Efficient Agent-Based Dissemination of Textual Information

Manolis Koubarakis, Theodoros Koutris,
Paraskevi Raftopoulou, Christos Tryfonopoulos

Dept. of Electronic and Computer Engineering
Technical University of Crete
University Campus - Kounoupidiana
73100 Chania, Crete, Greece
Tel: +30-821-037222
Fax: +30-821-037202
{ manolis | koutris | rautop | trifon } @intelligence.tuc.gr

Abstract. We study the problem of efficient dissemination of textual information over wide-area networks. Our dissemination architecture utilises middle-agents and sophisticated matching algorithms. The data model and query language is based on the well-known Boolean model from Information Retrieval. The main focus of this paper is the problem of matching incoming documents with submitted user profiles. We present four efficient main memory algorithms for this problem and compare them experimentally.

1 Introduction

The *selective dissemination of information (SDI)* to interested users is a problem arising frequently in today's information society. This problem has recently received the attention of various research communities including researchers from databases [17, 19, 1, 5], agent systems [4], digital libraries [6], distributed computing [3] and others. In an SDI scenario, information producers publish information to an SDI system. This information is forwarded to information consumers that have already subscribed to it with a matching *profile*. Agent implementations of SDI systems can rely on *middle agent*s to match documents with profiles while different kinds of *end agents* (producer agents and consumer agents) are doing the publishing and subscribing [4]. Similar implementation techniques are used in SDI systems implemented using other technologies [17, 19, 1, 5, 3].

Information in an SDI system is usually structured according to an information model, and profiles are expressed in an appropriate profile language. In SIFT [18, 19] and COINS [11], published information is in the form of free text interpreted under the Boolean or vector space model [2], and the profile language is based on keywords combined using Boolean operators. In the middle agent of project SHADE published information is expressed in the logic-based language KIF or the object-based language MAX, and profiles are queries in these languages [11]. In XFilter information is in the form of XML documents and the profile language is a version of XPath [1]. In event

dissemination systems Le Subscribe [5] and SIENA [3] published information is in the form of events (expressed in an appropriate event data model) and profiles are formulated as queries in this data model. Finally, in the middle agents of the multi-agent system RETSINA, the published information and posted profiles are service specifications expressed in the service description language LARKS [15].

In this paper we adopt the information model of SIFT [18] and COINS [11]. We assume that published information is in the form of free text interpreted under the Boolean model, and that profiles are expressed by conjunctions of keywords (as, for example, in modern search engines). Then we concentrate on *the matching problem*: Given a database of profiles and an incoming document *d*, how do we find efficiently, which profiles match *d*?

We formalize this problem in an appropriate mathematical framework and study efficient and scalable algorithms for its solution. For the matching problem, the main contributions of this paper can be seen as a continuation of the research carried out in SIFT by Yan and Garcia-Molina [18][1]. [18] have proposed four algorithms for the matching problem (see Section 2) and have carried out a *mathematical* evaluation of these algorithms under the assumption that profiles are stored on disk and the cost parameters of interest are storage space (in disk blocks), number of I/Os (block reads) and CPU time (measured in accesses of the associated main memory data structures). In this work we present main memory implementations of these algorithms and evaluate them experimentally.

The rest of the paper is organised as follows. Section 2 defines the matching problem formally and introduces the four algorithms developed in SIFT. Section 2 introduces the statistical models for documents and profiles that we will use in our experiments. Section 3 gives the results and conclusions of our experiments. Finally, Section 4 summarises the paper and discusses future work**.**

2 Algorithms for Matching

In this section we first define the matching problem formally. Then we go on to desribe algorithms for tackling this problem.

We assume the existence of a finite *alphabet* Σ. A *word* is a finite non-empty sequence of letters from Σ. We also assume the existence of two sets of words: the *document vocabulary* (denoted by V_d) and the *profile vocabulary* (denoted by V_p). It is assumed that $V_p \subseteq V_d$. In the rest of the paper, the *cardinality* of any set S will be denoted by $|S|$.

Definition 1. A *document d* is a bag of words from the vocabulary V_d. A *profile p* is a conjunction of words from the vocabulary V_p. A document *d matches* a profile *p* if every word of *p* is included in *d*.

