5.2.2. But basically limits the number of additionally “recommended” items.

5.2 Advanced optimizations

During the evaluation of all the functions some optimizations were made to the algorithms. This section will describe these optimizations and how these were tested.

Algorithm 1 Implemented ItemItem(Levenshtein)

```
for all Items I downloaded by the active user do
  //sort Co-Occurring items
  CO-items = CO-items.sort()

  for all CI = 0 to Top-X do
    //limit size to first 10 characters
    I = I[:10]
    CO-items[CI] = CO-items[CI][:10]
    edit_dist[CI] = Levenshtein(I, CO-items[CI], max_edit_dist)
  end for

  //sort edit-distances
  edit_dist = edit_dist.sort()
  for PI = 0 to max_additional_items do
    //update User-Item matrix
    UI[PI] = min(0.5/edit_dist[PI], 1.0)
  end for
end for
//UI is ready for Cosine function.
```

5.2.1 Long profile boost

During the research it was discovered that all new candidate similarity functions preferred users with small profiles. These users had the highest similarity, due to the large overlap between them and the test user. But because of this large overlap, recommending new items is difficult i.e. all items overlap so no new items could be recommended. To solve this problem, a weighing was implemented which causes users with a larger profile to be preferred.

$$Sim\left(u_1, u_2\right) = Sim\left(u_1, u_2\right) \cdot min\left(\frac{|I_{u_2}|}{L}, 1\right) \tag{5.16}$$

The function is shown in Equation 5.16, in which L is the preferred profile length.

The optimization was not applied to TriblerOld and TriblerOpt, because these two functions already preferred users with larger profiles. Additionally it was not applied to the basic Cosine function to show the differences in performance. Different values for L were tried and it was empirically proven that 40 was the optimal value.

In Figure 5.2 the average profilelength of the similar users is shown for Cosine and CosineBoost (CosineBoost is the Cosine function, with this newly introduced optimization). As can be seen, the boost increases the length of the average profile by roughly 30. Due to the design of the Cosine function, users with profiles that have a profilelength comparable to the own profilelength are preferred.

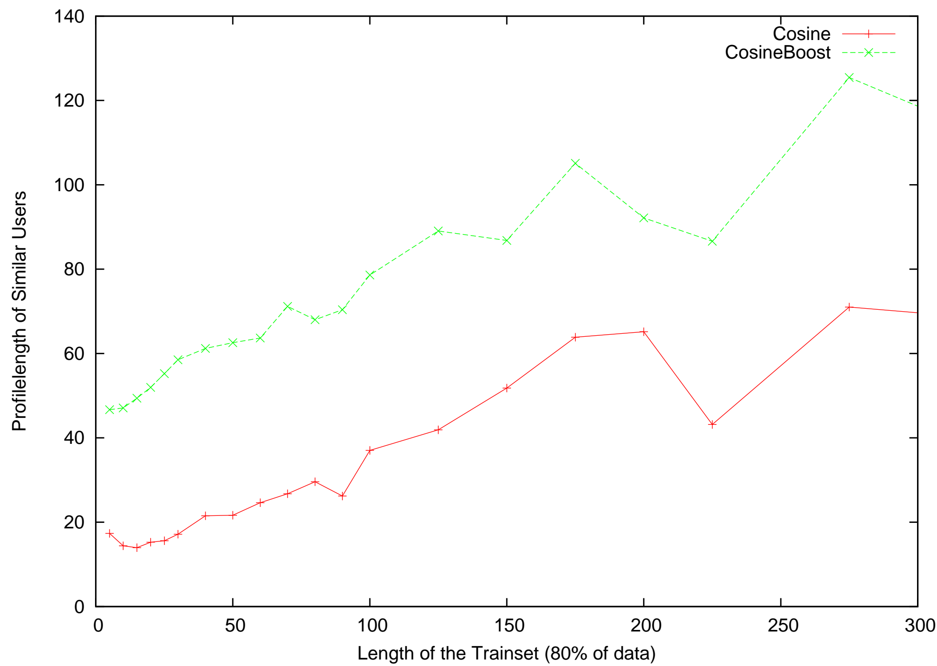


Figure 5.2: Differences between Cosine and CosineBoost, when comparing the profilelengths of the similar users. CosineBoost has an higher overall profilelength, which is caused by discounting the similarity of the users which have an profile less than 40.

This explains the increase in average profilelength, when the length of the trainset increases, as shown in the figure.

5.2.2 Balanced pseudo uservector

Another optimization which was applied to the pseudo uservector function limits the number of additionally promoted items. Each item in the original user profile can then only add 2 additional items, which was shown as the optimal value after empirical research. This prevents an item which has a lot of similar items to change the taste of a user due to overcrowding the profile with all different versions of this item. If more than 2 items were found, the most popular versions of the items were used (those which were downloaded the most and cause the most overlap with other users).

5.2.3 Levenshtein tuning

The Levenshtein function allows for various parameters to be tuned which influence the behavior of the function. They were varied and the impact on the performance was evaluated (the letters in front of the descriptions are used in the

graphs).

B The first parameter which was tested was the Top-X value, this value restricts the number of co-occurring items which were tested. Increasing Top-X would result in more items which are tested in the Levenshtein function, which could improve the number of matched items at a computational cost.

C The second parameter which was tested was the maximum edit distance. When decreasing this value would improve runtime, because the Levenshtein function can abort if during the computation a complete row has a higher value. When max edit is dropped below 1, then the first characters would need to match (which have a cost of 1), which is even faster to check. But by reducing the value, the number of items which are returned also decreases.

D The third "parameter" is not really a parameter but uses a variant of the Levenshtein function. This variant is know as the Damerau−Levenshtein distance which includes an operation that checks if 2 characters are switched, as described by Navarro [22]. This more complex function could possibly improve the matching, but at a higher computational cost.

E The fourth parameter was to use a different dynamic penalty. The default dynamic penalty causes the first 2 characters to have a cost of 1, this modification causes the first 3 characters to have this cost.

F The last parameter which was tried used a different normalization scheme, which used the maximum edit distance of the two strings to normalize.

To evaluate these parameters a small evaluation framework was created in which randomly 5000 items were chosen from the ground truth set (as introduced in Section 4.2.4), thus for each of these items the category was know. Then the different Levenshtein functions were applied and the number of recommended items which had the same category (correct results) were stored, wrong or no-category items were threated as incorrect results. Optimally we want the functions to return $10,000$ correct items, because the balanced pseudo uservector allows two additional items per item.

Figure 5.3 shows the differences in behavior, the baseline is option A. The other columns are the parameters in the order which they were introduced, the number above each column is the percentage of correctly identified items. Although the differences are small it can be seen that by decreasing the maximum edit distance (C), runtime and accuracy improves. The different dynamic penalty(E) also improves the accuracy of the item matching. Increasing the Top-X value (B) increases the absolute number of matches, but at a higher runtime. The other parameters have little or no impact, at the expense of a slight increase in runtime.

This leaves us with two possible candidates (max edit distance(C) and dynamic penalty(E)), but they both have a common downside. They also decrease the number of correctly matched items, which is a problem. One parameter which

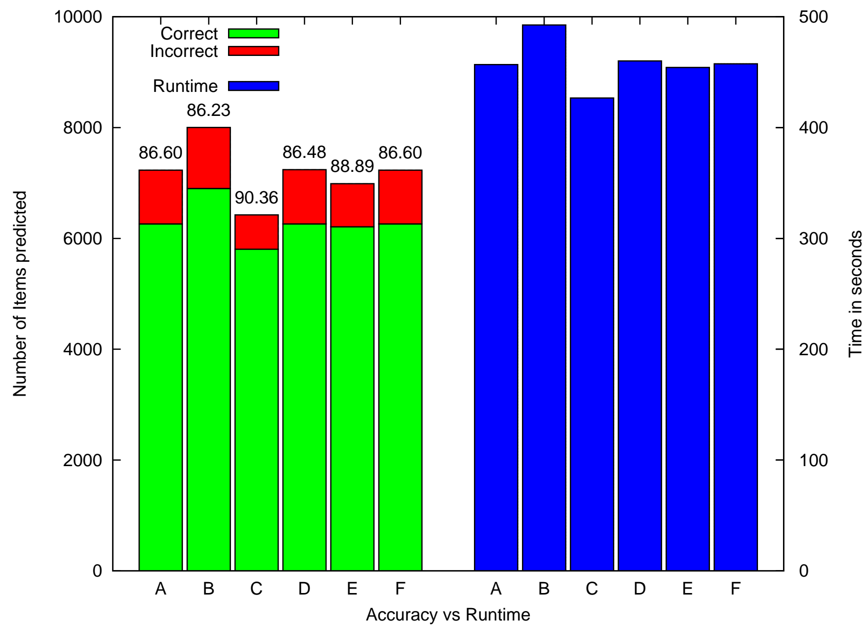


Figure 5.3: ItemItem tuning, number of correct/incorrect items and their runtime. Baseline A is compared to changes made to its parameters. C and E improve accuracy, but decrease number of items predicted. B increases number of item predicted but at a lower accuracy and increased runtime.

