Cache Placement Methods Based on Client Demand Clustering

Paul Barford , Jin-Yi Cai, Jim Gast

Abstract— **One of the principal motivations for content delivery networks, such as those that have been deployed over the past three years, is to improve performance from the perspective of the client. Ideally, this is achieved by placing caches close to groups of clients and then routing client requests to the nearest cache. In the first part of this paper we present a new method for identifying regions of client demand. Our method uses best path information from Border Gateway Protocol (BGP) routing tables to create a hierarchical clustering of autonomous systems (AS's). The method iteratively adds small clusters to larger clusters based on minimizing the Hamming distance between the neighbor sets of the clusters. This method results in a forest of AS trees where we define each tree root as an Internet backbone node. This forest representation of AS connectivity is an idealization of the Internet's true structure. We test for fidelity by comparing AS hop distances to the Internet backbone. One of the strengths of our AS clustering method is that it naturally lends itself to the cache placement problem. In the second part of this paper, we present two cache placement algorithms based on a tree graph of demand. The algorithms address the problems of placing single caches and multiple caches so as to minimize inter-AS traffic and client response time. We evaluate the effectiveness of our cache placement algorithms using Web server logs and show that they can greatly improve performance over random cache placement.**

Keywords— **Cache Placement, Hierarchical Clustering, Demand Analysis**

I. Introduction

Content Delivery Networks (CDNs) distribute caches in the Internet as a means for reducing load on Web servers, reducing network load for Internet Service Providers (ISPs) and improving performance for clients. In order to effectively deploy and manage cache and network resources, CDNs must be able to accurately identify areas of client demand. One means for doing this is by *clustering* clients that are topologically close to each other, and then placing caches in the areas where demand is typically large. This raises two immediate questions: how can clusters of clients be generated and once identified, how can caches be placed among the clusters so as to maximize their impact?

In this paper, we address the question of client clustering by presenting a new method that generates a hierarchy of client clusters. As opposed to recent work on IP client clustering described in [1], our method uses autonomous systems (AS's) as the basic cluster unit. We argue that clustering at the IP level results in cluster units which are too detailed, and too numerous and thus do not readily lend themselves to higher levels of aggregation. In contrast, clustering at the AS level provides a natural means for not only identifying clients which should experience similar performance from a given cache but also for aggregating AS's into larger groups which should experience similar performance.

Our interest in the ability to aggregate AS's stems from the desire to clearly understand demand and to effectively distribute caches. Our clustering method enables groups of AS's to be coalesced into larger groups based on best path connectivity extracted from BGP routing tables. We use best paths because these are typically the preferred route between an AS and its immediate neighbors. The difficulty is that best paths do not indicate anything about quality of a connection beyond immediate neighbors.

We address this problem by introducing notion of Hamming distance between a pair of connected AS's. Hamming distance was introduced in [2] as the minimum number of elements which must be changed to move from one string to another. For example, the Hamming distance between $\{1,3,5,7\}$ and $\{1,2,3,4\}$ is four because $\{2,4,5,7\}$ appear in one but not both of the sets. In our context, Hamming distance is applied as a measure of similarity of AS connectivity. Specifically, two nodes with a *short* Hamming distance indicate that they have many neighbors in common and are thus candidates for merging into a cluster.

Our clustering algorithm removes edges from the AS graph until all that remains in a forest of trees. The benefits of making a forest are: (1) objectively identifying a small number of vertices that can be treated as the backbone of the Internet and (2) assigning each AS to one and only one tree so that tractable algorithms can be used to predict the paths packets will take going to or coming from the backbone. It is implicitly assumed that the backbone vertices are tightly interconnected (ideally, a clique) and that packet transfers between backbone vertices are very fast.

Our algorithm starts by coalescing nodes whose path to the backbone is uncontested, forming small clusters of nodes whose only known path to the bulk of the Internet passes through a common parent. In the BGP tables we examined, clusters were seldom that obvious. In order to form larger clusters, the algorithm successively relaxes the Hamming distance requirements for clustering. If we relax the Hamming distance requirements too far we would eventually collapse the entire network to a tree with a single root node. Our intention, however, is to only collapse the topology to a size which readily enables evaluation of demand and facilitates our cache placement algorithms. The result of our clustering algorithm presented in this paper is a forest of 21 root AS trees. These root AS's consist of many of the major ISPs such as BBNPlanet and AT&T, but also some smaller ISPs such as LINX due to the nature of the algorithm. The root AS's connect on average with 7.29 other root AS's indicating a high level of connectivity between these nodes. The average out-degree of the root AS's (i.e.. the number of AS with whom they peer) is 198 with a median of 97 indicating that the root AS's facilitate Internet access to a large number of other AS's.

It is also important that the forest minimizes that amount by

Paul Barford, University of Wisconsin - Madison. E-mail:pb@cs.wisc.edu
Jin-Yi Cai, University of Wisconsin - Madison. E-Mail:jyc@cs.wisc.edu. Research supported in part by NSF CCR9820806 and a Guggenheim Fellowship
Jim Gast, University of Wisconsin - Madison. E-Mail: jgast@cs.wisc.edu. Research supported by the Anthony C. Klug fellowship in Computer Science

which it overstates the path lengths between vertices in the original graph. To test that we measured paths in terms of AS hops. In the original graph, the average number of AS hops to those 21 tree roots is 1.61. The average tree depth in our graph is 1.96. This gave us confidence that our forest does not misrepresent AS hop distance significantly. These characteristics indicate that while a forest is an idealization of the actual AS topology, it does not abstract away essential details.

One domain in which our tree hierarchy of AS's naturally lends itself is cache placement. Since our tree generation algorithm is based on best path information from BGP tables, it enables caches to be placed on AS hop paths which would actually be used in the Internet. This study assumes that placing a cache in a AS is sufficient to satisfy all demand from that AS (as well as the AS's children which are part of its cluster). We make this assumption based on the idea that most performance problems occur *across* AS boundaries and that performance *within* an AS is generally good. Our analysis of cache placement effectiveness focuses on the reduction of inter-domain traffic. There is clearly and additional benefit of improving client performance which is a simple extension of our work.

Placement of caches in trees has been treated as a dynamic programming problem in [3] however the means by which trees were created was not treated in that work. We address the issue of optimal cache placement by describing a dynamic programming algorithm in which each subtree calculates the optimal use for 0 to ℓ caches in its subtree. Each parent node can then discover the maximum benefit from ℓ caches by distributing all of the caches among its children or by retaining one cache for itself. We also present a greedy algorithm which iteratively chooses the AS with largest unsatisfied demand as the next site to place a cache.

We evaluate the effectiveness of these two algorithms by comparing their total cost of traffic when 0 to 50 caches are placed. We find that optimal placement of a small number of caches does measurably better than random placement, but that greedy placement performs surprisingly close to optimal when more caches are deployed.

The remainder of this paper is organized as follows: Section II discusses related work; Section III describes our process for constructing client clusters using BGP routing data; and Section IV describes the results of evaluating client demand from a Web log using our clustering results. In Section V we present our algorithms for optimally placing caches based on client demand distribution, and in Section VI we demonstrate the effectiveness of our cache placement methods. In Section VII, we summarize our results and conclude with directions for future study.