Example 1. The following is an example of a document:

{ *During, a, recent, holiday, in, Milos, I, stayed, in, a, wonderful, hotel* }

The following are two examples of profiles:

---

[1] The literature on the other closely related system COINS does not report any algorithms for the aforementioned problem [11].

holiday AND Milos, holiday AND Crete

The first profile matches the above document while the second one does not.

Let us now turn to algorithms for solving the matching problem. The algorithms and associated data structures are exactly the ones described in [18], but in our case all information is assumed to be in main memory.

We do not discuss the agent architecture that we use in our implementation, since it is not important for the purposes of this paper. Any agent architecture, e.g. RETSINA [4], could be used for implementing our SDI system. The only thing that has to change is the content language in order to correspond to the document and profile definitions, which are given above.

2.1 The Brute Force Algorithm

The *brute force* algorithm (BF) is a very simple one and is implemented only for comparison purposes. BF represents every document by its occurrence table. An *occurrence table* is a hash table that stores all the words appearing in a document (the keys are the words themselves). Profiles are stored sequentially in a linked list by BF. For each profile, BF stores a record that contains the profile identifier and a linked list with all the words in the profile.[2]

BF works by scanning the profile list sequentially. For each profile, it probes the document's occurrence table for all the words contained in the profile, failing as soon as a word does not exist in the document. If all the words of a profile are contained in the document, the profile identifier (and any other useful information) is added to a success list. When the algorithm terminates, the success list can be used for forwarding the document to appropriate subscribers.

If information about the *occurrence frequency* of words appearing in documents is available, then BF can exploit this information to improve its performance [18]. This can be done by adopting a ''fail-first'' strategy and probing the occurrence table first with the least frequent world, then with the next most frequent and so on. We have not evaluated a version of BF that takes into account frequency information.

2.2 The Count Algorithm

The Count algorithm *creates an index* over the database of profiles, and uses it for discovering quickly which profiles match an incoming document. The index is a hash table called the *profile directory*. For each word in the profile vocabulary, Count creates an entry in the profile directory, where it stores the identifiers of all the profiles that contain the specific word. For example, a profile that contains five different words can be found in five different hash table entries. Readers familiar with Information Retrieval techniques will notice that this is actually an *inverted index* over a database of profiles (not over a database of documents as it is used traditionally [2]).

Two arrays named *TOTAL* and *FOUND* with size equal to the number of profiles are also used by Count. For each profile p_i, the i-th element of array *TOTAL* is initially

[2] In a real system one has to store lots of other information related to a profile identifier (subscriber etc.). We will not consider the related issues in any detail in this paper.

set to the number of words in the *i*-th profile and never changes again. All elements of array *FOUND* are initially set to 0. As Count executes, elements of array *FOUND* are updated to keep track of how many times a profile identifier was found while probing the profile directory.

For the representation of an input document, Count uses a linked list, called the *distinct word list*. Each element of the list is a distinct word of the document. The number of elements in the distinct word list is usually smaller that the size of the document, since we will typically have multiple occurrences of the same word.

The matching procedure for Count is as follows. For each word in the distinct word list, we use the profile directory to find all the profiles that contain this word. Then we increment by one every element of array *FOUND* corresponding to these profiles. When this procedure is finished, we compare the entries of *TOTAL* and *FOUND* arrays for each profile. If the entry of array *FOUND* equals the corresponding entry of array *TOTAL* for some profile, then this profile matches the document and is added to the success list.

Variations of Count seem to be examined as candidate algorithms in many publish/subscribe systems today (see for example [12]).

2.3 The Key Algorithm

Like Count, Key utilises a profile directory as an index over the database of profiles. The difference is that instead of indexing the profiles according to all the words in them, Key uses a *randomly chosen word* as a key, and stores the rest of the words of the profile as a linked list in the hash table entry along with the profile identifier. For example, a profile with five words will be hashed in the directory according to one of these words. Consequently, its corresponding hash table entry will contain the profile identifier and the remaining four words. For the representation of an input document, Key uses both data structures used by BF and Count: the occurrence table and the distinct word list.

The matching algorithm, used by Key is as follows: for each word in the distinct word list, Key finds all the profiles that are hashed under this word. Then, for the remaining words of these profiles, it probes the occurrence table to see if each word appears in the document. As soon as a non-matching word is found, Key stops querying for this profile and examines the next one. If all the words of the profile are contained in the document, then the profile identifier is added in the success list.