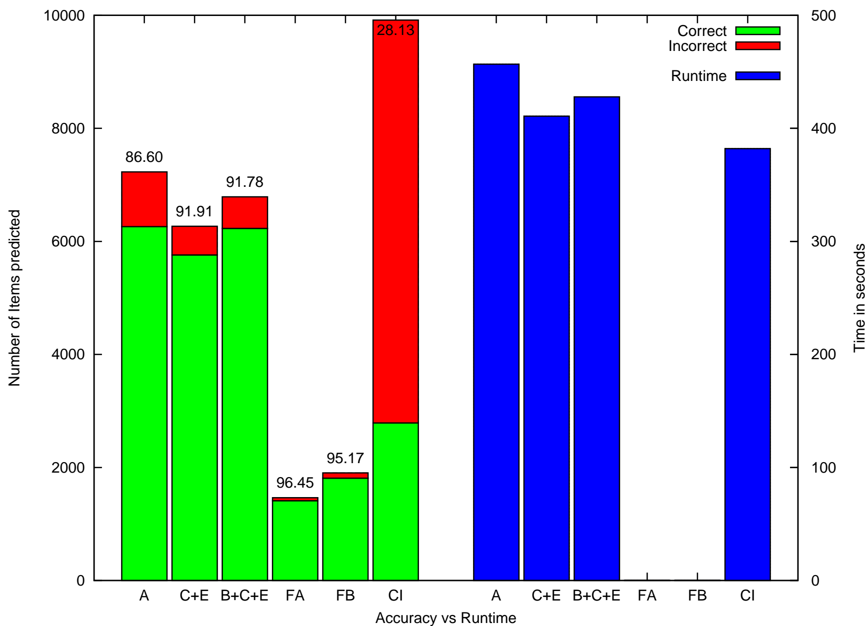


Figure 5.4: ItemItem tuning, number of correct/incorrect items and their runtime. Combining C+E does not increase the number of item predicted. B+C+E is the best option, increasing accuracy, number of items predicted at a slight increase in runtime. FileNames (FA and FB) provide accurate prediction at low runtime, but much less items are predicted. CosineItem (CI) is much less accurate.

increases the absolute number of returned items is the Top-X value(B), but this parameter increases the runtime. Combining all these parameters could solve all problems.

The results are shown in Figure 5.4. The first column contains the Levenshtein baseline(A), as in the previous figure. The second column is the combination of both the max edit distance and the new dynamic penalty (C+E), which causes a reduction in the number of files which are returned. In the third column this is combined with a larger Top-X value(B) which ensures that enough items are returned, while the low max edit distance results in an almost equal runtime (B+C+E). Comparing the best solution (B+C+E) to the baseline(A), the number of correctly recommended items stays the same while the number of incorrectly recommended item is decreased by 42%.

Additionally shown in Figure 5.4 are the two Filename matching heuristics. First the unique filename and filesize needs to match (FA), in the second variant only the filename has to match(FB). Comparing these to the Levenshtein variants clearly shows that the bottleneck of this method is the number of times it actually works. When removing the file size restriction this improves a bit, but still the number of returned similar items is small. Runtime is spectacularly low though, less then 1 second and accuracy is very high at 96.5% and 95.2%.

Finally in the last column CosineItem (CI) is shown. CosineItem is the original Cosine user similarity function, but now applied to items as described in Section 5.1.5. It does return almost 10,000 items, but accuracy is very low. Additionally, runtime is almost equal to the Levenshtein functions.

5.2.4 Pseudo uservector boost

Then after implementing this new and improved ItemItem(Levenshtein) function some strange behavior was observed. Instead of improving the performance of the Pseudo uservector performance got worse. After some research it was discovered that due to increasing the size of the Top-X value, more swarmnames were tested by the Levenshtein function. While this resulted in more similar swamnames and thus more similar items, those items were downloaded less on average. This is caused by the sorting of the Top-X co-occurring items list, which resulted in recommended items which were less popular. Less popular items, however causes less overlap between users. Thus reduces the impact of the Pseudo Uservector.

It is a similar problem as the original “Boost” improvement, by finding more similar but less popular items the impact of using a Pseudo Uservector reduced. Only the Levenshtein function had this problem and to solve this the edit distance was modified as shown in Equation 5.17.

$$Editdist(i_1) = Editdist(i_1) \cdot max\left(\frac{P}{|L_{i_1}|}, 1\right) \tag{5.17}$$

Which is very similar to the original boost function, but increases the editdist for

unpopular items. For P a value of 7 was empirically shown to be the best performing.

5.2.5 Additional remarks

We are confident that there is still room for more improvement of the Pseudo Uservector and Levenshtein functions. Preprocessing the swarmnames to reduce the number of stop words (words which do not add any information), sounds like a promising option. But due to time constraints and the likelihood that perfecting such a preprocessing technique would consume large amounts of time this was not included in this masters thesis.

5.2.6 Resulting functions

Most combinations of functions and optimizations were tried during the evaluation process. But in the next chapter the results of the following functions are shown.

TriblerOld
: The currently implemented similarity function as shown in Equation 5.1.

TriblerOpt
: The optimization of the currently implemented similarity function, shown in Equation 5.6. This function should give the exact same results, but at a lower runtime when performing a Full Update.

Cosine
: The Cosine similarity function, shown in Equation 5.7.

CosineBoost
: The combination of the Cosine similarity function (Equation 5.7) and the boost optimization (Equation 5.16 with L set at 40). This should improve the recall of the Cosine function, at the expense of a small decrease in accuracy.

ItemItem(Cosine)
: First Pseudo Uservector function, which uses the CosineBoost (Equation 5.7+5.16) function. But first preprocesses the uservector using the Cosine function (Equation 5.7).

ItemItem(Filename)
: Preprocessing done by FB (Shown in Figure 5.4), which only checks for exact matches of the unique filenames. This is an accurate method to increase the size of a userprofile, but a small one.

ItemItem(Levenshtein)
: Preprocessing done by B+C+E (Shown in Figure 5.4), top-X is set at 600, max edit distance at 0.99 and dynamic penalty with the first 3 characters

having cost 1. These values are used in Algorithm 1. Additionally popular items were boosted by Equation 5.17, with P set at 7.

Chapter 6

Results

The evaluation structure as introduced in Chapters 3 and 4 is now used to compare the candidate similarity functions. Using the 10 most similar users, items are recommended and compared to the testset of each user. Then, the search effectiveness and efficiency of all candidate functions can be determined.

6.1 Search effectiveness

First the search effectiveness of the candidates needs to be determined. Randomly 5000 users are selected and precision, recall, F1 score and NDCG figures are calculated as described in Section 3.1.

6.1.1 Precision and recall

In order to create the precision and recall graphs, the number of recommended items was varied between 5 and 1000, with an increment of 5. If a function could not recommend the requested number of items, due to not knowing any items which could be recommended, the list was appended with 0 values (which correspond to errors).

It has to be noted that fixing the number of similar users used to 10 is suboptimal for some similarity functions. But as implemented in Tribler only 10 peers will be used as neighbors. Using only these 10 peers, search will be provided. Modifying the number of neighbors used in the Tribler network is out of the scope of this research.

Figure 6.1 shows the result of the precision and recall experiments, but only the functions as configured in Section 5.2.6 are shown. The baseline is determined by TriblerOld and TriblerOpt, these functions need to be improved upon. As expected, both algorithms show equal performance.

Some thought on the extremely low precision figures which are magnitudes lower than those reported in other work on recommender systems [26] and shows that the P2P context in which these experiments are run is very different. As shown in

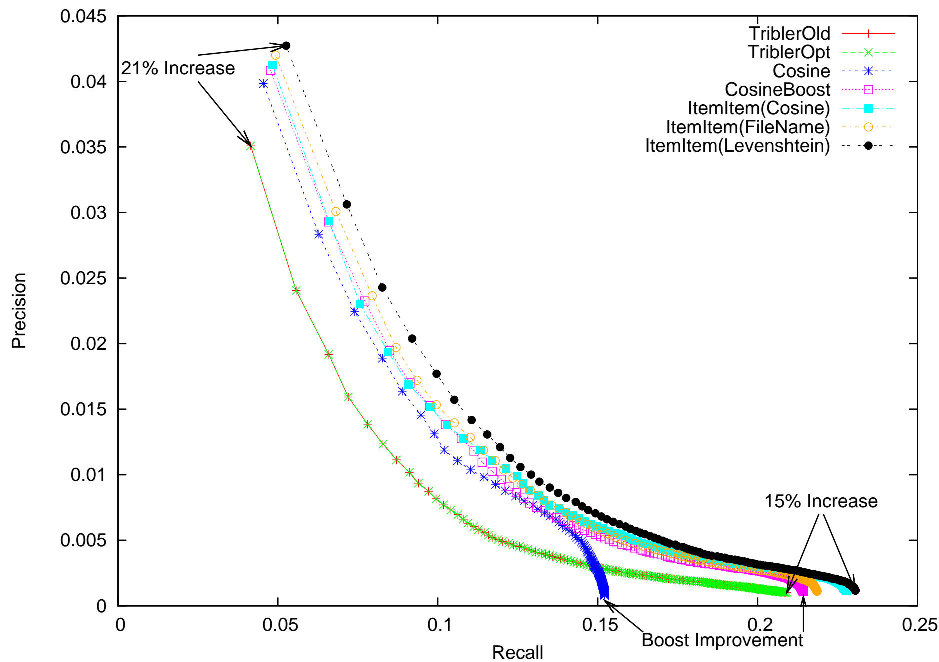


Figure 6.1: Precision and recall. All candidate functions improve upon the TriblerOld and TriblerOpt, except the Cosine function which has a lower recall.

