18.996: Topics on Internet Research Problems Lecture 3

Tracking Distributed Objects

Lecturer: R. Rajaraman *Scribe: M.T. Hajiaghayi and H. Zhou*

1 Introduction

The *data tracking* problem is one of the most basic problems in the Internet research area. Assume we have a set $1 \cdots m$ of objects and a set $1 \cdots n$ of nodes. As shown in Figure 1, we consider the whole *Internet* as a communication service such that each two nodes i and j can communicate via the Internet, but with some cost c_{ij}. In these notes, when we say two nodes i and j in the network are *neighbors*, we mean i and j are near each other by some logical measure such as *latency*, not necessarily *next-hop* neighborhood.

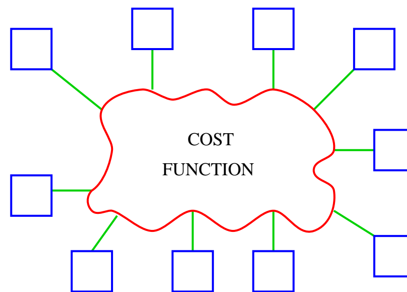

Figure 1: The Internet as a communication service

We also assume that each node contains a set of copies of objects. Now, our goal is to find a (nearby) copy of the requested object, insert or delete object copies and finally update control information as nodes join or leave the system. More precisely, we have the following operations.

1. *find(u,x)* in which node u issues a request to locate and obtain a copy of object x;
2. *insert(u,x)* in which node u inserts a new copy of object x into its storage space;
3. *delete(u,x)* in which node u deletes an existing copy of object x from its storage space;
4. *join(u)* in which node u joins the system; and
5. *leave(u)* in which node u leaves the system.

We note that the *insert* and *delete* operations refer to the nodes' public storage space. Thus, *insert(u,x)* makes a new copy of x at u available for access to all the network nodes, while *delete(u,x)* makes an existing available copy of x at u unavailable to the rest of the network. We also note that following a *join* or *leave* operation, it is quite likely that we have a series of *insert* or *delete* operations.

In tracking distributed objects, we assume that all nodes are equal and have the same power. For example, we do not have some central or special nodes which are more powerful than others. The main reason behind this assumption is that in real distributed systems like the Internet, nodes can join or leave very frequently and often we cannot assume that some nodes are very powerful. In addition, this assumption causes our system to be more fault-tolerant, e.g. if each node shuts-down abruptly, then we have only a leave operation and we can adapt the whole system very easily.

The data tracking problem has many applications. First, consider the *DNS* in which we need to map names to IP addresses. In this system, our objects are IP addresses whose copies are stored in some nodes of the Internet. Given a name, we want to find a copy of the object which contains its IP address. A second application is in peer-to-peer netwroks where each node acts as both a client and a server, and we need to provide efficient access to shared data using lightweight procedures. Again basic operations in these networks can be considered special cases of the operations mentioned above. Other examples are replicated servers, in which the join and leave operations may be ignored, and tracking mobile users, in which we have only one copy of each node (each node contains a unique object) [AP95].

Current popular commercial systems using peer-to-peer file sharing systems are *Napster*, *Gnutella* and *Freenet*. We discuss *Gnutella* and *Freenet* more in the next two subsections. Selected academic research projects in this area are *Oceanstore* at University of California at Berkeley which is conducted by Kubiatowics et al., *Chord* at Massachusetts Institute of Technology which is conducted by Stoica et al. and *Content Addressable Network (CAN)* again at University of California at Berekeley which is conducted by Ratnasamy et al.

1.1 Gnutella

Gnutella network is a decentralized, unmanaged system for sharing, searching, and acquiring files. The Gnutella network supports sharing and searching of any file type unlike Napster which only allows users to share certain types of files, namely MP3s. However Gnutella does not offer any extra functionality, like chatting. Gnutella is a peer-to-peer system, with client software that also acts as a server. Gnutella was created by a group of developers at Nullsoft, a subsidiary of America Online.