II. Related Work

Our clustering algorithm is analogous to generating a spanning tree for the AS graph. Prim's algorithm [4] is a standard method for constructing a minimum spanning tree if the root of the tree is known in advance. Starting at the root, use a breadth-first search to find all nodes. Each edge that lies on a shortest path from the root to any other node is a member of the minimum spanning tree. We cannot use Prim's algorithm since our graph does not have a pre-defined root. Kruskal's algorithm [5] does not require a starting point. It constructs a spanning forest that initially contains a tiny tree for each vertex. Trees are then combined by coalescing them at the shortest edges first. Any edge that does not cross between trees is redundant and any edge left over after all of the vertices have been visited are similarly not needed. Although Kruskal's algorithm serves as the inspiration for our algorithm, we still had to address the stopping criteria, since declaring a single root for the entire Internet would have artificially added several hops in the core of the Internet, where a dozen of the biggest transit providers are almost completely interconnected. Kruskal's algorithm finds a minimal spanning tree in the sense that the total of the edge lengths is minimized, even if it makes the tree deep. Our goal was subtly different, since we want a tree that has maximum fidelity to the traffic flow in the Internet. In particular, we want a shallow tree so that node representations are not mistakenly far from the backbone. Even outside the core, Kruskal's algorithm produces trees that are inappropriately deep when presented with neighborhoods of completely interconnected vertices.

Initial work on clustering clients and proxy placement was done by Cuñha in [6]. That work described a process of using *traceroute* to generate a tree graph of client accesses (using IP addresses collected from a Web server's logs). Proxies were then placed in the tree using three different algorithms and the effects on reduction of server load and network traffic were evaluated. Our work differs from this in our use of AS level information from BGP routing tables to create a tree which is simpler and more efficient. Our cache placement algorithms differ in that the coarser aggregation allows us to use a method that guarantees optimal placement. The next significant work on client clustering was done by Krishnamurthy and Wang in [1]. In that work, the authors merge the longest prefix entries (i.e.. those with the most detail) from a set of 14 BGP routing tables. This creates a prefix/netmask table of approximately 390K possible clusters. IP addresses from Web server logs are then clustered by finding the longest prefix match in the prefix/netmask table. While this approach generates client clusters which are topologically close and of minimal size, it does not provide for further levels of aggregation of clusters.

Content distribution companies (e.g.. Akamai, Digital Island) and wide area load balancing product vendors (e.g.. Cisco, Foundry and Nortel) also use the notion of client clustering to redirect client requests to distributed caches. These companies use the Domain Name System (DNS) [7] as a means for both determining client location and redirecting requests. The assumption made in DNS-redirection is that clients whose DNS requests come from the same DNS server are topologically close to each other. Initial work in [8] evaluates the performance of redirection schemes that access documents from multiple proxies versus a single proxy and shows that retrieving embedded objects from a single page from different servers is sub-optimal. Subsequent work in [9] indicates that clients and their nameservers are frequently *neither* topologically close *nor* close from the perspective of packet latency. However, Myers et al. show that the ranking of download times of the same three sites from 47 different mirrors was stable [10].

Caching has been widely studied as a means for enhancing performance in the Internet during the 1990's. These studies include cache traffic evaluation [11], [12], replacement algorithm performance [13], [14], cache hierarchy architecture [15], [16] and cache appliance design [17], [18]. A number of recent papers have addressed the issue of proxy placement based on assumptions about the underlying topological structure of the Internet [3], [19], [20]. In [3], Li et al. describe an optimal dynamic programming algorithm for placing multiple proxies in a tree-based topology. Their algorithm is comparable to ours although it is less efficient. It places M proxies in a tree with N nodes and operates in $O(N^3M^2)$ time where as our algorithm operates in $O(NM^2logN)$. Jamin et al. examine a number of proxy placement algorithms under the assumption that the underlying topological structure is not a tree. Their results show quickly diminishing benefits of placing additional mirrors (defined as proxies which service all client requests directed to them) even using sophisticated and computationally intensive techniques. In [20], Qiu et al. also evaluate the effectiveness of a number of graph theoretic proxy placement techniques. They find that proxy placement that considers both distance and request load performs a factor of 2 to 5 better than a random proxy placement. They also find that a greedy algorithm for mirror placement (one which simply iteratively chooses the best node as the site for the next mirror) performs better than a tree based algorithm.

Both router level and inter-domain topology have been studied over the past five years [21], [22], [23], [24], [25]. Our clustering algorithm uses BGP data thus inter-domain topology is most relevant to this work. In [24], Govindan and Reddy characterize inter-domain topology and route stability using BGP routing table information collected over a one year period. In that work the authors describe inter-domain topology in terms of diameter, degree distribution and connectivity characteristics. Inter-domain routing information can be collected from a number of public sites including NLANR [26], Merit [27] and Route Views [28] (our source of routing information). These sites provide BGP tables from looking glass routers located in various places in the Internet and peered with a large number in ISP's.

III. Topologically-guided Clustering

A study of sources and destinations of traffic in the Internet quickly becomes a search for a productive way to summarize large bodies of traffic into meaningful categories. Categorizations based on geography are natural, but they are an increasingly inaccurate representation of the topology of the Internet. Our algorithm discovers the topology of the Internet by reading the **best path** data from BGP routing tables [29]. BGP tables [29] contain a great deal of information about connections beyond the next hop. This enables us to construct an AS graph without having to query every BGP router in the world.

We simplify the graph of AS connectivity into a forest of trees to facilitate our analysis. We found clusters of nodes with high mutual affinity by comparing their neighbor sets. We then iteratively applied the same technique to identify clusters of clusters(super-clusters), and so on until there were only a few, very large clusters left. Our algorithm identified 21 such super-clusters. They form the first level of the forest of trees. As of 2001, a dozen of them are almost completely interconnected. Since the tree representation loses information about cross-links between branches of the tree, it is important that our algorithm minimize the impact on distance calculations using the trees.

Our work extends the IP clustering work done by Krishnamurthy and Wang [1] showing how BGP routing tables can be used to gain 99 percent accuracy in partitioning IP addresses into non-overlapping groups. All IP addresses in a group are topologically close and under common administrative control. Their client clustering paper shows other more involved techniques for gaining even higher accuracy and validating the results.

The basic unit of clustering used by our algorithm is the combination of all of the IP ranges that share a common AS number. Although clustering by AS is less specific than IP clustering, the IP addresses in our clusters share common routings. Without common routing, applications of clusters such as cache placement may not be meaningful.

A. Definitions

The following definitions are used throughout the paper:

- AS_n is a **neighbor** of AS_m if it immediately follows or precedes AS_m in any best path. To simplify the algorithm, AS_n is always added to its own list of neighbors.
- The **set of neighbors** of AS_n is denoted by N_n. The parent of AS_n is $p(n)$, initially 0.
- The **outdegree**, $outdegree(n)$ is the initial $|N_n|$. Although the neighbor set changes during the coalescing of clusters, it is important to note that outdegree always refers to the original outdegree, before any clustering. The outdegree of a cluster defined to be the outdegree of its *exemplar* AS.
- AS_n is said to **dominate** AS_m if $N_n \supset N_m$. In particular,

$$dom(n,m) \equiv (N_m \setminus N_n = \emptyset) \wedge (N_n \setminus N_m \neq \emptyset)$$

- The **Hamming distance** between AS_n and AS_m is the number of neighbors exclusive to only one of them.

$$hdist(n,m) \equiv |N_n \cup N_m| - |N_n \cap N_m|$$

- The **overhang** of AS_n over AS_m is the size of the set of Neighbors of n who are not also Neighbors of m.

$$overhang(n,m) \equiv |N_n \setminus N_m|$$

B. Clustering AS's using BGP routing data

To construct hierarchical trees of AS's we needed to find the best assignment of small clusters (AS's with small out-degree) to larger clusters. For this study, we extracted "best path" data from a routing table acquired from Oregon Route-views [28] dynamically on Feb. 20, 2001. BGP routers typically receive multiple paths to the same destination. The BGP best path algorithm decides which is the best path to install in the IP routing table and to use for forwarding traffic. These paths tend to use a highest-throughput lowest-latency link. Our algorithm has no other means to discover that information directly.

C. Why just use best paths?

Our study includes only best paths, thus some feasible routes are ignored. In particular, routes that connect AS's far from the backbone to other small AS's won't be seen. We investigated using all paths and found that low-bandwidth paths for fault tolerance and historical paths with comparatively low bandwidth made the clustering results volatile. Routing tables from different sources would significantly change the computed clustering.