The performance of Key can be improved if information about occurrence frequency of words is available [13]. We can index each profile according to its least frequent word, and store its remaining words in increasing order of frequency. In this way we make sure that each profile fails as soon as possible.

2.4 The Tree Algorithm

The main idea behind Tree is to store profiles in such a way so that similarities between them can be exploited during matching. This is achieved by introducing a *trie-like* data structure for profile storage [18]. This data structure assumes that

profiles are *sorted* sequences of words according to some order (e.g., lexicographic). The following definition is from [18].

Definition 2. Let p be a profile $(w_1, w_2, ..., w_k)$ of k words, and $0 \le i \le k$. Then $(w_1, w_2, ..., w_i)$ is called a *prefix* of p with $(w_{i+1}, ..., w_k)$ its corresponding *postfix*. A prefix $(w_1, w_2, ..., w_i)$ *identifies* p if $i = k$ or if there is no other profile, except those identical to p, that have $(w_1, w_2, ..., w_i)$ as a prefix. The shortest prefix that identifies a profile is called the *identifying prefix* of that profile (note that a prefix is the identifying prefix of two profiles if and only if they are identical).

As profiles arrive from subscribing users, Tree organises their identifying prefixes into a *profile tree.* The root of the profile tree (level 0) corresponds to the empty prefix. A node n at level i corresponds to a prefix $\sigma = (w_1, w_2, ..., w_i)$ of some identifying prefixes. All prefixes identical to σ, are represented by this node n. Its children are nodes corresponding to prefixes $(w_1, ..., w_i, u)$ of some identifying prefixes (where u is a word). A node n as above is implemented as a record with the following fields:

- *Children list*: a linked list of pairs (u, x) were u is a word such that $(w_1, ..., w_i, u)$ is the prefix corresponding to a child of n, and x is a pointer to that child.
- *Profiles list*: a linked list of profile identifiers of which σ is the identifying prefix.
- *Postfix list***:** a linked list of words that form the postfix of the identified profile(s).

To speed up searching, a hash table is used to index the children of the root of the prefix tree (as in the profile directory of Count and Key).

Figure 1 presents an example of a database of profiles and the corresponding profile tree as defined above. If we want to compare this tree with the profile directory used by Key and Count, we can say that the root corresponds to the directory and each sub-tree forms a tree-structured inverted index.

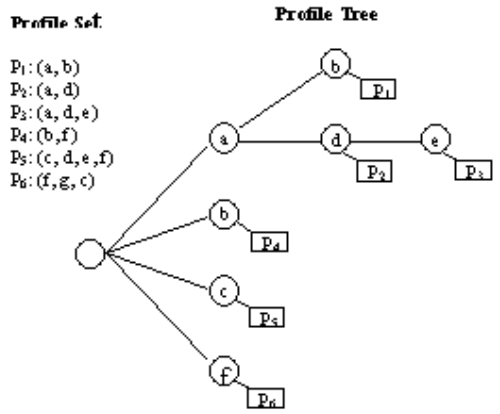

Figure 1. A profile tree created from a profile set

When an input document comes along, the profile tree is searched in a breadth-first fashion to discover all matching profiles. The details of this are as follows. At first, for each distinct word in the document, the child of the root of the profile tree corresponding to this word is inserted in a queue (the hash table is used here). While the queue is not empty, we examine the first node in the queue. If there are elements (u, x) of the children list of this node such that word u is in the document, these children are inserted in the queue. Then the postfix list of this node is checked against the occurrence table. If all words in the postfix list are in the document or the postfix is

empty, then all profiles in the profile list of the current node, match the document and are inserted in the success list.

The performance of Tree can be improved if information about frequency of occurrences of words is available [18]. Then profiles can be sorted in ascending order of occurrence frequency thus ending up with more profiles stored in sub-trees corresponding to low-frequency words. It is very likely then that these sub-trees will be visited less often.

3 Document and Profile Models

The previous section presented four algorithms for the matching problem originally proposed in [18]. There are two ways for carrying out an experimental evaluation of these algorithms. The first choice is to base the evaluation on real databases of profiles and documents that have been collected from various applications over a period of time. The second choice is to synthesize documents and queries that follow certain realistic application scenarios. In this paper we follow the second approach partly because it is more suited to experimenting with a various possibilities. Additionally, we are not aware of any existing databases of profiles and documents that are characteristic of an information dissemination application (and thus could have been used in our study).

