Yenta: A Multi-Agent, Referral-Based Matchmaking System

Leonard N. Foner

MIT Media Lab
20 Ames St, E15-305
Cambridge, MA 02139
foner@media.mit.edu
617/253-9601

Abstract

Many important and useful applications for software agents require multiple agents on a network that communicate with each other. Such agents must find each other and perform a useful joint computation without having to know about every other such agent on the network. As an example, this paper describes a *matchmaker* system, designed to find people with similar interests and introduce them to each other. The matchmaker is designed to introduce *everyone*, unlike conventional Internet media which only allow those who take the time to *speak* in public to be known.

The paper details how the agents that make up the matchmaking system can function in a decentralized fashion, yet group themselves into clusters which reflect their users' interests; these clusters are then used to make introductions or allow users to send messages to others who share their interests. The algorithm uses *referrals* from one agent to another in the same fashion that word-of-mouth is used when people are looking for an expert. Several prototypes of various parts of the system have been implemented, and the most recent results, including simulations of up to 1000 such agents, are presented.

Introduction

Many useful software agents are *facilitators*—they act as *intermediaries*, bringing people together for one reason or another. Such agents often need only an approximate knowledge of the domain of interest, leveraging most of their power off users' own knowledge. For example, systems such as Webhound/Webdoggie [7], or HOMR/Ringo/Firefly [9] use *automated collaborative filtering* to find either Web pages or music, respectively, that fit each individual user's taste. Neither system needs to understand Web pages or music in any real way—instead, they allow users to easily share their preferences with each other in a way that the *users* understand.

Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of ACM. To copy otherwise, or to republish, requires a fee and/or specific permission. Agents '97 Conference Proceedings, © 1997 ACM.

While the two agents above match up users' tastes to make recommendations, their focus is not explicitly matchmaking users and introducing them to each other. The research described in this paper is focussed on introducing users who are interested in similar topics. There are a number of reasons why one might want to do this:

- People are often working on similar projects without realizing it—be it two people down the hall from each other reinventing the same wheel, or two doctors both doing research on similar cases but having no idea that both of them are studying the same literature.
- It is often the case that people need to find an expert in some field, but finding such an expert can be difficult and time-consuming. Those who are not well "plugged-in" via word of mouth can find this even more difficult.
- There is potential for a great deal of social collaboration on the Internet, but it is often underutilized. "Lurkers" who read but do not post to mailing lists or newsgroups, for example, are an undiscovered resource to the community, invisible because they do not contribute to public discussion.

Current communications systems on the Internet are not well-designed for this sort of matchmaking. In almost all media on the Internet, only people who take the time to write a piece of prose and transmit it somewhere, whether by mail, news, or making a Web page, are ever seen by anyone else. Two people who are both working on the same problem, or who share an interest, may never know if they themselves are not actually writing about it. The matchmaking system described here is designed to aid these "lurkers" who are not part of the public discussion nonetheless find each other and establish a community. It does so using a *multi-agent* strategy and a *completely decentralized, peer-to-peer* architecture.

Why having multi-agent systems helps

Many currently-implemented agents use a *centralized* architecture, in which one agent serves either one or many users. A centralized architecture has its advantages: for example, if there is no effective way for peers to find each other, a centralized solution may be the only workable solution. Unfortunately, there are problems with a centralized architecture:

To appear in: The First International Conference on Autonomous Agents (Agents '97), Marina del Rey, CA

- Scaling such an architecture to large numbers of users is difficult; in systems which must correlate user interests, for example [9], straightforward approaches to this problem generally require a quadratic-order matching step somewhere.
- If the system requires either high availability (due to constant demand for its services) or high trustability (because it handles potentially sensitive information, such as personal data), a centralized server provides a single point where either accidental failure or deliberate compromise can have catastrophic consequences.

For these reasons and others, many foreseeable future applications for software agents involve large numbers of agents interacting with each other. Users may have a number of agents operating on their behalf, and agents of any particular user may have to communicate with other agents elsewhere on the network in order to share information.

