Available online at www.sciencedirect.com
ScienceDirect
Applied Soft Computing 7 (2007) 890–898
Applied Soft Computing
www.elsevier.com/locate/asoc

Evolutionary design of en-route caching strategies

Jürgen Branke [a,*], Pablo Funes [b,**], Frederik Thiele [c]

[a] *Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany*
[b] *Icosystem Corp., 10 Fawcett ST. Cambridge, MA 02138, USA*
[c] *i+o Industrieplanung und Organisation GmbH & Co. KG Römerstrasse 245, 69126 Heidelberg, Germany*

Received 3 May 2005; received in revised form 28 February 2006; accepted 6 April 2006
Available online 5 June 2006

Abstract

Nowadays, large distributed databases are commonplace. Client applications increasingly rely on accessing objects from multiple remote hosts. The Internet itself is a huge network of computers, sending documents point-to-point by routing packetized data over multiple intermediate relays. As hubs in the network become overutilized, slowdowns and timeouts can disrupt the process. It is thus worth to think about ways to minimize these effects. Caching, i.e. storing replicas of previously-seen objects for later reuse, has the potential for generating large bandwidth savings and in turn a significant decrease in response time. En-route caching is the concept that all nodes in a network are equipped with a cache, and may opt to keep copies of some documents for future reuse [X. Tang, S.T. Chanson, Coordinated en-route web caching, IEEE Transact. Comput. 51 6 (2002) 595–607]. The rules used for such decisions are called ''caching strategies''. Designing such strategies is a challenging task, because the different nodes interact, resulting in a complex, dynamic system. In this paper, we use genetic programming to evolve good caching strategies, both for specific networks and network classes. An important result is a new innovative caching strategy that outperforms current state-of-the-art methods.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Genetic programming; En-route caching; Robustness

1. Introduction

The Internet is a distributed, heterogeneous network of computers. From a user's point of view, it can be regarded as a large database of data objects—''documents'' that are available for retrieval via their *uniform resource locator* (URL). Access to files on the net is based on a client-server architecture: a client computer generates a request, opens up a connection to a server host, and retrieves the document from the server.

Applications based on Internet protocols, such as web servers and browsers, are usually unaware of the underlying transport layer. They see the net as an all-to-all, fully connected network where each host can talk directly to any other.

Underneath, there is a real network of computers connected to each other via physical links of various kinds (coaxial and fiber optics cables, satellite links and so on; see Fig. 1) that relay each message, passing it along until it reaches its destination. The average number of intermediate steps (*hops*) can be quite substantial depending on the topology of the network, the origin and destination, and the routing algorithm.

The coexistence of multiple superimposed paths creates the potential for bottlenecks and congestion. Internet users experience *latency* when there is an extended wait between the moment a document is requested and that of its reception, and low perceived bandwidth when the transmission of the document is slow. In order to avoid infinite queues from forming, relaying hosts usually implement a timeout feature, killing documents when the transmission is delayed more than a certain threshold, so some documents never arrive at their destinations.

These problems can be prevented either by expanding the capacity and bandwidth of overloaded links, in order to match peak time demand, or by making a more efficient use of the existing capacity. One such proposal is *en-route caching*, an approach to minimize network traffic by exploiting regularities in document request patterns [1,2].

Popular documents on the net (portals, for example) are requested all the time, while others are almost never requested. Therefore, it makes sense to store copies (*replicas*) of popular documents at several places in the network. This phenomenon has prompted hosting companies (e.g. Akamai.com) to position

* Corresponding author. Tel.: +49 721 608 6585; fax: +49 721 608 5998.
** Corresponding author. Tel.: +1 617 520 1030; fax: +1 617 492 1505.
E-mail addresses: branke@aifb.uni-karlsruhe.de (J. Branke), pablo@icosystem.com (P. Funes).

1568-4946/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2006.04.003

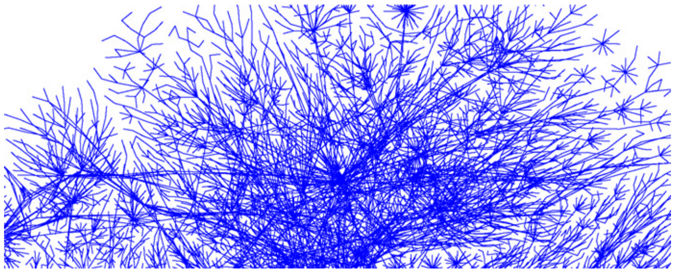


Fig. 1. Map of the Internet (fragment). Nodes correspond to ip addresses, edges to links between them. Based on data by The Opte Project (http://www.opte.org/).

hosts all over the world, creating forms of mirroring to save bandwidth by servicing requests from hosts that are closer, in Internet topology terms, to the clients making the requests. However, this solution works only for long-term data access patterns in which a commercial interest can be matched with monetary investments in distributed regions of the globe.

For the en-route caching perspective, observe that when two neighbors in the same street request the same web page, each one of them creates a channel to the remote server hosting the document, even though both requesting computers are connected to the same trunk line. The same data is sent over from the host twice, and relayed by the same intermediate routers. It would make sense, for any of the intermediate hosts, to keep a copy of the document, allowing it to service the second request directly, without having to contact the remote host at all.

