Complexity of Adaptive Spatial Indexing for Robust Distributed Data

by

Matthew Vincent Mahoney

Master of Science
Electrical Engineering
Florida Institute of Technology
1987

A thesis
submitted to the Graduate School of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
January, 1998

We the undersigned committee
hereby approve the attached thesis

Complexity of Adaptive Spatial Indexing for Robust Distributed Data

by
Matthew Vincent Mahoney

Philip K. Chan, Ph.D.
Assistant Professor, Computer Science
Thesis Advisor

William D. Shoaff, Ph.D.
Program Chair and Associate Professor, Computer Science
Thesis Committee

Jay Yellen, Ph.D.
Program Chair, Operations Research
Associate Professor, Applied Mathematics
Thesis Committee

Abstract

Complexity of Adaptive Spatial Indexing for Robust Distributed Data

by
Matthew Vincent Mahoney

Thesis Advisor: Philip K. Chan, Ph.D.

A spatial index suitable for implementation of a multidimensionally keyed database (such as text retrieval system) in an unreliable, decentralized, distributed environment is shown to have complexity comparable to the Internet's Domain Name Service and better than USENET or Web search engines. The index is a graph mapped into Euclidean space with high smoothness, a property allowing efficient backtrack-free directed search techniques such as hill climbing. Updates are tested using random search, then edges are adaptively added to bypass local minima, network congestion, and hardware failures. Protocols are described. Empirical average case complexity for n data items are: storage, O(n log n); query, O(log n $\log^2$ log n); update, O($\log^2$ n $\log^2$ log n), provided that the number of dimensions is fixed or grows no faster than O(log n).

Table of Contents

Acknowledgments

I thank Dr. Philip Chan for his advice on preparing the thesis, both in pointing out technical difficulties and in organizing the document, even before he was officially my thesis advisor or on my committee. I also thank Dr. William Shoaff and Dr. Jay Yellen for their time spent on the thesis committee and for feedback on this document.

Section 1

Introduction

Is an Internet database service feasible? Such a service, if it existed, would allow one to post any text-based message to the Internet, which anyone else could then retrieve by keyword search, without regard to location.

Of course we can already do this. One way is to post a news article to USENET. Some search engines collect these posts and offer a text-retrieval service, allowing one to recall all posts containing certain words, ranking them by relevance. Alternatively, one could put the data on a Web server and retrieve it using one of the search engines. Both methods have their drawbacks: USENET articles typically expire after a few weeks, and Web-based search engines may not update their indexes for several days.

The problem is more fundamental than this. Both USENET and Web indexes require $O(n^2)$ storage, where n is the number of articles or Web pages. USENET articles are copied to every server, and search engines build complete indexes independently. As the Internet grows, so must the number of servers. These servers already require enormous resources, and it will get worse. Although memory, disk, and CPU performance has been doubling about every 18 months, the Internet has been doubling in size every 12 months [Kantor, 1996].

There are more efficient systems, however. The Domain Name Service (DNS), which maps host names to IP addresses, uses a distributed tree with redundant servers for reliability. Private distributed relational databases (such as airline reservation systems) sometimes have a similar organization. These systems have a different problem: If the root of the tree fails, then the whole system fails. This is true even if the root is replicated on independent servers, because a single user or organization has access to every copy, and could potentially make an error when updating the master copy.

We describe a distributed data structure that is both efficient, like DNS, and which lacks a root, like the Web and USENET. Users would be able to make updates immediately accessible (unlike the Web) and have control over deletion (unlike USENET). Nobody would "own" this data structure or decide how it is organized (unlike DNS), and there would be no single point of failure. Servers could easily run on small computers without using significant resources.

The structure we describe is a smooth randomized graph; *smooth*, meaning that the graph can be traversed efficiently, usually without backtracking; *randomized* meaning that search and update are non-deterministic; performance is not guaranteed but is good on average.

Recall the title: "Complexity of Adaptive Spatial Indexing for Robust Distributed Data".

- Complexity -- We are interested in predicting performance: how fast, and how much storage? The answer: about as efficient as DNS and much better than USENET or the Web.
- Adaptive -- The data structure adapts to changes in organization, usage patterns, hardware failures, etc. The technique is a simple one: updates are immediately tested by simulating queries, and we modify the graph (adding edges) as needed to improve performance.
- Spatial -- The data keys are mapped into a space over which a distance function is defined. We can use Euclidean space in one dimension to implement a dictionary, simply ordering the keys alphabetically, or we can use the multidimensional space of term vectors for text retrieval. The technique generalizes to non-Euclidean spaces. For example, the keys could be sets, because we can say that sets are close or far apart according to the number of common elements.
- Indexing -- We are concerned with distributing the index -- knowledge about the location of data -- not just the data itself.
- Robust -- Most data is still accessible when parts of the graph are removed. There is no single point of failure.
- Distributed -- The query and update algorithms are concurrent, and can be implemented using message passing.
- Data -- The technique applies to any data which is retrieved by matching a key, either exactly or by a distance metric (i.e. text retrieval).

First, we discuss the background of distributed relational databases, text retrieval systems, USENET, search engines, and DNS. Then we implement the adaptive spatial index as a smooth randomized graph, describing the query and update operations and estimating complexity. Next, we run simulations to measure average case complexity. Finally, we show how the algorithms could be distributed, describing a 3-level protocol that maintains graph consistency and implements the update and query operations concurrently.

Section 2

Background

We examine the distributed relational model and contrast it to text retrieval, which is easier to use and where higher error rates are tolerable. Then we examine three widely used data systems on the Internet: USENET, Web search engines, and DNS.

Distributed Databases

Most of the work in distributed databases [Bell, 1992] is toward implementing the relational model [Korth, 1991; Tansel, 1993] across multiple servers, and deals with issues such as concurrency, consistency across redundant copies, query optimization, etc. It is usually assumed that the organization of the data (the metadata) and associated access privileges is simple enough that it can all be kept in one place (or copied to every computer that needs it), and furthermore, that the data can be centrally managed. That is not the case with very large systems such as the Internet, where anyone may add data for his or her own purpose, regardless of whether it might conflict with somebody else's data.

