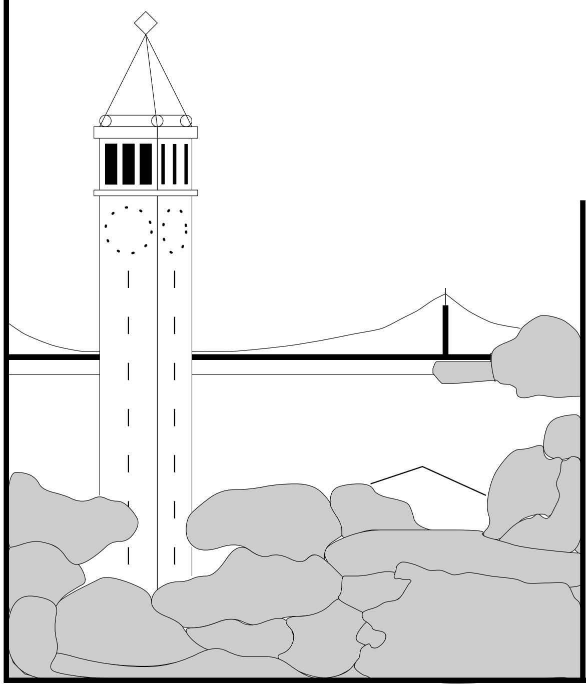
Another Way to Find the Nearest Neighbor in Growth-Restricted Metrics

Kirsten Hildrum *John Kubiatowicz* *Satish Rao*



Report No. UCB/CSD-03-1267

August 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

In this paper, we give sequential and distributed dynamic data structures for finding nearest neighbors in certain growth restricted metrics.

In particular, we give a sequential data structure that uses linear space, and requires $O(\log n)$ expected time and $O(\log n)$ time for lookups with high probability. This improved the results of Karger and Ruhl, whose data structure takes $O(n \log n)$ space with comparable expected time bounds. Also, we describe a dynamic, load-balanced data structure using $O(\log n)$ space per node, matching the bound of Karger and Ruhl.

We note that our algorithm is significantly different in structure from those of Karger and Ruhl [4], and perhaps substantially simpler. It is based on a technique used for object location developed by Plaxton, Rajaraman and Richa in [7], which gives it an application to peer-to-peer networks.

1 Introduction

This nearest-neighbor algorithm is motivated work in peer-to-peer networks, more precisely, those peer-to-peer networks that are based on the scheme presented by Plaxton, Rajaraman and Richa [7]. Relevant schemes include [1, 6, 8, 10]. In these systems, nodes in a network are assigned random identifiers. For the routing to work properly, each node must know the closest node with certain ID prefixes. Further, when a new node enters the system, it must be able to find the nearest node with the given prefix, and may also need to update the information for other nodes. In a special class of metric spaces, these systems provably perform well.

Essentially, this requires maintaining a nearest-neighbor search data structure in a metric space. In general metric spaces, this problem is extremely difficult, but these systems assume a restricted space, and in this restricted space, the problem becomes tractable.

Karger and Ruhl [4] were the first to look at the nearest neighbor problem in the restricted class of metric spaces relevant to this application. They defined a slightly-broader class of metric spaces they called *growth-restricted*. This is similar in spirit, but stronger than, requiring that the growth be polynomially bounded. Their construction is randomized. Recently, Krauthgamer and Lee [5] presented a sequential, deterministic construction that has applications to a broader class of metric spaces.

In the PRR-like networks ([1, 7, 8, 10]), nodes (computers) and objects (files) in the network are given IDs uniformly at random. Objects have home

Scheme	Avg Space	Max Space	Comments
Karger and Ruhl	$O(\log n)$	$O(\log n)$	balanced, dynamic
Krauthgamer and Lee	$O(1)$	$O(1)$	dynamic
This paper, basic	$O(1)$	$O(\log n)$	
This paper, balanced	$O(\log n)$	$O(\log n)$	balanced
This paper, dynamic	$O(1)$	$O(n)$	dynamic
This paper, both	$O(\log n)$	$O(\log n)$	balanced,dynamic

Table 1: A comparison of various versions of our scheme to the previous work. The scheme presented here is better suited applications in PRR-like networks.

the node with ID that most closely matches their ID.This node is the *root*. The root always stores either a copy of the object or a pointer to the object.