Section 4 this could be due to the dataset being magnitudes more sparse, caused by the large amount of items. Addtionally, almost 50% of all items are downloaded once. These items are very difficult to recommend, due to the active user being the only one which has downloaded those. But if these items are recommended due to the top-10 similar users having downloaded them, then the active user did not download them resulting in an incorrect prediction.

As can be seen in the graph, all candidate functions improve on the current Tribler implementation. The Cosine function is struggling a bit, due to the problem introduced in Section 5.2.1 which prevents it from recommending enough items. In general the similar users found by this function have small profiles and thus the number of new items which can be recommended is not large enough to reach a higher recall than 0.15. Using the "Boost Improvement", the recall improves to 0.21.

The Pseudo uservector approach which incorporates Item-Item similarity, again improves performance. Recall is higher (more items correctly predicted) and precision is higher (more accurate when predicting few items). But unsurprisingly the most simple approach, ItemItem(Cosine), does not improve performance by much. Only recall is higher, which is expected due to additionally created overlap.

ItemItem(FileName) is a small improvement over ItemItem(Cosine) when not recommending large amounts of items. The low number of similar items found by the Item-Item function is a problem and reduces the performance when recommen-

ding more items. ItemItem(Levenshtein) is a significant step in performance, the overall performance is higher than all other similarity functions. Recall is increased by 15% compared to TriblerOld, precision by 21%.

The functions which are not shown here such as the similarity coefficients, behaved similar to the CosineBoost function. The only two functions which did not improve upon the TriblerOpt function are SimpleMatching and the modified Cosine. Additional information can be found in Appendix C, where all raw results are shown.

Hitrate

As suggested in Section 3.1 recall could be interpreted as hitrate. In Figure 6.1 the number of recommended items ranges between 1 and 1000, and the maximum recall achieved by ItemItem(Levenshtein) is 0.23. When a peer is connected to 10 neighbors, it has access to a maximum of $55,000$ items, due to each peer having a database of at most 5000 items ($10 \cdot 5000$ (neighbors) + 5000 (local database)). But during our experiments by only using 1000 items instead of $55,000$, a hitrate of roughly 23% is allready achieved.

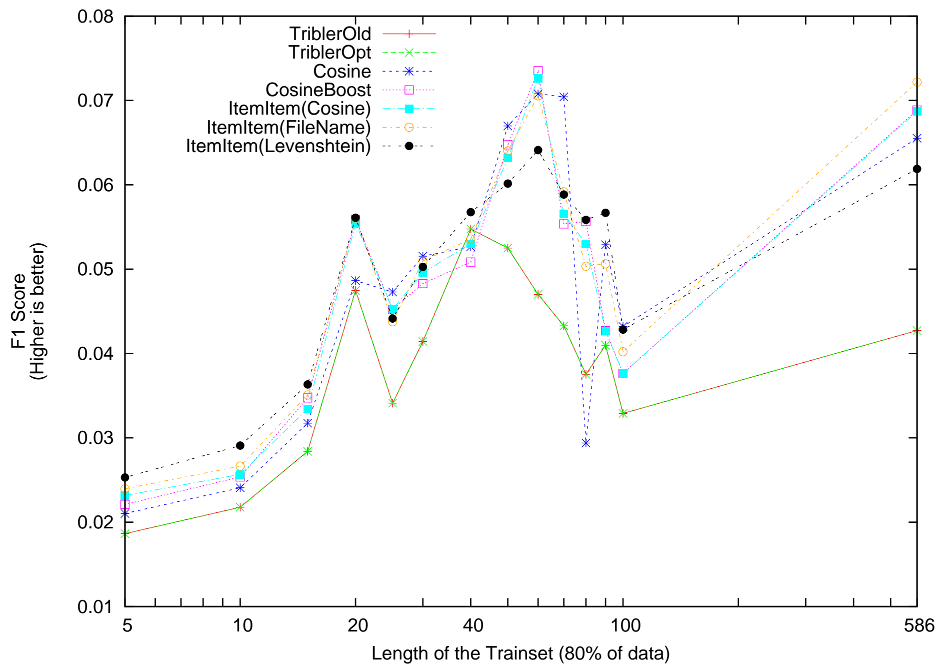
This figure alone is impressive, but to be able to recommend 1000 items using the 10 most similar users, these users need at least 101 items. One item to create overlap, 100 to construct 1000 not-downloaded items to recommend. But in our dataset only 1388 users have downloaded 101 items or more, which is equal to 2.8%. Obviously there will be some overlap between these users reducing this figure even more. Thus the hitrate or recall number on average is based on much less than 1000 items. This is shown in the precision and recall figures, in which it is shown that recall increases slowly if the number of recommended item is increased. And if 300 items are recommended still a recall/hitrate of 20% is achieved. Combining these recall numbers with the additional caching implemented in Tribler (peers store at most 5000 items), will probably result in an even higher hitrate.

ItemItem(Levenshtein) has a 3% higher recall compared to TriblerOld (23% vs. 20%). This translates to an improvement of 15% when comparing these two numbers.

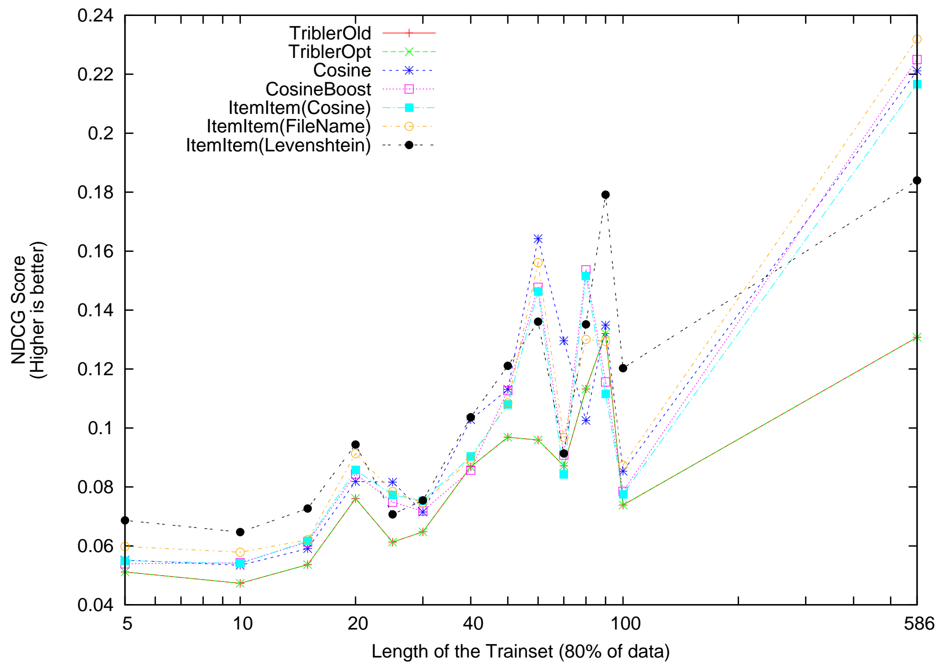
6.1.2 F1 and NDCG performance

In Figure 6.2 both the F1 and NDCG scores are shown. The number of recommended items is fixed to 10, but the differences in performance is shown based on the length of the userprofile. Due to a small number of users with large profiles, the users with a trainset larger than 100 are grouped (only 1388 users have an profilelength higher than 100, which corresponds to 2.77%).

For users with small profiles, the ItemItem(Levenshtein) function provides a boost in performance, as can be seen in both F1 and NDCG score plots. This boost fades for users with larger profiles. TriblerOld and TriblerOpt are the worst



(a) F1 score, shown in log-scale in respect to trainset length.



(b) NDCG, shown in log-scale in respect to trainset length.

Figure 6.2: Search effectiveness expressed in F1 and NDCG scores. Both scores increase for users with larger profiles. ItemItem(Levenshtein) is the best performing function until roughly a trainset length of 40.

	F1 score (All)	NDCG (All)	F1 score (≤ 35)	NDCG (≤ 35)
1 TriblerOld	0.0298	0.0597	0.0274	0.0544
2 TriblerOpt	0.0298	0.0597	0.0274	0.0544
3 Cosine	0.0341	$0.0696^{1,2}$	0.0307	0.0612
4 CosineBoost	$0.0356^{1,2}$	0.0687	0.0324	0.0610
5 ItemItem(Cosine)	$0.0357^{1,2}$	$0.0690^{1,2}$	0.0325	0.0615
6 ItemItem(FileName)	$0.0366^{1,2}$	$0.0723^{1,2}$	$0.0334^{1,2}$	$0.0648^{1,2}$
7 ItemItem(Levenshtein)	$0.0378^{1,2}$	$0.0781^{1,2,4,5}$	$0.0350^{1,2}$	$0.0716^{1,2,3,4,5}$

Table 6.1: Average F1 and NDCG scores, with significance. If a function is a significant improvement, this is indicated in superscript to which it is. First two columns show the averages for all users, the last two columns users with a trainsetlength less or equal to 35.

performing functions. In both figures, these functions clearly are the worst performers.