A smooth randomized graph (SRG) is not a relational database, but it does implement many of the same functions. In theory, the tuples (table rows) of a set of relations (tables) could be encoded as strings:

"Owner, Timestamp, Table-name: Attr1=Value1 Attr2=Value2 ..."

For example:

"XYZ Corp, 11/15/1997 Employees: Name=John Smith Phone=x3287"

Then the query *SELECT FROM Employees WHERE Name="John Smith"* would retrieve the record with the phone number. Competing data would be resolved using the owner (assuming there was a secure way to identify users). Multiple copies of old and new data would be resolved using the timestamp. Arbitrary relational operations could be performed by downloading all of the necessary data and performing the operations locally.

Text Retrieval

A text retrieval system answers queries such as *Give me all documents containing "XYZ Corp," and "John Smith".* These systems match words or terms between documents and queries, returning those documents that contain all or most of the words. Words are weighted so that matches between infrequent terms (such as "John Smith") are ranked higher than matches between common words such as "and". A term is usually a word, but may be a group of words ("John Smith") or the root of a word after removing common suffixes such as "-s" or "-ed". [Stanfill 1986; Harman, 1995; Grossman, 1998; Schäuble, 1997; Sparck Jones, 1997].

A text string (document or query) can be represented as a fuzzy bag. Recall that a bag is a set allowing duplicate elements (allowing a word count), and that a fuzzy set is a set allowing continuous membership values between 0 and 1 (allowing terms to be weighted). Therefore, a fuzzy bag is simply a vector in which all elements are real and non-negative. A simple text retrieval algorithm is:

> Let Q be a query, a fuzzy bag of terms
> Let D be a database containing documents as fuzzy bags of terms
> Sort the elements D_i of D on increasing $|Q - D|$
> Return the first k documents of D

The parameter k allows one to trade off *precision* against *recall*. Precision is the fraction of documents in the returned set that are relevant to Q. Recall is the fraction of documents in D relevant to Q that are returned. The best text retrieval systems yield precision + recall of about 100% (out of a possible 200%) when matching query to document [Harman, 1995], and about 150% when matching document to document [Stanfill, 1986].

This algorithm is obviously inefficient. The best general purpose sorting algorithms have a time complexity of O(n log n) in the number, n, of data items, although there are O(n) algorithms that pick the k best elements from a list [Cormen, 1990]. A spatial index would allow fast lookup of documents "near" Q without sorting. One such index is the k-d tree, which is O(log n) for a fixed number of dimensions [Bentley, 1975]. However it is not clear that a k-d tree would be practical for databases with tens of thousands of dimensions (one per word), or that it could be distributed robustly. A k-d tree is a binary tree sorted on one of d dimensions at each level, repeating the cycle every d levels.

USENET

USENET is the Internet news group posting service [Krol, 1992]. It provides a hierarchically organized set of bulletin boards to which anyone may post articles for others to read. As a database, it implements a simple mapping from the news group name to the set of recently posted articles. Some Internet search engines also index and archive articles so that they can be retrieved by keyword search.

USENET fully replicates the database at every server. When a user posts an article, it is propagated from server to server until everyone has a copy. This makes it easy to recall articles, but results in $O(n^2)$ storage and network traffic for n articles. This is because the number of articles posted depends on the number of users, and the number of servers (assuming fixed load capacity) also depends on the number of users. In other words, the storage cost is O(mn), where there are m servers and n articles, and we are assuming that m and n must be proportional.

The system of replicating all articles on all servers worked well at first, but as the Internet has grown (doubling in size each year from 1980 through 1996 [Kantor, 1996]), USENET has become unwieldy. Servers typically have to limit the number of news groups, expire articles quickly, and limit the number of users. The service is rarely free any more. Users must typically belong to an organization or pay an Internet service provider.

Search Engines

Web search engines allow Web pages to be searched by keyword, as in a text retrieval system. They are based on the older Archie and Veronica systems, which indexed FTP and Gopher sites [Krol, 1992]. The search engine scans all known web sites and builds a term index, learning about new sites either by following hypertext links or by allowing users to register sites [Steinberg, 1996].

Search engines are quite useful, but they suffer the same $O(n^2)$ storage complexity as USENET, since every site essentially builds a copy of the same index. Again we are assuming that the number of servers and the size of each copy of the index are both proportional to the number of users. Although an index of the Web is much smaller than the Web itself, search engines still require a substantial investment in hardware and network bandwidth that can only grow worse.

Domain Name Service

The Domain Name Service (DNS) maps Internet host names such as *tuck.cs.fit.edu* to IP addresses such as *163.118.22.3* [Hunt, 1992]. This is a true distributed database. Instead of replicating the database at every server (as was done originally when the Internet had only a few hundred computers), the data is organized into a tree:

```
            .
       / | \
  com  edu  uk ...
       / | \
  mit  fit  berkeley ...
       / | \
    ee  cs  math ...
       / | \
 zach tuck ss5a ...
```

Fig. 2.1. The Domain Name System tree

Each node resides on a separate set of servers (a primary server and one or more secondary servers for reliability), and is administered independently. Each server knows only the addresses of the servers above and below it. To look up an address, a client (called a resolver) would start at the root (indicated by a dot) and go down the tree. A server can respond iteratively, referring the client to the child server, or recursively, querying the child server itself and relaying the answer back to the client. Servers and resolvers can also cache information by making local copies to improve performance, starting below the root of the tree if possible.

If the average number of children at each node is constant, then the depth of the tree grows with O(log n) in the number of data items. Since a query accesses one server at each level of the tree, query complexity is also O(log n).

Leaf nodes can be updated locally. If a DNS server is added or removed, however, the parent server must be updated, a manual process that must be coordinated between the two administrators. Caching introduces another difficulty, as there is no cache update mechanism. Since DNS data does not change frequently, this is not a major problem. Cached data typically expires after a few hours or days.