Why multi-agent systems are hard to build

While decentralized, multi-agent systems have several important advantages, one of the largest problems with them is *how agents are supposed to find each other.* Each agent should not have to know about (and, indeed, probably cannot know about) every other agent, user, or resource on the network. Instead, some mechanism by which agents may locate only the useful agents on the network must be arranged.

This research focuses on the problems of a *matchmaking* service, one designed to find groups of people with similar interests and bring them together to form coalitions and interest groups. The intended scale of the matchmaking is that of the entire Internet, an environment in which there are potentially millions of users and millions of agents corresponding to them. The domain and the large number of agents presents difficult coordination problems, such as:

- hierarchical, single-inheritance trees are often used for such coordination, but there is no obvious a priori hierarchy in this application by which to organize the agents (why would any one person's interests be at the top of any hierarchy? how would we know whom to pick, anyway?);
- asking other agents *at random* resembles diffusion in a gas and is extremely slow—it means each agent could be required to ask every agent on the network, guaranteeing a solution that scales poorly; and
- a centralized approach runs into the problems mentioned above of quadratic scaling, and also is subject to single-point-of-failure problems if the central system either fails or is compromised—an important point for an application handling potentially sensitive data.

Finding the right cluster of peer agents: the core idea

To address these problems, this research considers an overall organization which borrows ideas from *computational ecology* [4], in which agents have only local knowledge, but self-organize into larger units. The *core ideas* in the approach taken here are to

- compare the agents' information in a *peer-to-peer, decentralized* fashion,
- use *referrals* from one agent to another and an algorithm resembling *hill-climbing* to find other, more appropriate agents when searching for relevant peers, in order to
- build *clusters* or *clumps* of like-minded agents, and to
- *use these clusters* of similar or like-minded agents (whose users therefore share similar interests) to *introduce* users to each other and enable cluster-wide *messaging* between users whose interests match.
- use a *persistent* agent that runs most of the time, for long periods; the user does not start up the agent, get an immediate result, and shut it down, but instead runs it in the background for hours or weeks, while it uses "word of mouth" to find and join appropriate groups of agents whose users share the same interests.

How the resulting clusters can be used

Once agents have formed clusters—an ongoing and continuous process for real agents on the Internet, due to the scale and constantly-changing environment involved—how can we use these clusters? There are many applications; this is a short summary:

- Messaging into the group. A user whose agent is in some particular group can send a message into the group—either those other agents known directly by the user's agent to be in the same cluster, or transitively through all other agents in the cluster by following cluster cache information in a flooding algorithm. Thus, given some particular granule on the user's local agent, the user could ask his agent to send a message to all other agents in the clump of which this granule is a member.
- Introductions. The chain of referrals themselves can be useful information, and can be exposed to the user under certain circumstances. Not only can the user send message to particular individuals (whether pseudonymously or not), but the agent itself can facilitate a "flirtatious" sort of introduction in which information can be symmetrically and gradually revealed, via cryptographic protocols. Users could ask for an explicit introduction to particular members of the cluster, or could instruct their agent to accept or solicit introductions when it looked like there was a particularly good match available.
- Finding an expert. By using a combination of messaging into the group and introductions, the clusters that a user's agent finds itself in can potentially be used to find experts on the subject, since presumably such experts (if they, too, are running the agent) will have their interests reflected in the clustering. Here, a user could prepare a small piece of prose, or find some existing message, which talks about the subject for which the user wants an expert; the clustering algorithm could then generate a granule for this grain and attempt to find a suitable cluster. Once found, it could start the introduction process to acquaint the questioner and the expert.

What is described in this paper

The following sections describe the algorithm used in a prototype of the clustering system, and some recent simulation results evaluating its performance. Earlier simulation results, along with a description of the algorithm which emphasized different aspects of Yenta-Lite's operation, can be found in [2].

Note that the algorithms described below are but a small piece of the overall task. In particular, since the system handles sensitive information such as people's interests, fielding the system on the Internet requires cryptographic privacy safeguards are the subject of current research and, to keep this paper short, are described elsewhere [1][3]. The entire system, including such cryptographic safeguards, a user interface, and other necessary elements, is called *Yenta*; to avoid confusion, the prototype described here is called *Yenta-Lite* or *YL* for short.