Requests for objects are sent toward the root for that object, and once they reach that point, the request can be satisfied ([1, 7, 10] attempt to shortcut this process by placing object pointers in the network).

Suppose the node A has ID α. We will imagine that this is an ID is some radix b, that is, each digit is in the range 0 to $b-1$. Then let α_i denote the first i digits of α. Then for each i, and for $j \in [0, b-1]$, A stores the closest node with prefix $\alpha_i j$ (that is, α_i followed by j). This enables prefix routing. That is, at each step, one additional digit of the prefix is "fixed."

If A receives a request for and object with ID β, A forwards the request to some node with the prefix β_1. Then that node is able to find some node with prefix β_2, and so on. For example, a request for an object 1234 starting at 0000 would first go to a node beginning in 1, then a node beginning in 12, then a node beginning in 123, and finally would reach 1234.

As described, this requires no nearest neighbor information. But when i is small, there are many nodes with prefix $\alpha_i j$, and if the node A picks the closest one, the overall route length will be small (in certain metric spaces). A particular destination defines a tree on this network, made up of all the links that can be used to route toward this destination. Intuitively, two nearby nodes will likely share an ancestor in this tree. This is the intuition; the situation is complicated by the fact that one "parent" isn't enough, so the structure no longer is a tree.)

The nearest neighbor scheme presented here is in some sense a "backward" version of this. It uses the same tree, and notices that the nearest neighbor of a query point q likely shares a parent with q. As such, it uses the almost the same data structures (the exception is the dynamic version of the algorithm, which requires additional data), searching for the subtrees that q might belong to with the idea that q's nearest neighbor also belongs to those subtrees.

A less efficient version of this algorithm appeared in [3]. That version fixed a parameter k, and showed that for $k = O(\log n)$, the algorithm would find the nearest node with high probability with running time $O(k \log n) = O(\log^2 n)$. The version presented here is "Las Vegas", and implicitly chooses k to be only as big as it needs to be, and so find the nearest neighbor with only $O(\log n)$ communication steps.

Karger and Ruhl in [4] present an algorithm for finding the nearest neighbor in the same class of metric spaces as used here. Their algorithm uses $O(\log n)$ expected space, and runs in time $O(\log n)$ with high probability. Their overall idea is to halve the distance between the current point and the query point. After $\log n$ successful halving steps, the algorithm finds the nearest neighbor of the query point. To implement these halving steps, they use a random permutation of the nodes. The algorithm described here follows this very general outline, but the resulting algorithms are different.

Recently, Krauthgamer and Lee [5] described a deterministic construction that is in some ways similar in structure to the one presented here. Their algorithm has applications in a broader class of metric spaces, though it is not yet clear whether it can be used as a distributed algorithm.

Most closely related is the algorithm presented by the same authors in [3], which used $O(\log n)$ space and $O(\log^2 n)$ time. This technical report improves that result to use $O(\log n)$ time. This also gives a sequential algorithm that uses linear space and $O(\log n)$ time, but Krauthgamer and Lee present an better sequential algorithm.

The approach is similar to that of Clarkson in [2], and the sampling technique used by Thorup and Zwick [9] for approximate distance oracles is similar to our technique. We also note that the general idea of our algorithm is very similar to the idea used by Plaxton, Rajaraman and Richa [7] to find a nearby copy of an object.

2 Preliminaries

In this paper, we will use the term metric and network interchangeably. A network can be naturally described as a graph, where the computers are nodes and the edges represent the wires connecting them. In turn, a graph induces a metric. Given a metric space of points S, we would like to find the closest point in the metric space to a given point s. This application that motivated this algorithm requires that both the space and the work be distributed.

For an algorithm to be distributed, the data structures needed can be divided up among the nodes in such a way that each node keeps about the same amount of data, and the algorithm only needs to use the data from a few of the nodes. In our model, we can access the metric space in a very limited way–we can query an oracle for the distance from s to t (one can think of this as s pinging t).

