Latency-dependent fitness in evolutionary multithreaded Web agents

Melania Degeratu
Computer Science Department
Columbia University
New York, NY 10027
melania@cs.columbia.edu

Gautam Pant and Filippo Menczer
Management Sciences Department
University of Iowa
Iowa City, IA 52242
{gautam-pant,filippo-menczer}@uiowa.edu

Abstract

The World Wide Web creates opportunities for search systems using adaptive distributed agents. This paper presents a threaded implementation of *InfoSpiders*, a client-based system that uses an evolving population of intelligent agents to browse the Web at query time. We consider different fitness functions based on network resource consumption and show that taxing agents in proportion to latency results in better efficiency without penalties in the quality of the retrieved documents. The tool is available to the public as a Java applet.

1 Introduction

Search engines make use of the state of the art in information retrieval in conjunction with automated programs, called *crawlers*, that continuously visit large parts of the Web in a blind, exhaustive fashion. The crawled pages are stored in a static index mapping every term from a controlled vocabulary onto the collection of URLs containing that term. The index is used at query time to return the sets of URLs containing the terms in users' queries.

The Web is dynamic, with pages being added, deleted, updated, moved, and linked to each other in an unstructured manner. It is also very large and growing at a very fast pace [2]. Therefore the set of relevant pages for any given query is also highly dynamic, leading to a *scalability* problem — the assumption of an accurate and complete static image of the Web breaks down with its rate of change. As search engines fail to satisfy the user's need for complete and recently updated information, it becomes highly desirable to improve the coverage and recency of search engines.

One approach to address the coverage issue is the use of meta-search techniques to combine relevant sets from multiple search engines. Meta-search has been shown to improve recall compared to single "traditional" search engines [1]. Yet even meta-search engines cannot locate recent pages unknown to the individual engines whose results are combined. Therefore we suggest a more radical solution to the scalability problem: complementing index-based search engines with intelligent search agents at the user's end.

This paper describes an evolutionary multi-agent system designed to browse adaptively on behalf of the user to complement search engines. We discuss the implementation of such agents in Java and the design of appropriate fitness functions for this application.

2 Sequential InfoSpiders

InfoSpiders [3] is a multi-agent system for online, dynamic Web search. Each agent checks its information neighborhood (defined by hyperlinks) looking for new documents relevant to the user's query and having little or no interaction with other agents. InfoSpiders are able to display an intelligent behavior by evaluating the relevance of the document content with respect to the users query, and by reasoning autonomously about future actions that mimic the browsing habits of human users. Adaptation occurs at both individual and population levels, by evolutionary and reinforcement learning. The goal is to maintain diversity at a global level, trying to achieve a good coverage of all aspects related to the query, while capturing the relevant features of each agent's local information environment.

InfoSpiders employ an evolutionary computation approach based on *local selection* [4]. An agent's "energy," related to the relevance of the pages visited by that agent, is accumulated over time. We want to reward agents that find relevant pages first, and not agents that visit those pages subsequently. For this we employ a caching system, and the interaction among individuals is limited to sharing access to this cache. Selection occurs in a distributed fashion, when the energy level of an agent reaches a certain fixed threshold. This selection maintains diversity in a way similar with fitness sharing without a global selection bottleneck.

InfoSpiders rely on traditional search engines to obtain a set of starting URLs pointing to pages supposedly relevant to the query submitted by the user. The user also provides a maximum number of pages the population of agents may visit. The starting documents are prefetched, and each agent is "positioned" at one of these documents and given an initial amount of energy. An agent estimates each outgoing link by looking at the occurrence of keywords in the vicinity of the link. The agent then uses these link relevance estimates to choose the next document to visit.

After a document has been visited, the agent needs to update its energy. Energy is used to move and survive, so the agent will be rewarded with energy based on the estimated value of the visited documents. Since the energy dynamics of an agent determine its reproductive rate, determining an appropriate fitness function to map the quality and cost of visited pages into energy is crucial for the success of the system. Alternative fitness functions are discussed in the next sections.