Proxy servers sometimes have caching capability, being able to optimize Internet access for a group of users in a closed environment, such as a corporate office or a campus network. However, much better savings and scalability are possible by using this strategy at all levels: If the campus proxy fails to retrieve the page from the cache, or even, if the two requests come from neighboring university campuses in the same city, then a node further down the chain would have the opportunity of utilizing its cache memory, for the same opportunity exists in every single router on the Internet. The proposal of en-route web caching is to provide every intermediate host with the option to respond to a request by sending back a cached copy of a document that was previously requested.

The difficult question is how to decide which documents to store, and where to store them. With finite memory, it is impossible for individual hosts to cache all the documents they see.

A global policy control in which a centralized decision-making entity distributes replicas among servers optimally is impractical, for several reasons: the tremendous complexity, because the Internet is dynamically changing all the time, and because no global authority exists. Thus, each server has to decide independently which documents it wants to keep a replica of. The rules used for such decisions are also known as *caching strategies.*

Today, many routers with caching—such as a campus network proxy—use the well-known least recently utilized (LRU) strategy: objects are prioritized by the last time they were requested. The document that has not been used for the longest time is the first to be deleted. Although this makes sense for an isolated router, it is easy to see why LRU is not an optimal policy for a *network* of caching hosts. In our example above, all the intermediate hosts between the two neighbors that requested the same document, and the server at the end of the chain, will store a copy of the document because a new document has the highest priority in LRU. However, it would be more efficient if only one, or a few, but not all intermediate nodes kept a copy—leaving room for caching other highly requested documents. In isolation, a caching host tries to store all the documents with highest priority. In a network, a caching host should try to cache only those documents that are not cached by its neighbors.

The possible economic benefits of en-route web-caching are obvious: it has not only the potential to remove congestions and thus reduce latency and save time for the end user, but it could also reduce bandwidth requirements and even the load of highly popular servers by shielding off some traffic and serving the requests locally. Finally, the distributed structure of en-route web caching could reduce the impact of link or server failures. As the Internet grows in size and importance and file sizes grow due to an increasing amount of multimedia content, these issues will gain even more importance.

However, designing good en-route caching strategies for a network is a non-trivial task, because it involves trying to create global efficiency by means of local rules. Furthermore, caching decisions at one node influence the optimal caching decisions of the other nodes in the network. The problem of cache similarity of the above example is one of *symmetry breaking*: when neighbors apply identical, local-information based strategies, they are likely to store the same documents in their caches. In this scenario, the network becomes saturated with replicas of the same few documents, with the consequent degradation of performance.

In this paper, we attempt to design good caching strategies by means of genetic programming (GP). As we will show, GP is able to evolve new innovative caching strategies, outperforming other state-of-the-art caching strategies on a variety of networks.

The paper is structured as follows: first, we will cover related work in Section 2. Then, we describe our GP framework and the integrated network simulator in Section 3. Section 4 goes over the results based on a number of different scenarios. The paper concludes with a summary and some ideas for future work.

2. Related work

There is a huge amount of literature on caching at the CPU level (see e.g. [3]). Caching Internet documents is a special field of caching. For a survey, see Wang [4]; Davison [5], and, with a particular focus on cache replacement strategies, Podlipnig and Boszormenyi [6]. But even those are primarily concerned with caching on the receiving end, like browser caches or proxy caching. As we have argued in the introduction, much higher benefits can be achieved by allowing documents to be cached anywhere on the network. Such network or web caching has only recently received some attention.

Generally, the literature on web caching can be grouped into two categories:

(1) Centralized control: this is the idea of a central authority overseeing the entire network, deciding globally which replicas should be stored on which server. It is usually assumed that the network structure and request pattern are known.
(2) Decentralized control: in this group, decentralized caching strategies are proposed, i.e., rules that allow each server to independently decide which recently seen documents to keep. Since these rules are applied on-line, it is important that they can be processed efficiently and that they do not cause additional communication overhead.

The centralized control version is also known as the ''file allocation problem''. For a classification of file allocation algorithms, see Karlsson et al. [7]. In Loukopoulos and Ahmad [8], an evolutionary algorithm is used to find a suitable allocation of replicas to servers. Besides retrieval cost, Sen [9]; Pierre et al. [10] additionally consider the issue of maintaining consistency after a document is modified, an aspect which we deliberately ignore here. In any case, while the centralized approach may be valid for small networks, it becomes impracticable for larger networks, as the data analysis, the administrative costs, and the conflict between local authorities make it impracticable.

Given the difficulties with a centralized approach, in this paper we focus on decentralized control. We try to find simple strategies that can be applied independently on a local level, thereby making a global control superfluous. This approach is more or less independent of the network structure and request pattern, and can thus adapt much quicker to a changing environment.

Early work on web caching has simply used or adapted traditional caching strategies from the CPU level, such as LRU and least frequently used (LFU), see e.g. Williams et al. [11]. The disadvantages of these, namely that they lead to cache symmetry, were described in the introduction. An example of a network-aware caching strategy is greedy dual size frequency (GDSF), which considers the number of times a particular document has been accessed, its size and the cost to retrieve that document from a remote server (in our case, the distance, measured in hops, to the server holding the next replica).

GDSF's cost-of-retrieval factor avoids the cache repetition problem by reducing the priority of documents that can be found in nearby caches. It has been shown to be among the best caching strategies for networks [12,13]. Another comparison of several web caching strategies can be found in Bahn et al. [14].

An interesting alternative has recently been suggested in Tang and Chanson [1]. There, a node holding a document and receiving a request uses a dynamic programming based method to determine where on the path to the requesting node replicas should be stored. While this approach certainly holds great potential, it requires that all nodes in the network cooperate, and the computation of the optimal allocation of replicas to servers is time-consuming. Furthermore, request frequency information must be stored not only for documents in the cache, but all documents ever seen.