Suppose that a node u wants to find an object like a web-page which contains some keywords. Gnutella works by *flooding* protocol in which node X sends its request to all its neighbors and then these neighbors send the request to all their neighbors until the request is reached by a node Y containing the object and then node Y returns a copy of the object to X (see Figure 2). In this file-system, each request is satisfied by a nearby copy and thus it is efficient in terms of the *find* cost. The flooding process is somehow *controlled* in the system, that is, each node has a searchable index in a database which prevents flooding via some wrong paths. However this system is not scalable and in the worst-case, the entire network may be flooded. Also for eliminating loops, the system uses a *time-to-live (TTL)* field, which may prevent excessive flooding.

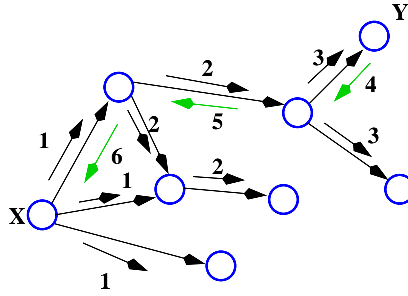

Figure 2: Finding an object in Gnutella

1.2 Freenet

Free Network (Freenet) is a large-scale peer-to-peer decentralized network which pools the power of member computers around the world to create a massive virtual information store open to anyone to publish or view information of all kinds freely. Freenet dynamically replicates and relocates information in response to demand to provide efficient service and minimal bandwidth usage regardless of load. In addition, Freenet is private and secure, i.e. information stored in Freenet is protected by strong cryptography to guard against malicious tampering or counterfeiting. This network is an enhanced Open Source implementation of the system described by Ian Clarke's 1999 paper *"A distributed decentralized information storage and retrieval system."* The first version (Version 0.1) of this system was released in March 2000.

To find an object, Freenet uses a DFS of the network, which results in a *sequential version of flooding* in the worst-case (Figure 3 illustrates a sequence of query and response messages in Freenet). Here we have a trade-off between efficiency and scalability, i.e. a request may have to be forwarded along a long chain of nodes before being satisfied, but little congestion is caused due to a single request.

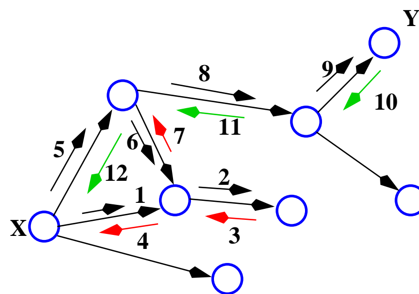

Figure 3: Finding an object in Freenet

1.3 Measures

Communication cost between two nodes is an idealized function of latency, bandwidth, queue size, etc. For

analysis, we assume that our cost function is a metric static cost. The *cost of an operation* is the total communication cost of messages it sends to conduct the operation.

Let v be a node nearest to u that has a copy of x. We define *stretch of find(u,x)* operation as $\frac{cost\ of\ find(u,x)}{cost(u,v)}$. The stretch of insert and delete operations are defined similarly [BFR95]. For join and leave operations, we use the number of nodes that are updated as a measure instead of communication cost.

Memory overhead is defined to be the maximum amount of *control information* stored at a node of the network. Here by *control information*, we mean the list of nodes to which node forwards requests or the list of objects of which the node is aware. We also define the *static load* of a node to be the number of objects it is aware of and *dynamic load* of a node to be the number of find operations affecting the node per unit time.

2 Algorithms for data tracking

In this section, we introduce three algorithms for efficient data tracking. The first one is a simple algorithm which presents the main ideas. The second algorithm is due to Awerbuch and Peleg [AP90]. Awerbuch and Peleg motivated the idea of *locality preserving* in which, given that a task concerns only a subset of the sites located in a small region in the network, one would like the execution of the task to involve only sites in or around that region and the cost of the task to be proportional to its locality level. The concepts of *sparse neighborhood covers* and *hierarchical clustering decompositions* (see Subsection 2.2) were first introduced in this paper. Using these concepts, they prove near-optimal bounds on the stretch factor. We see the main disadvantage of the algorithm when we have join and leave operations. The second algorithm, which is a simpler flat tracking scheme, is due to Plaxton et al. [PRR99]. The paper considers *partially locality preserving* and *static load balancing* and forms the data location component of Oceanstore (see Subsection 2.3). Due to time constraints, we were unable to cover recent work on data tracking algorithms, including the Chord and CAN projects.