In general all tested similarity functions show an increase in F1 and NDCG score, compared to the profilelength, but the F1 score seems to be roughly stabilizing. NDCG differs from the F1 score in two aspects, the notion of ordering and the ground truth set to identify near misses.

An explanation would then be that the ordering of the recommending items is improving, but the number of correct items does not. This seems intuitive, because if more data is available the similarity functions can more accurately find similar users which in turn leads to better item recommendations.

Another possible explanation for the stabilization of the F1 score is the ground truth set. Which is used by the NDCG to identify near misses instead of exact matches, but not by the F1 score which requires exact matches.

A final remark regarding the F1 score statistic is that, during the evaluation when the testset (20% of all data) is less than 10, the F1 score that cannot reach a value of 1. This is the case for 97% of all users and explains the slow start until a trainset length of 40 (when the testset reaches 10 items). NDCG compares the ideal score to the actual score, thus does not have this problem.

Significance

Table 6.1 shows the average F1 and NDCG scores, when recommending 10 items. Using a one-way ANOVA combined with a Tukey HSD post hoc, the significance is also shown (indicated by a number to which it is an significant improvement in superscript). The significance is calculated using SPSS as described in Section 3.2.

Four similarity functions provide a significant improvement over the currently

implemented function (TriblerOpt performs the same as TriblerOld). Additionally, while the Boost Improvement (Section 5.2.1) increases the F1 score of the Cosine function, it degrades the ordering (NDCG) of the recommendations. This is expected due to preferring users with larger profiles, which are less similar.

The ItemItem(Levenshtein) function is the best performing function, both in F1 and NDCG scores. Additionally the function is also a significant improvement on NDCG compared to the CosineBoost and ItemItem(Cosine). This proves that using additional semantic information will improve the effectiveness of the similarity functions.

If only users with a trainset less or equal to 35 are compared, then the ItemItem(Levenshtein) function is additionally a significant improvement over the Cosine function. As stated, the Boost Improvement degraded the NDCG performance, but the additional meta-data matching corrects this. If only large profiles are considered the significance could not be determined. But as shown in Figure 6.2, it is unlikely that there are significant differences between the candidate functions.

6.2 Computational efficiency

Another important evaluation metric is the efficiency of all algorithms. More complex similarity functions, which are shown in the previous section to significantly improve the search effectiveness, are expected to need more time to run. But this does not necessarily impact the user experience. Of course an optimal similarity function must balance both performance metrics.

6.2.1 Runtime

Just as in the previous section, the baseline is determined by TriblerOld. The TriblerOpt function was designed to improve upon the full update (calculating the similarities for all other users) performance of the original TriblerOld function, thus there should be some differences between these two.

The comparison of the runtime of the various similarity functions is shown in Figure 6.3. TriblerOld and TriblerOpt are performing as expected, the TriblerOpt algorithm is faster during a Full Update, but worse during a Single Update. Both Cosine and CosineBoost improve upon the TriblerOpt function, compared to the baseline they perform better in Full Updates and in Single Updates. The ItemItem functions are slower, but when comparing them to the baseline, TriblerOld, they do not perform a lot worse but the Single Updates are slower. A notable exception is the ItemItem(Filename), which due to a simple and efficient implementation is almost as fast as the Cosine and CosineBoost functions.

Additionally, the figure shows the lower and upper quartiles. These figures represent the value at 25% and 75% of all datapoints (after sorting). If the box, determined by those lower and upper quartiles, is larger then there is more difference in the runtimes for users. This is the case for the TriblerOld, ItemItem(Cosine)

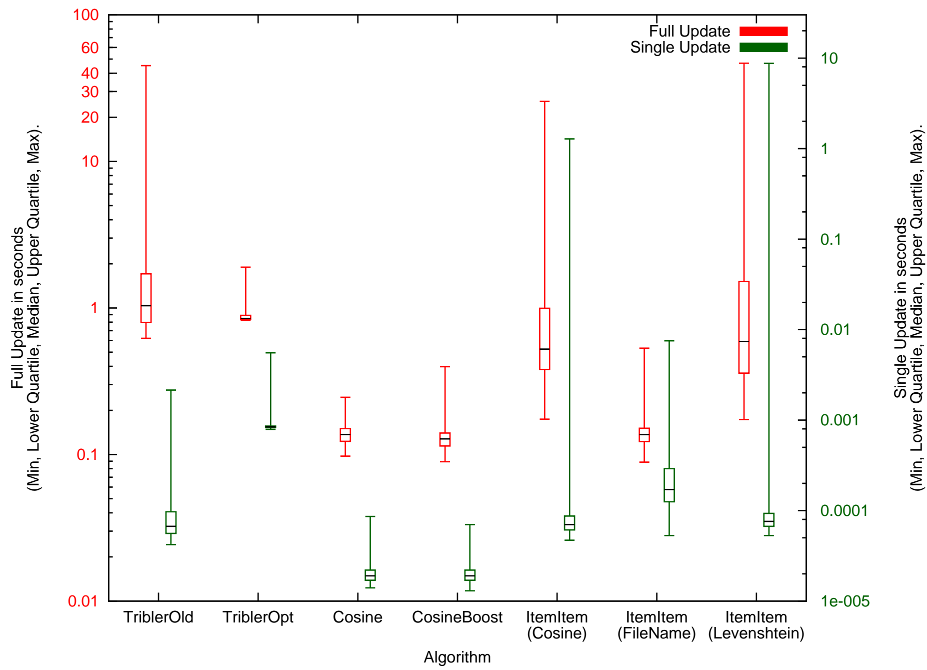


Figure 6.3: Average runtime. The box represents all data between the lower and upper quartile, the black line the average. A larger box indicates less stable runtimes. On average all candidate functions improve upon both TriblerOld and TriblerOpt.

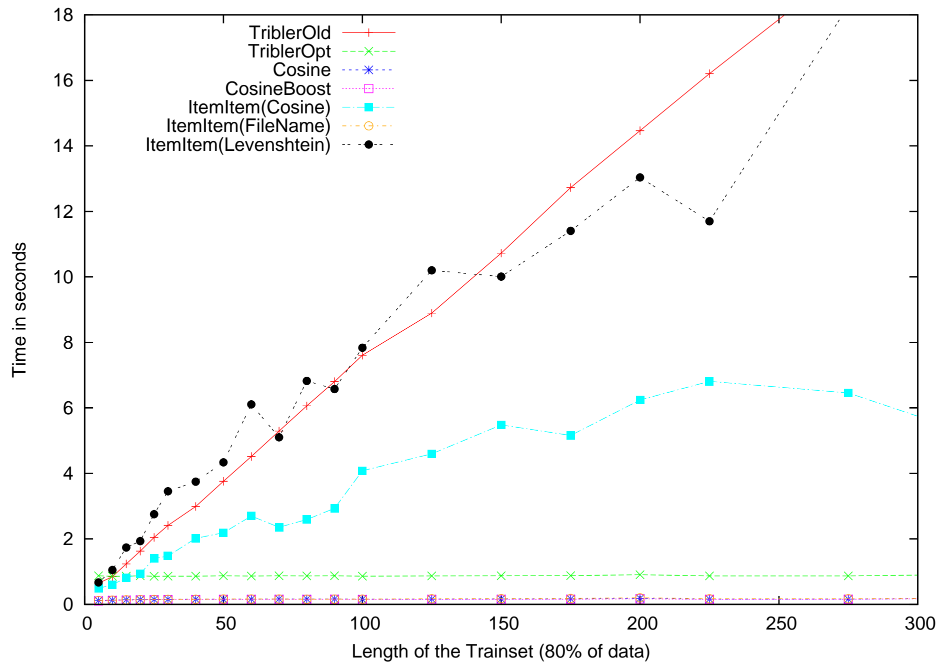


Figure 6.4: Average runtime, in regard to userprofile length. TriblerOld and ItemItem(Levenshtein) both show linear increase in runtime. ItemItem(Cosine) is a small improvement, but only the performance of the other candidates is acceptable.

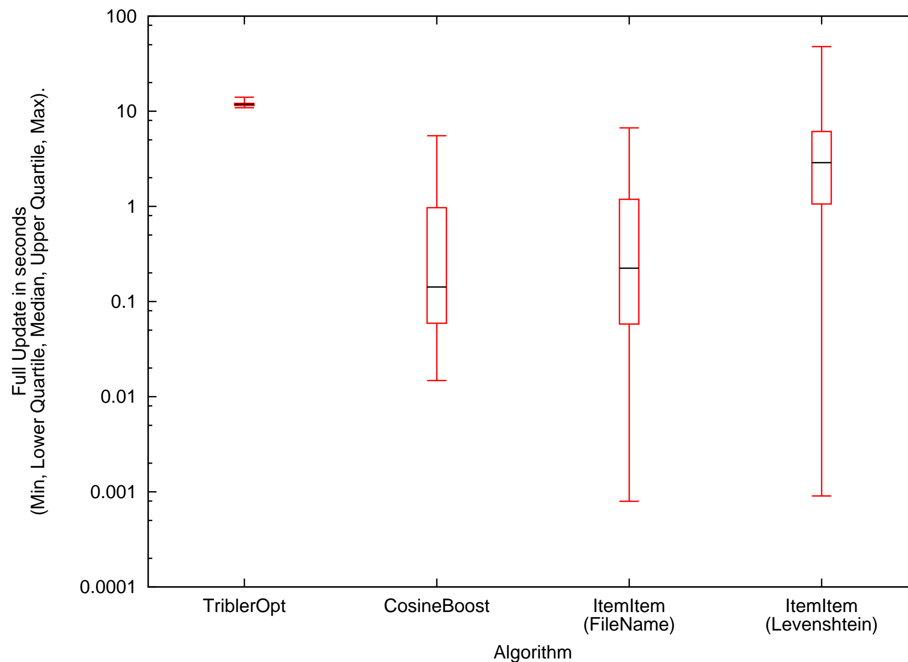


Figure 6.5: Full Update, using database. TriblerOpt almost always has a runtime of 12 seconds, due to database inserts. Other candidates provide better performance.

and ItemItem(Levenshtein) functions and could incidate a slowdown for users with more data to process.