Let us now present the statistical models we used to create the documents and profiles used in the evaluation of the algorithms. The parameters of our models are summarised for ease of reference in Table 1.

Table 1. The parameters used for the experiments

Parameter	Base Value	Description
$\|V_d\|$	900,000	Document vocabulary size
S_d	12,000	Document size (in words)
$\|V_p\|$	9,000	Profile vocabulary size
S_p	5	Profile size (in words)
N	500,000	Number of profiles
l	$3 < l < 10$	Letters in word
θ	0.9	Skewness factor for the Zipf distribution
q	0	Similarity degree

3.1 The Document Model

We first describe the way input documents are generated in our experiments. We follow the same methodology as [16] and [18]. Words in the document vocabulary V_d are identified by a number in the range 1 to $|V_d|$. This number is called the *rank* of the word. Each word in the vocabulary has an occurrence frequency associated with it. We assume that words in each input document follow Zipf's law, which states that the frequency of each word is inversely proportional to its rank [20]. We use the general

form of Zipf's law given in [7] (page 290) according to which the probability that a word w appears in a document is given by the following probability distribution:

$$\Pr(word\ is\ w) = \frac{1}{w^{\theta} \cdot \sum_{u=1}^{|V_d|} \left(\frac{1}{u}\right)^{\theta}} \quad , \quad 0 < \theta < 1 \qquad (1)$$

In other words, the distribution is such that words are ranked in non-increasing order of occurrence frequency (i.e., word 1 occurs most frequently, followed by word 2, etc.). The skewness parameter θ in the above distribution is the parameter controlling its form. As θ increases from slightly above 0 to slightly below 1, the distribution varies from uniform to Zipfian.

A document consists of S_d words from the document vocabulary V_d (where $|V_d| > S_d$). The words of each document are generated by S_d independent and identically distributed trials. In our implementation a document is generated as follows. We use the generator introduced in [8] to produce S_d integers ranging from 1 to $|V_d|$ such that the probability that a produced integer is w is given by the above distribution. Each actual word then is a string formed by concatenating the integer w produced and a fixed 2-character string. This has the effect that the length of the word varies from three to ten letters and it is only used so that word comparison operations become realistic.

3.2 The Profile Model

The profile model used in our simulations is a modification of the one described in [18]. The words in the profiles are selected from a subset of the document vocabulary called the *profile vocabulary*, by $N \cdot S_p$ independent and identically distributed trials. An important question that arises immediately is which words from the document vocabulary to include in the profile vocabulary and what distribution these words should follow. The profile model of [18] is based on the query model of [16] where the problem of querying a distributed database of texts is studied. This model has been developed after experimenting with a database of legal documents to determine sensible values of various parameters. [16] proposes to form the profile vocabulary with words 1 up to $|V_p|$ of the document vocabulary where $|V_p| = 0.01 \cdot |V_d|$. This allows one to discard the very infrequent words at the tail of the Zipf distribution because these words can be either misspellings or words that are used infrequently and will probably never appear in a user profile. The very frequent words from the beginning of the Zipf distribution are included in the profile vocabulary since it is possible to have profiles involving such words (e.g., rock fans might include the word WHO while computer technology experts might include the words IT or NEXT). Finally, if one considers the cumulative word occurrences in the database of [16], the above value for V_p allows to cover slightly more than 96% of all these occurrences. Inspired by [16], Yan and Garcia-Molina have used the same ratio of $|V_p|/|V_d|$ in the profile model of [18]. Since we are not aware of any other study that offers good estimates of this parameter, we have decided to use the same ratio for our generation of user

profiles. Thus our profile vocabulary is formed from the first (i.e., more frequent) 9,000 words of the document vocabulary.

Assuming the above ratio $|V_p|/|V_d|$, the question that naturally arises is the following. Given a database of profiles *db* and an incoming document *d, w*hat is the percentage of profiles in *db* that match *d*? [18] presents its experiments without any reference to this question. But using the analytical model and the base values for all the parameters given in [18] we can calculate that the probability that a profile matches an incoming document is 0.01%. For our base values (see Table 1) this number is 0.15%. The exact calculations are not given here due to the limited space available.