The Approach

The overall goal is to form clusters of agents whose users share similar interests. In order to do this, we must answer the following questions:

- What does it mean to have an interest, and how do agents know about these interests?
- How do we determine similarity of interests?
- How does a particular agent know which other agents to contact?
- How can we form clusters of similar agents?

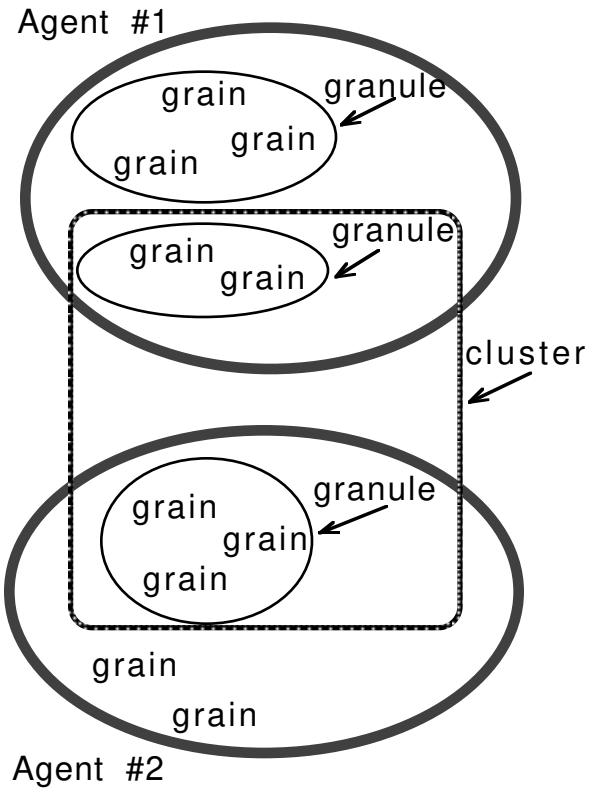
What does it mean for a user to have an interest, and how do we capture that computationally?

For the purposes of matching people by their interests, we assume that these interests are *capturable* in some computer-based form. At the moment, Yenta only deals with text, such as electronic mail messages, the contents of various newsgroup articles, the contents of the user's files in a filesystem, and so forth. The architecture of Yenta supports somewhat different sources of information as well (such as World Wide Web hotlists and homepages)—the crucial requirements for any interest are *a)* they are represented in some electronic form, hence captured by the computer, and *b)* there is some way of comparing two potential interests and assigning a *degree of similarity* between them.

As currently implemented, Yenta-Lite can examine the contents of email messages, newsgroup articles, and user files that the user has received, read, or written. The tests described in this paper used *newsgroup articles* and *email messages* only, as discussed in the section on evaluating Yenta-Lite's performance. Each individual message, article, or file being compared is considered a *document*; however, since Yenta might eventually be comparing nontextual documents, we use the term *grain* to refer to any individual chunk of bits associated with a user.

A user is deemed to *have* an interest if several grains are *similar* to each other. Such a collection of similar grains is called a *granule*. A user may own many granules, each corresponding to some separate interest; for example, a user who regularly reads newsgroup articles on dogs and cars would presumably have two granules reflecting these disparate interests.

Two users, A and B, are deemed to *share* an interest if A has at least one granule that is similar to at least one of B's granule. Two or more users who share an interest are *conceptually* in a *cluster* at the instant that they both possess similar granules; they are *actually* in clump at the instant their two agents discover this similarity. A diagram illustrating this is below.



Suppose we have three users, A, B, and C. Suppose that A and B are in a clump, and B and C are in a clump. Are A, B, and C all in a clump together? Not necessarily. If A is interested in dogs and cars, his associated granules are A_{dogs} and A_{cars}. If the other granules are B_{dogs}, B_{zebras}, C_{dogs}, and $C_{guitars}$, then A, B, and C are all in a clump, because they all share an interest in dogs. However, if C_{dogs} was instead C_{zebras}, then we have two clumps, one reflecting A and B's interests in dogs, and one reflecting B and C's interest in zebras. B in this case is in two clumps, while A and C are each in one clump.