Storage complexity appears to be O(n), but is actually O(n log n) due to the need to replicate the servers near the root to handle client load. Because every access involves one server at each level (in most cases), the number of servers needed at each level is proportional to the number of users, which we assume is proportional to n. Update complexity is O(log n), given the cost of replication

DNS is not decentralized like the Web or USENET. The organization that administers the root node (currently InterNIC) effectively decides the overall structure of the data. The Web and USENET have no controlling organization, but at the cost of $O(n^2)$ storage due to full replication on every server. As we shall see, an SRG has the desirable properties of a DNS tree, but lacks a root, and therefore has no need of centralized management.

Section 3

Data Structures and Algorithms

In this section, we describe the data structures and the algorithms for a *spatial index*, where there is a mapping between the database keys and a space such as
$\mathbf{R}^d$ -- d-dimensional Euclidean space. We describe a property of graphs which we call *smoothness*, which allows a graph to be searched efficiently using greedy algorithms [Cormen, 1990] such as hill climbing [Mitchell, 1997]. We describe an *adaptive* update algorithm; in which we test updates by searching for them from random points, adding edges as needed to maintain smoothness. The adaptive algorithm repairs damage that might occur in an unreliable, distributed implementation by forming new routes around defects or congested nodes. Because the update algorithm is randomized, we call the data structure a *smooth randomized graph*, and we say that it implements an *adaptive spatial index*.

Efficient indexing structures based on graphs are well known. In one-dimensional space, a sorted binary tree with n vertices has O(log n) search complexity and O(n) storage complexity. A k-d tree [Bentley, 1975] has the same complexities in multidimensional space. Both of these data structures have a root, a fixed starting point for searching, but we can describe structures (smooth graphs) that have the same complexities with arbitrary starting points.

Unfortunately, we do not know of any distributed index with O(n) storage cost. In all of the graphs described, the longer edges (when the vertices are mapped into space) are traversed more frequently than the shorter edges. This is irrelevant in a sequential implementation but results in an unbalanced load distribution in a distributed implementation. The solution is to add redundant long-distance edges, but this increases storage complexity to O(n log n).

We first define a spatial mapping, using a one-dimensional space as an example. Next we describe search strategies, of which the simplest is hill climbing, and describe the graph properties that make search efficient in a distributed implementation. We analyze complexity for the special case of one-dimensional space, arguing (but not proving) that search complexity is $O(\log^2 n)$. Then we describe an update algorithm and show that it preserves the properties required for efficient search in one dimension. Next, we describe *randomized hill climbing*, and use it to produce more efficient search and update algorithms. Finally, we describe some multidimensional data structures that have the same complexities as the one-dimensional case.

Spatial Mapping

We define a *space* (S, dist) as a set S and a real-valued distance function *dist(a, b)* defined over pairs of elements in S, such that for all elements a, b, c, d $\in$ S:

- dist(a, a) = 0

- dist(a, b) = dist(b, a) ≥ 0
- dist(a, b) > dist(a, c) ∧ dist(a, c) > dist(a, d) ⇒ dist(a, b) > dist(a, d)

For example, d-dimensional Euclidean space $\mathbf{R}^d$ is a space over d-dimensional vectors, where dist(a, b) ≡ |a - b|. Note that in a Euclidean space, we have the triangle inequality, which states that dist(a, b) + dist(b, c) ≥ dist(a, c). Although we study only mappings into Euclidean spaces, the algorithms that we describe can be applied to spaces where the triangle inequality does not hold. For instance, we could define dist(a, b) ≡ $|a - b|^2$.

Many types of data can be mapped into space. For instance, historical weather data could be mapped into the 3-dimensional space of longitude, latitude, and time. More generally, strings from an alphabet Σ could be mapped into the 1-dimensional space [0, 1], where dist(a, b) ≡ |a - b|, by interpreting the characters in the string as the digits 1 through |Σ| in a fractional number in base |Σ| + 1, i.e. ordering the strings alphabetically. For instance, if Σ = (A=1, B=2, ..., Z=26), then map("CAB") = $3/27 + 1/27^2 + 2/27^3 = 0.1125844$. A phone directory, keyed by name, might look like this:

Map	**Name**	**Phone**
0.0		
	Alice	123
0.1	Bill	456
	Bob	720
0.2		
	Jane	562
0.3	John	965
	Mark	389
0.4	Matt	612

Fig. 3.1. A database mapping into [0,1]

The next step is to build a data a graph and assign each datum to a vertex. Some of these vertices may be connected by edges. When possible, we will show graphs so that the position of each vertex in the diagram corresponds to its mapping into space. For instance:

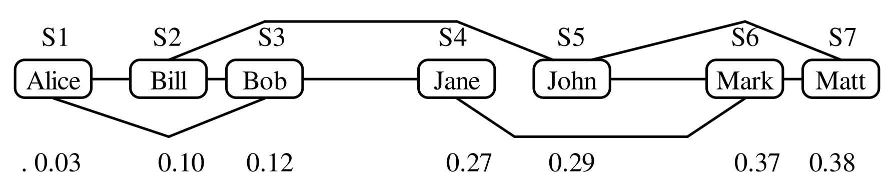

Fig. 3.2. Mapped database keys inserted into a graph

We say that the *length* of an edge between vertices x and y is dist(x, y). We also use terms like *shorter, longer, closer,* etc., in reference to the distance function in the obvious way.

Search Strategies

We are interested in finding efficient algorithms for searching and updating the graph that could be implemented on a distributed system. Each vertex or datum might be stored on a separate server, and each server would have only local knowledge about the graph, as indicated by the edges. For example, server S1 in fig. 3.2 would know the following three facts: that the value of key *Alice* is *123*, that server S2 knows the value of *Bill*, and that server S3 knows the value of *Bob*. A user on S1 wanting to know the phone number for Jane, would only know that map(Jane) = 0.27, but would not know that the data is stored on S4. Only the neighbors of S4 (S3 and S6) would know where *Jane* is stored.