We now define growth-restricted. Let the ball around x of radius r be all nodes of distance less than or equal to r from x. We say this ball has volume s if there are s nodes in the ball. Then,

Definition 1 *A metric is growth-restricted with constant* c *if, for any* x *and* r*, when the ball around* x *of radius* r *has volume* s*, the ball around* x *of radius* $2r$ *has volume less than or equal to* cs*.*

For example, points on a d dimensional grid have this property with $c = 2^d$.

A simple data algorithm for finding the nearest neighbor would be to simply list all the nodes in the network, query the distance to each of them, and thereby find the nearest. This would, however, be a lot of work. A simple way to improving on this idea is to clustering the nodes. The query point then tests each cluster to see if it might contain the nearest neighbor. Still, then either there are many clusters to search, or searching a cluster is very expensive, so we decompose the clusters further, creating $\log_b n$ levels of clustering.

Now we explain how we use randomness to create these clusters. (Krauthgamer and Lee take a similar approach and give a simple and elegant deterministic algorithm to find these clusters.)

For $i \in [0, \log_b n - 1]$, we produce a random sample of the network, such that level $(i+1)$ is a subset of level i. A node is in the ith sample with probability $1/b^i$. For level $\log_b n$, we pick exactly one node to be the root.

Next, we connect these nodes together as follows. Each level i node now chooses as its parent the closest level $i+1$ node. In addition, each level i node has a pointer to all its children. Notice that each level i node points to exactly one level $i+1$ node. Second, notice that there are b times as many level $i+1$ nodes as level i nodes, so in expectation, each level i node has b children.

3 Finding the Nearest Neighbor

At the high level, the algorithm is simple. Start with the root node in the current list. Then ask each node in the current list for their children. Choose the children close enough to have descendants that are "close" to be the in the next level current list. Then query those, and so on. The problem is determining which nodes could have nearby descendants. In this section, we will make the assumption that we have extra knowledge. In particular, we will assume we know that distance to the closest level-i node for each i. This is not a reasonable assumption in general, but we will later show how to dispense with it.

For an index i, let $d_i(x)$ be the distance from x to the closest level i node. We will drop the x when clear from context.

Let $q_0(x) = d_0(x)$, and for $i > 0$, $q_i(x) = \max(3d_i(x), 3q_{i-1}(x))$. Lemma 1 shows that all level-$(i-1)$ nodes within $q_{i-1}(x)$ have parents within $q_i(x)$ of the query node.

Given the q_is and the single $\log_b n$ level root node, we can find the nearest neighbor of a node x, as follows. First, x queries the root for its children, and keep all the children within q_i for the i corresponding to that level. For all those nodes, query their children, and keep the children within q_{i-1} and so on. Pseudocode for this case is shown in Figure 2. In Section 4, we will show how the q_i's can be found.

Lemma 1 *If we query all the level i nodes within q_i of x, then we can find all the level $i-1$ nodes within q_{i-1} of x.*

Proof:

We will ask all the level i nodes within q_i for their children (the level $i-1$ nodes that point to them). We want to prove that no level-$(i-1)$ node within distance q_{i-1} is missed.

Suppose A is a level $i-1$ node within distance q_{i-1} of x. If the parent of A is within q_i of x, then we will query A's parent and so find A. We show that $d(A, \text{parent}(A)) < q_i$.

Figure 1 shows this situation. Notice that A's parent must be closer to A than the closest level i node to the query point (call this node u_i), since A chooses the closest among the level i nodes. Mathematically, we know that $d(A, \text{parent}(A)) \leq d_i + q_{i-1}$.

We know that the distance between A and the query point is bounded by q_{i-1}, so the distance between the query point and A's parent is bounded by $2q_{i-1} + d_i$, but q_i was chosen to be greater than $2q_{i-1} + d_i$, so we are done. $\square$

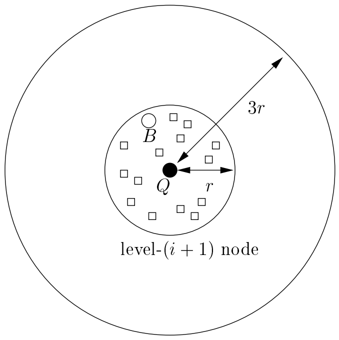


Figure 1: The parent of A must lie within the big circle.

```
method FindNearestNeighbor (rootNode, q)
  currList ← [rootNode]
  nextList ← ∅
  for i = maxLevel to 1
      for n ∈ currList
          nextList ← nextList ∪ GetChildren(n)
      currList ← KeepWithDist(nextList, q_i)
      nextList ← ∅
  return (currList)
end FindNearestNeighbor
```

Figure 2: If the q_i's are giving, finding the nearest neighbor is straightforward.