The adaptive representation of InfoSpiders roughly consists of a list of keywords, initialized with the query terms, and of a feed-forward neural net. The keywords represent an agent's opinion of what terms best discriminate documents relevant to the user from the rest. The neural net is used to estimate links.

An agent can modify its behavior during its life by comparing the relevance of the current document (as evaluated once the document is visited) with the estimation that was made from the previous page, prior to following the link that led to the current one. Reinforcement or prediction learning can be employed to achieve this goal. The relevance is computed by an agent as a similarity measure between its keywords and the current document. The learning scheme is completely unsupervised.

InfoSpiders adapt not only by learning neural net weights, but also by evolving keyword representations. If an agent accumulates enough energy, it clones a new agent in the location (page) where it is currently situated. At reproduction, the keyword vector of the offspring is mutated by replacing the least useful (discriminating) term with a term that appears better correlated with relevance. The two agents can continue the search independently of each other. If an agent runs out of energy, it is destroyed. This way, agents are allocated to promising areas of the information space.

```
InfoSpiders(query, INIT_POP, MAX_PAGES) {
   starting_urls := search_engine(query, INIT_POP);
   for agent (1..INIT_POP) {
      initialize(agent, query, one_of(starting_urls));
      agent.energy := THETA / 2;
   }
   foreach agent {
      while (alive & (visited < MAX_PAGES)) {
         pick_link_from_current_document(agent);
         p := fetch_new_page(agent);
         lock(cache_semaphore);
         update(cache);
         unlock(cache_semaphore);
         agent.energy += fitness(p);
         learn(agent);
         if (agent.energy > THETA) {
            offspring := mutate(clone(agent));
            offspring.energy := agent.energy / 2;
            agent.energy -= offspring.energy;
         }
         elseif (agent.energy < 0) death(agent);
      }
   }
}
```

Figure 1: Pseudocode of multithreaded InfoSpiders.

The output of the algorithm is a flux of links to documents, ranked by estimated relevance. The algorithm stops when the population goes extinct for lack of relevant information, visits the maximum number of documents, or is terminated by the user. Additional details on the algorithm can be found elsewhere [3].

3 Multithreaded MySpiders

Due to the parallel nature of the InfoSpiders algorithm, mutithreading can be expected to provide better utilization of resources as compared to a single thread (sequential) implementation. Since Java has built-in support for threads and allows for classes to be loaded at runtime over the Web, we implemented a multithreaded version of InfoSpiders as a Java applet.

The multithreaded implementation allows one agent thread to use the network connection to retrieve documents, while other agents can use the CPU, or access cache information on the local disk. Figure 1 illustrates the multithreaded InfoSpiders applet. The only addition to the algorithm described in the previous section is a lock mechanism to allow concurrent access to the shared cache.

Unfortunately, there are other issues that weaken the Java choice, the most important of which are the low speed of execution of Java byte code and the need to provide a mechanism for granting privileges to the applet. In fact, to be able to open network connections to hosts other than the one from where the applet itself

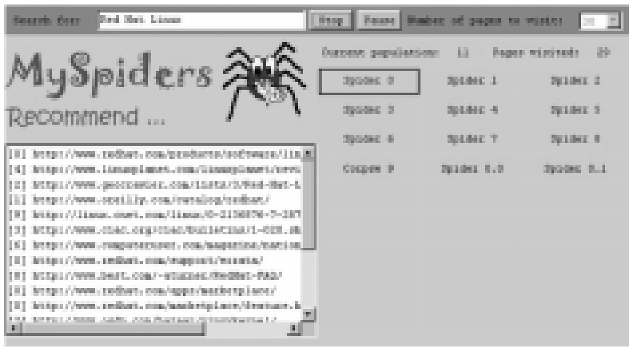


Figure 2: MySpiders applet screenshot.

was downloaded, and to access the local disk for cache I/O operations, the applet has to bypass the browser's security manager. One way to accomplish this task is to digitally sign the applet code to request additional privileges from the user. The problem was first addressed by using the Netscape Signing Tool, but this solution narrowed the portability of the application. Therefore, as an alternative solution we used Java's Signing Tool. The digital signatures created this way can be verified using a Java plug-in. The problem here is that the plug-in's verification procedure has platform dependencies. A different way for the applet to acquire the required privileges is the use of policy files. For this purpose, in addition to the plug-in, the user needs to save a policy file under platform specific directories. The current system uses a hybrid of the two technologies mentioned above — signed applets and policy files — to support multiple platforms and provide ease of use. Through the development experience, the team has learned that applet portability is not as easy as it was meant to be.