How do we determine similarity of interests?

The fundamental assumption behind Yenta's assessment user similar of user interests is this: If two users both have several documents which are similar to each other, then the users are assumed to share an interest themselves.

In order to function at all, Yenta demands that any two grains

can be compared to yield some measure of similarity. It is also required that this measure be (at least) partially-ordered; for example, a floating-point number, which reflects how similar two grains are, is an acceptable representation. The Yenta architecture allows more sophisticated similarities than scalar numbers, but Yenta-Lite, and the results reported here, use only scalars. At the moment, it is also assumed that this comparison operator is symmetric, e.g., that if A's similarity to B is 0.74, then B's similarity to A is likewise 0.74. Future work may explore the stability of the clustering algorithm in the face of nonsymmetric comparison operators.

Since Yenta-Lite's grains are all exclusively textual, we use a straightforward keyword-vector text comparison metric. (This is similar to SMART [10], which was used in an earlier version of Yenta-Lite.) To compute similarity, we first stem all words in any given document (e.g., remove prefixes and suffixes and otherwise canonicalize the text), compute an inverse-frequency metric for each word in the document (so that rare words with greater power to discriminate two documents from each other have greater weight than common words which appear in most documents), and compute a vector which describes the document based on these. Given two different documents, we can then take the dot-product of their associated keyword vectors to compute similarity.

This is not the only way to do this, of course. For example, consider WordNet [8], which is a semantic net that allows comparing words based on how many links away one word is from another, and in what direction (e.g., synonym, antonym, superset, etc). Future implementations of Yenta may combine keyword vectors and WordNet if the advantages (e.g., possibly more resilience in the face of synonyms that rarely co-occur in a single document) outweigh the disadvantages (e.g., greater semantic "fuzz" in the comparison due to the greater number of words investigated in any given document).

Forming clusters via referrals

We now come to the heart of the clustering algorithm. Given that we have a multiplicity of agents with no central node and no hierarchy, how can we reasonably form clusters which reflect the interests of the users?

The major steps are:

- Intra-agent initialization, known as *preclustering*: Combine grains into granules within a single agent.
- Inter-agent initialization, known as *bootstrapping*: Find at least one other agent with which to communicate.
- Walk referrals and cluster: Form clusters of like-minded agents.

Only a quick summary of Yenta-Lite's preclustering is given here; more details have appeared elsewhere [2], along with earlier simulatation results.

Preclustering

When an agent first starts running, it must determine what interests its user possesses. It does this by collecting some subset of the user's email, newsgroup articles, and files; each such item is known as a *grain*. Each separate grain is considered for membership in a growing collection of granules.

Each grain is converted to a keyword vector, then compared with all other grains not yet part of a granule via dot-products. An agglomerative algorithm is used to increase the size of any given granule until no grain is within a high enough threshold (which also depends on the variance in the data) to "stick" to the granule; we then start building another granule This process of producing granules is relatively time-consuming (it has several $O(n^2)$ steps in it), but must be done only once for any given collection of the user's grains, and it appears to produce acceptable results. As the user's collection of files or new messages grows, additional grains may be incrementally added to existing granules, and new granules created when the pile of un-added grains becomes too large.[2]

Boostrapping

The next phase requires finding at least one other agent with which to communicate; finding more after that is easier—due to other agents' rumor caches—in that it is less likely that we will require either ad-hoc heuristics or user intervention. In Yenta-Lite, we finess this problem and assume that we can always find another agent. Several heuristics are availabie for true Yenta, including broadcasts and directed multicasts on local network segments to find other agents in the same organization; asking a central registry which contains a *partial* list of other known agents; and asking the user for suggestions. All of these heuristics have various advantages and disadvantages, but we shall not pursue them here.