Figure 6.4 gives insight into those differences. It shows the average time it takes to compute the similarity, shown in respect to the length of the user profiles (when more information needs to be processed). A larger userprofile has more data to process, thus an increase in runtime is expected. But as shown, the current implementation, TriblerOld, does so in a linear fashion which will degrade the user experience and is unacceptable. This linear performance proves that the behavior as seen in Section 4.2.3 in the top-50 Tribler users and their download behavior could very well be caused by the current similarity function.

Tribler will simply not be able to cope with large profiles, resulting in users that stop using Tribler after downloading 600-800 items. Full updates are being calculated when a user has a new item and on a regular interval. During a Full Update Tribler would need, if we extrapolate the data, roughly between 43 and 56 seconds computation time. This is unacceptable and because these experiments were all performed with all data in memory, the actual runtime could be magnitudes longer. Additionally, ItemItem(Cosine) and ItemItem(Levenshtein) also show this behavior.

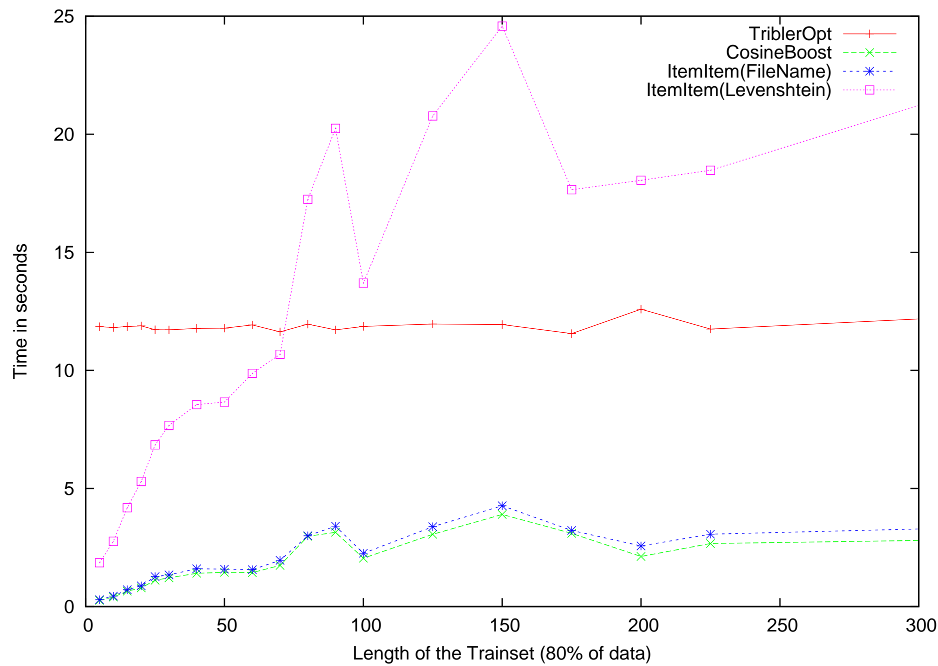


Figure 6.6: Full Update, using database. ItemItem(Levenshtein) still shows an linear increase.

6.2.2 Database impact

All previous experiments were run with all information in memory, which is about 1 GB. Obviously when implementing a new similarity function in Tribler, the functions need to perform with memory constraints. Thus the functions were converted into efficient database queries. In this step only the three best performing functions were used, which are CosineBoost, ItemItem(FileName) and ItemItem(Levenshtein). As a reference the baseline TriblerOpt was also included.

For the design of the database the currently implemented database of Tribler was used, including all indexes etc. A single modification was made to allow the identifying filename to be stored for each item, as required by the ItemItem(Filename) function. Examples of the SQL statements are shown in Appendix B. Figures 6.5 and 6.6 shows the efficiency of the database implementations.

The TriblerOpt function performs worse compared to the new functions, even the more complex ItemItem(Levenshtein) function is faster. This is due to the nature of this function, it needs to determine the owners of each item a user has downloaded. This is not required by the other functions. Additionally, TriblerOpt will always return a small similarity between users even if no overlap is present between them. The new functions only return a similarity when there is overlap. All similarities are written to the database, which results is a large constant factor in the TriblerOpt function (roughly 10 seconds).

	Effectiveness	Efficiency	Totals
TriblerOld	- -	- -	- -
TriblerOpt	- -	-	-
Cosine	-	++	□
CosineBoost	□	++	+
ItemItem(Cosine)	□	□	□
ItemItem(Filename)	+	+	+
ItemItem(Levenshtein)	++	-	□

Table 6.2: Function rankings. Effectiveness is a summary of the F1 and NDCG scores. Efficiency a summary of the runtimes.

CosineBoost and ItemItem(FileName), perform roughly the same, stabilizing at 3 seconds. The additional filename matching only causes a small overhead, which proves how efficient this can be implemented. But this function still needs database changes, which is less convenient.

ItemItem(Levenshtein) performs as expected, it has a linear increase in runtime when the userprofile increases in size, which is unacceptable.

6.3 Best of both worlds

Using ranking as shown in Table 6.2, a simple ordering can be made for all options. Using this ranking the CosineBoost or ItemItem(Filename) function would be the optimal choice, if we would select a single function. But can we improve upon this result by combining two functions? As shown in Figure 6.2 the effectiveness of a similarity function improves when more information is known on a user. But additionally as shown in Figure 6.4, the efficiency decreases (especially in the case of ItemItem(Levenshtein)).

The idea is straightforward, using the most effective function (ItemItem(Levenshtein)) until a certain profilelength, then switching to a more efficient function, efficiency can be be controlled. While the effectiveness of the more complex ItemItem(Levenshtein) function will help if less information is known and the runtime is still acceptable.

In the case of Tribler the value specifying the switch could even be controlled by the user, if the user requires a more effective matching then value could be increased. If the runtime would be unacceptable then the value could be decreased.

Another option would be to monitor the runtime of the similarity function each time it performs a Full Update, if the runtime would be higher then a predetermined threshold. Tribler itself could decide if the more efficient similarity function needs to be used. If Tribler is run on a very powerful computer, the switch would be made later. If it is run an old computer, then this switch could be made sooner.

ItemItem(Levenshtein) was combined with the Cosine, CosineBoost and ItemItem(Filename) functions. Table 6.3 shows the F1-Score and NDCG for the dif-

ferent combinations. The switch is made at a profilelength of 35, after several different values were tried and this value was empirically prove to be the best performing one.

	F1 score	NDCG
ItemItem(Levenshtein)	0.035443	0.075895
Combination(Levenshtein,Cosine)	0.035542	0.076821
Combination(Levenshtein,CosineBoost)	0.035130	0.074870
Combination(Levenshtein,FileName)	0.035275	0.075387

Table 6.3: ItemItem(Levenshtein) combined with different functions. No significant changes.

No significant differences can be observed, due to the majority of users in the dataset having a profilelength of less than 35. Thus only using the ItemItem(Levenshtein) function. The only difference is the runtime as can be observed in Figure 6.7.

The combination of ItemItem(Levingshtein) and the normal Cosine actually improves the effectiveness by a very small margin, while also allowing for an easy and simple method to control the efficiency.

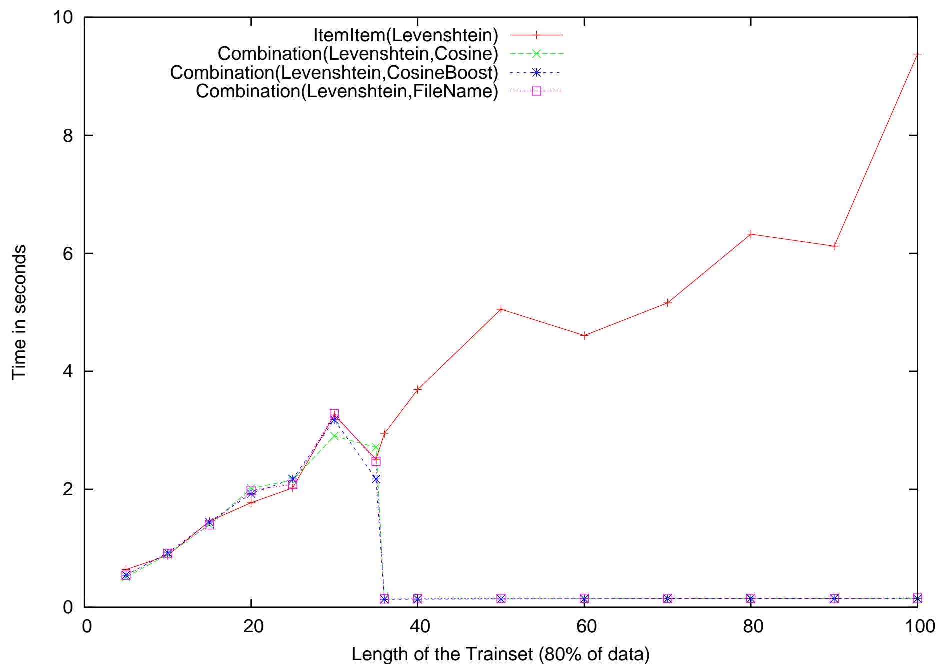


Figure 6.7: Combining ItemItem(Levenshtein) with other candidates to reduce runtime. Switching between functions at a trainset length of 35, as can be observed by the drop in runtime.