A simplified version of InfoSpiders, called MySpiders, was implemented as a multithreaded Java applet and is deployed on a public Web server (`http://myspiders.biz.uiowa.edu`). Figure 2 shows the user interface of the MySpiders applet in the course of a query search. MySpiders hides from the user many parameters in order to keep the user interface as simple as possible.

Once the start button is pressed, an automated browsing process is initiated. The user can scan through a ranked list of relevant pages found up to that time. The relevant URLs are preceded by a number that represents the agent (or spider) that found the page. If the user likes a page, clicking on the spider that found it provides a list of other pages found by that spider. The user has the choice to stop the search process at any time if she has already found the desired results, or if the process is taking too long.

4 Evaluation of Fitness Functions

A good choice of fitness function to map visited pages into energy income is essential for the MySpiders applet, due to its interactive nature. Users will only use tools that are both effective and efficient. For this reason we consider two components in our fitness functions: a benefit to reflect the estimated relevance of pages with respect to the user query, and a cost to account for the network resources used to download pages. For the benefit component, previous experiments have shown that the frequency of query terms in a page is an appropriate measure [3]. For the cost component, we have used in earlier implementations a simple model that assumes equal costs for all pages. Now we want to compare this model with a latency-dependent scheme in which agents are charged more for visiting large pages and slow servers.

For the constant cost model we want the population to visit at least `MAX_PAGES` new documents before running out of its initial energy, therefore we set

$$cost_{const} = \frac{\texttt{INIT\_POP} * \texttt{THETA}}{2 * \texttt{MAX\_PAGES}}.$$

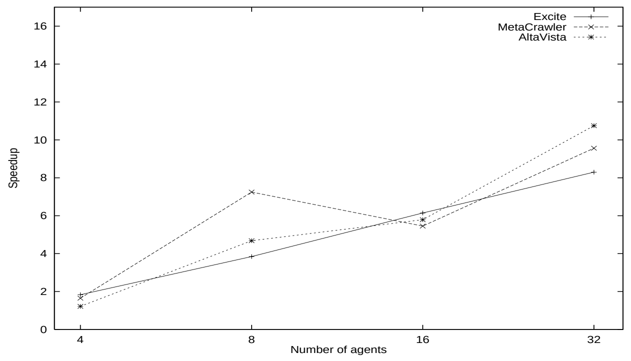
For the latency dependent cost model, we compute

$$cost_{latency}(p) = 2 * cost_{const} * \frac{time(p)}{\texttt{TIMEOUT}}$$

where $time(p)$ is the real time in which the page is downloaded and `TIMEOUT` is the timeout parameter for the socket connection. Note that the expected value of the two cost models is the same under a uniform latency assumption.

Figure 3 plots the speedup obtained for different sizes of the initial population and for the constant and latency-dependent fitness functions. These preliminary results are very encouraging, with the speedup reaching values as high as 15-fold for larger populations. Furthermore, the speedup for large populations are approximately 50% higher when the cost function is latency-dependent.

This result leads to questioning the quality of the documents retrieved during these runs: Does the more efficient concurrent implementation bias the search toward pages and servers that — while faster — contain less relevant information? Figure 4 shows the rates at which new relevant pages are visited, plotted versus the size of the initial populations for constant and latency-dependent costs. Again we observe a better performance for the latency-dependent fitness function, suggesting that it affords better efficiency without a penalty in the quality of the results.