One search technique is called *hill climbing* [Mitchell, 1997]. Given a starting vertex (say, S1), and a goal (say, 0.27), the idea is to traverse the graph along the edges, at each step choosing the neighbor that is closest (in mapping space) to the goal. For instance, from S1 we would go to S3 (Bob) rather than S2 (Bill) because dist(Bob, Jane) = 0.15 < dist(Bill, Jane) = 0.17. From S3, we could then find Jane on S4. Formally, let G be a graph, and let GOAL and START be vertices in G:

```
Hill-Climb(G, GOAL, START)
    Let CLOSEST := START
    For each neighbor X of START do
        If dist(X, GOAL) < dist(CLOSEST, GOAL) then
            Let CLOSEST := X
    If CLOSEST = START then return START (a minimum)
    Else return Hill-Climb(G, GOAL, CLOSEST)
```

Hill climbing either returns the goal or a local minimum, a vertex that is closer to the goal than any of its neighbors (vertices connected by an edge). We define the *smoothness* of the graph as the fraction of hill climbs between all possible n^2 pairs of n vertices that succeed in finding the goal. A graph is *smooth* if it has a smoothness of 1.

The example above is not smooth because hill-climb(S3, "John") = S4, which is the vertex representing Jane. From S3 (Bob), we first go to S4 (Jane) because it is closer to John than any of Bob's other neighbors, S1 or S2 (Alice or Bill). However, we cannot proceed further toward John because every path from Jane to John must first move in the wrong direction, away from John.

One way we can cope with the problem of local minima is by repeating the search from a different starting point picked at random, repeating until successful. Let GOAL be a vertex in graph G:

```
Find(G, GOAL)
    Repeat
        Let START := a random vertex in G
        Let FOUND := Hill-Climb(G, GOAL, START)
    Until FOUND = GOAL
    Return FOUND
```

Find must eventually succeed, if only by the lucky guess START = GOAL.

Obviously we are interested in smooth graphs to minimize the number of iterations in *find*. In one dimensional space, one such graph is an ordered list, in which each vertex is connected to the two closest vertices above and below it in the mapping space. In other words, if we label the n vertices from 1 to n in order of ascending map value, then the i'th vertex (labeled i in fig. 3.3) is connected to the vertices labeled i + 1 and i - 1 (except for vertices 1 and n).

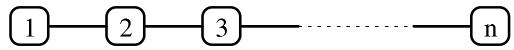

Fig. 3.3. A sorted linked list

Hill climbing on a sorted list of n vertices has O(n) search complexity in the worst case. We can improve search time by adding "shortcuts", or edges connecting more distant points. For instance, given n nodes, we could add n/2 edges of length 2, n/4 edges of length 4, n/8 edges of length 8, and so on, for a total approaching 2n edges. Specifically, if a vertex label has the form $i2^m$, $m \geq 0$, then we add edges to $(i - 1)2^m$, and $(i + 1)2^m$ if those vertices exist. For example:

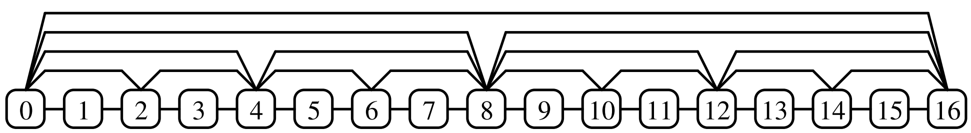

Fig. 3.4. A smooth graph with "shortcut" edges

Storage complexity is still O(n), although there are twice as many edges as an ordered list. Hill climbing now requires O(log n) edge traversals. At worst case, we can go from any node to any other node by taking $\lfloor 2 \log_2 n \rfloor - 1 = O(\log n)$ steps of 1, 2, 4,..., n'/4, n'/2, n', n'/2, n'/4, ..., 4, 2, 1, where n' is the highest power of 2 less than n The first half of the search starts with a step size of 1 or more, followed by steps that at least double in length up to the longest step, no more than n. The second half goes back down, each step at most half the length of the previous one. Usually we can skip some of these steps. For instance, we can hill climb from node 1 to node 14 by traversing nodes 2, 4, 8, and 12, with steps of size 1, 2, 4, 4, 2.

The graph in fig. 3.4 is not well suited for a distributed implementation, even assuming perfectly reliable hardware, because the longer edges are traversed more frequently, increasing the load on the servers at the ends of the longer edges and creating a load imbalance. If an edge connects vertices $i2^m$ and $(i + 1)2^m$, then it is traversed whenever the start and goal are on opposite sides of the edge and the starting vertex is in the range $(i - 1)2^m + 1$ through $i2^m$ or $(i + 1)2^m$ through $(i + 2)2^m - 1$. For example, the edge (8, 12) is traversed whenever the start is in the range 5-8 and the goal is 12 or higher, or the start is 12-15 and the goal is 8 or less. The size of these ranges is proportional to the length of the edge.

Another way to see the problem is to note that hill climbing traverses equal numbers of edges whose length ranges from k/2 to k for all k from 2 to n. For instance, the worst case step sequence (1, 2, 4, ..., n', ..., 4, 2, 1) traverses 2 edges in each range [1,2), [2,4), [4,8), etc. The problem is that there are fewer long edges than short ones. There are n edges in [1,2), n/2 in [2,4), n/4 in [4,8), and so on. To balance the load, we would need to add redundant edges in the higher ranges.

Since there would be n edges in each range, and $\log_2$ n ranges, storage complexity (since edges require storage space too), would be O(n log n).

Returning to fig. 3.4., it should be clear that we could randomly remove some of the shortcut edges without reducing smoothness (as long as we retain the sorted list as a subgraph), and that hill climbing would still be faster than the plain sorted list. In fact, we could just place the shortcut edges randomly and still see an improvement. This is called a *smooth randomized graph* or SRG.