As has already been noted in the introduction, in this paper we attempt to evolve a decentralized caching strategy by means of GP. Previous examples of using GP for the design of caching strategies were demonstrated in Paterson and Livesey [15]; O'Neill and Ryan [16] for the case of the instruction cache of a microprocessor. Some of their fundamental ideas are similar to ours: GP is a tool that can be used to explore a space of strategies, or algorithms for caching. However, the nature of the CPU cache is different from the problem of distributed network caching because it does not involve multiple interconnected caches.

The present paper is a more detailed and extended version of a previous publication [17].

3. A genetic programming approach to the evolution of caching strategies

In this section, we will describe the different parts of our approach to evolve caching strategies suitable for networks. We will start with a more precise description of the assumed environment and the network simulator used, followed by the GP implementation.

3.1. Network simulator

The overall goal of our study was to evolve caching strategies for complex data networks like the Internet. However, for testing purposes, we had to design a simplified model of the network.

3.1.1. Edges and nodes

Our network consists of a set of servers (nodes), connected through links (edges). Each server has a number of original documents (which are never deleted), some excess storage space that can be used for caching, and a request pattern. All hosts function simultaneously as clients, requesting documents, and content providers, distributing their original documents.

We assume that the shortest paths from each server to all other servers, as well as the original locations of all documents, are known. Thereby, we are obviating the problem of *routing* and focusing only on the *caching* decisions.

When a host asks for a document, it sends out a request along the route (shortest path) to the remote server that keeps the original copy of the file (publisher). The request goes initially to the first host in the path, which passes it along to the next one, and so on until reaching the destination. Each one of these intermediate nodes check whether they have a cached replica of the requested document. If not, the request is passed through. Whenever a cached replica or the original document have been found, the document is sent back to the requesting server following the inverse route. All the intermediate nodes receive the document's packets and pass them along the path to the next node, until the document arrives at its destination. An example is depicted in Fig. 2.

3.1.2. Bandwidth, queuing and packetizing

When a document travels through the network, it is divided into many small packets. Each link has an associated

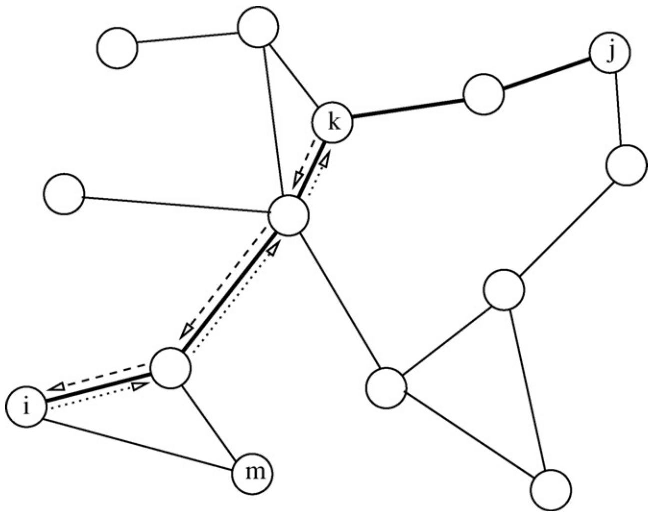


Fig. 2. When node *i* requests a document from server *j*, the request is sent along the shortest path (bold), and the first replica found in this path is sent back (say from node *k*). Other possible replicas not on the path are ignored (e.g., a replica on node *m*).

bandwidth, being able to deliver a limited number of bytes per second. Excess packets wait in an infinite FIFO queue until they can be serviced. We simplified the network operation somewhat by ignoring timeouts.

3.1.3. *Efficiency*

The main goal is to minimize the average latency of the requests, which is defined as the average time from a request until the arrival of (the last packet of) the document.

3.2. *Evolving caching strategies*

3.2.1. *Caching as deletion*

It is easy to see that servers which have not yet filled up their memories should store every single document they see. Even if an oracle was available to tell that a particular object will never be accessed again, there would be no harm in storing it. The problem comes when the caching memory fills up, and a storage action requires the deletion of some other previously cached object. Internet hosts have finite memory and disk space and so they must eventually discard old copies to make room for new requests.

In order to evolve caching strategies we thus focused on the problem of deletion. The nodes in our simulated networks store all the documents they receive, until their caches are filled up. If, however, there is not enough memory available to fit the next incoming object, some space must be freed. In order to do that, all cached objects are first sorted, according to a priority function. Then, the document with the lowest priority is trashed. The operation is repeated until enough documents have been deleted so that there is enough space to save the newcomer. In the remainder of this paper we shall define *caching strategy* as the priority function used for deletion.

3.2.2. *Genetic programming*

Given the difficulties to define a restricted search space for caching strategies, we decided to use genetic programming (GP) [18,19], which allows an open-ended search of a space of more or less arbitrary priority functions.

GP is a well-established technique within the realm of Evolutionary Computation (which includes Genetic Algorithms (GAs) and other related methods). As with other evolutionary algorithms, GP solvers maintain an "evolving" set of candidate solutions. The candidates are evaluated with respect to a certain metric (latency in our case). A new set of candidate solutions ("next generation") is then built iteratively by randomly picking some candidates from the previous generation—with the better ones having higher probability of being selected—and introducing new variations by altering ("mutation") and re-combining ("crossover") the previous ones.