2.1 A simple algorithm: a tree-based distributed tracking

One naive approach for data tracking is embedding a tree into the network and then each node informs all its ancestors about its own objects (see Figure 4). Now for $find(u,x)$, we forward the request through the path from u to the root of the tree until we find the first ancestor which contains control information about object x and then we retrieve the object from an appropriate node. In fact, the tree and its embedding determine the location of control information among the network nodes. This approach has some problems. The first one is that the root of the tree must have control information about all objects. In other words, we have memory overhead $\Omega(m)$. The second problem is that the embedding may not respect network locality. For example, consider a ring of nodes $0, \cdots, n-1$ in which c_{ij} is the number of hops between node i and j in the ring plus one (the cost function is metric). One can observe that in every tree embedding there are two nodes of distance one in the ring whose distance in the network is $\Omega(n)$. In the next Subsection, we

consider the sparse neighborhood covers introduced by Awerbuch and Peleg [AP90] to overcome the second problem. The main idea here is to try to embed multiple "tree-like structures" in the network. This idea further was considered by Bartal et al. in their paper on hierarchically well-separated trees [Bar98].

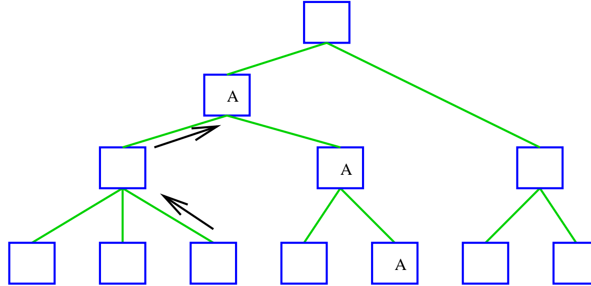

Figure 4: Tree-based distributed tracking

2.2 Sparse neighborhood covers

The *r-neighborhood* of a vertex u is defined as $N_r(u) = \{v | c(u,v) \leq r\}$. Recall that $c(u,v)$ is the distance between u and v in the network. A *sparse r-neighborhood cover* M_r is a collection of sets (also called *clusters*) of vertices $S_1, \cdots, S_l$ with the following properties:

1. for every vertex v there exists some $1 \leq i \leq l$ such that $N_r(v) \subseteq S_i$;
2. diameter of each cluster S_i (the diameter of the subgraph induced by S_i), $1 \leq i \leq l$, is in $O(r \log n)$; and
3. each node belongs to $O(\log n)$ clusters.

A *sparse neighborhood covers data structure* is a family of sparse r-neighborhood covers for different values of r. Applications often require the construction of sparse r-neighborhood covers for $O(\log(Diam(G))$ successively doubled values, namely $r = 1, 2, 4, 8, \cdots$. Here G is our network graph.

The algorithm for finding a sparse r-neighborhood cover M_r is as follows.

- Initially all the nodes are unmarked.
- Repeat the following until all nodes are *removed*.
  1. Pick an unmarked node u.
  2. Find smallest j such that $2|N_{jr}(u)| \geq |N_{(j+1)r}(u)|$.
  3. Either $j \leq \log n$ exists or $N_{r \log n}(u)$ includes all nodes; in the latter case, set $j = \log n$.
  4. Include set $N_{(j+1)r}(u)$ in cover.

5. Mark all nodes in $N_{(j+1)r}(u)$ and *remove* all nodes in $N_{jr}(u)$ from further consideration.
6. If there exists any unmarked node u, go to step 1, otherwise first unmark all nodes and then go to step 1.

First we observe that, when a node v is *removed*, its $N_r(v)$ is included in some cluster (see Figure 5). Using the fact that in each cluster at least half of the nodes will be removed, one can prove that each node will be in at most $O(\log n)$ clusters (see the details in [ABCP99]). Linal and Saks [LS93] showed how a distributed randomized algorithm can be used to compute sparse covers. In this approach, each node u starts with $N_r(u)$ as a cluster. Then in each round, every cluster grows simultaneously, but some clusters stop because of the others (the ties are broken randomly). The running time of this algorithm is poly-logarithmic time. The reader is referred to the original paper for more details.