Data structures used in finding referrals and clusters

We now come to the step in which the various granules in agents form clusters with other granules. For concreteness, assume that we have two agents, named A and B, which each have a few granules in them, e.g., G_{A0}, G_{A1}, etc. Each agent also contains several other data structures:

- A *cluster cache, CC,* which contains the names of all other agents currently known by some particular agent as being in the same cluster. Thus, if agent A knows that its granule 1 is similar to granule 3 of agent B, then CC_A contains a notation linking G_{A1} to G_{B3}. There are two important limits to the storage consumed by such caches: g_l ("local granules"), the number of separate granules that any given agent is willing to remember about itself; and g_r ("remote granules"), the number of granules this agent is willing to remember about other agents. The total size of *CC* is hence g_l times g_r. In Yenta-Lite, these are essentially unbounded; in an implementation that wishes to save space, limiting g_r before limiting g_l would seem to make the most sense, as this limits the total number of other agents that will be remembered by the local agent, while not limiting the total number of disparate interests belonging to the user that may be remembered by the local agent.
- A *rumor cache, RC,* which contains the names and other information (described below) from the last r agents that this agent has communicated with. In Yenta-Lite, r is arbitrarily set to 5, and it should definitely be bounded in true Yenta as well, since otherwise any given agent will

remember *all* of the agents it has ever encountered on the net and its storage consumption will grow without bound. Reasonable values for bounds in real-life operation with large numbers of agents are currently unknown, but are suspected to be on the order of 20 to 100.

- A *pending-contact* list, *PC*, which is a priority-ordered list of other agents that have been discovered but which the local agent has not yet contacted.

The rumor cache contains more than just the names of other agents encountered on the network. It also contains some subset, perhaps complete, of the text of each granule corresponding to those agents. Exactly how much of this text is stored has several tradeoffs [2]; in particular, storing more text makes possible heterogeneous comparison metrics between different versions of Yenta running on the network, at a possible cost in space and security (due to the privacy implications of compromised agents) which must be ameliorated using cryptographic protocols not discussed here[3].

Getting referrals and doing clustering

Now that we have all this mechanism in place, performing referrals and clustering is relatively uncomplicated.

The process starts when some agent (call it A) has finished preclustering and has found at least one other agent (call it B) via bootstrapping. Agent A then performs a comparison of its local granules with those of agent B, using a process reminiscent of the preclustering phase but simplified. A builds an upper-triangular matrix describing the similarities between each of its local granules and those locally held by B. Then, rather than taking averages and standard deviations, it simply finds the highest score (e.g., closest similarity) between any given granule (say, G_{A1}) and B's granules. If there is no such value above a particular threshold, then the local granule under consideration does not match any of B's granules, although some other local granule, e.g., G_{A2}, might match.

The comparison process is simpler in the clustering (inter-agent) phase than in the preclustering (intra-agent) phase in part because two agents talking to each other cannot assume that they have complete information about either each other or the space of all possible other granules on the network. Thus, we do not bother trying to calculate averages and standard deviations; as observed in the prototype, a simpler, threshold-based match appears to work well enough.

When we are done comparing granules from A with granules from B, agent A may have found some acceptably close matches. Such matches are entered, one pair of granules at a time, in A's cluster cache. B is likewise doing a comparison of its granules with A and is entering items in its own cluster cache.

Whether or not any matches were found that were good enough to justify entering them in a cluster cache, the next step is to acquire *referrals* to agents that might be better matches. In the example here, agent A asks agent B for the entire contents of its rumor cache, and runs the same sort of comparison on those contents that it did on agent B's own local granules. Good matches are added to A's cluster cache, the rest of the data is added to A's rumor cache, and A's namelist is updated by adding to it those other agents which showed good matches to A, that is, those agents which had granules that went into A's cluster cache. These agents will be contacted next, after A finishes with B and any other entries in its namelist. The various caches belonging to B that A has been consulting were gathered by B in a similar way; every agent participating in this protocol is thus building up a collection of data for its own use and for the use of other agents.

This procedure acts somewhat like human word of mouth. If Sally asks Joe, "What should I look for in a new stereo?" Joe may respond, "I have no idea, but Alyson was talking to me recently about stereos and may know better." In effect, this has put Alyson into Sally's rumor cache (and, if Joe could quote something Alyson said that Sally found appropriate, perhaps into Sally's cluster cache as well). Sally now repeats the process with Alyson, essentially hill-climbing her way towards someone with the expertise to answer her question.