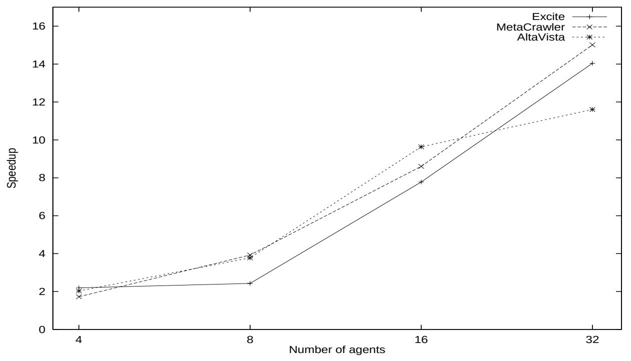
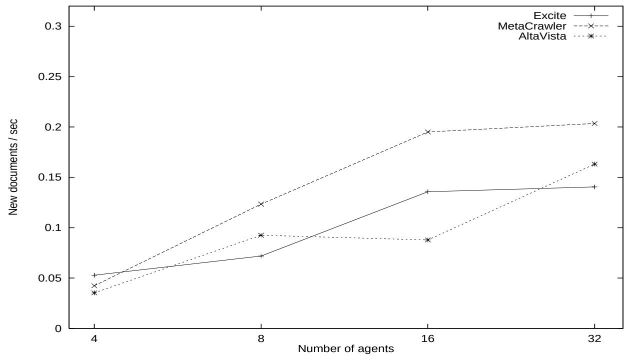


Figure 3: Speedup acquired for constant (left) and latency-dependent (right) cost. In this experiment we use a single query ("`neural networks`") and 3 search engines to get the starting points.



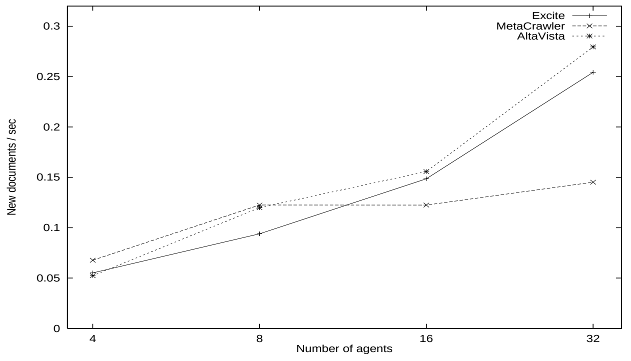


Figure 4: Relevant new pages retrieved per unit time, for constant (left) and latency-dependent (right) fitness functions. A document is assumed to be relevant if its similarity to the query is above a 2% threshold.

5 Conclusions

We have described a multithreaded implementation of the InfoSpiders system in which evolutionary information agents can browse the Web concurrently. We experimented with two fitness functions and found that taxing agents in proportion to their usage of network resources results in better efficiency without apparent penalties in the quality of the retrieved documents. The multithreaded approach has proven very efficient, achieving speedups of an order of magnitude. The latency-dependent fitness model has been incorporated into the public MySpiders applet.

Our preliminary experiments suggest that complementing static search with an dynamic component has a potential to improve the quality of the pages presented to the user, and that evolutionary multi-agent systems can perform this task satisfactorily.

However, to fully understand the benefits of using InfoSpiders on top of static search results, the system has to undergo more extensive tests in conjunction with performance assessments from users. We also intend to test further variations of the fitness function, considering factors such as document popularity in addition to latency and query similarity.

References

[1] S. Lawrence and C. Giles. Context and page analysis for improved web search. *IEEE Internet Computing*, 2(4):38–46, 1998.

[2] S. Lawrence and C. Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[3] F. Menczer and R. Belew. Adaptive retrieval agents: Internalizing local context and scaling up to the web. *Machine Learning*, 39(2/3):203–242, 2000.

[4] F. Menczer, M. Degeratu, and W. Street. Efficient and scalable pareto optimization by evolutionary local selection algorithms. *Evolutionary Computation*, 8(2):223–247, 2000.