D. Multiple passes across the graph

Clustering is performed by successive passes through the graph building large clusters by visiting small clusters and merging them into an existing larger cluster.

For each clustering pass, each node, n, without a parent (i.e. $p(n) = 0$) tries to find a suitable parent. Conceptually, the candidate parents are the nodes which dominate it, $C_n = \{m \in N_n | dom(m, n)\}$. In practice, this is too strict a requirement and we will define C_n more correctly below. Now, find the nearest among the candidate parents, $m \in C_n$. The best parent is

$$nearest(n) = \min_{m \in C_n} \{hdist(n, m)\}$$

If $C_n \neq \emptyset$, Node n is merged into the cluster of the best parent, m. Now $p(n)$ is set to m and n is removed from N_m. Note that n is not removed from other neighbor lists, since n might later be chosen as a parent by an even smaller cluster.

An interesting design decision happens in situations where $N_m = N_n$, neither neighbor list is a proper superset of the other and neither dominates. We defined domination in this way so both nodes are free to become siblings under some other parent, keeping the tree comparatively shallow. If n or m had been arbitrarily chosen as parent, the other (and her subtree) would appear to be one AS hop farther from the backbone.

It would also have been meaningful to define the best parent as the farthest candidate parent. This would cause AS's to choose AS's with very high out-degree as their preferred parent. The result would have been a shallower tree that more closely matches the distance to the backbone, but it also would have lost the useful categorization of AS's into clusters with very similar sets of neighbors.

In practice, many AS's connect to more than one major provider. These AS's are not strictly dominated by any one of the nodes they have links to. To relax the domination requirement, a tolerance factor grows with each pass through the nodes without parents. The tolerance, δ, allows a node to become a child of any node with a higher out-degree if the overhang is less than the current tolerance. δ drives the speed at which the clustering completes. So the actual computation for **the set of candidate parents** is:

$$C_n = \left\{ m \in N_n \middle| \begin{array}{c} overhang(n, m) \leq \delta \bigwedge \\ outdegree(m) > outdegree(n) \end{array} \right\}$$

E. Cluster generation example

A simple example demonstrates how the clustering operates in practice. In Figure 1, AS 2, AS 3, and AS 6 are connected to many other nodes. In this example $N_7 = \{4, 7\}$ is dominated by $N_4 = \{4, 5, 7\}$ so $dom(4, 7) = true$. For each pass, each node makes a list of candidate parents. During the first pass, AS 7 coalesces with AS 4. AS 4 is now the *exemplar* for a cluster and AS 7 is removed from N_4 reducing it to $\{4, 5\}$. The parent of AS 7, $p(7)$, is set to 4. Similarly, AS 8 is dominated by AS 5. During the second pass, AS 4 coalesces with AS 5 to form an even bigger cluster with AS 5 as the *exemplar*.

In the third pass, the algorithm has nothing to coalesce, since no node is dominated by any single neighbor. In this case $N_1 = \{1, 2, 3, 5\}$ is not dominated by AS 2, AS 3, or AS 5. Since AS 1 connects to one node (AS 2) missing from the AS 5 list, $overhang(1, 5) = 1$. Similarly, $overhang(5, 1) = 1$ because of AS 6.

In a later pass, the tolerance grows above 1.0 and the candidate parent set of AS 1 becomes $C_1 = \{3, 5\}$. The nearest of these is AS 5, so AS 1 coalesces with AS 5. During the same pass, the candidate parents of AS 5 becomes $C_5 = \{3\}$. Note that AS 1 is not a candidate parent of AS 5 because it originally had a smaller outdegree.

In the example, AS 7 would be denoted as AS3.5.4.7. The name shows the relationship that AS 7 is a child of the progressively larger super-clusters. Clients in AS 7 would benefit (albeit progressively less) from caches on the path to the backbone.

F. Results of AS clustering

For this study a δ tolerance growth of 0.25 per pass was chosen. Figure 2 shows the number of clusters at the end of each pass through the list of AS's. The first four passes cluster all of the easily-classified AS's with small out-degree. Passes five through ten found a large number of national, government, and educational transit AS's. After the pass 37, further reduction in the number of clusters takes much longer. To avoid excess layers at the top of the tree, we stopped the algorithm at pass 40 and declared the 21 remaining *exemplars* to be the roots of the forest of 21 trees.

Figure 2 compares the cumulative distribution of distances to the backbone in both the original full graph and the tree left at the end of clustering. The maximum distance from the backbone was 5 in the full graph but rose to 8 in the forest. There were only 56 nodes in the forest farther than 5 hops from the backbone. This matched our goal for the backbone since over 90 percent of the 6395 nodes are within 2 hops of a backbone node in the graph and within 3 hops of a backbone node in the forest. The average node is 1.61 hops away from the 21 "backbone" nodes in the full graph, and 1.96 hops away from those same 21 nodes in the computed forest.

The resulting clustering contains 21 large trees, each headed by a particular AS. Table III-F shows the names of those Autonomous Systems. The list does not contain some of the AS's with high out-degree. Presumably, this is because they were dominated (at some small tolerance) by an AS that is on the list. Alternet had the largest number of immediate children at 492, a little over half of its out-degree (878) in the full graph. There were 2515 AS's at the second level of the tree, making the average number of children per backbone node 120. The top three levels include a total of 4833 AS's that are within 2 hops of the backbone.

G. AS clustering limitations

BGP routing tables don't show peering relationships that often permit packets to take shortcuts through the Internet. This is because routers will intentionally NOT advertise peers if they do not want to provide transit services for those peers. We have not studied the extent that these relationships improve global traffic statistics.

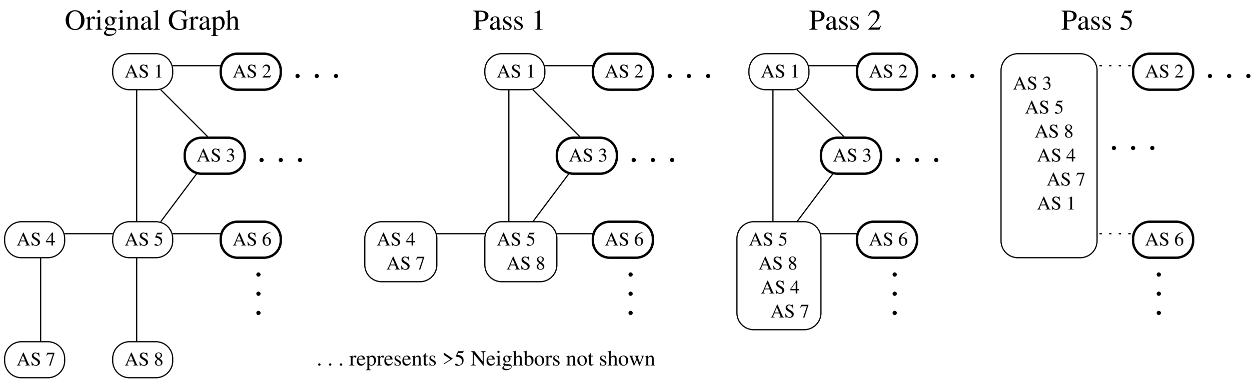
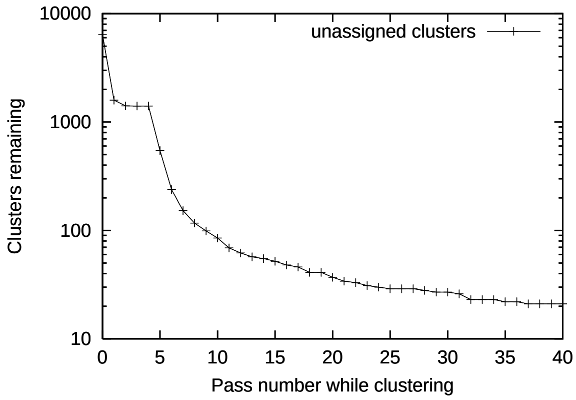


Fig. 1. Walk-through of the clustering algorithm



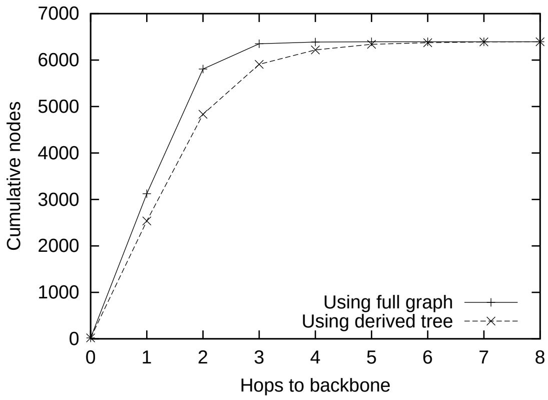


Fig. 2. Results of AS cluster formation. The left graph shows how the number of clusters declines as clusters are coalesced. The right graph shows how the path length in the derived tree compares to the path length in the original graph of best paths.