Experimental Evaluation of the Algorithm

Previous work has investigated the number of messages that must be exchanged to reach a high level of convergence, and the quality of the resulting clusters, for small numbers of agents (20 or less) [2]. That work showed that, relative to a complete n-by-n crossbar comparison (the naive algorithm which uses no clustering and no referrals), Yenta did not exchange an unreasonably larger number of messages (less than a factor of two), yet achieved several important advantages over such a centralized, brute-force solution:

- The clusters are grown incrementally for each agent, so at any given time, each agent sees at least some of many clusters.
- No agent need retain knowledge of all other granules in the system at any time.
- If an agent were to disappear from the system, the only lasting effect would be for other agents to "forget" it—the rest of the clusters would still form.

Yenta has not yet been made available for widespread use on the Internet, hence truly large-scale data on its performance and behavior is not yet available. However, recent simulations have supported this assertion for larger numbers of agents.

In particular, the Yenta clustering algorithm has been simulated for various numbers of interests, typical sizes of its rumor cache, and up to 1000 Yentas, and shown good performance and convergence. Graphical results of these simulations are presented on the last page of this paper, and discussed immediately below.

Three different simulations are presented; for each, the format of presentation is identical. Each simulation is shown as a series of images taken at various timesteps. The final state of any given simulation is the large image on the right; the six smaller images to the left of that image represent earlier stages of the simulation, reading from left to right and top to bottom.

Each Yenta in any given simulation was given a random interest from the total number of interests available, and then the size of its cluster cache was examined at each simulation step, which indicates how successful it has been at finding other Yentas which share its interest. For all Yentas that share the same pair of parameter values (e.g., rumor cache size versus number of Yentas for the first simulation) and are hence in the same bar of the display, the size of their cluster caches were averaged. This average is then compared to the total number of Yentas that *could* have conceivably been in the cluster cache (if all Yentas sharing the interest had been found), and that ratio is expressed as the percentage height of the bar.

The first simulation shows the effect of varying the size of the rumor cache for up to 1000 Yentas, given 30 different interests split amongst the Yentas. Roughly speaking, it shows that the size of the rumor cache does not make much difference in the speed of cluster formation for more than around 400 Yentas.

The second simulation varies the number of possible interests shared amongst the Yentas with the total number of Yentas, given a rumor cache size of 50. As might be expected, it takes longer to find all the other Yentas one would want as the number of interests increases or as the number of total Yentas increases.

Finally, the third simulation shows the effect of varying the size of the rumor cache for various numbers of interests, given 1000 Yentas. This seems to show that a rumor cache size of 15 is enough for small numbers of interests, and that raising this size beyond 35, even for large numbers of interests, does not buy us much.

Related Work

There are many efforts in distributed AI and multi-agent systems which could be considered relevant; here we consider only other matchmaking systems and related approaches.

A common technique in systems that support computation amongst a group of users is to centralize a server and have its users act like clients. Systems that match user interests to each other, and have such a centralized structure, include Webhound/Webdoggie [9] and HOMR/Ringo/Firefly[7].

Kuokka and Harada [6] describe a system that matches advertisements and requests from users and hence serves as a brokering service. Their system certainly is a matchmaker, but it assumes a centralized architecture and a highly-structured representation of user interests.

Others have taken a more distributed approach. For example, Kautz, Milewski, and Selman [5] report work on a prototype system for expertise location in a large company. Their prototype assumes that users can identify who else might be a suitable contact, and use agents to automate the referral-chaining process; they include simulated results showing how the length and accuracy of the resulting referral chains are affected by the number of simulated users and the accuracy and helpfulness of their recommendations. Yenta-Lite differs from this approach in using ubiquitous user data to infer interests, rather than explicitly asking about expertise.

Conclusions

Yenta-Lite demonstrates that referral-based matchmaking can provide acceptable results without requiring any one agent to know about all other agents, and without requiring unreasonable messaging traffic or computational demands.

Acknowledgments

I would like to thank undergraduate Bayard Wenzel for implementing the initial prototype of these ideas, and express my especially heartfelt thanks to Barry Crabtree of British Telecom for his recent simulation results for large numbers of Yentas. I would also like to thank my advisor, Dr. Pattie Maes, for her advice and her support of this research. This research has been supported in part by British Telecom.