The main difference between GP and other evolutionary algorithms is that in GP, candidate solutions are mathematical expressions built from a set of *functions* of one or more parameters, and a set of *terminals* (see Fig. 4 for an example). A *Mutation* operation introduces variations by replacing a subexpression with a different random subexpression, and a *Crossover* operation introduces variations by combining two parents, replacing a subexpression from one of the parents with a subexpression taken from the other one.

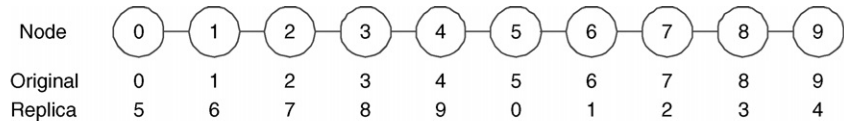


Fig. 3. Optimal distribution of replicas on a simple linear network.

```
(* (* b (+ (+ (* b (+ (+ a (- 0.994 b)) (* b (+ (* b e) (exp
(exp e)))))) (* d e)) (exp e))) (iflte (iflte (+ (exp f) (*
(iflte 0.694 f d a) (- b b))) (exp (exp (* (* (+ a f) f) (exp
c)))) (iflte b (* c (- 0.139 (* 1.616 a))) 0.444 (iflte e
(iflte d a 0.601 (- f (- a e))) c f)) b) e 0.507 (exp (* c
(+ (+ (exp d) d) (* a b))))))
```

Fig. 4. Caching strategy for the linear network (in LISP form for brevity), denoted as BESTGP in the paper. The terminals *a* through *f* are defined in Table 2.

Table 1
Functions used for GP

Function	Meaning	Function	Meaning
add(a,b)	$a + b$	sin(a)	$\sin(a)$
sub(a,b)	$a - b$	cos(a)	$\cos(a)$
mul(a,b)	$a \times b$	exp(a)	$e^{a}, a \in [-100,100]$
div(a,b)	$\begin{cases} \frac{a}{b} : (b \neq 0) \\ 1 : (b = 0) \end{cases}$	iflte(a, b, c, d)	$\begin{cases} c : (a < b) \\ d : (a \geq b) \end{cases}$

Table 2
Terminals used for GP

Variable	Long name	Meaning
A	TimeCreated	Time when replica was stored in cache
B	Size	Size of document in kilobytes
C	AccessCount	Number of times the document has been accessed
D	Last time accessed	Time of last access to document
E	Distance	Distance from node that sent the document (in number of hops)
F	Frequency	Observed frequency of access (in accesses per second)
$\mathfrak{R}$		Random constant

In order to explore the space of caching policies, we employed a generic set of GP functions (Table 1). The set of terminals represents the local information available about each document (Table 2). Each node computes this *observable* information about the objects as they are sent, received and forwarded. One of the observables, for example, is the number of hops a document traveled before reaching the host (distance from sender). Other measures of distance, such as ''distance to nearest replica'' were not used because they could only be determined through additional communication.

Another important decision was to avoid metrics that need to be recomputed before each usage. This allows a host to maintain its cache sorted rather than re-computing and re-sorting before each deletion. Measures such as ''time elapsed since last usage'', or ''access frequency'' change with every tick of the clock, for all cached objects. They would force re-computing each time they need to be used. Instead we have used ''time of last usage'' and ''inverse mean time between accesses'' which do not change their value over time, unless the object itself is accessed again (see [14]).

Since our focus was more on the application than the search for optimal parameter settings, we used rather standard settings for the test runs reported below: GP has been run with a population size of 60 for 100 generations, the initial population has been generated randomly with depth of at most 6. Tournament selection [20] with tournament size of 2 was used, and a new population was generated in the following way:

- The best 1/3 of the individuals were simply transferred to the next generation (*elite*).
- 1/3 of the individuals were generated by crossover of parents selected by tournament selection.
- 1/3 of the individuals were generated by mutation of parents selected by tournament selection.

Crossover was the usual swapping of sub-trees, mutation replaced a sub-tree by a new random tree.

3.3. Evaluating caching strategies

Evaluating caching strategies analytically is difficult. Instead, we tested them, using the simulation environment described in Section 3.1. There are two possible scenarios: if the network topology, and the location and request patterns of all documents are known, GP can be used to tailor a caching strategy exactly to the situation at hand. Evaluation is deterministic, as we can simulate the environment exactly and simply test the performance of a particular caching strategy in that environment.

On the other hand, such topology and patterns may be known only approximately (e.g., ''the document request frequencies follow a scale-free distribution''). We would like to evolve caching strategies that perform well over a whole class of possible scenarios. In other words, we are looking for a solution that applies in general to all networks within the range of characteristics we expect to find in the real world. Since it is impossible to test a caching strategy against all possible networks, we defined classes of scenarios and tested each caching strategy against a random representative from that class. Of course, that made the evaluation function stochastic. Following observations from other evolutionary algorithms used for searching robust solutions [21], we decided to evaluate all individuals within one generation by the same network/request pattern, changing the scenario from generation to generation. Naturally, elite individuals transferred from one generation to the next have to be re-evaluated.

Note that using a simulation for evaluation is rather time consuming. A single simulation run for the larger networks used in our experiments takes about 6 min. Thus, even though we used a cluster of 6 Linux workstations with clock rates of 2 GHz each, and even though we used a relatively small population size and only 100 generations, a typical GP run took about 5 days.