Of course, the above numbers are arbitrary and whether they are realistic or not depends on the application at hand. We can easily imagine applications with lower matching percentage or higher matching percentage. Imagine for example a news alert service such as the one available at www.cnn.com/EMAIL/ at times of an important international event (war, terrorist act, etc.). Then most published news articles will refer to this important event and thus match a great percentage of profiles also created in response to this event. In the experiments of Section 4, we study the performance of the four algorithms by varying the matching percentage while the ratio $|V_p|/|V_d|$ is kept equal to 0.01.

Let us also point out that the lower the ratio $|V_p|/|V_d|$ is, the higher the matching percentage achieved, as the use of more frequent words in the profiles increases the probability that these profiles match. To study this in detail, we calculated the probability P_m that a profile matches an incoming document, using our probability distribution shown in Formula 1 of Section 3.1. Working as above we have the following:

$$\Pr_m = \left(\frac{\sum_{w=1}^{|V_d|} (1 - (1 - \Pr(word\ is\ w))^{S_d})}{|V_p|} \right)^{S_p} \qquad (2)$$

In Figure 2 we graph this probability for various values of the parameter $|V_p|$ while all other values (S_d, S_p, θ) are as in Table 1. In Figure 2 this curve is the one labeled "curve calculated analytically". For various values of the parameter $|V_p|$, we also generated randomly 500,000 profiles and a single document and counted how many matching profiles we have. The results are the points plotted in Figure 2 and can easily be seen to validate our analytical model.

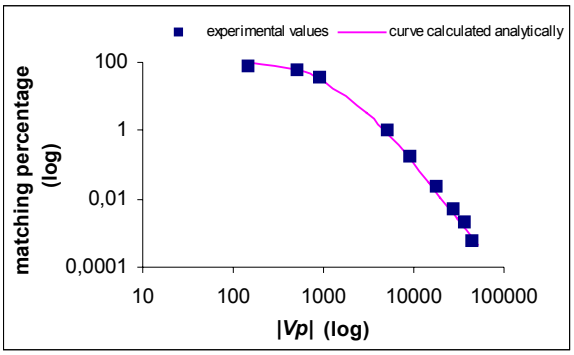

Figure 2. Matching percentage vs. $|V_p|$

Let us close this section by describing the details of how profiles are generated in our implementation. We generate randomly an integer in the range of 1 to $|V_p|$ following the uniform distribution. Then we concatenate this integer with a fixed 2-character string to form a word between three and ten letters (this is as in the document generation). In this way we can produce S_p words for each of the N profiles in an experiment.

In order to produce a profile set with *controlled matching percentage* we produce two kinds of profiles: profiles that are formed by randomly chosen words from the input document (matching profiles), and profiles that are formed from randomly chosen words not contained in the document (non-matching profiles). Thus, if we wish to set the matching percentage of a profile to k, then each profile is chosen with probability k, to be a matching one. Moreover, since non-matching profiles are created only from words not contained in the document, it follows that they fail at their first word as it follows from the description of the four algorithms given earlier.

4 Results

In this section we evaluate the performance of the four algorithms *experimentally*, assuming that both documents and profiles are stored in *main memory*. These are the first two crucial differences with the evaluation of these algorithms done in [18], where profiles are assumed to be stored in secondary storage, and the evaluation is most of the times analytical.

4.1 Experiments

For our experiments, the algorithms of Section 2 were implemented in C and were run on a PC with a Pentium III 800MHz processor and 768 MB RAM, running Linux. No other processes were run on the PC and the time shown in the graphs is *elapsed time* in milliseconds (ms). Documents and profiles were generated randomly according to the statistical models given above.

Each one of our experiments proceeds as follows. At first, we generate the input document and the database of profiles according to the above document and profile models. Then we populate the data structures needed for each algorithm. Finally, we run each algorithm and measure the time needed to match the incoming document against the database of profiles.

4.2 Varying the Number of Profiles

The first parameter that we vary is the number of profiles N. The values used for the other parameters are the base values shown in Table 1, while the matching percentage is set to 20%.

In Figure 3 we show the time taken by the four different matching algorithms as the number of profiles increases from 500,000 to 2,500,000. BF and Count are in this case the most sensitive ones to the increase of N. Moreover, Count appears to perform almost like BF when the number of profiles is about 500,000.