TABLE I

CLUSTERS IDENTIFIED AS BACKBONE BY THE ALGORITHM. PEERS ARE THE NUMBER OF CONNECTIONS THE EXEMPLAR HAS TO OTHER EXEMPLARS ALSO IN THIS LIST.

Clstr #	Exemplar AS	Members	Out Degree	Peers	Depth
1	2914: Verio	150	235	13	5
2	1: BBNPlanet	171	284	12	4
3	701: Alternet	492	878	12	8
4	7018: AT&T	281	374	11	4
5	2828: Concentric	30	85	9	5
6	3549: Globalcenter	33	60	9	2
7	3561: Cable&Wireless	287	482	9	5
8	6453: Teleglobe	57	124	9	6
9	293: ESnet	41	112	8	5
10	1239: Sprint	407	645	8	5
11	2497: JNIC	45	82	8	6
12	3356: Level3	33	60	8	3
13	209: QWest	83	112	7	4
14	3300: Infonet-Europe	21	40	6	3
15	702: UUNet-Europe	56	80	5	5
16	1221: Telstra	27	61	5	1
17	1755: EBone	59	97	4	6
18	5378: INSNET	32	59	4	8
19	1849: PIPEX	26	47	3	5
20	2548: ICIX	158	189	2	3
21	5459: LINX	26	49	1	4

Other complications can make the AS path less accurate. In RFC 1772 [30], Route Aggregation allows an AS to advertise an aggregate route in which contiguous IP addresses can be collapsed to a single entry. The rules of BGP4 require that the aggregated route contain all of the AS numbers for any portion of the aggregation. This sometimes overstates the length of the AS path. It is also possible to use an atomic aggregate, thus effectively hiding some AS numbers from appearing in the AS path.

Our algorithm also depends on the AS path being a sequence, an ordered list of the AS numbers traversed to deliver a packet to a given IP address range. The BGP4 specification allows an AS path to be an unordered AS set, but requires that it become an AS sequence before it is passed as a advertisement to a neighboring AS. In theory, this means that any BGP4 AS path farther than 1 hop away from its ultimate destination must be an AS sequence and our algorithm assumes this to be true.

Route Views [28] is a standard source for timely, composite BGP information. It collects BGP information from routers widely distributed throughout the Internet. Nonetheless, initial investigation indicates that adding other routing tables would be likely to affected our clustering.

Finally, our algorithm creates a forest that sometimes makes an AS appear farther from the backbone than it really is. This most often occurs because the cluster with the least overhang over a subject cluster is preferred when the subject cluster picks a parent. The average depth of the cluster tree was 1.961, whereas the average number of hops to the backbone in the full graph was 1.595. The right-hand graph in Figure 2 shows how these two metrics compare.

IV. Client Demand Analysis

To map demand into our AS hierarchy, we needed to know the quantity and the composition of client requests that come from each leaf cluster. A simple case is a web server with a single host name. To demonstrate our cache placement techniques, we analyzed a single commercial web server log.

A. Converting IP addresses to AS numbers

The process of converting IP addresses to AS numbers is analogous to the way an IP routers match the longest prefix of the IP address contained in the composite routing table obtained in the prior step. The demand summary [31] for each web server log is a compact file, suitable for sending across the network to a collection point. Each demand summary file contains one line for each AS number that had non-zero requests. The line contains the AS number, the count of successful requests, and the number of bytes in replies.

B. Web server log

For this study, we use a log from a commercial web server collected in February, 2001. The log contained 18 hours of requests that were globally diverse containing 402,955 requests making up 3.69 Gigabytes. There were requests from 791 different autonomous systems. The 50 AS's with the highest demand accounted for 232,991 requests and 2.19 Gb. To avoid complex error scenarios, we filtered out all of the requests except HTTP GET requests with successful result (codes 200 to 203).

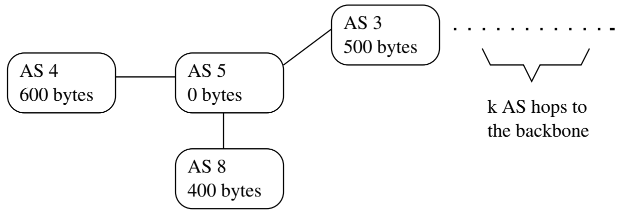


Fig. 4. Tadpole Graph Example

C. Demand Aggregation

Figure 3 shows the aggregate demand from each of the 21 major clusters in both bytes and byte-ASHops. The graphs show that the commercial web server had clients that were concentrated in certain areas of the Internet. The 3 busiest were the clusters whose exemplars were Verio, Alternet, and AT&T with 64 percent of the bytes and 63 percent of the byte-ASHops in replies. The BBNPlanet cluster was particularly interesting because it was also one of the best trees for delivering the test data in the fewest ASHops (2.752 ASHops including 1 for BBNPlanet and 1 for the root). The clusters with averages above 3.5 ASHops were those represented by ESNET, UUNET-Europe, LINX and EBONE.

V. Cache Placement

The result of our clustering algorithm is a forest of trees containing clusters of AS's in increasingly detailed groups. The fundamental assumption is that analysis of a load pattern against this model will yield a useful, objective measure of the value of placing caches into this forest. The problem is similar to that posed by Li, et al. [3], but we simplified it by setting the delivery cost to be the number of Autonomous Systems that the reply entered times the number of bytes in the reply.

To do this, we assign a weight to each leaf node equal to the number of bytes given to it in successful replies. Parent clusters of that leaf are responsible for finding the optimal use of m proxy caches for each value of ℓ up to the total number of proxy caches we can afford to place. Each node can choose to distribute those ℓ caches in any amounts among its children and can choose to keep one for itself. We visualize this as pebbles placed onto the tree wherever a proxy cache is indicated. Our cache placement study assumes that any proxy cache will completely satisfy all requests sent to it. We assume that all requests are sent to web servers on the backbone. The cost of each reply is the number of AS's that see the reply (including the originating AS) multiplied by the size in bytes of the reply. The cost of the requests is ignored.

Figure 4 shows a subtree near the bottom of a large tree. In the absence of caches, the 600 bytes of replies for AS 4 would be seen by $k+3$ systems as they traveled from the backbone. Placing a pebble at AS 4 will satisfy its 600 byte demand locally. If that were the only pebble placed, the other 900 bytes of demand would escape and their cost would be $(500(k+1)+400(k+3))$. So, the total cost of the AS 3 subtree given only a single pebble (and placing it at AS 4) is $600 + (1700 + 900k)$.