4. Results

4.1. Linear networks

First, we tested our approach by evolving strategies for linear networks, for which we were able to determine the optimal placement of replicas by means of complete enumeration.

The first test involved a purely linear network with 10 nodes connected in a line (i.e. the first and last node have exactly one neighbor, all other nodes have exactly two neighbors). Every node has one original document, each document has the same size, and each node has excess memory to store exactly one additional document. The request pattern is uniform, i.e. each document is requested equally often (on average). For such a

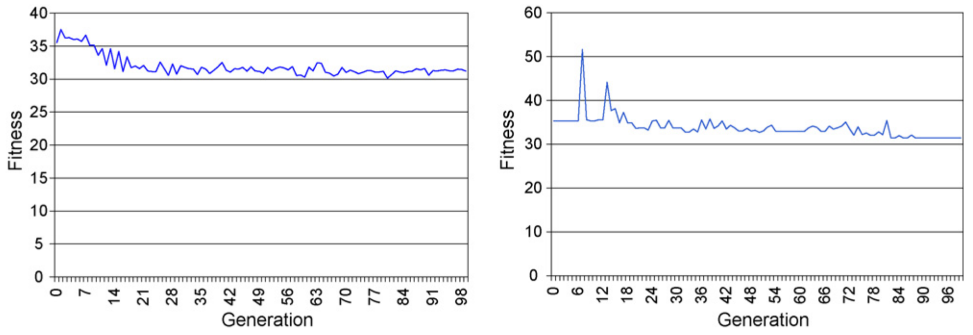


Fig. 5. Sample GP run on the linear network problem. (a) Fitness of best individual per generation. (b) Performance of best individual on 10 other request patterns.

Table 3
Average latency of different caching strategies on the linear network ± S.E.

Caching Strategy	∅ latency
OPTIMAL	31.58 ± 0.03
BESTGP	31.98 ± 0.06
GDSF	47.67 ± 1.17
DISTANCE	50.40 ± 1.24
RANDOM	61.65 ± 0.14
LRU	74.77 ± 0.19

simple setting, the optimal placement of replicas can be determined by complete enumeration and is depicted in Fig. 3.

A typical test run is shown in Fig. 5, where part (a) shows the observed average latency of the best individual in each generation, as observed by GP, and part (b) shows the performance of these individuals on 10 additional random request patterns to test how the solution generalizes. The caching strategy evolved is rather complex and difficult to understand (see Fig. 4). Its performance, however, was excellent.

Table 3 compares the latency of different caching strategies over 30 additional tests with different random request patterns. Note that GP is a randomized method, and starting with different seeds is likely to result in different caching strategies. Therefore, we ran GP 5 times, and used the average performance over the resulting 5 caching strategies as result for each test case. OPTIMAL is the latency observed with the optimal distribution of replicas on this linear network, which is a lower bound. As can be seen, the strategy evolved by GP (BESTGP) was able to find a solution that performs very close to the lower bound. Other standard caching strategies such as GDSF or LRU, perform poorly, LRU even worse than RANDOM[1], where it is randomly decided whether to keep or delete a particular document. Looking at distance only (DISTANCE) is better than looking at the last time accessed only (LRU), but also much worse than the evolved strategy.

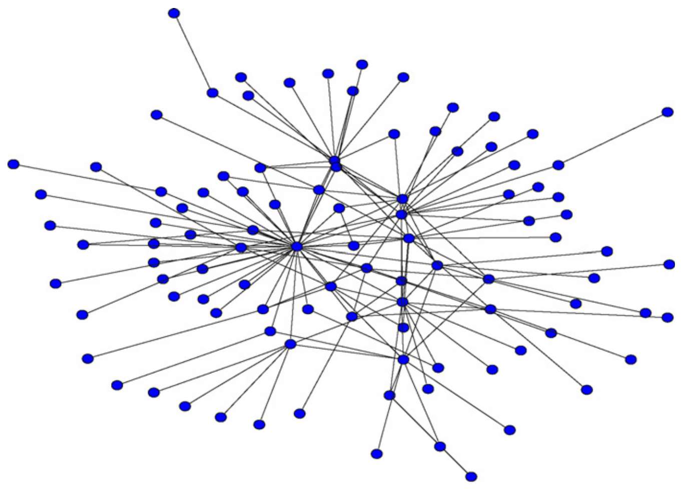


Fig. 6. Sample random scale-free network with 100 nodes, as generated with our network generator and laid out for visualization using a spring algorithm [28].

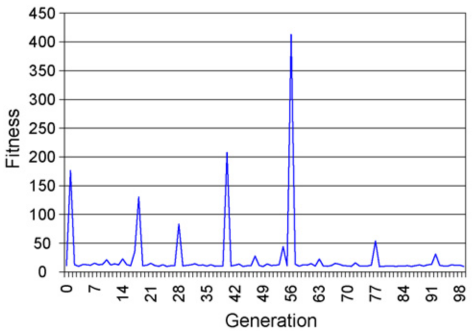
4.2. Scale-free networks

The physical layer of the Internet, composed of hosts, routers, physical links and wireless links, has a scale-free structure, with a few highly connected servers, and many servers with just a few connections [22–24]. The tests in this subsection are obtained using a scale-free network with 100 nodes. We used the generalized linear preference (GLP) growth model described in Bu and Towsley [25] to generate random scale-free networks with the same characteristics as the Internet (Fig. 6 shows an example).