References

[1] Foner, Leonard, “Clustering and Information Sharing in an Ecology of Cooperating Agents, or How to Gossip without Spilling the Beans,” *Proceedings of the Conference on Computers, Freedom, and Privacy '95 Student Paper Winner,* Burlingame, CA, 1995.

[2] Foner, Leonard, “A Multi-Agent Referral System for Matchmaking,” *PAAM ‘96 Proceedings*, London, England, 1996.

[3] Foner, Leonard, “A Security Architecture for a Multi-Agent Matchmaker”, submitted to *Autonomous Agents ‘97,* Marina del Rey, 1997.

[4] Huberman, B.A., editor, *The Ecology of Computation*, Elsevier Science Publishers B.V., 1988.

[5] Kautz, Henry, Milewski, Al, and Selman Bart, “Agent Amplified Communication,” *AAAI '95 Spring Symposium Workshop Notes on Information Gathering in Distributed, Heterogeneous Environments,* Stanford, CA.

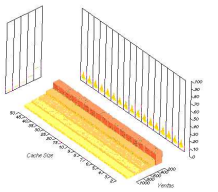
[6] Kuokka, Daniel, and Harada, Larry, “Matchmaking for Information Agents,” *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) '95,* 1995.

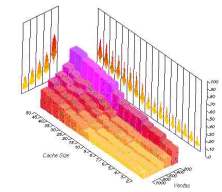
[7] Lashkari, Yezdi, Metral, Max, and Maes Pattie, “Collaborative Interface Agents,” *Proceedings of the Twelfth National Conference on Artificial Intelligence*, MIT Press, Cambridge, MA, 1994.

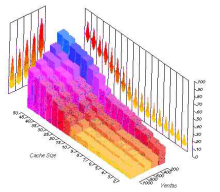
[8] Miller, George, Beckwith, Richard, Fellbaum, Christiane, Gross, Derek, and Miller, Katherine, “Introduction to WordNet: An On-line Lexical Database,” Princeton University Technical Report, 1993.

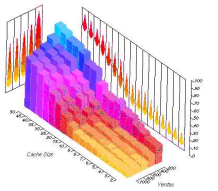
[9] Shardanand, Upendra, and Maes Pattie, “Social Information Filtering: Algorithms for Automating ‘Word of Mouth,’” *Proceedings of the CHI '95 Conference, 1995.*

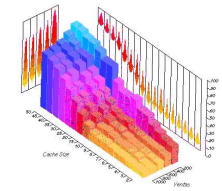
[10] Zumoff, Joel, “Users Manual for the SMART Information Retrieval System,” Cornell Technical Report 71-95.