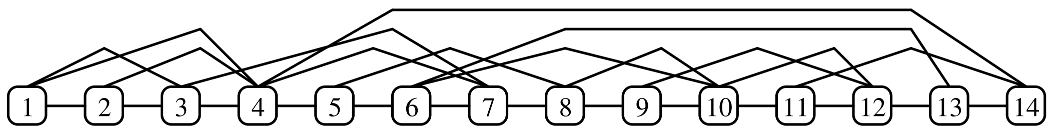

Fig. 3.5. A smooth randomized graph

Update Algorithm

It would be difficult to add or delete a vertex in a regular structure such as fig. 3.4, but the process is simple for a randomized graph such as fig. 3.5. To delete a vertex, we simply remove it along with its adjacent edges. Because the distribution of edges by length is the same (on average) for each vertex as for the graph as a whole, deletion should not affect this distribution. Deletion may reduce smoothness, causing some degradation of the *find* algorithm.

To add a vertex, we insert it with no initial edges, then search for it from several random locations. Whenever a search returns a local minimum instead of the goal, we add an edge between the minimum and the goal so that a future search from the same point would succeed. We repeat the tests until some fraction (say, 1/2) of the tests succeed.

Let G be a graph. Let GOAL be the vertex to add:

```
Update(G, GOAL)
        Add GOAL to G with no connecting edges
        Repeat until hits > misses
                Let START := a random vertex in G
                Let FOUND := Hill-Climb(G, GOAL, START)
                If FOUND = GOAL then count a hit
                Else add an edge between FOUND and GOAL and count a miss
```

We need to show that (1) *update* preserves smoothness, (2) it preserves a distribution that allows efficient (logarithmic) search, and (3) that *update* itself is efficient. Recall that for the one-dimensional case, the proper distribution of edge lengths to achieve load balancing is one with equal numbers of short and long edges. More precisely, we need an approximately uniform distribution of edge lengths over intervals [k/2, k) for all k in [2/n, 1] for an n-vertex graph mapped into [0, 1]. Since there are $\log_2$ n disjoint subintervals of the form [k/2, k) in [1/n, 1], and we need about n edges in the first interval [1/n, 2/n) to have an ordered list as a subgraph to ensure smoothness, there must be n $\log_2$ n edges.

We now argue (2), that *update* preserves a uniform distribution of edge lengths (on a logarithmic scale) when updates are randomly distributed in one dimension. Without loss of generality, we consider the mapping space [0, 1]. If the graph has n vertices, then the average distance between adjacent vertices (in space) is 1/n. Given an initially smooth graph G and an update vertex C, let A be the closest vertex in mapping space, and let B be the closest vertex in the opposite direction from C (see fig. 3.6). Because G is smooth, the first iteration of *update(G, C)* will add an edge between A and C, because hill climbing from any vertex in G (except C) will reach a minimum at A. Subsequent hill climbing will then succeed from all starting vertices and *update* will terminate. Note that G is no longer smooth, as hill climbing from C to B fails.

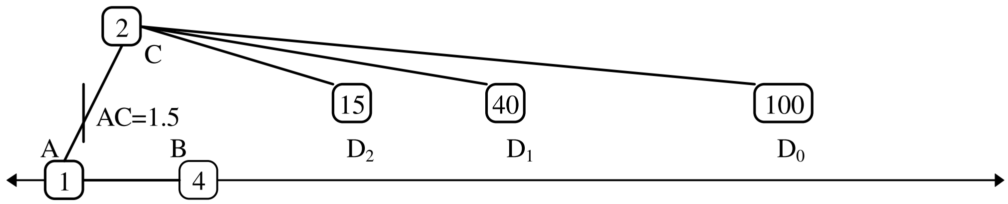

Fig. 3.6. How *update* adds long edges

Let D_0 be the next vertex to be updated such that D_0 is closer to C than to A in mapping space (to the right of AC, the point between A and C, in fig. 3.6) and the update algorithm randomly chooses a starting vertex so that the search path passes through C. The search fails, so an edge is added between C and D_0. Because map(C) has a mean of 1/2, the mean of dist(C, D_0) is 1/2 the distance to the boundary of the mapping space, or 1/4.

Let D_i, i > 0, be the first vertex to be updated after D_{i-1} such that D_i is closer to C than to D_{i-1} or A (between AC and the point midway between C and D_{i-1}), and the update algorithm randomly chooses a starting vertex that results in hill climbing through C. Then the search will fail and an edge will be added between C and D_i. Note that if the update is anywhere else, then hill climbing can proceed from C to either A or D_{i-1}. The distance from C to D_i is between 0 and 1/2 of dist(C, D_{i-1}) or 1/4 of dist(C, D_{i-1}) on average. (We are neglecting the small contribution of dist(A, C) ≈ 1/n on the probability distribution).

If we now count the D_i vertices, whose mean distances are 1/4, $1/4^2$, $1/4^3$, ... 1/n, we find that there are about $\log_4$ n of them. Of course, this analysis assumes that the random position of D_i is uniformly distributed, but it may not be because its position may affect the probability that the update algorithm will route a test search through C. If D_i is further away, then it is more likely that an edge could route the search around C that would not be taken of D_i were closer. This is illustrated below:

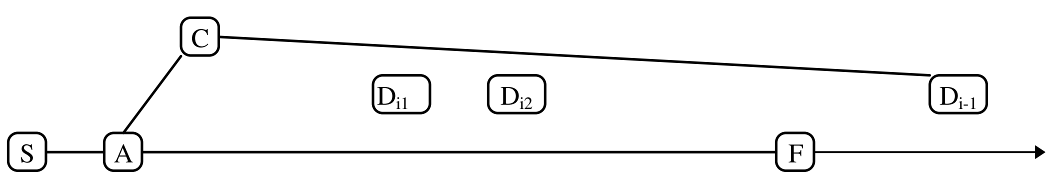

Fig. 3.7. How *update*d edges may be distributed nonuniformly

Both D_{i1} and D_{i2} are closer to C than to D_{i-1}, so both update(S, D_{i1}) and update(S, D_{i2}) would result in an edge to C, except that there may be another vertex F such that D_{i1} is closer to C than F but D_{i2} is closer to F than C. Therefore, the next D_i in the sequence will be D_{i1}, the one closer to C. This effect biases the distribution of edges to C so that each is less than 1/4 of the length of the previous one. If the effect is independent of scale, then we can still say that E(dist(C, D_i)) = E(d(C, D_{i-1}))/k for some k (not necessarily 4). The expected values of distances from C form the sequence 1/k, $1/k^2$, $1/k^3$, ..., 1/n, which has about $\log_k n$ = O(log n) elements. Furthermore, the distribution of edges is still the one required for load balancing in a distributed environment: equal numbers of edges of each length on a logarithmic scale.