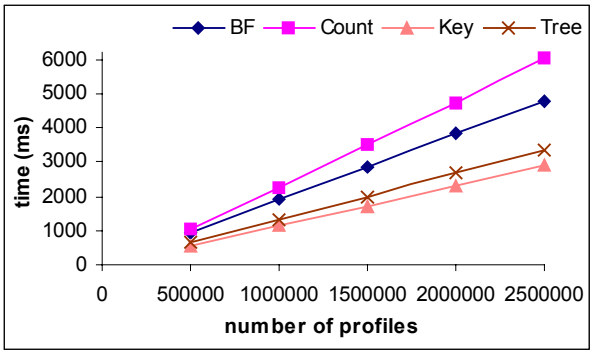

Figure 3. Time vs. number of profiles for 20% matching

This happens because the performance of Count is heavily dependent on the size of the distinct word list of the document, and because the comparison of the two main-memory arrays *TOTAL* and *FOUND* adds significant overhead to the algorithm. On the other hand, Key and Tree are shown to be less sensitive to the increase in the number of profiles in this case. These two algorithms perform much better than BF and Count since they need about 50% less time to match the profiles against the documents (about 3 seconds to match a document against 2,500,000 profiles). This happens due to the more sophisticated indexing techniques they use.

4.3 Varying the Matching Percentage

Let us now study the performance of the four algorithms when the number of profiles matching an incoming document varies while all other parameters are set to their base values given in Table 1. The results are shown in Figure 4.

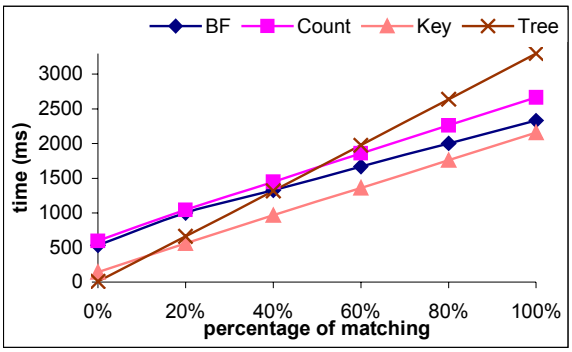

Figure 4. Time vs. percentage of matching for 500,000 profiles

As we can observe all algorithms are affected by an increase in the number of profiles matching an input document, though some of them show greater sensitivity

than others. BF examines all the profiles sequentially. This means that the greater the number of profiles matching an input document, the greater the number of lookups BF has to do in the document's occurrence table. Count is also greatly affected from increasing the matching percentage. As the matching percentage rises, so does the number of words in the distinct word list that are expected to be in the profiles. This results in more probes in the profile directory thus more processing time for Count. As the matching percentage increases, Key also has similar difficulties: in fact, for large number of matching profiles it performs similarly to BF. The increase in the matching time of this algorithm should also be expected, since the number of probes in the occurrence table increases as the matching percentage increases. Nevertheless, Key appears to be the best choice when the matching percentage is low (below 40%) since it needs around 66% of the time needed by the best of the other algorithms.

An interesting point in this graph is the behaviour of the Tree algorithm. As we can observe in Figure 4, Tree is a winner if the matching percentage is below 15%, and becomes worse than all the other algorithms when the matching percentage rises above 50%. This behaviour is heavily related to the formation of the tree that indexes the profiles. From this experiment we can derive that Tree is a good choice when the matching percentage of the profile set is relatively small. Finally, the similarity in the behaviour of Key and Tree below 20% matching can be explained as follows. As the matching percentage is relatively low, the work that has to be done by the two algorithms is quite similar: a single directory (hash table) lookup for most of the words in the distinct word list.

5 Conclusions

In this paper we studied the problem of matching incoming documents to submitted user profiles in an information dissemination system based on the Boolean model. We studied experimentally the tradeoffs involved in tackling this problem using four algorithms proposed by Yan and Garcia-Molina [18].

A lot of interesting work remains to be done in this area. Here we briefly mention some questions we are addressing in our current work:

1. Our current analysis needs to be extended to consider the effect of other parameters like similarity of profiles etc.
2. It would be nice to have real collections of profiles (presumably coming from various applications) where these algorithms could be tested. To the best of our knowledge, such profile collections currently do not exist although experiments with document collections are now standard practice in Information Retrieval (see TREC at http://trec.nist.gov).
3. How can we deal with more sophisticated languages for user profiles e.g., with negation and disjunction? Negation can be easily incorporated into our algorithms ([18] have already discussed this) but incorporating disjunction as well while remaining very efficient can be trickier. It would also be interesting to consider more sophisticated document models, query languages and their corresponding matching algorithms (e.g., we could allow structured documents as in [10] or [1] or interpret documents and profiles under the vector space model [2]).