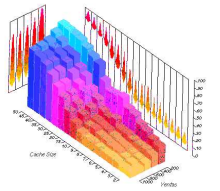

30 Interests Time: 10

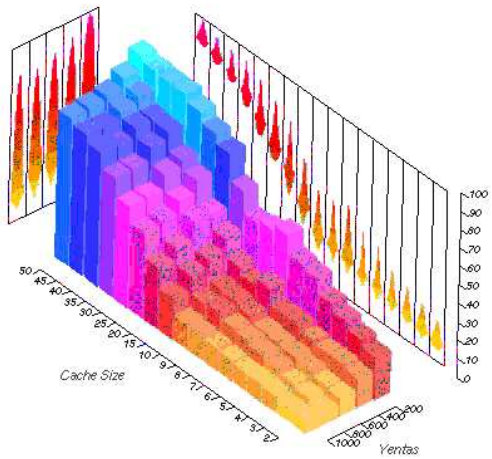

30 Interests Time: 100


30 Interests Time: 200

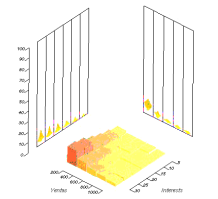

30 Interests Time: 290

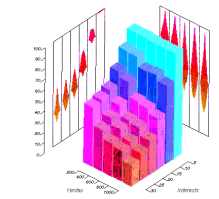

30 Interests Time: 370

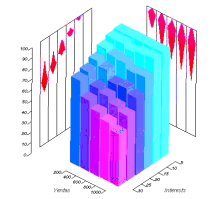

30 Interests Time: 460

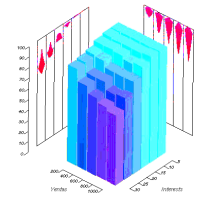


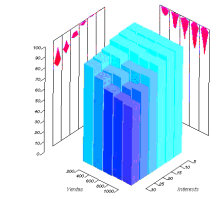
30 Interests Time: 500

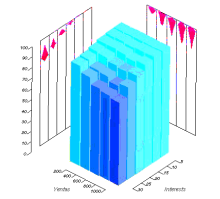

Rumour Cache = 50 Time: 10

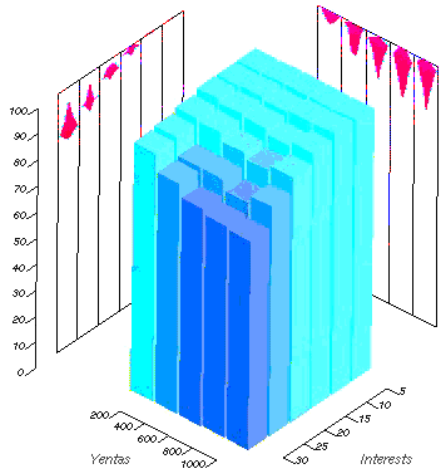

Rumour Cache = 50 Time: 90


Rumour Cache = 50 Time: 190

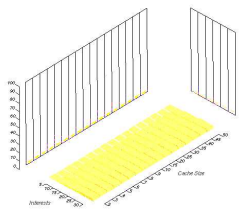

Rumour Cache = 50 Time: 290

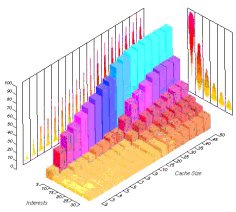

Rumour Cache = 50 Time: 370

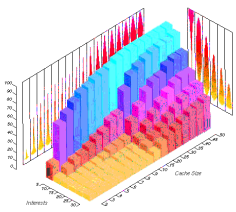

Rumour Cache = 50 Time: 460

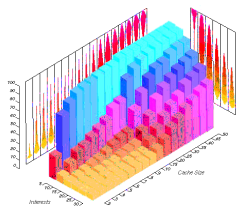


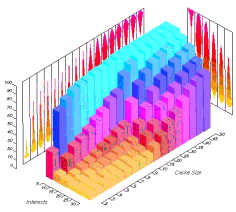
Rumour Cache = 50 Time: 500

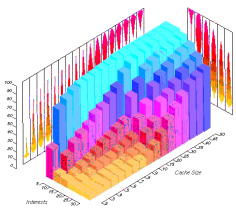

1000 Ventas Time: 10

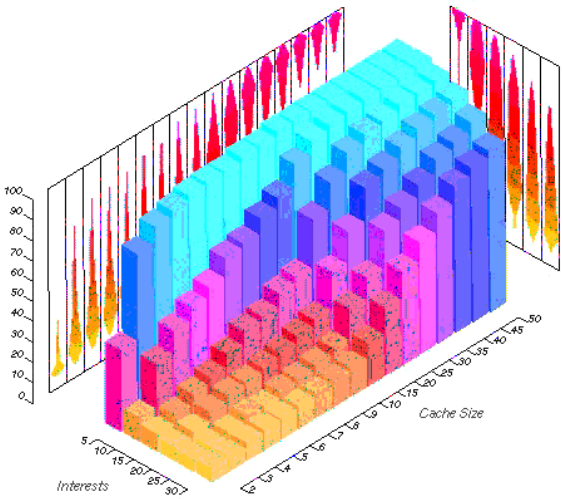

1000 Ventas Time: 80


1000 Ventas Time: 150


1000 Ventas Time: 230


1000 Ventas Time: 310


1000 Ventas Time: 390



1000 Ventas Time: 490