Chapter 7

Discussion

We will now expand on the results in previous chapters. First the problems found in the original Tribler function will be highlighted, after that the performance improvements of the candidate similarity functions.

7.1 Identifying the problems

Although the Tribler has been using the TriblerOld function for a long time it was known that it had some problems, hence this study. After extracting data from the superpeer logs, there were indications that there indeed were problems for users with large profiles. Still the discovery of the linear increase in the runtime of the current algorithm, TriblerOld, as shown in Figure 6.4 is remarkable.

Additionally the design of the algorithm, which always returns a similarity even if no overlap is found between users, causes problems when updating the database. To “fix” this, some caching strategies are currently implemented in Tribler. These include only updating the similarity field in the database if it has changed more than 10%. This increases memory usage and resembles a quick fix more, then a reliable solution.

But by removing the preference for users with longer profiles from the similarity function which is currently implemented, its Achilles heal is revealed. As a starting point TriblerOpt (Equation 5.5) is used, with μ set to 1 and divided into three parts:

$$TriblerOpt\,(u_1, u_2) = Scaling \cdot (ActualPref + ProfilePref) \tag{7.1}$$

$$Scaling = \frac{k}{|I_{u_1}|} \tag{7.2}$$

$$ActualPref = \sum_{j \in I_{u_1 \wedge u_2}} \frac{1}{|L_j| + 1} \tag{7.3}$$

$$ProfilePref = \frac{|I_{u_2}|}{|I|} \cdot \sum_{j \in I_{u_1}} \frac{1}{|L_j| + 1} \tag{7.4}$$

PI1	PI2	PI3	PI4	PI5	RI1	RI2	RI3	RI4	RI5	
										Active User
										User 1
										User 2

Table 7.1: TriblerOld preference for users with overlapping rare items. Gray areas are the downloaded items. In this example the Active User would have a higher similarity for User 1 than for User 2. Because of one overlapping rare item, instead of 5 overlapping popular items.

The first part is called Scaling, this part could easily be dropped due to being only dependent on User 1. When performing a Full Update, this value will be the same for all users, thus could be considered a constant.

Analyzing only the AcualPref function (as shown in Equation 7.3), shows that it only uses the popularity of the items downloaded by both users. This is by far the most simplistic of all candidate functions and does not include M_{10} (the number of items which only the active user has downloaded) or M_{01} (the number of items which only the other user has downloaded) at all.

Using only M_{11}(the number of items both users have downloaded) causes problems: i.e. if similarity between an active user and two other users is calculated. Both users have downloaded the same number of items, as shown in Table 7.1 (both have downloaded 5 items).

User 1 has one overlapping rare item and User 2 has 5 overlapping popular items. Then similarity with User 1 could be higher, due to $\frac{1}{|L_j|}$ which is higher for rare items. Rare or unpopular items are indeed more indicative for a users taste, but ignoring the ratio of overlap causes the problems as described above.

Lastly ProfilePref (as shown in Equation 7.4) adds an preference for users with larger profiles. One could argue that M_{10} and M_{01} is included in ProfilePref, due to these values being included into the enumeration over I_{u_1} and $|I_{u_2}|$. This is indeed the case, but the enumeration over I_{u_1} will have the same value when calculating similarity between this user and all others. This could thus be considered a scaling value, like the Scaling part. Which is exploited by TriblerOpt in order to reduce computational complexity. But causes ProfilePref to behave like $(M_{01} + M_{11})\cdot$ a constant, which is similar to the Long Profile Boost as suggested in Section 5.2.1.

7.2 Improved search effectiveness

As shown in Section 6.1 the search effectiveness of all candidate functions are an improvement over TriblerOld and TriblerOpt. A significant improvement has been made in the F1 score for all four candidate functions. Additionally the NDCG score, which is an indicator for the quality of the order of the recommendations, is significantly improved resulting from a better matching of users.

The candidate functions thus will improve hitrate in Tribler during search. If

only max-recall is used as an indicator for search effectiveness, then the best solution (ItemItem(Levenshtein)) is an improvement of 15% over both TriblerOld and TriblerOpt as shown in Section 6.1.1.

Previous versions of Tribler included a feature which recommended items similar to the item which is selected (similar to the Amazon.com example shown in Figure 5.1). If this feature were to return in the Tribler client, an improvement of more than 20% can be achieved based on the results of the NDCG. And because the item recommendation process is not tuned during the research (just the top-10 users are used), a larger improvement could be achieved.

7.3 Improved computation efficiency

While removing the "we like users with large profiles" from the similarity function allowed for great improvements in runtime, due to the removal of the unnecessary database inserts. The runtime of the candidate functions with all data in memory is also an improvement over TriblerOld, with the exception of ItemItem(Levenshtein). The combination of ItemItem(Levenshtein) with the Cosine function, resolves this without a reduction of search effectiveness.

If a new function required to have an even lower runtime, then the CosineBoost and ItemItem(Filename) functions could be used. These functions provide a significant improvement over the TriblerOld and TriblerOpt functions with both a lower runtime and a higher search effectiveness.

7.4 Implementation details

During the research the CosineBoost function was implemented in Tribler. This allowed for several optimizations in the code. Instead of specific methods which calculated similarity between one user (Single Update), and all users (Full Update), a single method was called which only implemented Single Update. This resulted in a very inefficient implementation, because reuse of values was impossible.

This was changed to a separate method for Single Update and Full Update. Additionally all caching strategies were removed, no one currently working on the Tribler project knew exactly why, what and how much data was being cached to improve the performance. Thus removing it allowed for cleaner code and a more simple architecture.

Two database queries were used, one optimized for Single Updates and the other optimized for Full Updates.

Due to the CosineBoost function not always returning a similarity score for all users causes problems though. Tribler actually expects to find at least 10 users with a similarity score in the database to connect to in the overlay. But when a new user starts Tribler for the first time, using the suggested new similarity functions, no users with a similarity can be found (no overlap, thus no similarity).

This is the ColdStart problem, and to solve it, the logic which was previously implemented inside the TriblerOld function is moved to the part which selects the 10 most similar users from the database. If less than the required users are found, then the "select users which have a large profile" kicks in. Which is a much better solution then integrating this logic into the similarity function.

Chapter 8

Conclusions and Future Work

We will now revisit the research questions and recommend which steps which will lead to the most improvement for the Tribler network.

8.1 Conclusion

The research question as introduced in the problem description will now be answered using the key deliverables in this study.

> Find a similarity function from which an efficient P2P topology for keyword search emerges.

The first part of the research question actually asks for an analysis of the current algorithm, because if this algorithm was sufficient we could basically stop the research there and then.

> A linear increase in runtime is discovered in the current implemented algorithm TriblerOld.

As expected some flaws were discovered in this algorithm, TriberlOld, most notable the linear increase in runtime. This problem was first introduced in Section 4.2.3, where after analyzing the dataset strange behavior of the Top-50 most active users gave an indication that there was a problem. Then in Section 6.2.1 the problem was shown to exist.

When a similarity function from which an efficient P2P topology for keyword search emerges is requested, a function which has a linear increase in runtime is simply not sufficient. TriblerOpt, an improvement over the TriblerOld function, did not show this behavior. But including a preference for users with large profiles in this algorithm proved to be an additional problem.

> The time spent to update all similarities in the database, due to algorithm design in TriblerOld and TriblerOpt, is unacceptable.

	Effectiveness	Efficiency	Remarks
CosineBoost	□	++	
ItemItem(Filename)	+	++	Database changes required
Combination (Levenshtein,Cosine)	++ ↔ □	□ ↔ ++	Efficiency easily controlled

Table 8.1: Final Rankings. Combining ItemItem(Levenshtein) and Cosine results in a very effective, very efficient solution which can be easily controlled.

From the perspective of a similarity function, this preference could not be considered a design flaw. But when implementing the similarity function into Tribler, the preference causes a considerable slowdown which is simply unacceptable. Caching strategies which are currently implemented in Tribler and try to correct this are simply not sufficient.

Combining these two facts, resulted in the need for a new similarity function. A large number of functions were tried and from these three possible options emerge. CosineBoost, ItemItem(Filename) and Combination(Levenshtein,Cosine). ItemItem(Cosine) is discarded due to its computational efficiency. Table 8.1 ranks the functions based on effectiveness and efficiency.