We have argued (but not proved) (2) above, that search is efficient, but we have not shown that (1), update preserves smoothness. In fact it does not. In fig. 3.6, a search for B from A (or from its left) fails after C is updated. It may be necessary to periodically update old data to restore lost smoothness.

We are now ready to analyze the average case complexity of *find* and *update* in 1-dimensional space. We have just argued that storage complexity is O(n log n), or that the average degree (edges per vertex) is O(log n). From our previous discussion, the average path length for hill climbing is O(log n), but at each step, we must choose from among O(log n) neighbors to find the one closest to the goal. As a result, hill climbing is $O(\log^2 n)$. The *find* algorithm repeats hill climbing until successful, so its complexity is $O((\log^2 n)/\text{smoothness})$. But *update* runs until at least 1/2 of search tests succeeds, implying a smoothness between 1/2 and 1. Therefore *find* has average time complexity of $O(\log^2 n)$.

The *update* algorithm runs until 1/2 of search tests succeed. Since each failure results in an edge, and the average degree is O(log n), this implies O(log n) successes and O(log n) failures. Since each test is $O(\log^2 n)$, *update* has average time complexity of $O(\log^3 n)$.

Randomized Hill Climbing

There are more efficient query and update algorithms than the ones based on hill climbing. Suppose that in the innermost loop, instead of comparing each neighbor to find the closest to the goal at each step, we take a step as soon as we find a neighbor that is any closer, abandoning the rest of the comparisons. We would need to traverse more edges, since we would make less progress at each step by choosing a sub-optimal neighbor, but we would do fewer comparisons at each step.

Let G be a graph. Let GOAL and START be vertices in G.

```
Randomized-Hill-Climb(G, GOAL, START)
    Let S be the set of neighboring vertices of START
    While S is not empty, loop
        Remove vertex V from S, choosing randomly
        If dist(GOAL, V) < dist(GOAL, START)
            Then return randomized-hill-climb(G, GOAL, V)
    Return START
```

Recall the uniform distribution of edge lengths on an exponential scale. The distribution looks something like this.

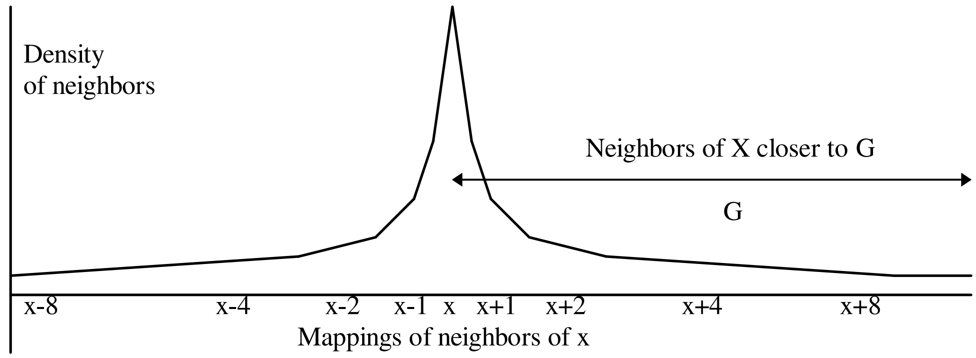

Fig. 3.8. Distribution of neighbors of vertex x in **R**

Deterministic hill climbing, like a binary search, converges exponentially toward the goal, taking large steps at first, and then successively smaller ones. If the goal is mapped to, say, X + 5, then hill climbing would take the best step possible, say X + 4. Randomized hill climbing, on the other hand, might have several other choices, say X + 1, X + 2, X + 8, etc. At first, about half of the neighbors make forward progress, but this fraction decreases as we approach the goal.

Suppose that n vertices are mapped uniformly into an interval in **R**, and each vertex has an average of k = O(log n) neighbors distributed as above. Then at the start of randomized hill climbing, almost k/2 neighbors make forward progress, so we do 2 comparisons per step. If we approach the goal at an exponential rate, then at each step the number of neighbors that makes forward progress decreases at a constant rate, so the total number of comparisons is

$$k/(k/2) + k/(k/2 - 1) + \ldots + k/3 + k/2 + k/1 \approx k \ln k/2 = O(\log n \log \log n)$$

However, we approach the goal at a slightly less than exponential rate because we take sub-optimal steps. We later show empirical evidence which suggests that the overall complexity of randomized hill climbing is $O(\log n \log^2 \log n)$, probably because the average search path has increased by a factor of O(log log n). Nevertheless, this is an improvement. Since *update* still performs O(log n) tests, its complexity would be O(log n) higher than search, or $O(\log^2 n \log^2 \log n)$, again an improvement over deterministic hill climbing for the average case.

Multidimensional Data

Often we find it useful to map data into more than one dimension. For instance, if we keyed weather data by longitude and latitude, we could compose queries of the form "give me the weather forecast for all cities within 100 miles of Chicago". In a text retrieval system, we would map documents into a very high number of dimensions (one per word or term) so that we could request documents that are "similar" to some other query or document, i.e., containing many words in common.

We now give examples of regular data structures of n vertices in d-dimensions, $1 \le d \le \log_2 n$ that have the same storage and search time complexities as the one-dimensional spatial index described above. The simplest example is a hypercube in $d = \log_2 n$ dimensions. A hypercube is a graph G = {V, E}, $V = \{0, 1\}^d$,
$E = \{(a, b): a, b \in V, \text{dist}(a, b) = 1\}$. The following is a 4-dimensional hypercube (d = 4).