From the point of view of AS 3, the cost of the traffic he will be different depending on how many pebbles are used. We will use ℓ to represent the number of pebbles available. If $\ell = 0$, AS

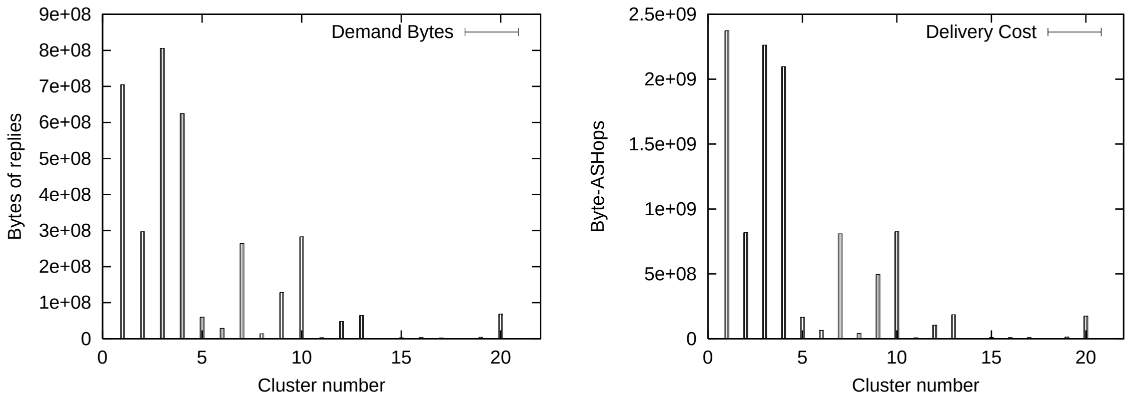


Fig. 3. Demand aggregated to the 21 backbone nodes

3 can place 0 pebbles and its cost is $0+(3500+1500k)$. If AS 3 can place $\ell = 4$ pebbles, the cost of its subtree is 1500, although in this case, the pebble placed at AS 5 is not valuable.

An interesting problem lies in comparing the options AS 3 has if offered only 1 pebble. At $k = 0, \ell = 1$, AS 3 should place the pebble at AS 4 for a total cost of 2300. But at $k = 100$, AS 3 would choose to put the only pebble on AS 3 for a total cost of 3500. Clearly, cost is not a simple function of k.

A. Simultaneous placement algorithm

Given a rooted tree with n vertices. Every leaf v is associated with a non-negative weight $w[v]$. There are m pebbles, where m is at most the number of leaves. Consider any placement of up to m pebbles on any vertices of the tree. A placement of pebbles is called *feasible* if every leaf with a non-zero weight $w[v] > 0$ has an ancestor which has a pebble on it. Here the ancestor relation is the reflexive and transitive closure of the parent relation, in particular every vertex is an ancestor of itself. The cost of any feasible placement P is defined as follows:

$$c(P) = \sum_v c(v),$$

where the sum is over all leaves v, and the cost associated with the leaf v, denoted by $c(v)$, is $(\lambda + 1) \cdot w[v]$, where λ is the distance from v to the closest pebbled ancestor of v. Here the distance between two vertices of the tree is the number of edges on the unique shortest path between them. For technical reasons we define the cost of an infeasible placement to be ∞.

The goal is to find a feasible placement P with at most m pebbles such that $c(P)$ is minimized.

B. Binary tree case

We first consider the case of binary trees, where every vertex has at most two children. Of course it is a leaf when there are no children. Thus for non-leaves, there are either a unique child, or there are two children, in which case we order them as left and right arbitrarily.

For any vertex v of the tree T, denote the subtree rooted at v by T_v. Generically, if v has a unique child then we denote that child by v_1, and if there are two children then we denote them v_1 and v_2 respectively. For $k \geq 0$ we consider a *tadpole graph* (T, k) defined as T appended by a single path extending upwards from the root of T with k extra vertices. Note that $(T, 0) = T$.

For $\ell \geq 0$, We will consider the optimal placement of at most ℓ pebbles in T_v, and denote the minimal cost by $f_v(0, \ell)$. More generally, for $k > 0$ and $\ell \geq 0$, we will consider the optimal placement of one pebble at the tip of the sperm graph (T_v, k) which has distance k from the root v of T_v, and at most ℓ pebbles within T_v. We denote by $f_v(k, \ell)$ the minimal cost $c(P)$ of all feasible pebbling P of (T, k) with at most ℓ pebbles in T_v, and where if $k > 0$ we stipulate that one additional pebble is placed at the tip of the external path from v. If $k = 0$ and $\ell = 0$ then we have a feasible pebbling if and only if all weights in T_v are zero, in which case $f_v(0, 0) = 0$. Note that for any $k, \ell \geq 0$ and $k + \ell \geq 1$, a feasible pebbling exists. For $k = \ell = 0$, and if some non-zero weights exist in T_v, and thus no feasible pebbling exists, we denote $f_v(0, 0) = \infty$.

We will compute $f_v(k, \ell)$ for all $k, \ell \geq 0$, inductively for v according to the height of the subtree T_v, starting with leaves v.

More formally, let L_v be the number of leaves in T_v. Let $d_v = d_v(T)$ be the depth of v in T, i.e., the distance from the root of T to v (note that by our definition of distance, the depth of the root is 0). Let $h(T_v)$ be the height of the tree T_v, which is the maximum depth of all leaves in T_v, i.e., $h(T_v) = \max_u d_u(T_v)$, where u ranges over all leaves in T_v. Note that a tree with a singleton vertex has height 0. Inductively for $0 \leq h \leq h(T)$, starting with $h = 0$, we compute $f_v(k, \ell)$, for all $v \in T$ such that the subtree T_v has $h(T_v) = h$, and for all $0 \leq k \leq d_v$, and for all $0 \leq \ell \leq L_v$.

Base Case $h = 0$:
In the base case $h = 0$ and we are dealing with a singleton leaf, together with an extension of a path of length k if $k > 0$, and no extensions if $k = 0$.

Thus, for $k = 0$,

$$f_v(0, 0) = \begin{cases} 0 & \text{if } w[v] = 0 \\ \infty & \text{otherwise,} \end{cases}$$

and for $\ell = 1$, (note that $h(T_v) = h = 0$ implies that $L_v = 1$),

$$f_v(0, 1) = w[v].$$

Now for $k \geq 1$,

$$f_v(k, 0) = (k + 1) \cdot w[v],$$

and for $\ell = 1$,
$$f_v(k,1) = w[v].$$
Inductive Case $h > 0$:
For the inductive case $h > 0$, we have some v with $h(T_v) = h$, and we assume we have computed all $f_{v'}(k,\ell)$ for children v' of v. There are two cases, v has either one or two children. First we consider v has a unique child v_1. For either $k = 0$ or $k > 0$, we can consider either placing a pebble at v or not placing it there. But we claim that without loss of generality we don't need to place it there. This is because v has only one child, and if an optimal pebbling places a pebble at v, we can obtain at least as good a pebbling by moving the pebble from v to v_1, and if v_1 is already pebbled we can remove one pebble. Thus, we have an optimal pebbling of (T_v, k) using at most ℓ pebbles in T_v without a pebble at v. Hence,
$$f_v(0,\ell) = f_{v'}(0,\ell),$$
and for $k > 0$,
$$f_v(k,\ell) = f_{v'}(k+1,\ell).$$
Suppose now v has two children v_1 and v_2. Basically we must decide how to distribute ℓ pebbles in the subtrees T_{v_1} and T_{v_2} with ℓ_1 and ℓ_2 pebbles each. There is a slight complication as to whether to place a pebble at v, the root of T_v, which affects how many pebbles there are to be distributed, either $\ell_1 + \ell_2 = \ell$ or $\ell - 1$.