> Significantly improved item recommendations for CosineBoost, ItemItem(Filename) and Combination(Levenshtein,Cosine) over the currently implemented TriblerOld.

All three options provide a significant improvement over TriblerOld, both in effectiveness and efficiency. All three options would thus be an improvement over the current situation, but there are differences between them.

If an exceptionally efficient similarity function is required then CosineBoost or ItemItem(Filename) should be considered. They both are very fast and can be translated to an efficient database query. But ItemItem(Filename) does need a change in the current database of Tribler, which might not be an option. Additionally while using the unique filename as an identifier is very accurate, it only works for a small fraction of items. Resulting in a small improvement over CosineBoost, which after using the boost improvement (Section 5.2.1) proves to be an very effective method to find similar peers.

If computational efficiency is less important, then the Combination(Levenshtein,Cosine) as described in Section 6.3 could be used. This function combines the approach as suggested by Melville et al.[20], using metadata to increase the size of the userprofile, and the efficiency of the Cosine similarity function if the profile of a user has reached a certain size. No database changes are required for this option and Tribler itself could decide when the switch is made. If the runtime of the function is not acceptable, then the switch could simply be made and the much more efficient Cosine or CosineBoost function would be used. Using this method the function would simply scale with the type of computer

the user has. If it is a very new and highly capable computer, the switch would be made at a larger profile length. If it is an old computer, then this switch could be made sooner. Which results in a similarity function which is a combination of the best of two worlds.

> Combining ItemItem(Levenshtein) with Cosine or CosineBoost results in a optimal compromise between effectiveness and efficiency.

This would then be the answer to the research question. The Combination(Levenshtein,Cosine) will result in an efficient P2P topology for keyword search, because it can scale with the resources that a user has available and provides the best possible search effectiveness from all options discussed in this research. Resulting in what we think is a significant step forward in optimizing the topology of the semantic overlay.

8.2 Future work

The structural changes made to Tribler to allow for the implementation of the CosineBoost function will allow for a simple implementation of the Combination(Levenshtein,Cosine). This would be recommended as a first step. If Tribler were to control the switch as suggested in the text, then a combination between Levenshtein and CosineBoost would be preferred. Because CosineBoost provides better matchings for users with a small profiles and Cosine would otherwise hurt the search effectiveness for users with a small/old computer.

A combination of three similarity functions could also be tried. Using ItemItem(Levenshtein) until a certain profile length. Then switching to Cosine, but if this length is dynamically determined and falls below a certain value, then CosineBoost would be used. Which performs better for users with small profiles (max-recall is much higher).

After implementing one of these options then the Levenshtein function should be further optimized. There is still room for more improvement and as shown in Section 5.2.3 the Levenshtein function depends on multiple parameters which could be tuned to improve performance. Additionally, preprocessing the swarmnames would allow for an even more effective matching of items.

Bibliography

[1] Tribler.org wiki. `http://www.tribler.org`.

[2] L. Adamic and B. Huberman. Zipfs law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[3] A. Bakker, J.J.D. Mol, J. Yang, L. dAcunto, J.A. Pouwelse, J. Wang, P. Garbacki, A. Iosup, J. Doumen, J. Roozenburg, Y. Yuan, M. ten Brinke, L. Musat, F. Zindel, F. van der Werf, M. Meulpolder, J. Taal, and B. Schoon. *Tribler Protocol Specification*.

[4] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. pages 43–52. Morgan Kaufmann, 1998.

[5] Bram Cohen. Incentives build robustness in bittorrent. Technical report, bittorrent.org, 2003.

[6] Arturo Crespo and Hector G. Molina. Semantic overlay networks for p2p systems, 2002.

[7] Ayhan Demiriz. Enhancing product recommender systems on sparse binary data. *Data Min. Knowl. Discov*, 9:2004, 2002.

[8] Ken Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. Technical Report UCB/ERL M00/41, EECS Department, University of California, Berkeley, 2000.

[9] Tarun Gupta and Ham Id Seifoddini. Production data based similarity coefficient for machine-component grouping decisions in the design of a cellular manufacturing system. *International Journal of Production Research*, 28:1247–1269, 1990.

[10] Peng Han, Bo Xie, Fan Yang, and Ruimin Shen. A scalable p2p recommender system based on distributed collaborative filtering. *Expert Systems with Applications*, 27(2):203 – 210, 2004.

[11] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. *SIGOPS Oper. Syst. Rev.*, 40(4):359–371, 2006.

[12] Adriana Iamnitchi, Matei Ripeanu, and Ian T. Foster. Small-world file-sharing communities. *CoRR*, cs.DC/0307036, 2003.

[13] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[14] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 247–254, New York, NY, USA, 2001. ACM.

[15] Jian Liang and Rakesh Kumar. Pollution in p2p file sharing systems. In *In IEEE INFOCOM*, pages 1174–1185, 2005.

[16] Jian Liang, Rakesh Kumar, and Keith W. Ross. The kazaa overlay: A measurement study. page 25, 2004.

[17] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. 7(1):76–80, Jan.–Feb. 2003.

[18] Yong Liu, Yang Guo, and Chao Liang. A survey on peer-to-peer video streaming systems. *Peer-to-Peer Networking and Applications*, 1(1):18–28, March 2008.

[19] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[20] Prem Melville, Raymod J. Mooney, and Ramadass Nagarajan. Content-boosted collaborative filtering for improved recommendations. In *Eighteenth national conference on Artificial intelligence*, pages 187–192, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[21] Bradley N. Miller, Joseph A. Konstan, and John Riedl. Pocketlens: Toward a personal recommender system. *ACM Trans. Inf. Syst.*, 22(3):437–476, 2004.

[22] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[23] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.

[24] Hun Myoun Park. Comparing group means: T-tests and one-way anova using stata, sas, r, and spss. 2009.

[25] 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. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, 2008.

[26] R. Rafter and B. Smyth. Passive profiling from server logs in an online recruitment environment. In *Proceedings of IJCAI Workshop on Intelligent Techniques for Web Personalization (ITWP2001)*, Seattle, Washington, U.S.A., August 2001.

[27] Amir H. Rasti, Daniel Stutzbach, and Reza Rejaie. On the long-term evolution of the two-tier gnutella overlay. In *INFOCOM*, 2006.

[28] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[29] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proc. First International Conference on Peer-to-Peer Computing*, pages 99–100, 27–29 Aug. 2001.

[30] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: search methods. *Comput. Netw.*, 50(17):3485–3521, 2006.

[31] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Syst.*, 9(2):170–184, 2003.

[32] Nima Sarshar. Percolation search in power law networks: Making unstructured peer-to-peer networks scalable. In *In Proceedings of IEEE P2P04*, pages 2–9. IEEE Computer Society, 2004.

[33] J. Ben Schafer, Joseph A. Konstan, and John Riedl. E-commerce recommendation applications. *Data Min. Knowl. Discov.*, 5(1-2):115–153, 2001.

[34] Hendrik Schulze and Klaus Mochalski. Internet study 2008/2009. `http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009`, 2007.

[35] R. William Soukoreff and I. Scott MacKenzie. Measuring errors in text entry tasks: an application of the levenshtein string distance statistic. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 319–320, New York, NY, USA, 2001. ACM.

[36] Moritz Steiner, Wolfgang Effelsberg, and Taoufik En-najjary. Load reduction in the kad peer-to-peer system. In *In Fifth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P*, 2007.

[37] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed dht. In *Proc. 25th IEEE International Conference on Computer Communications INFOCOM 2006*, pages 1–12, April 2006.

[38] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, page 5, Berkeley, CA, USA, 2005. USENIX Association.

[39] Spyros Voulgaris and Maarten van Steen. *Epidemic-Style Management of Semantic Overlays for Content-Based Searching*. 2005.

[40] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.

[41] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA), 1949.

Appendix A

Ground truth

A.1 TF-IDF

Creating of the ground truth set consists of two steps. The first step consists of processing all swarmnames using the tf-idf function, to extract the most important categories. In each swarmname all . _ - where converted to spaces and # and ' where ignored. Next the swarmnames are processed by the tf-idf function and for each separate word in a swarmname the co-occurrences with all other words in the swarmname is stored. The tf-idf library, which can be found at `http://code.google.com/p/tfidf/`, is used but modified to allow for the co-occurrences.

After all swarmnames were processed, the stopwords were written to a file. These words are too common, thus can be removed safely. A threshold is used to determine if a word is a stopword, which basically determines that if a word is present in more then X documents it is a stopword. The used threshold is 1%, this was the default threshold in the used library.

Besides a list of stopwords, the most important words (which were not stopwords) are written to a file together with the number of swarmnames in which they occurred. Because of using the default threshold, the resulting important words file

Stopwords
xvid
the
avi
dvdrip
2007
rar

Important Words
exe
collection
pdf
video
ac3
torrentazos

Table A.1: Output of the tf-idf process. Stopwords are ignored, important words are used to determine categories.