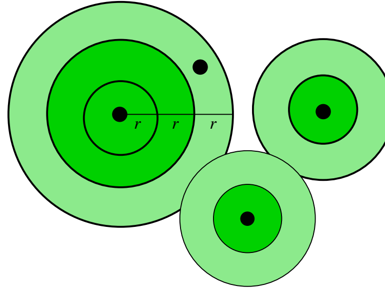

Figure 5: Growing regions to obtain an sparse neighborhood cover

Now, we assume that we have sparse neighborhood covers data structure which contains all M_{2^i}, $1 \leq i \leq \log(Diam(G))$. Our solution for data tracking is based on the hierarchy of covers in the network. For each cluster S in each cover M_{2^i}, we elect a leader $l(S)$ and provide internal routing services by constructing a tree routing component for S rooted at $l(S)$. For each i, we associate with every node u a *home cluster*, $home_i(u) \in M_{2^i}$, which is the cluster containing $N_{2^i}(u)$. Thus each node has $\log(Diam(G))$ home clusters. Now, consider $find(u,x)$ operation. Node u first tries using the lowest level cover M_{2^1}, i.e. forward its request for object x to its first home cluster leader, $l(home_1(u))$. If this trial fails, i.e. $l(home_1(u))$ does not know any control information about x, u retries sending its message, this time using cover M_{2^2}, and so on, until it finally succeeds. Suppose a node v nearest to u which contains x has distance d to u. Since v is contained in $N_{2^{\lceil \log d \rceil}}(u)$ and diameter of each cluster in M_{2^i} is at most $2^i \log n$, we have:

$$cost\ of\ find(u,x) = O(\sum_{i=0}^{\lceil \log d \rceil} 2^i \log n) = O(d \log n).$$

Thus the stretch of find is $O(\log n)$.

For $insert(u,x)$ or $delete(u,x)$, node u informs the leader of each cluster containing u in M_{2^i} for all $1 \leq i \leq \log(Diam(G))$. The worst-case cost for these two operations is in $O(Diam(G)\ polylog(n))$; however,

an amortized stretch of $O(polylog(n))$ can be achieved [BFR95]. The cost of join and leave operations is in $\Omega(n)$, since we need to update all nodes in the worst case. Finally, since some leader nodes need to store control information of all objects, the memory overhead of this algorithm is in $\Omega(m)$. In the next, Subsection we present an algorithm with a better memory overhead.

2.3 A Flat Tracking Scheme

If, for each object in the system, we maintain a logical tree that randomly maps to the nodes(hopefully respecting locality), then the set of objects will be evenly distributed among the nodes and we will have good load balance. However, there continues to be a scalability problem in that each node must know its neighbors in the tree for every possible object tree. To address this, we present a simple flat tracking scheme that uses randomized embedding of logical trees to achieve static load balance while requiring low memory overhead. For a restricted class of cost functions, it also achieves asymptotically efficient cost. This scheme forms the data location component of Oceanstore, a data tracking system developed at Berkeley.

In this scheme, unique IDs are assigned to nodes and objects. The node whose bits match the largest prefix of the object ID becomes "root" in that object's tree and is responsible for information necessary to locate at least one copy of the object. The thrust of this is to route by successively matching a longer prefix of the object ID from node to node until we arrive at one that knows about a copy of the object searched for. Note that random node IDs mean that the tracking scheme is topology-sensitive.

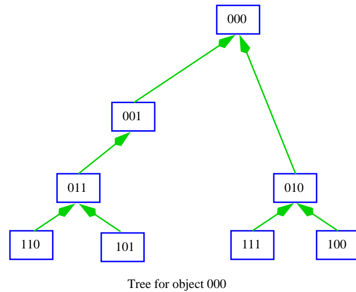

Figure 6: Object access tree

As shown in figure 6, the parent of a node u is the closest node(by some network metric, e.g. latency) whose ID matches the object's ID in a longer prefix than u's ID. For example, node 011 matches 000 in a prefix of length 1, better than node 110 which matches no prefix at all. As 011 is the closest such node to 110, it becomes the parent of 110 in the access tree for 000. Similarly, node 000(root for object 000) happens to be closest to 010, and so becomes the direct parent of 010, so that 010 would be able to get to object 000 in one hop.