3.1 Bounding the number of nodes

In this section, we bound the number of nodes contacted during this process. To facilitate this, we define the notion of a certificate. A certificate for x has for each i, a list of all the level-i nodes within distance $q_i(x)$. We view the certificate as being divided up in pieces, called subcertificates. Two adjacent levels i and $i-1$ are in the same subcertificate if $q_i = 3q_{i-1}$. The lowest level in a subcertificate is called a base level. By definition, level 0 is always a base level.

The certificate described above has $O(\log n)$ nodes in expectation if the metric space is growth restricted. To show this, we start with the following

Lemma.

Lemma 2 *Suppose i is a base level (i.e., the lowest level in some subcertificate), and the base ball has volume s. Then the expected size of that subcertificate is bounded by $O(s/b^i)$, provided that $c^2 < b$.*

Proof: For a given j, we must find all the level $i+j$ nodes within a factor 3^{j+1} times the original radius. If the original ball had volume s, then each factor of 3 increase in radius increases the volume of the ball by no more than a factor of c^2. So the ball of radius $3^{j+1}d_i$ has volume bounded by $s_i(c^2)^{j+1}$, where d_i and s_i are the base radius and base volume, respectively. For a given j, we only need to store the level $(i+j)$ nodes. The probability that a node is an $i+j$ node is $b^{-(i+j)}$. Combining these two facts with a little algebra, we expect to have no more than $s/b^{i-1}(c^2/b)^{j+1}$ level $(j+i)$ nodes in the certificate. Summing over all possible j, this gives an upper bound of $O(s/b^{i-1})$ nodes in the certificate for base level i, and since b is a constant, $O(s/b^{i-1}) = O(s/b^i)$. (This final step makes a later proof a bit tidier.) □

Now, we can prove the main size lemma.

Lemma 3 *The total expected size of the certificate is $O(\log n)$ if b is larger than c^2, where c is the expansion constant of the network.*

Proof:

We bound the total size of a subcertificate at level i by considering the expected size of the subcertificate when there is no base level larger than i. This is an overcount since some levels may be charged to more than one base level.

Let s_i be the size of the base ball at level i. If $s_i = s$, that means the first s nodes were not part of the ith sample. Using this fact, we get

$$Pr[s_i > cb^i] \leq (1 - 1/b^i)^{cb^i} \leq e^{-c}.$$

Now, we know that

$$E[\text{baselevel}_i] = \sum_{k=1}^{\infty} E[\text{baselevel}_i \Big| (k-1)b_i \leq s_i < kb^i] Pr[(k-1)*b^i \leq s_i < k \cdot b^i] \leq \sum e^{-k} kb^i/b^{i-1}.$$

For fixed b, this is a constant.

Finally, since there are at most $\log n$ subcertificates, the total certificate size is $O(\log n)$ ■

3.2 High Probability Bound

Lemma 4 *Suppose we are given the base ball sizes $s_1 \dots s_k$. Then with high probability, the size of the certificate is less than $O(\sum_i s_i/b^i)$*

Proof: We want to argue that we can view each base level independently. If that is the case, then we can apply a Chernoff bound, as before, as we are done. The problem is that the levels are not independent.

But consider the following related variable X, where $X = \sum_{i,j} X_j^{(i)}$ and $X_j^{(i)}$ is one if node j is in a subcertificate for level i with base ball s_i. Then X is the sum of independent random variables, and we can apply the Chernoff bound. Since $E[X] = \sum s_i/b_i = O(\log n)$, we will be able to get a high probability bound.

Now, consider the random variable $Y = \sum Y_i$, where Y_i is one if node i is in the certificate. Notice that $Pr[Y \geq k] \leq Pr[X \geq k]$, and we bound X by the use of a Chernoff bound. $\square$

Next, we bound the probability that $\sum_i s_i/b^{i-1}$ is large.