ItemNr	Swarmname
9267	SUSE Linux 10.1
23,909	suse-10.1-boxed.set.dvd.iso
365,162	Sams.openSUSE.Linux.Unleashed.Nov.2007.eBook-BBL
-1985	de_Lillios_Suse_Avgarde-(1986)_Norsk_pop_FROYDIS

Table A.2: The Suse Linux category. All items containing “Suse” or “Linux” are assigned to this category, then manually checked. A negative ItemNr indicates an incorrect assignment.

was polluted with stopwords. Our approach include manually selecting categories from the various output files, thus no time was spend in optimizing the threshold value. A third file which was created consisted of the word co-occurrences. This file proved to be the most useful. It still was polluted, but much less than the important words list. The first 6 lines of the stopwords and important word files are shown in Table A.1.

A.2 Categorization

After manually selecting 34 categories using the output files as a reference, all swarmnames were processed again and a category was assigned if at least one of the words of a category was in the swarmname. These files were then manually processed to remove all errors.

An example of the “suse linux” category is shown in Table A.2. If an item did not belong to this category, its id was negated (an example would be item 1985). Because at least one of the words of the category should exists in the swarmname, some simple spelling errors could be corrected. An example of this is item 365,162 where “openSuse” was used instead of only “Suse”.

Appendix B

SQL statements

Shown below some of the more important SQL Statements which are used when running the similarity function on the database.

B.1 CosineBoost

```
SELECT peer_id, nrprefs, COUNT(torrent_id)
FROM Peer
JOIN Preference ON Peer.peer_id = Preference.peer_id
WHERE torrent_id IN( ? ) AND Peer.peer_id <> ?
GROUP BY Peer.peer_id
```

This simple SQL statement is a join between all peers and their preferences. As a parameter a list of items of the peer are passed and its peerid is used to exclude itself from the returned list.

The efficiency of this statement is caused by the fact that it will only return peers which have overlap. Thus less data has to be transferred between the database and the program.

This SQL statement can be used with all Similarity Coefficients (Section 5.1.4 and both Cosine functions).

B.2 ItemItem(Filename)

```
SELECT torrent_id, count(Preference.torrent_id)
FROM Torrent
JOIN Preference ON Torrent.torrent_id = Preference.torrent_id
WHERE file_ident = ?
GROUP BY Torrent.torrent_id
```

This simple SQL statement will return a list of items which have the same unique file identifier and the number of times these items have been downloaded. This list

is used by the ItemItem(Filename) function. If more than 2 items are found, then the number of times an item has been downloaded is used to prefer popular items. This logic could also be integrated into the SQL statement, but during the research the number of items which additionally could be recommended was not fixed thus it was chosen not to do so during the experiments.

B.3 ItemItem(Levenshtein)

```
SELECT torrent_id, name, count(Preference.torrent_id)
FROM Torrent
JOIN Preference ON Torrent.torrent_id = Preference.torrent_id
WHERE Torrent.torrent_id IN( ? )
GROUP BY Torrent.torrent_id
```

And finally a SQL statement which returns the swarmname and the number of times an item has been downloaded. A list of discovered items is passed as a parameter. After receiving this list from the database, the swarmnames were further processed by the ItemItem(Levenshtein) function, which then tries to find similar items.

Appendix C

Raw results

In Chapter 6 not all candidate functions as introduced in Chapter 5 are evaluated. This is because not all function showed an improvement or a better performing candidate function in their class was shown. This appendix shows all raw results. In total roughly 100 function/parameter combinations were tried, but only the best performing combination for each candidate solution are shown here.

	F1-Score	NDCG	Max-Recall
1 TriblerOld	0.030174	0.060674	0.212942
2 TriblerOpt	0.030174	0.060674	0,212942
3 Cosine	0.034268	0.071864	0.151457
4 CosineMod	0.027865	0.061500	0.111969
5 Jaccard	0.033983	0.069309	0.146664
6 Anderberg	0.035520	0.071639	0.159649
7 SorensenDice	0.034084	0.069730	0.146664
8 SimpleMatching	0.022733	0.053676	0.089460
9 CosineBoost	0.034946	0.069543	0.211743
10 AnderbergBoost	0.034729	0.069658	0.202062
11 ItemItem(Cosine)	0.034622	0.069798	0.224746
12 ItemItem(FileName/Size)	0.035644	0,072878	0.215267
13 ItemItem(FileName)	0.035753	0.073360	0.216552
14 ItemItem(Levenshtein)	0.036709	0.078010	0.226791

Table C.1: Average F1 and NDCG performance. ItemItem(Levenshtein) is the best performing, second are ItemItem(FileName) and ItemItem(FileName/Size). Anderberg is also performing well, but is dropped because of lower Max-Recall.

Some thoughts on the results as shown in Table C.1 and Figure C.1. Two functions are not performing at all, SimpleMatching (Equation 5.13) and CosineMod (Equation 5.9). SimpleMatching is the only function which includes the use of the number of items not rated by both users (M_{00}). This is probably not working because of the huge number of items and users in a P2P environment, which results

in a very large M_{00} and a similarity coefficient which is almost always very close to 1. The difference in behavior as introduced in Section 5.1.4 between Cosine and CosineMod causes the performance problems of CosineMod. The users which are recommended have an even smaller profile resulting in a lower recall.

Anderberg (Equation 5.11) with its preference for users with larger profiles has a higher recall if compared to Cosine. But after applying the Long Profile Boost as introduced in Section 5.2.1, it has a slightly lower performance in Max-Recall. Because of this CosineBoost was chosen instead of AnderbergBoost. Comparing ItemItem(FileName/Size) and ItemItem(FileName) shows a small improvement in Max-Recall and NDCG, which resulted in dropping the FileSize restriction for our final candidates.

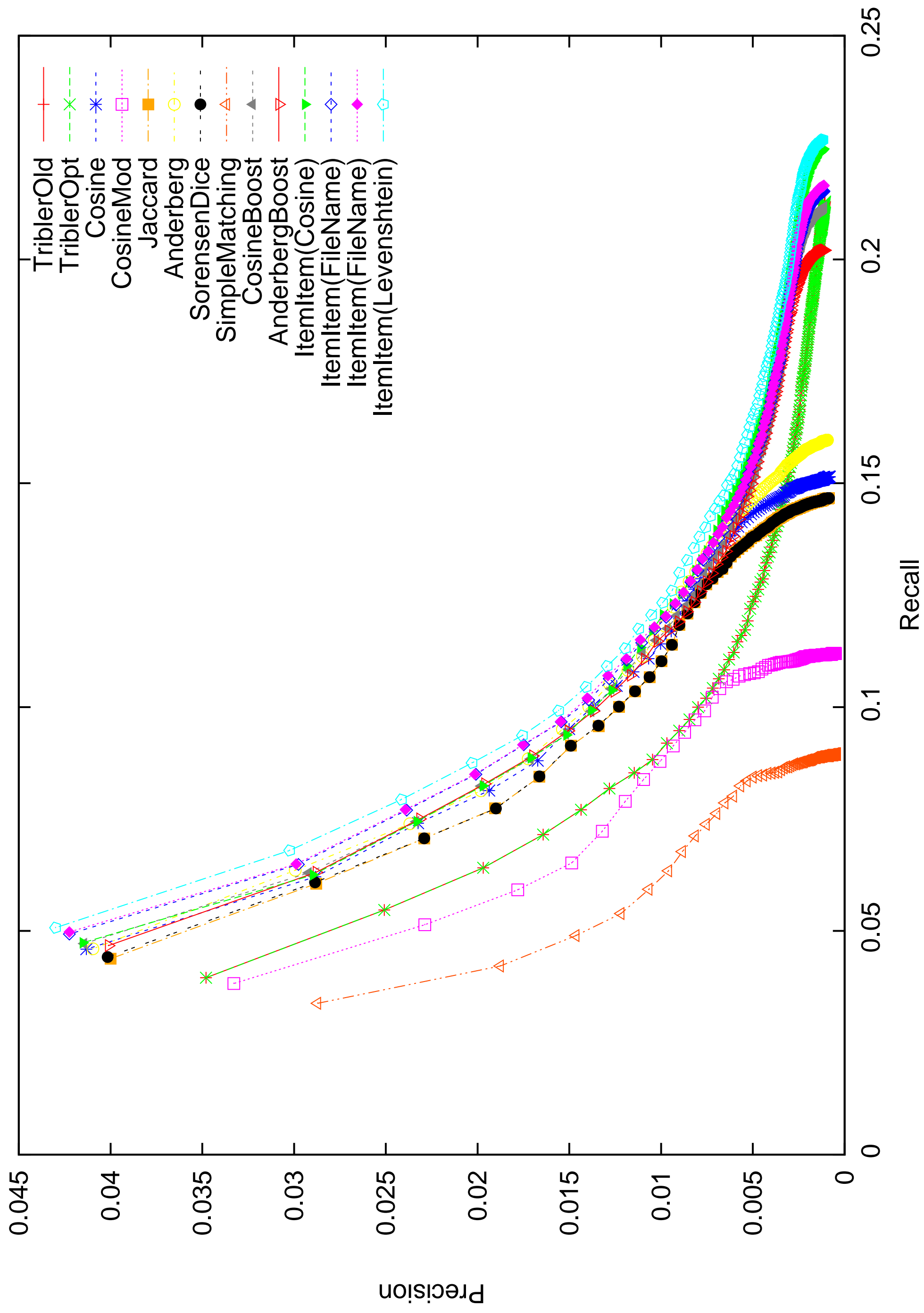


Figure C.1: Precision and Recall.