First $k = 0$. If we place a pebble at v, (which of course presupposes $\ell > 0$), then there are $\ell_1 + \ell_2 = \ell - 1$ pebbles to be distributed in T_{v_1} and T_{v_2}, but with respect to these two subtrees the "k" values are both 1, i.e., we have $f_{v_1}(1,\ell_1) + f_{v_2}(1,\ell_2)$, minimized over all pairs $\ell_1 + \ell_2 = \ell - 1$. (To be precise, all pairs (ℓ_1, ℓ_2), such that $0 \le \ell_1 \le L_{v_1}$, $0 \le \ell_2 \le L_{v_2}$ and $\ell_1 + \ell_2 = \ell - 1$; but we will not specify this range explicitly in the following.)

If we don't place a pebble at v, then there are $\ell_1 + \ell_2 = \ell$ pebbles to be distributed in T_{v_1} and T_{v_2}, and since $k = 0$ for T_v, with respect to these two subtrees we still have the "k" values 0. So we have $f_{v_1}(0,\ell_1) + f_{v_2}(0,\ell_2)$, minimized over all pairs $\ell_1 + \ell_2 = \ell$.

The optimal cost $f_v(0,\ell)$ is the minimum of these two minimizations, i.e.,
$$f_v(0,\ell) = \min \left\{ \begin{array}{c} \min\limits_{\ell_1+\ell_2=\ell-1} \{f_{v_1}(1,\ell_1) + f_{v_2}(1,\ell_2)\}, \\ \min\limits_{\ell_1+\ell_2=\ell} \{f_{v_1}(0,\ell_1) + f_{v_2}(0,\ell_2)\} \end{array} \right\}.$$
(It is understood that in case $\ell = 0$, the first minimization is vacuous and should be omitted. This is the standard convention, a minimization over an empty set (no non-negative ℓ_i sum to -1) is ∞. Also the second minimization is merely $f_{v_1}(0,0) + f_{v_2}(0,0)$ which is typically ∞ unless all weights in T_v are zero, in which case it is 0.)

We consider the case $k \ge 1$ next. For $\ell = 0$ we have
$$f_v(k,0) = f_{v_1}(k+1,0) + f_{v_2}(k+1,0).$$
Suppose $\ell > 0$. Again we have the possibilities of placing a pebble at v or not. Thus,
$$f_v(k,\ell) = \min \left\{ \begin{array}{c} \min\limits_{\ell_1+\ell_2=\ell-1} \{f_{v_1}(1,\ell_1) + f_{v_2}(1,\ell_2)\}, \\ \min\limits_{\ell_1+\ell_2=\ell} \{f_{v_1}(k+1,\ell_1) + f_{v_2}(k+1,\ell_2)\} \end{array} \right\}$$

This completes the description of the computations of $f_v(k,\ell)$. The final answer is $f_r(0,m)$, where r is the root of T and m is the number of pebbles. If m is given, (typically much smaller than the number of leaves), in the above computations one never needs to compute for ℓ, the number of pebbles allowed, beyond m, i.e., all $\ell \le m$.

We estimate the complexity of the algorithm. Let $H = h(T)$ be the height of the tree. Typically $H \approx O(\log n)$. For leaves, the algorithm spends $O(d_v) = O(H)$ time per leaf. For each vertex with one child the time is $O(d_v \min\{L_v, m\}) = O(Hm)$. For each vertex with two children it is $O(d_v \min\{L_v, m\}^2) = O(Hm^2)$. Hence the total running time is at most $O(nHm^2)$, which is only $O(nm^2 \log n)$ with $H \approx O(\log n)$.

It is also clear that the above algorithm can be easily modified to compute the actual optimal algorithm in addition to the optimal cost.

Theorem 1: There is a polynomial time algorithm that computes the optimal pebbling placement as well as the optimal cost of the pebbling placement. The running time is $O(nHm^2)$, for a binary tree of n vertices, height H, and m pebbles.

C. General trees

We now generalize the above algorithm to an arbitrary tree. First, for a leaf node v, we define $f_v(k,\ell)$ to be the minimal cost $c(P)$ of all feasible pebbling P of (T_v, k) with at most ℓ pebbles in T_v, and where if $k > 0$ we stipulate that one additional pebble is placed at the tip of the external path from v. Note that in the case of leaf node, T_v is a singleton, and if $k > 0$ then (T_v, k) is a single path of length k. Also $0 \le \ell \le L_v = 1$, and $0 \le k \le d_v$.

Thus, the computation for the leaves are identical to that in the binary tree. If $k = 0$, then
$$f_v(0,0) = \begin{cases} 0 & \text{if } w[v] = 0 \\ \infty & \text{otherwise,} \end{cases}$$
and for $\ell = 1$,
$$f_v(0,1) = w[v].$$
For $k \ge 1$,
$$f_v(k,0) = (k+1) \cdot w[v],$$
and for $\ell = 1$,
$$f_v(k,1) = w[v].$$

We now consider non-leaf nodes v. Let Δ be the number of children of v, let $v_1, v_2, \ldots, v_\Delta$ be its children from left to right, and let the subtrees rooted at the children of v be $T_{v,1}, T_{v,2}, \ldots, T_{v,\Delta}$ respectively. Denote by $T_{v,[d]}$ the subtree of T_v induced by the vertex set of $\{v\} \cup \bigcup_{i=1}^{d} T_{v,i}$, for $1 \le d \le \Delta$. Denote by $L_{v,d}$ the total number of leaves in $T_{v,[d]}$.

Define $f^b_{v,d}(k,\ell)$, where $b = 0$ or 1, $1 \le d \le \Delta$, $0 \le \ell \le L_{v,d}$, and $0 \le k \le d_v$, as follows. First let $k = 0$. If $b = 0$, $f^0_{v,d}(0,\ell)$ is the minimal cost of a pebbling placement of the subtree $T_{v,[d]}$, where we use at most ℓ pebbles in $T_{v,[d]}$, and no pebble is placed on v. (When no feasible pebbling placement exists with this constraint we have $f^0_{v,d}(0,\ell) = \infty$.) If $b = 1$, $f^1_{v,d}(0,\ell)$ is the same as above except v is placed with a pebble out of ℓ pebbles.

This definition is generalized for $k \geq 0$. For $f^b_{v,d}(k,\ell)$, we consider $(T_{v,[d]}, k)$ in place of $T_{v,[d]}$ and for $k > 0$ we stipulate that one additional pebble is placed at the tip of the external path from v of distance k from v. As before this additional pebble is not counted in ℓ.

We then define

$$f^b_v(k,\ell) = f^b_{v,\Delta}(k,\ell),$$

and

$$f_v(k,\ell) = \min\{f^0_v(k,\ell), f^1_v(k,\ell)\}.$$

Again we will compute $f_v(k,\ell)$ for all $k,\ell \geq 0$, inductively for v according to the height of the subtree T_v, starting with leaves v. The base case $h = 0$ having already been taken care of, we assume $h > 0$ and $h(T_v) = h$.

First we consider the left most subtree ($T_{v,1}$ with $d = 1$, i.e., we compute $f^b_{v,1}(k,\ell)$ for $(T_{v,[1]}, k)$.

If $k = 0$ and $b = 0$, then

$$f^0_{v,1}(0,\ell) = f_{v_1}(0,\ell).$$

Note that $h(T_{v_1}) < h$ and thus inductively $f_{v_1}(k,\ell)$ have been all computed already.

Similarly for $k = 0$ and $b = 1$, then

$$f^1_{v,1}(0,\ell) = \begin{cases} \infty & \text{if } \ell = 0 \\ f_{v_1}(1,\ell-1) & \text{if } \ell \geq 1. \end{cases}$$

Note that in the last equation the "k" value in f_{v_1} is 1 due to the stipulation that by $b = 1$ we placed a pebble on v.

Now we consider $k \geq 1$. Again if $b = 0$,

$$f^0_{v,1}(k,\ell) = f_{v_1}(k+1,\ell).$$

Similarly for $k \geq 1$ and $b = 1$,

$$f^1_{v,1}(0,\ell) = \begin{cases} \infty & \text{if } \ell = 0 \\ f_{v_1}(1,\ell-1) & \text{if } \ell \geq 1. \end{cases}$$

We proceed to the case of $1 < d \leq \Delta$. This time we inductively assume that we have already computed not only all $f_{v'}(k,\ell)$ with $h(T_{v'}) < h$, but also the relevant quantities for $(T_{v,[d-1]}, k)$.