Lemma 5 *For the s_i defined as before, $\sum_i s_i/b^{i-1}$ is $O(\log n)$.*

Proof Let $S = \sum_i s_i/b^i$. The probability of a given configuration $s_1, s_2, \dots s_k$ is

$$\begin{aligned}
(1-1/b)^{s_1} \prod_i (1-1/b^i)^{s_i - s_{i-1}} &\leq \exp\left(-s_1/b_1 + -\sum_{i=2}^{k}(s_i - s_{i-1})/b^i\right) \\
&= \exp(-S + 1/b(S - s_k)) \\
&\leq \exp(-S + S/b) \\
&\leq \exp(-S(1-1/b)) \\
&\leq \exp(-\frac{1}{2}S)
\end{aligned}$$

The number of ways to get a given sum S from k terms is S choose k, so the probability that of a sum is S is less than $(Se/k)^k \exp(-\frac{1}{2}S)$.

Now, let S_0 be such that $S_0/k > 4\log(S_0 e/k)$. Then

$$\begin{aligned}
&(S_0 e/k)^k \exp(-\frac{1}{2}S_0) \\
&\quad = \exp(-\frac{1}{2}S_0 + k\log(S_0 e/k)) \\
&\quad \leq \exp(-\frac{1}{2}S_0 + k\frac{1}{4}S_0/k)
\end{aligned}$$

```
method GetClosest (x,rootNode, maxLevel)
  for i = 0 to maxLevel - 1
      InitializeHeap(heap[i])
      closest[i] ← ∞
  Insert(heap[maxLevel], rootNode, d(x,rootNode))
return GetNext(x, 0, ∞)
end GetCertificate

method GetNext (x, level, maxDist)
  if AllHigherLevelsEmpty(heap[i]) then return null
  do
      if peek(heap[level]) < maxDist then
          nextDist ← 3·Max(closest[level + 1], Peek(heap[level]))
      else
          nextDist ← 3·Max(closest[level + 1], maxDist)
      next ← getNext(x,level + 1, nextDist)
      if notNull(next) then
          if closest[(level + 1)] = ∞ then closest[(level + 1)] ← d(x, next)
          AddNodeToCertAndChildrenToHeap(next,level)
  while (notNull(next))
  if peek(heap[level]) < maxDist
    return getMin(heap[level])
  else
    return null
end GetNext
```

Figure 3: Anything that leaves a heap is part of the certificate.

$$\leq \quad \exp(-\frac{1}{4}S_0)$$

Finally, note that the $Pr[S \geq S_O] \leq \sum_{S \geq S_0} \exp(-\frac{1}{4}S) \leq \exp(-\frac{1}{4}S_0) \sum_i (e^{\frac{1}{4}})^i \leq 5\exp(-\frac{1}{4}S_0)$. This means that when $S = O(\log n)$, the certificate size is greater than S with probability one over polynomial in n.

4 Generating the Certificate

This section explains how to find the nearest neighbor without knowing the q_i's in advance. This section shows an algorithm that needs to contact only nodes that are children of those in the certificate. Since each node has an expected constant number of children, if the certificate size is $O(\log n)$, in expectation, we do not contact more than $O(b \log n)$ in the certificate generation process.

The high-probability argument is a little bit more difficult. Note, however, that if A is a level-i node within distance q_i, only its children within distance $2q_i$ could matter, since any child at distance $2q_i$ from A is at least q_i from x. What this means is that no children further than $2q_i$from a level-i in the certificate need to be queried. To get a high probability bound on the number of nodes contacted, we need to bound the number of level-i nodes within $2q_{i+1}$ of x. This can be done using the same technique as Lemma 4, where the constant in the big-O would be different.

To motivate the algorithm, notice that if we could access the nodes on a given level in order of their distance from the query point, the problem would be solved. Starting with $i = 0$, we find the closet node at level i, which we can use to compute q_i.

Figure 3 shows pseudocode. The algorithm maintains, for each level, a list of candidates for the certificate. Nodes removed from these candidate lists are placed in the certificate, and their children become candidates (see line 10). To ensure that this does not place any extra nodes in the certificate, we require than when a level-i is added to the certificate, there are no closer level-i nodes not yet in the certificate. This may require a check for nearby level-$i+1$ nodes.