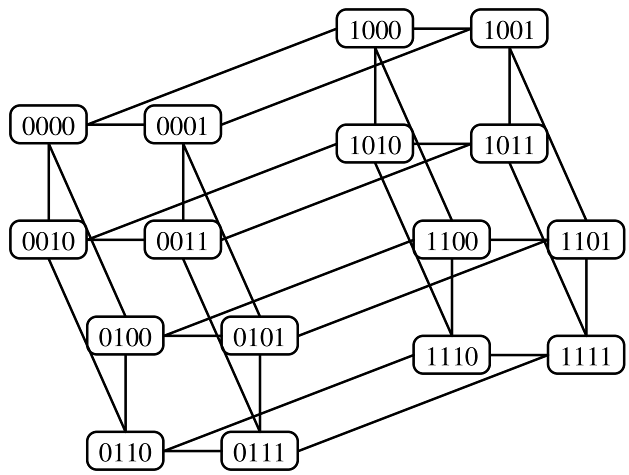

Fig. 3.9. A 4D hypercube

Each vertex has $d = \log_2 n$ neighbors, so there are $n/2 \log_2 n$ edges, or O(n log n) storage complexity. Hill climbing traverses at most d edges, and must examine d neighbors for each traversal, so search is $O(d^2) = O(\log^2 n)$ worst case. Randomized hill climbing also traverses d edges, but because path length does not increase, complexity is O(log n log log n) by our previous argument. We do not analyze update complexity, because the algorithm is not applicable to regular structures.

When the number of dimensions is between 1 and $\log_2 n$, we can construct a hypergrid and use "shortcut" edges in each dimension to reduce search complexity. We define a hypergrid of order k in d dimensions as a graph G = (V, E): $V = \{0, 1, 2, ..., k\}^d$, $E = \{(a, b), a, b \in V, \text{dist}(a, b) = 1\}$. Vertices are regularly spaced in d dimensions, with at most 2d neighbors (one on either side in d dimensions). A hypercube is a special case of an order k = 1 hypergrid.

An order k hypergrid in d dimensions has $n = (k + 1)^d$ vertices and nd edges, so has O(nd) space complexity. Worst case search complexity is at least O(dk), because the path length between diagonally opposite corners (such as $\{0\}^d$ and $\{k\}^d$) is dk.

We could reduce the maximum path length to O(d log k) by adding "shortcut" edges along each dimension as we did for the 1-dimensional list. In the following example (k = 4, d = 2), every pair of vertices is connected by a path of length
$d \log_2 k = 4$ or less.

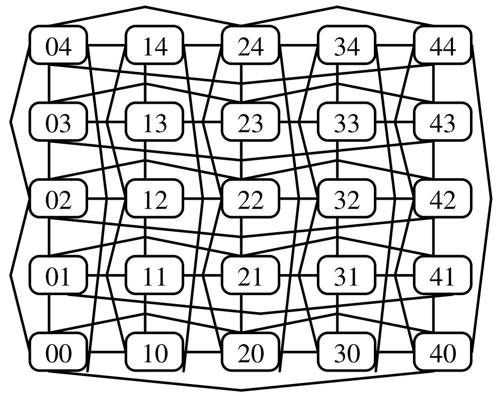

Fig. 3.10. An order-4 2D hypergrid with shortcut edges.

The rule is that if two vertices $a = (a_1, a_2,..., a_d)$ and $b = (b_1, b_2,..., b_d)$ differ in only one dimension, i ($a - b = (0, 0, ... a_i - b_i, ..., 0)$), and $a_i = c2^m$, and $b_i = (c \pm 1)2^m$ for some integers c and m, then a and b are neighbors. However, it should be clear that *any* edges added to a hypergrid will reduce the average number of edges that must be traversed during hill climbing.

Hill climbing in d dimensions can be reduced to d cases of hill climbing in one dimension From our previous analysis of one-dimensional lists, we saw that we could obtain logarithmic search complexity by doubling the number of edges, but that this technique did not extend to distributed systems because the longer edges would be more heavily traveled. Once again, this is the case. To achieve load balancing, we must increase the number of edges along rows in each dimension by O(log k). Since there are $n = (k + 1)^d$ vertices, and the average degree is $O(d \log k) = O(\log k^d) = O(\log n)$, we have a space complexity of $O(n \log n)$.

Hill climbing in d dimensions, requires O(log k) steps in each of d dimensions, or $O(d \log k) = O(\log n)$ steps. Each step examines O(log n) neighbors, so time complexity is $O(\log^2 n)$, just as in the one dimensional case. Randomized hill climbing examines fewer neighbors, but may traverse more edges, but we should expect a complexity between O(log n) and $O(\log^2 n)$.

Summary

We described data structures for mapping a database index into a space. We described two algorithms, hill climbing and randomized hill climbing, for searching the graph, though other algorithms are well known. These algorithms are greedy, making decisions based on local knowledge. The effectiveness of these algorithms (do they succeed?) depends on a property we called smoothness. Efficiency (how fast do they report success or failure?) depends on the distribution of edge lengths. Logarithmic search complexity is possible with O(n) storage in a centralized implementation, but this is not known to be true for distributed implementations. It appears that O(n log n) storage is required but we have no proof of this.

For the special case of one-dimensional data, we analyzed search and update complexity using both deterministic and randomized hill climbing, based on the assumption of O(n log n) edges and a uniform distribution (on a logarithmic scale) by length. We presented a plausible argument that the update algorithm generates the necessary distribution, but a proof would require that we show that the distribution of new vertices affecting the addition of edges to an older vertex is independent of the configuration of the graph on average, or at least that the distribution is independent of scale.