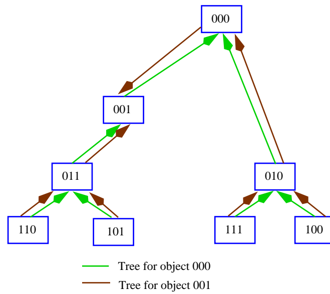

Figure 7: Overlapping neighbors

Clearly, as shown in figure 7, there will be overlap among the neighbors of a given node across different access trees. In fact, with a little thought, one can see that maintaining the closest node whose ID matches the object's ID in a longer prefix for all objects is equivalent to simply maintaining the closest node whose ID matches the current node's ID in the first i bits and differs in the $(i+1)$st bit (assuming n a power of 2). Thus, the total number of distinct neighbors a node has to store is only $\log n$. A typical neighbor table is shown in figure 8.

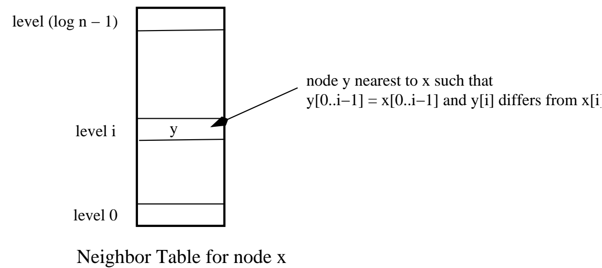

Figure 8: Node neighbor table

In addition to neighbors, a node also maintains object pointers to objects that it knows about. An object request is routed from node to node, successively matching longer node ID prefixes, until it reaches one that has a pointer to the object. If the object is in the system at all, then the root node for that object will point to it, so that the lookup is guaranteed to terminate. Any pointers not at root are like cached queries: they sometimes alleviate the need to go all the way to the root.

When a node u inserts an object copy into the system, it propagates the location of the copy through the object's search path, leaving an object pointer at each node along the path that is unaware of the object. If it encounters a node w that points to an existing copy that is closer to w than u, it changes nothing at w and the propagation ends. Otherwise, if u is closer, it updates the pointer at w and continues the propagation.

This flat tracking scheme has several advantages. The randomized ID assignment distributes the set of object pointers evenly over the nodes. The lookup scheme maintains the neighbor tables at $\log n$ size. Thus, the system scales well. Under certain assumptions about communication cost functions, the system can also be shown to be efficient in that the expected access cost is within a constant factor of optimal. In particular, we require that for every node x and real $r \geq 1$, the ratio of the number of nodes within distance(cost) $2r$ of x to the number of nodes within distance r is bounded from above and below by constants. [PRR99]

There are also several limitations. As mentioned above, the efficiency proofs depend on restricted cost functions. It does not take into consideration dynamic load on nodes, and the overhead of forwarding the requests through several nodes may be significant. The system has no distributed scheme for dynamic node joins and leaves. Furthermore, the number of nodes affected by a join or leave may be large, from a practical standpoint.

3 The Load Assignment Problem

(Notes essentially from Adrian Vetta's presentation)

INPUT:

- A set $S = \{s_1, s_2, \ldots, s_n\}$ of sources.
- Associated with a sink s_i is a load of size l_i.
- A set $T = \{t_1, t_2, \ldots, t_r\}$ of sinks.
- Associated with a sink t_j is a cost function $c_j()$.
- There is an edge ij if a load from s_i may be routed to t_j.

OUTPUT: A minimum cost assignment of the loads to sinks.

A solution is an assignment $x = \{x_1, x_2, \ldots, x_m\}$ where x_e is the load on edge e.

THE COST FUNCTIONS

Given a source s_i, let $\mathbb{F}_i$ be the set of sinks incident to s_i. Given a sink t_j, let Γ_j be the set of sources incident to t_j. Let $X_j = \sum_{i \in \Gamma_j} x_{ij}$ be the total load at sink t_j.