There are 100 original documents scattered uniformly among the nodes in the network, with document size uniformly distributed between 0.1 and 2 MB. In each of 1000 simulated seconds, an average of 1100 requests are generated in a simulated Poisson process. Request frequencies for each document follow a random power-law distribution as well [26], so that some documents are requested much more often than others. All nodes and links are assumed to be of identical capacities in terms of storage size and communication bandwidth.

With the characteristics of the network defined only as distributions, we searched for individual strategies that could perform well over a wide range of networks in the class. As

[1] A random strategy, in spite of being completely blind, has the advantage of breaking symmetry (cf. page 4) because all hosts use different random number seeds, thus their caches tend to complement each other.



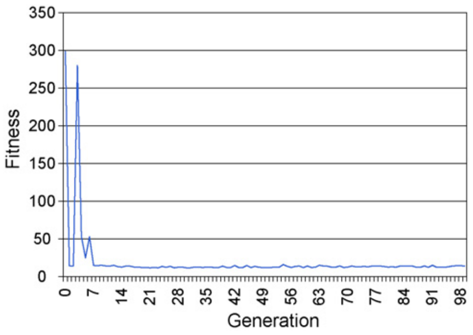


Fig. 7. Exemplary GP-run on the class of scale-free networks. (a) Fitness of best individual per generation. (b) Performance of best individual on 20 other instances.

explained in Section 3.3, this was achieved by using a different random network and request pattern in each generation.

Our test case generator produced a different combination of network topology, object sizes, and object request frequency each time. The resulting problems ranged from ‘‘easy’’ cases—where even without caching, the network bandwidth was sufficient to service requests with minimal latency—to ‘‘impossible’’ cases—where even the best caching strategy was overwhelmed by too many requests, leading to huge waiting times.

As predicted by basic queuing theory, a link can be either *underutilized* (bandwidth > request load) or *overutilized* (bandwidth < request load). In the first case, latency remains close to zero, and in the second it grows to infinity. In order to provide for an adequate level of challenge for caching strategies, we have deliberately set the parameters in our simulation so that they generate networks around to the saturation point. The interesting cases, from the point of view of caching, are those which are neither ‘‘impossible’’ nor ‘‘easy’’—thus providing an opportunity for a good strategy to make a difference.

This high variance between scenarios became a difficulty for our GP algorithm. We decided to evaluate each individual three times per generation, on three different random networks, and use the average as fitness for selection. Even then, the randomness was substantial, as can be seen in the oscillations in performance of the best solution in Fig. 7 (again, part (a) shows the fitness of the best individual as observed by GP, while part (b) shows the performance of that individual on 20 additional tests).

Since the fitness is a random variable, the best solution in the final generation is not necessarily the truly best solution discovered during the run. Furthermore, most of the strategies generated were rather complicated (as the one in Fig. 4). In the following, we will focus on the performance of *two* strategies: the best strategy of the final generation (denoted **GPfinal**), and a strategy that we found particularly appealing because—besides a promising fitness—it was very simple. It could be succinctly expressed as:

$$\text{Priority} = \text{last time accessed} \times (\text{distance} + \text{access count})$$

The excellent performance of this strategy seems odd at a first glance, as it adds quantities from different units, namely the distance the document has traveled and the number of times it has been accessed. But the combination has meaning: it keeps the documents that have either a very high access count, or a very high distance, but only if they have been accessed recently (otherwise last Time Accessed would be small). We call this evolved strategy **RUDF** for ‘‘Recently Used Distance plus Frequency’’.

Again, we used 30 test runs to evaluate the strategies, with a new random network and request pattern generated for each test instance. Due to the large variance between test instances, the mean latency is very dependent on some outliers and a comparison based on mean latency has only limited explanatory power. We therefore focus primarily a non-parametric measure, namely the rank the caching strategy achieved when compared to the other strategies tested (with 1 being the best rank, and 6 being the worst rank), averaged over all test cases.

The results depicted in Table 4 show that both evolved strategies (GPfinal and RUDF) work very well over the randomly generated range of networks in the considered class of scale-free networks. They significantly outperformed all the other strategies tested, with a slight advantage of RUDF over GPfinal. GDSF is the best conventional strategy with respect to average rank, but worse than DISTANCE with respect to latency, which means that it produces better results in the majority of cases, but if it performs worse, the performance is much worse.

In order to test how the evolved strategies would perform on completely different networks, we additionally tested them on the linear network (Section 4.1) from above, and on two scale-free network classes with 30 and 300 nodes, respectively.

RUDF demonstrated to be quite robust to such changes from the original environment. On the linear network, it resulted in a

Table 4
Comparison of different caching strategies on the class of scale-free networks with 100 nodes

Caching Strategy	∅ Latency	∅ Rank
RUDF	3671	1.53 ± 0.12
GPfinal	6373	1.70 ± 0.11
GDSF	24414	3.80 ± 0.15
DISTANCE	8473	4.00 ± 0.30
LRU	25390	4.70 ± 0.16
RANDOM	24903	5.27 ± 0.14

Table shows latency and rank (1=best, 6=worst) averaged over 30 runs ± S.E.

Table 5
Comparison of different caching strategies on the class of scale-free networks with 30 nodes

Caching strategy	∅ Latency	∅ Rank
RUDF	94.1	1.03 ± 0.03
GDSF	100.0	2.20 ± 0.09
GPfinal	101.1	2.76 ± 0.11
LRU	103.7	3.93 ± 0.06
RANDOM	114.7	5.20 ± 0.07
DISTANCE	118.7	5.70 ± 0.10

Table shows average latency and rank ± S.E.