Thus, for $k = 0$ and $b = 0$,

$$f^0_{v,d}(0,\ell) = \min_{\ell'+\ell''=\ell}\{f^0_{v,d-1}(0,\ell') + f_{v_d}(0,\ell'')\}.$$

To be precise the minimization is over all pairs (ℓ',ℓ''), such that $0 \leq \ell' \leq L_{v,d-1}$, $0 \leq \ell'' \leq L_{v_d}$ and $\ell_1 + \ell_2 = \ell \leq L_{v,d}$.

For $k = 0$ and $b = 1$,

$$f^1_{v,d}(0,\ell) = \min_{\ell'+\ell''=\ell}\{f^1_{v,d-1}(0,\ell') + f_{v_d}(1,\ell'')\}.$$

Note that in f_{v_d} we had the "k" value 1 since by $b = 1$ we have stipulated that a pebble is placed on v. The range of (ℓ',ℓ'') are the same as before except in fact ℓ' must be ≥ 1, otherwise the value ∞ will appear. (In particular, for $\ell = 0$ the minimization is ∞.)

Finally we consider the case $d > 1$ and $1 \leq k \leq d_v$. For $k \geq 1$ and $b = 0$, we have

$$f^0_{v,d}(k,\ell) = \min_{\ell'+\ell''=\ell}\{f^0_{v,d-1}(k,\ell') + f_{v_d}(k+1,\ell'')\}.$$

And for $k \geq 1$ and $b = 1$, we have

$$f^1_{v,d}(k,\ell) = \min_{\ell'+\ell''=\ell}\{f^1_{v,d-1}(k,\ell') + f_{v_d}(1,\ell'')\}.$$

Note that in the last equation, in fact the minimization is over all pairs (ℓ',ℓ'') with $\ell' \geq 1$, as well as $\ell' \leq L_{v,d-1}$, $0 \leq \ell'' \leq L_{v_d}$ and $\ell_1 + \ell_2 = \ell \leq L_{v,d}$. But we do not need to explicitly state that $\ell' \geq 1$, since for $\ell' = 0$, $f^1_{v,d-1}(k,0) = \infty$ can be shown by an easy induction. Also note that the "k" value in f_{v_d} is 1, due to the stipulation by $b = 1$ that v is pebbled by one of the ℓ' pebbles.

We have completed the description of the algorithm. The final answer is $f_r(0,m)$, where r is the root of T and m is the number of pebbles. Again, no need to compute for any value $\ell > m$, if m is the total number of pebbles given.

The complexity of the algorithm can be easily estimated as before. For leaves, the algorithm spends $O(d_v) = O(H)$ time per leaf. Thus the total work spent on leaves is at most $O(nH)$. For any non-leaf v, suppose the degree of v is Δ_v, then the computation work spent for v is $O(\Delta_v H m^2)$. Thus the total amount of work spent for non-leaves is $O(\sum_v \Delta_v H m^2) = O(nHm^2)$. Hence the total running time is at most $O(nHm^2)$, which is again only $O(nm^2 \log n)$ with $H \approx O(\log n)$.

Theorem 2: There is a polynomial time algorithm that computes the optimal pebbling placement as well as the optimal cost of the pebbling placement. The running time is $O(nHm^2)$, for any rooted tree of n vertices, height H, and m pebbles.

D. Implementation

Our simultaneous placement algorithm is a *dynamic programming* algorithm that visits each node exactly once to determine the best use of m caches in its subtree. The algorithm discovers the optimal placement of all values of ℓ caches from 0 to m so as to minimize the total cost of traffic.

The result of running the evaluation on any node v is a $k \times m$ matrix $f_v(k,\ell)$ containing the total costs of the subtree where $0 \leq \ell \leq m$ is the number of caches and k is the distance to the nearest source of the data. For each element of the matrix, the node must choose how many pebbles to give to each of its children and whether or not to keep a pebble for itself.

Leaf nodes can compute their cost matrix $f_v(k,\ell)$ easily. If they are given one or more pebbles, their cost is simply the number of bytes of replies needed by that AS. Assume t_v is that number of local traffic bytes at node v. If a leaf is given zero pebbles, his cost is $k * t_v$. In the implementation, we used a matrix that is 15 rows high, representing values of k from 0 to 14. In our study, the maximum number of pebbles, m, is set to 50 but could be increased at the cost of running time and memory consumed by the algorithm.

Define $f^b_{v,d}(k,\ell)$ to be the cost of a daughter subtree of T_v, where $1 \leq d \leq \Delta$ is one of the Δ daughters of node v.

Each row k of $f^0_v(k,\ell)$ is computed using row $k+1$ from the daughters. Start with the first daughter's $k+1$ row intact. Then for each subsequent daughter, test all distributions of $\ell' + \ell'' = \ell$ pebbles in which ℓ' pebbles are given to the prior daughters and ℓ'' pebbles are given to the new child.

$$f^0_{v,d}(k,\ell) = \min_{\ell'+\ell''=\ell} \{f^0_{v,d-1}(k,\ell') + f_{v_d}(k+1,\ell'')\}.$$

When all Δ children have been combined, the resulting $f^0_{v,\Delta}(k,\ell)$ matrix is $f^0_v(k,\ell)$.

Now we construct $f^1_v(k,\ell)$. The first element, $f^1_v(0,0)$ is ∞, because no pebble is available. To find the rest of $f^1_v(k,\ell)$, take the first row of $f^0_v(k,\ell)$ and shift it down by 1 pebble because the children will only have $\ell - 1$ pebbles to distribute. Note that all other rows of the matrix are copies of row 0.

$$f^1_v(k,\ell) = f^0_v(0,\ell-1)$$

Finally, each element $f_v(k,\ell)$ is the minimum of $f^1_v(k,\ell)$ and $f^0_v(k,\ell)$.

To compute the best placement for the whole tree, we compute the cost matrix of the root, $f_{root}(k,\ell)$. The row $k = 0$ contains the minimum cost for the whole tree for values of $0 \leq \ell \leq m$.

E. Practical computational cost

Let i be the number of interior (non-leaf) nodes in the tree (1594 in our study). Let H be the height of the tree, the maximum number of AS-hops for any path (15 in our study). Let m be the maximum number of proxy caches placed (50 in our study).

Each AS is visited exactly once to compute his cost matrix. The total number of cost matrices computed is i.

Each cost matrix has K rows. The total number of cost rows computed is $i * K$.

Each of those rows is a combination of the contributions from all of the children of the node. Let δ be the number of children of node v. As previously noted, there will be K rows at node v. Each of those rows will have $m + 1$ items representing values from 0 to m pebbles. The initial local cost matrix of the parent will be combined δ times with other matrices (once for each child).

After several simple optimizations, our test run with a tree of 21 backbone nodes totaling 6395 nodes had 69,486 row combinations in the 6395 matrix combinations.

VI. Evaluation of Cache Placement Impact

To measure the benefit of each new cache added to the tree, we compute the total traffic seen by the sample web server log. Figure 5 shows the total traffic normalized to the traffic that would result if 0 caches are used. In our test data, 3.41 Gigabytes of replies came from 790 of the 6395 clusters. Using the tree produced by the clustering algorithm, on average traffic touched 3.07 AS's including the AS at the backbone and the originating AS. The total cost of traffic in this test data was 10.46 Gigabyte-ASHops.

A. Random Placement

For comparison, we compute costs for a placement algorithm that more closely matches the way caches might be placed opportunistically in a practical case. We randomly chose 50 locations out of the top 200 demand sites. The results in Figure 5 show that an occasional good guess causes a noticeable decrease in traffic. The random algorithm took 193 caches to reduce the normalized traffic below 7 Gigabyte-ASHops. Averaging a number of random runs would smooth the curve, but would be unlikely to lower it.