In Figure 3, the candidate lists are implemented as heaps, since we naturally only want to pull out the minimum. To guarantee that we have the closest possible, use Lemma 1 as follows. Before moving a level-i node from the candidate list to the certificate, we ensure that there are no unexplored (or candidate) level-$(i+1)$ nodes that could have a child closer to the query point than the node at the top of the level-i heap.

Before arguing this is correct, we point out two facts.

Fact 1 *If a node A is in the the certificate for a query point x, then it must be that the parent of A is also in the certificate.*

If we restate this, it says that if a level-i node A is within q_i, the parent of A is within q_{i+1}, which is true by construction and Lemma 1.

Fact 2 *If a level-i node A is in the certificate, then any level-i node B closer to x than A is also in the certificate.*

To see this, recall that the certificate is defined by the q_i's, so if A is in the certificate, $d(A, x) \leq q_i$, but since $d(B, x) \leq d(A, x)$, then $d(B, x) \leq q_i$, so B is also in the certificate.

The algorithm in Figure 3 will work by pulling out level-i nodes in order of their distance from x. The function is GetClosest does some initialization of global variables, and then calls the recursive GetNext. For each level, we keep the distance to the closest found at a given level. Also, for each level, there's a list of nodes whose parents are in the certificate, but who are not themselves in the certificate yet. We will call these candidate nodes.

The function GetNext takes the query point, x, a level l, and a distance $d_{\max}$. Then, if there is a level-l node within the given distance, it will return the closest such node not already in the certificate, and otherwise returns null.

Suppose, then we call GetNext with the query point x, the level is l, and the distance set to d_{max}. Before returning the closest node among the candidate level-l nodes, it must guarantee that there are no closer level-l nodes. By Lemma 1, we get a bound on the distance between x and any the parent of any closer level-l node. Then, we call GetNext with the query point x, level $(l+1)$ and this distance bound. If nothing is found, then we know that the candidate is the correct one, and return it. If something is found, we add it to the certificate, add its children to the list of candidates, and check again whether the closest candidate is the closest overall.

To find the nearest neighbor, GetClosest calls GetNext with query point x, level 0, and the distance ∞ in line 3. This will cause many recursive calls to GetNext, and we could like to argue that each such call returns a node in the certificate. By Fact 2, we can simplify the problem and argue that the last node returned by GetNext on level l is in the certificate, since by construction, GetNext returns level l nodes in order of their distance from the query source.

The proof will be by induction on l. The base case is the nearest level-0 node, which is also the last level-0 node returned, and is clearly in the certificate. Now assume the last level-$(l-1)$ returned is in the certificate. We will show that the last level-l node returned must also be in the certificate.

Let A be the last level l node returned by a call to GetNext. There are two cases: The last level-$(l-1)$ is a child of A, or it is not. If it is a child of A, then clearly, A must be in the certificate by fact 1. So suppose it is not. Then we will argue that $d_{\max}$ is no more than q_l, so if A is returned, its distance to x is less than $d_{\max}$ and so less than q_l.

Consider the call stack, and particularly the $d_{\max}$ at level-$(l-1)$. Since we know that the last level-$(l-1)$ node is not a child of A, it must already be in the heap. If it will be returned, by the induction hypothesis, it is at distance less than q_{l-1}, and so that implies that $d_{\max}$ is less than q_l. But suppose it has already been returned. Then we can bound the $d_{\max}$ on the recursive call in a similar way, and repeat the argument backward.

A simpler, but not as provably good, approach appears in the appendix.

While this describes how to find the nearest neighbor, the algorithm can be extended to find the closest k nodes for any k merely by repeating calls to GetNext. Further, it can be used to find all nodes within distance r. The cost of these changes depends on the most distant node returned.

This is important because we need this functionality for the original purpose described in [3].

5 A Dynamic Algorithm

The algorithm can be easily extended to allow nodes to be inserted at the lowest level. In the process of finding its nearest neighbor, a node finds its nearest level 1 node, and that node becomes it parent. However, a truly dynamic algorithm must allow the insertion at any level, and a level i node needs to find its level $i-1$ children, and the data structure we presented does not make this easy, because children can be arbitrarily far away from their parents. In expectation, the distance is small, but is not bounded. Krauthgamer and Lee get around this problem because their construction ensures that parents are close to children.