For a sink t_j we will assume that the cost function $c_j()$:

- Is a function of X_j, i.e. $c_j(X_j)$

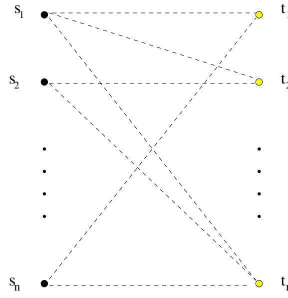

Figure 9: Load assignment problem

- Has decreasing marginal costs $c'_j(X_j)$.

Note that this second property is just concavity, since $c''_j(X_j) \leq 0$.

A Mathematical Formulation

$$\begin{array}{rcll} & \min & \sum_{t_j \in T} c_j(X_j) & \\ \sum_{t_j \in \mathbb{F}_i} x_{ij} & = & l_i & \forall\, s_i \in S \\ x_{ij} & \geq & 0 & \forall\, ij \in E \end{array}$$

3.1 A Hardness Result

Set Cover

Input: Elements $\{v_1, v_2, \ldots, v_n\}$ and sets $\{S_1, S_2, \ldots, S_r\}$.

Output: A collection $\mathcal{S}$ of sets of minimum cardinality that covers every element.

It is known that the Set Cover problem can not be approximated to within an $O(\log n)$ factor.

Reduction to Load Assignment

There is an approximation preserving reduction from Set Cover to the Load Assignment problem:

- There is a source s_i for each element v_i.
- There is a sink t_j for each set S_j.
- There is an edge ij if the element v_i is in the set S_j.
- The load l_i is one for each source.

- The marginal costs at sink j are $(1, 0, 0, \ldots, 0)$.

Theorem. The Load Assignment problem can not be approximated to within an $O(\log n)$ factor.■

3.2 The Structure of a Solution

OBSERVATION

We may view solutions as permutations of $\{1, 2, \ldots, r\}$. To see this, note that some optimal solution x has the following two properties:

Property I. For any source s_i, all of its load l_i is assigned to a single sink.

Proof. Suppose not. Let some of the load be assigned to the sink t_1 and some to sink t_2. Now if $c_1'(X_1) \leq c_2'(X_2)$ then, by concavity, the total cost may be reduced by re-routing the load from t_2 to t_1. Similarly if $c_2'(X_2) \leq c_1'(X_1)$.■

From now on we may assume that the load at each source is one.

3.3 Saturated Sinks

Sink t_j is saturated if each source in Γ_j assigns its load to t_j.

Property II. At least one sink is saturated.

Proof. Suppose not. Then, by concavity,

$$\exists s_1 \in \Gamma_1 \text{ st load } l_1 \text{ is routed to } t_{f(1)} \neq t_1 \Longrightarrow c'_{f(1)}(X_{f(1)}) < c_1'(X_1)$$

$$\exists s_2 \in \Gamma_2 \text{ st load } l_2 \text{ is routed to } t_{f(2)} \neq t_2 \Longrightarrow c'_{f(2)}(X_{f(2)}) < c_2'(X_2)$$

$$\vdots$$

Consider a directed graph with a node for each sink, and arcs $(j, f(j))$. Then as each no sink is saturated, each vertex has out degree one. Thus the graph contains a cycle C. Summing around the cycle, we obtain

$$\sum_{t_j \in C} c'_{f(j)}(X_{f(j)}) \quad < \quad \sum_{t_j \in C} c_j'(X_j)$$

Let k be the last node in the cycle, so that f(k) is the node we started out from. The above telescopes to give

$$c'_k(X_k) < c'_{f(k)}(X_{f(k)})$$

which is a contradiction. ∎

WLOG assume that t_1 is saturated. Then by similar arguments:
$\exists t_2$ that is saturated with respect to $S - \Gamma_1$
$\exists t_3$ that is saturated with respect to $S - \Gamma_1 - \Gamma_2$ etc ...