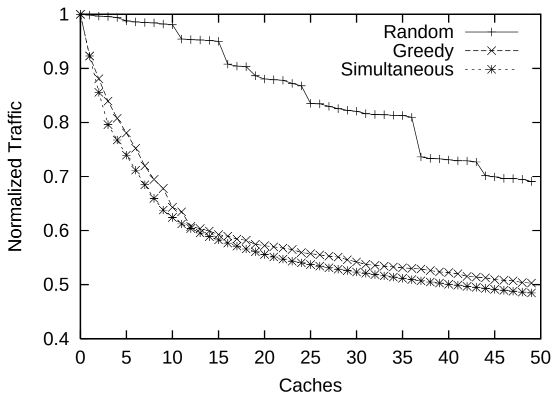


Fig. 5. Performance versus random and greedy placement

B. Greedy Placement

Figure 5 also shows the results of a greedy placement algorithm that incrementally places each cache at the hottest remaining site in the forest. Two greedy algorithms were attempted with very similar results. Assume p caches have already been placed. Incremental placement is accomplished for the $(p + 1)$ cache by pre-defining locations for the prior p pebbles. The algorithm is then run with only one pebble allocated to the entire Internet. In fact, figure 5 shows an even simpler algorithm to determine the placement of a single, new cache. It chooses the uncached AS with the highest local demand. We were surprised to see how well the greedy algorithms performed and how closely their performance matched each other. The greedy algorithm reduced the total traffic below 7 Gigabyte-ASHops by using the 10 AS's with the highest local demand. In fact, the first 11 locations chosen by the greedy algorithm matched the first 11 locations chosen by the simultaneous placement algorithm (albeit in a different order).

Moreover, these incremental placement algorithms (random and greedy) more closely model the financial reality that moving a cache from one location to another is typically not economic.

C. Simultaneous Placement

Running the dynamic programming algorithm discovered ways to cut the total traffic gigabyte hops by half using 42 caches. This is 10 fewer caches than a greedy placement and it is also a point at which extra caches give little benefit. With 200 caches, the simultaneous placement algorithm was able to reduce the traffic to 4 Gigabyte-ASHops.

Perhaps the most benefit of the simultaneous placement algorithm is the shape of the graph. Figure 5 clearly shows diminishing returns beyond placing 11 caches. By running the algorithm once, the analyst can see what the optimal result is for the entire range of 0 to m caches and compare the benefits to the cost per cache.

VII. Conclusions

In this paper we have described methods for creating AS clusters based on BGP routing data. The algorithm for creating a forest of AS numbers objectively discovers the AS's that form a highly interconnected backbone for the Internet. The resulting forest slightly overstates the average number of hops from any point in the Internet to a common backbone, but is close enough to allow the study of client demand and cache placement.

We have also presented a new, optimal method for placing caches in the AS hierarchy generated by our clustering method. We compared the effectiveness of our algorithm to two incremental techniques using a commercial Web log. We found that greedy placement of caches worked nearly as well as the sophisticated, optimal technique when the number of caches was small or large.

VIII. Acknowledgements

The authors would like to thank Peter Andreae and Venkat Chakaravarthy for discussions of old and new algorithms for spanning trees and Jude Shavlik for insights into nearest neighbor computations.

References

[1] B. Krishnamurthy and J. Wang, "On network aware clustering of Web clients," in *Proceedings of ACM SIGCOMM '00*, Stockholm, Sweden, September 2000.

[2] R.Hamming, "Error detecting and error correcting codes," Tech. Rep. 29-147, Bell System Technical Journal, 1950.

[3] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby, "On the optimal placement of Web proxies in the Internet," in *Proceedings of IEEE INFOCOM '99*, New York, New York, March 1999.

[4] R. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, pp. 1389–1401, 1957.

[5] J. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," in *Proceedings of the American Mathematical Society*, 1956.

[6] C. Cuñha, *Trace Analysis and its Applications to Performance Enhancements of Distributed Information Systems*, Ph.D. thesis, Boston University, 1997.

[7] P. Mockapetris, "Deomain names - concepts and facilities," IETF RFC 1034, November 1987.

[8] J. Kangasharju, K. Ross, and J. Roberts, "Performance evaluation of redirection schemes in content distribution networks," in *Proceedings of 5th Web Caching and Content Distribution Workshop*, Lisbon, Portugal, June 2000.

[9] A. Shaikh, R. Tewari, and M. Agrawal, "On the effectiveness of DNS-based server selection," in *Proceedings of IEEE INFOCOM '01*, Anchorage, Alaska, April 2001.

[10] A. Myers, P. Dinda, and H. Zhang, "Performance characteristics of mirror servers on the Internet," in *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[11] H. Braun and K. Claffy, "Web traffic characterization: An assessment of the impact of caching documents from NCSA's Web server," in *Proceedings of the Second International WWW Conference*, Chicago, IL, October 1994.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[13] R. Wooster and M. Abrams, "Proxy caching that estimates page load delays," in *Sixth First International World Wide Web Conference*, Santa Clara, California, 1997.

[14] J. Dilly and M. Arlitt, "Improving proxy cache performance: Analysis of three replacement policies," *IEEE Internet Computing*, vol. 3(6), November 1999.

[15] S. Glassman, "A caching relay for the World Wide Web," *Computer Networks and ISDN Systems*, vol. 27, no. 2, 1994.

[16] S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V.Jacobson, "Adaptive Web caching: Towards a new global caching architecture," in *Proceedings of the 3rd Web Caching Workshop*, Manchester, England, June 1998.

[17] Squid Internet Object Cache, "http://www.nlanr.net/squid," 2001.

[18] P. Cao, J.Zhang, and K. Beach, "Active cache: Caching dynamic contents on the Web," *Distributed Systems Engineering*, vol. 6(1), 1999.

[19] S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt, "Constrained mirror placement on the Internet," in *Proceedings of IEEE INFOCOM '01*, Anchorage, Alaska, April 2001.

[20] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of Web server replicas," in *Proceedings of IEEE INFOCOM '01*, Anchorage, Alaska, April 2001.

[21] R. Govindan and H. Tangmunarunkit, "Heuristics for internet map discovery," in *Proceedings of IEEE INFOCOM '00*, April 2000.

[22] J.-J. Pansiot and D. Grad, "On Routes and Multicast Trees in the Internet," *Computer Communications Review*, vol. 28, no. 1, January 1998.

[23] R. Siamwalla, R. Sharma, and S. Keshav, "Discovering internet topology," Tech. Rep., Cornell University Computer Science Department, July 1998, http://www.cs.cornell.edu/skeshav/papers/discovery.pdf.

[24] R. Govindan and A. Reddy, "An analysis of internet inter-domain topology and route stability," in *Proceedings of IEEE INFOCOM '97*, Kobe, Japan, April 1997.

[25] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of ACM SIGCOMM '99*, Boston, Massachusetts, September 1999.

[26] National Laboratory for Applied Network Research, "http://www.nlanr.net," 1998.

[27] Merit Internet Performance Measurement and Analysis Project, "http://nic.merit.edu/ipma/," 1998.

[28] Route Views, "University of oregon," http://www.antc.uoregon.edu/route-views.

[29] Y. Rekhter and T. Li, "RFC 1771: A Border Gateway Protocol 4 (BGP-4)," Mar. 1995, Obsoletes RFC1654. Status: DRAFT STANDARD.

[30] Y. Rekhter and P. Gross, "RFC 1772: Application of the Border Gateway Protocol in the Internet," Mar. 1995.

[31] NetCity Project, "NetCity Demand Summary File Format, sample files, and scripts to produce NCD files," http://www.cs.wisc.edu/netcity.