6 Acknowledgments

This work was supported in part by project DIET (IST-1999-10088) funded by the IST Programme of the European Commission, under the FET Proactive Initiative on "Universal Information Ecosystems". We would like to aknowledge the contributions of all partners of DIET to this work. Special thanks go to Franscisco Valverde-Albacete for enlightening discussions concerning the statistical models and experiments reported in this paper.

References

1. Altinel M. and Franklin M.J. Efficient Filtering of *XML* Documents for Selective Dissemination of Information. *Proceedings on the 26th VLDB Conference, pp. 89-98*, 2000.
2. Baeza-Yates R. and Ribeiro-Neto B. *Modern Information Retrieval*. Addison-Wesley, New York, 1999.
3. Carzaniga A., Rosenblum D. and Wolf A.L. Interfaces and Algorithms for a Wide-Area Event Notification Service. *Proc. of the 19th ACM PODC*, 2000.
4. Decker K., Sycara K. and Williamson M. Middle-Agents for the Internet. *Proceedings of IJCAI-97*, 1997.
5. Fabret F., Jacobsen H.A., Llirbat F., Pereira J., Ross K.A. and Shasha D. Filtering algorithms and implementation for very fast publish/subscribe systems. *Proceedings of ACM SIGMOD*, 2001.
6. Faensen D., Faulstich L., Schweppe H., Hinze A. and Steindinger A. A Notification Service for Digital Libraries. *Proceedings of the Joint ACM/IEEE Conference on Digital Libraries (JCDL '01)*, 2001.
7. Gonnet G.H. and Baeza-Yates R. *Handbook of Algorithms and Data Structures (2nd edition),* Addison-Wesley, New York, 1991.
8. Gray J., Sundaresan P., Englert S., Baclawski K. and Weinberger P.J. Quickly generating billion-record synthetic databases. *Proc. Of ACM SIGMOD*, 1994.
9. Kernigham B.W. and Ritchie D.M. *The C Programming Language (2nd edition),* Prentice Hall, 1988.
10. Koubarakis M. Boolean Queries with Proximity Operators for Information Dissemination. *Proceedings of the Workshop on Foundations of Models and Languages for Information Integration (FMII-01), Viterbo, Italy,* September 16-18, 2001. In LNCS (forthcoming).
11. Kuokka D. and Harada L. Matchmaking for Information Agents. *Proceedings of IJCAI '95, pp. 672-678, Montreal, Canada*, 1995.
12. Marrow P. et. al. Agents in Decentralised Information Ecosystems: The DIET Approach. *Symposium on Information Agents for E-Commerce, AISB'01 Convention, University of York, United Kingdom*, March 21- 24, 2001.
13. Pereira J., Fabret F., Llibrat F. and Shasha D. Efficient matching for web-based publish/subscribe systems. *In Proc. of the Int Conf. On Cooperative Information Systems (CooPIS)*, 2000.
14. Schwartz E.S. A Dictionary of Minimum Redundancy Encoding. *Journal of the ACM, 10(4)*, October 1963.
15. Sycara K., Klusch M., Widoff S. and Lu J. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record, 28(1): 47-53*, 1999.
16. Tomasic A. and Garcia-Molina H. Performance of inverted indices in distributed text document retrieval systems. *Proceedings of the 2nd International Conference on Parallel*

and Distributed Systems, December 8-17, 1993. Full version appears as Stanford Tech. Report No 8090.

17. Yan T.W. and Garcia-Molina H. Distributed Selective Dissemination of Information. *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems (PDIS), pp. 89-98, Austin, Texas,* September 1994.
18. Yan T.W. and Garcia-Molina H. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM TODS, 19(2):332-364,* 1994.
19. Yan T.W. and Garcia-Molina H. The SIFT Information Dissemination System. *ACM TODS, 24(4): 529-565,*1999.
20. Zipf G.K. *Human Behaviour sand Principle of Least Effort*, Addison-Wesley, Cambridge, Massachusetts, 1949.