3.4 A Greedy Algorithm

GREEDY
Calculate the average saturated cost of each sink, i.e. $\frac{c_j(|\Gamma_j|)}{|\Gamma_j|}$.
Assign Γ_{j^*} to t_{j^*}, the sink with the minimum average saturated cost.
Repeat on $S - \Gamma_{j^*}$ until each source is covered

RUNNING TIME
The running time of the greedy algorithm is linear in the number of edges. It takes $O(m)$ time to calculate the initial average saturated costs and $O(m)$ time, over all subsequent iterations, to update these average saturated costs.

3.5 Analysis of Greedy Algorithm

The optimal solution $T^* = \{t_1^*, t_2^*, \dots, t_k^*\}$ has cost $c(T^*) = \text{OPT}$.
The greedy solution $T = \{t_1, t_2, \dots, t_s\}$ covers n_i sources in step i.

Theorem. Greedy is an $O(\log n)$-approximation algorithm.
Proof. Note that OPT/n is just a weighted average of sink average costs. Thus, we see that

$$\text{At step 1 some sink in } T^* \text{ has average cost} \leq \frac{\text{OPT}}{n}$$

$$\text{At step 2 some sink in } T^* \text{ has average cost} \leq \frac{\text{OPT}}{n-n_1}$$

$$\vdots$$

At step s some sink in T^* has average cost $\leq \frac{\text{OPT}}{n-n_1-\cdots-n_{s-1}}$

Now, by concavity, the average saturated cost of a sink is at most the average cost. Thus, the greedy algorithm has a total cost

$$c(T) \leq n_1 \frac{\text{OPT}}{n} + n_2 \frac{\text{OPT}}{n - n_1} + \cdots + n_s \frac{\text{OPT}}{n - n_1 - \cdots - n_{s-1}}$$

$$\leq (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) \text{ OPT}$$

$$= H_n \text{ OPT} \quad = \quad O(\log n) \text{ OPT}$$

where in the second inequality we've used the fact that

$$\frac{n}{m} \leq \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-m+1} \blacksquare$$

3.6 Some Special Cases

CONCAVE MARGINAL COST FUNCTIONS

The actual performance guarantee of greedy is $\frac{1}{\alpha}$ where

$$\alpha = \min_j \frac{1}{c'_j(0)} \frac{c_j(|\Gamma_j|)}{|\Gamma_j|} < \min_j \frac{1}{c_j(1)} \frac{c_j(|\Gamma_j|)}{|\Gamma_j|}$$

As a corollary, for example, if the marginal cost function is concave then we have a 2-approximation algorithm.

CONSTANT NUMBER OF "TRICKY" SINKS

If there is only a fixed number k of sinks with non-constant marginal cost functions then we can solve the problem optimally.

CONVEX COST FUNCTIONS

If the cost function is convex then the optimal solution can be found by a minimum cost flow algorithm.

3.7 Open Problems

PROBLEM I

What if there are capacities at the sinks?

PROBLEM II

What if the cost functions are more complex?

PROBLEM III

What if there is added structure regarding source-sink links?

4 Another Open Problem: Online Assignment

This is essentially the same as the load assignment problem with two key differences: the assignment must be performed online, and assigning unit load from client i to server j incurs c(i,j) cost.

One might consider two possible server load models: a capacity C_j for server j, or a function f_j of load that gives additional cost for each unit load that is served by j.

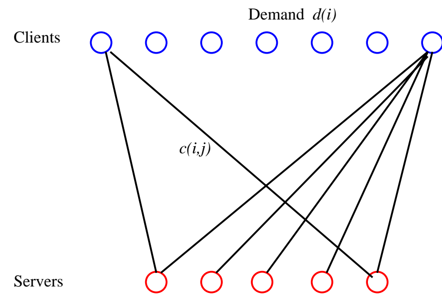

Figure 10: Online load assignment

References

[ABCP99] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. *SIAM Journal of Computing*, 28:263–277, 1999.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[AP95] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 37:1021–1058, 1995.

[Bar98] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, May 1998.

[BFR95] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. *Journal of Computer and Systems Sciences*, 51:341–358, 1995.

[LS93] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13:441–454, 1993.

[PRR99] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.