Table 6
Comparison of different caching strategies on the class of scale-free networks with 300 nodes

Caching strategy	∅ Latency	∅ Rank
RUDF	66786	1.10 ± 0.05
GPfinal	93055	2.17 ± 0.10
DISTANCE	118104	3.00 ± 0.19
GDSF	150109	4.20 ± 0.11
RANDOM	157801	5.20 ± 0.15
LRU	151731	5.33 ± 0.13

Table shows average latency and rank ± S.E.

latency of 35.8 ± 0.42, i.e. worse than the caching strategy evolved particularly for the linear network, but still better than all other tested strategies. The results on the two scale-free network classes are shown in Tables 5 and 6. As can be seen, RUDF significantly outperforms all other strategies independent of the network size.

While GPfinal was very competitive on the 100-node networks assumed during evolution, it is much less robust than RUDF. Its average rank worsened from 1.7 on the 100-node network to 2.17 on the 300 networks and 2.2 on the 30-node networks, making it third out of the 6 strategies tested. On the linear network, it failed completely with a latency of 117.0 and a very large standard error of 11.0. This indicates that perhaps the evolved rules were already overfitted to a particular type of network, and that the simplicity of RUDF allows it to generalize better to other networks. Note that the unnecessary complexity of the evolved rules (called "bloat") is a well-known problem in GP. We have not attempted to use bloat-prevention strategies other than limiting the allowed depth of expressions (maximum depth was set to 20) and always preferring, between two solutions with identical performance, the one that has a smaller number of nodes [27].

The performance of DISTANCE seems to be very dependent on the network size; it scores second on large networks, but worse than RANDOM on smaller ones.

4.3. Using metadata

So far, we have assumed that each node has information on the request patterns of the documents stored in its cache. Now we want to examine how performance could be improved by maintaining metadata on the request patterns of all documents seen by a router. In other words, when a document is deleted from the cache, the information collected about it (size, last time accessed, access frequency and so on), called *metadata*, is kept in a small record so it is available for use in future caching decisions.

Table 7
Comparison of different caching strategies on the class of scale-free networks with 100 nodes, depending on whether metadata is maintained only for cached documents (standard) or all documents (all metadata)

Caching strategy	∅ Latency	∅ Rank
RUDF	1191.7 ± 27.1	1.33 ± 0.13
GPMD w/metadata	1340.8 ± 18.7	2.13 ± 0.15
GDSF w/metadata	6206.8 ± 80.2	3.0 ± 0.16
RUDF w/metadata	9470.8 ± 115.5	4.27 ± 0.15
GPMD	3162.8 ± 38.4	4.3 ± 0.17
GDSF	18060.5 ± 135.6	5.96 ± 0.03

Table shows average rank according to latency ± S.E.

Table 7 shows the results. As one might expect, GDSF benefits from the more accurate information contained in the metadata, and moves from the worst rank (5.96) to an average rank of 3.0. For RUDF, which was evolved without extra metadata memory, adding that information not only does not help, but in fact significantly reduces performance (rank 1.33 without metadata, rank 4.27 with metadata). The reduced performance of RUDF might be due to the loss of appropriate balance between the *distance* and *accessCount* variables: with too much back memory, access counts can grow to be orders of magnitude larger than distance, thus becoming the single dominant factor in the caching decision.

The rule denoted as GPMD was evolved using GP with all metadata available during the simulations. It is the best performing rule when metadata is provided. If metadata is not available, it is still better than GDSF, but surpassed by RUDF, which was evolved without metadata.

Note that GPMD has a comparably low standard error in latency. Although it is often slightly worse than the other methods if metadata is not available, it does not suffer from occasional very bad outliers and therefore has a relatively good average latency despite the relatively bad average rank. Interestingly, RUDF without metadata performs better than GPMD with metadata. This may be due to the uncertainty in the fitness function and the randomness in GP, which may lead to varying solution qualities from one run to another. Independent of whether metadata is used or not, the corresponding evolved solutions performed better than GDSF, demonstrating the suitability of the GP approach.

5. Conclusions

A key inefficiency of the Internet is its tendency to retransmit a single blob of data millions of times over identical trunk routes. Web caches are an attempt to reduce this waste by storing replicas of recently accessed documents at suitable locations. Caching reduces network traffic as well as experienced latency.

The challenge is to design caching strategies which, when applied locally in every network router, exhibit a good performance from a global point of view. One of the appeals of GP is that it can be used to explore a space of algorithms. Here, we have used GP to search for caching strategies in networks that resemble the Internet, with the aim to find strategies that minimize latency. We have shown that GP is able to successfully generate near-optimal caching strategies on simple linear networks. For scale-free networks, a new rule called RUDF was evolved, which is very simple yet outperformed all other tested caching strategies on all the scenarios examined, thus setting a new benchmark.

An additional advantage of the GP approach described here is that caching strategies can be tailored to the general characteristics of the network itself. We evolved efficient rules specifically for linear networks, then scale-free networks with power-law request frequency distributions, finally the same networks but when the network nodes are allowed to hold on to information or ‘‘metadata’’ about objects seen a long time ago.

An important obstacle we faced was measuring fitness, because fitness could only be determined indirectly through simulation, and different random seeds resulted in a high variance in latency (the criterion we used as fitness). Nevertheless, averaging and multiple-seed evaluation techniques allowed us to evolve robust strategies that are efficient in a wide variety of conditions.