To solve this problem, we require that each node write a pointer to itself on every node in its certificate. That is, if A is in the certificate of B, then A has a pointer to B. Further, when A gets a new child, it notifies B.[1] Suppose a node C enters that should be a part of B's certificate. Then C's parent must also be in B's certificate. But then the parent of C has a pointer to B, and so B will be notified about C's entrance, and C will have a pointer to B.

The problem with this scheme is that the root has to store pointers to all the nodes, since it is in every node's certificate. One way of dealing with this is to use load balancing similar to that in PRR. More precisely, assign each node a ID in base b. Each node then defines a tree, where the nodes matching in the first i digits are the level-i nodes. Each node is in

[1] An alternate solution was presented in [3]. That paper noted that with high probability, only the closest $O(\log n)$ level-$(i-1)$ nodes are ever children. These nodes can be found as described above.

the static tree for every other node, storing only its parent and children. As described in [3, 7], these trees can all be maintained with $O(b \log_b n)$ storage per node. Each node is also in one dynamic tree (that of a node with the longest possible matching prefix), and the information for the dynamic tree is used to maintain the static trees.

6 Conclusion

We have presented an algorithm for finding the nearest neighbor in a growth-restricted metric space. Our algorithm takes only constant space per node, and runs in time polynomial in the network. Preliminary simulations suggest that its practical performance will be similar to theoretical bounds.

Finally, we believe these techniques, in combination with the techniques of Abraham, Malkhi, and Dobzinski [1] could be used to present a linear space, load balanced, dynamical nearest neighbor data structure.

References

[1] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. Land: Locality aware networks for distributed hash tables. Technical Report Leibnitz Center TR 2003-75, The Hebrew University, June 2003.

[2] K. L. Clarkson. Nearest neighbor queries in metric spaces. In *Proc. of the 29th Annual ACM Symp. on Theory of Comp.*, pages 609–617, 1997.

[3] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, 2002.

[4] David Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. of the 34th Annual ACM Symp. on Theory of Comp.*, pages 741–750, May 2002.

[5] Robert Krauthgamer and James Lee. Navigating nets: Simple algorithms for proximity search. manuscript, 2003.

[6] Xiaozhou Li and C. Greg Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 82–89, October 2002.

[7] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th Annual Symp. on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[8] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, pages 329–350, 2001.

[9] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *Proc. of the 33th Annual ACM Symp. on Theory of Comp.*, pages 183–192, 2001.

[10] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Computer Science Division, 2001.

A Alternate Certificate Generation

A potentially simpler approach is to upper bound the d_i's, and so get upper bounds on the q_i's, and using these too-big q_i's, do the search as before. If this is done, we will find all the nodes in the certificate, including the nearest node. Unfortunately, nodes outside the certificate may also be contacted, so the bounds on certificate size are not directly applicable.

However, because it does not use a heap, it may be simpler to implement, and would probably perform equally well in practice. A simple way to upper bound the d_i's would be to query only the closest k nodes at every level (for k some small value like one or three), and use the results to get d_i's.

```
method GetCertificate (x,rootNode)
certificate[maxLevel] ← {rootNode}
queryDist[maxLevel] ← ∞
level ← maxlevel - 1
while level > 0
    (certificate[level], queryDist[level]) =
        GetNextList(x, certificate[level + 1], level,
        queryDist[level + 1])
    while 3* queryDist[level] > queryDist[level + 1]
            queryDist[level + 1] = 3*queryDist[level]
            level ← level + 1
    level ← level - 1
end GetCertificate

method GetNextList (x,neighborlist, level, queryDist)
nextList ← ∅
minDist ← queryDist
for n ∈ neighborlist
    if (d(n,x) ≤ queryDist
        temp ← GetChildren(n, level)
        for m ∈ temp
            if d(m, x) ≤ minDist
                minDist ← d(m, x)
return (nextList, minDist)
end GetNextList
```

Figure 4: A different method of certificate generation. This method does not have provable bounds, but it may be simpler and easier to implement in practice.