Currently, we are extending the presented work in several directions. First, we are testing the newly evolved caching strategy on a larger variety of conditions, including heterogeneous networks with links and hosts of different characteristics.

Then, the caching strategies could be made dependent on node characteristics (e.g., location in network, number of connections, and so on), moving away from the assumption that all nodes should apply identical caching strategies.

Finally, an intriguing idea is to have independent GPs running on every node, so that each node evolves its own adaptive caching rule, creating an online system of coevolving caching strategies.

References

[1] X. Tang, S.T. Chanson, Coordinated en-route web caching, IEEE Transact. Computers 51 (6) (2002) 595–607.

[2] K. Li, H. Shen, Optimal methods for object placement in en-route web caching for tree networks and autonomous systems, in: International Workshop on Grid and Cooperative Computing, vol. 3033 of LNCS, Springer, 2004, pp. 263–270

[3] A.J. Smith, Cache memories, ACM Computing Surveys 14 (3) (1982) 473–530.

[4] J. Wang, A survey of web caching schemes for the internet, ACM SIGCOMM Computer Comm. Rev. 29 (5) (2001) 36–46.

[5] B.D. Davison, A web caching primer, IEEE Internet Computing 5 (4) (2001) 38–45.

[6] S. Podlipnig, L. Boszormenyi, A survey of web caching replacement strategies, ACM Computing Surveys 35 (4) (2003) 374–398.

[7] M. Karlsson, C. Karamanolis, M. Mahalingam, A framework for evaluating replica placement algorithms. Tech. Rep. HPL-2002-219, Hewlett-Packard, 2002.

[8] T. Loukopoulos, I. Ahmad, Static and adaptive data replication algorithms for fast information access in large distributed systems, in: International Conference on Distributed Computing Systems, IEEE, 2000, pp. 385–392.

[9] S. Sen, File placement over a network using simulated annealing, in: Symposium on Applied Computing. ACM, 1994, pp. 251–255.

[10] G. Pierre, M.V. Teen, A. Tanenbaum, Dynamically selecting optimal distribution strategies for web documents, IEEE Transact. Comput. 51 (6) (2002) 637–651.

[11] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, E.A. Fox, Removal policies in network caches for World-Wide Web documents, in: SIGCOMM’96: Conference proceedings on applications, technologies, architectures, and protocols for computer communications, Palo Alto, California, United States, ACM Press, New York, NY, USA, 1996, pp. 293–305, ISBN 0-89791-790-1.

[12] L. Cherkasova, G. Ciardo, Role of aging, frequency, and size in web cache replacement policies, in: B. Hertzberger, A. Hoekstra, R. Williams (Eds.), High-Performance Computing and Networking, vol. 2110 of LNCS, Springer, 2001, pp. 114–123.

[13] S. Jin, A. Bestavros, Greedydual* web caching algorithm, Computer Commun. 24 (2) (2001) 174–183.

[14] H. Bahn, S.H. Noh, S.L. Min, K. Koh, Efficient replacement of nonuniform objects in web caches, IEEE Comput. 35 (6) (2002) 65–73.

[15] N. Paterson, M. Livesey, Evolving caching algorithms in C by GP, in: Genetic Programming Conference. Morgan Kaufmann, 1997, pp. 262–267.

[16] M. O’Neill, C. Ryan, Automatic generation of caching algorithms, in: Evolutionary Algorithms in Engineering and Computer Science, John Wiley & Sons, 1999 pp.127–134.

[17] J. Branke, P. Funes, F. Thiele, Evolving en-route caching strategies for the Internet, in: K. Deb, et al., (Eds.), Genetic and Evolutionary Computation Conference, vol. 3103 of LNCS, Springer, 2004, pp. 434–446.

[18] J. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection., MIT Press, Cambridge, 1992.

[19] J. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press, Cambridge, 1994.

[20] D.E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: G.J.E. Rawlins (Ed.), Foundations of Genetic Algorithms, Morgan Kaufmann, San Mateo, California, USA, 1991, pp. 69–93.

[21] J. Branke, Reducing the sampling variance when searching for robust solutions, in: L.S. et al. (Ed.), Genetic and Evolutionary Computation Conference (GECCO’01), Morgan Kaufmann, 2001, pp. 235–242.

[22] M. Faloutsos, P. Faloutsos, C. Faloutsos. On power-law relationships of the internet topology, in: Proceedings of the ACM SIGCOMM. ACM, 1999, pp. 251–262.

[23] S. Yook, H. Jeong, A. Barabasi, Modeling the Internet’s large-scale topology, in: Proceedings of the National Academy of Sciences 99 21 (2002) 13382–13386.

[24] A. Barabasi, R. Albert, H. Heong, Scale-free characteristics of random networks: the topology of the world-wide web, Physica A 281 (2000) 2115.

[25] T. Bu, D. Towsley, On distinguishing between internet power law topology generators, in: Proceedings of INFOCOM, 2002, IEEE, 2002, pp. 638–647.

[26] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: Evidence and implications, in: INFOCOM’99, IEEE, 1999, pp. 126–134.

[27] E.D. De Jong, R.A. Watson, J.B. Pollack, Reducing bloat and promoting diversity using multi-objective methods, in: L.S. et al. (Ed.), Genetic and Evolutionary Computation Conference. Morgan Kaufmann, 2001, pp. 11–18.

[28] T. Kamada, S. Kawai, An algorithm for drawing general undirected graphs, Information Processing Lett. 31 (1) (1989) 7–15.