Finally, we looked at special cases from 1 up through log n dimensions, showing O(n log n) storage complexity, O($\log^2$ n) search complexity using deterministic hill climbing, and between O(log n) and O($\log^2$ n) search complexity using randomized hill climbing. These should not be considered proofs, because we have not shown that the analyses can be extended to the randomized graphs that would be produced byt the update algorithm in multidimensional space.

Section 4

Complexity Measurements

In section 3 we described algorithms for searching and updating spatially indexed data using a randomized graph with high smoothness. We have also calculated average space and time complexity under certain conditions, and found them to be worse than what could be obtained in a non-distributed implementation. The special cases we examined were n vertices in d-dimensional spaces, $1 \leq d \leq \log_2 n$, and the vertices are distributed either uniformly at random in one dimension, or regularly in more than one dimension. Under these conditions, space complexity is O(n log n), search complexity is $O(\log^2 n)$ or better (but worse than O(log n)), and update complexity in one dimension is $O(\log^3 n)$ or better (but worse than $O(\log^2 n)$).

Those were average cases, but we can easily analyze worst case complexity for the general case:

Query time: O(n) for smooth graphs, infinite for non smooth graphs. In the smooth case, the following would be built by adding vertices in ascending order in one dimensional space:

1 --- 2 --- 3 --- ... --- n

The update algorithm does not guarantee smoothness, however. Consider a disconnected graph:

1 2

Such a graph could result if we add node 2 using the update algorithm, then choose node 2 as the random starting point for testing. The test succeeds, so we have one hit and no misses, and we stop. A query for 2 will eventually succeed by randomly choosing 2 as the start, but there is no guarantee of when.

We can reduce the incidence of disconnected updates by forcing a minimum number of tests, say m. This reduces the probability of a disconnected update to $1/n^m$.

Update time: O(n) to construct a fully connected graph. It is possible to construct a graph G = (V, E) such that dist(a, b) = C > 0 for some constant C and for all $a \neq b$ in V, in which case the only possible smooth graph is a fully connected one. This could happen with d = n dimensions, where the i'th vertex has a value of the form 0000...001000..., with n coordinates set to 0, except for the i'th coordinate, which is set to 1. The distance between every pair of distinct vertices is the same ($2^{1/2}$), therefore hill climbing (either deterministic or randomized) will fail if the start and goal vertices are not already connected.

Storage: $O(n^2)$ for the edges of a fully connected graph.

The purpose of the next subsection is to experimentally determine the complexity for cases of interest: randomly distributed data in d-dimensional space.

Experimental Setup

We now construct smooth randomized graphs and estimate complexity empirically. For our first test, we first insert all of the vertices to build an n-vertex graph with no edges. Then we adaptively add edges by picking pairs of distinct vertices and searching for one from the other by deterministic hill climbing, adding an edge between the goal and any local minima found, repeating until exactly half of the tests succeed. Finally we test the graph by sequentially searching for every vertex from random starting points, not giving up until the goal is found. The data is uniformly distributed in d-dimensional space, $[0, 1]^d$.

```
BUILD-SRG:
    Create a disconnected graph of n vertices
    While hits ≤ misses do
        Pick 2 distinct vertices START ≠ GOAL at random from G
        Let FOUND := Hill-Climb(G, GOAL, START)
        If FOUND ≠ START then connect them and count a miss
        Else count a hit
```

Note that this is different than using the update procedure described in section 3. Although the update procedure is adaptive, it only ensures the accessibility of a vertex from previously added vertices, not from vertices added later. In an actual implementation, it might be better to continuously test the availability of data as the graph changes, and add edges as needed so that data remains accessible at all times.

Also, the update algorithm does not require that the start and goal vertices be distinct. We chose distinct vertices to avoid the pathological case of counting a hit on the first iteration, which could occur with probability 1/n for n vertices. Another solution would be to force a minimum number of iterations. Either solution should have a negligible effect on the results when n is large.

Deterministic Hill Climbing

Space Complexity

For our first experiment, we build graphs of $n = 2^3, 2^6, ..., 2^{18}$ vertices in spaces of $d = 1, 8, 64$, and 512 dimensions. We then plot the average degree (2 edges/n) vs. n for each of the four values of d. For clarity, we also plot the point n = 1, where we know that degree = 0. Note that the X axis (n) is logarithmic, and the Y axis (degree) is linear.

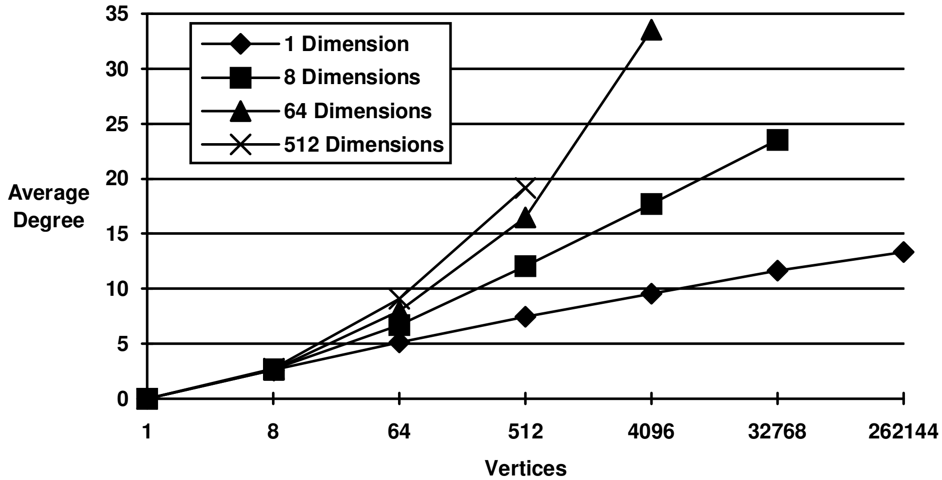

Fig. 4.1. Average degree per vertex.

We observer the following. For d = 1, the line is straight, suggesting that degree increases logarithmically with n, confirming our earlier expectations. In fact, the measured degree is very close to ln n + 1, as seen in the following table:

n	Degree	ln n + 1	Trials
1	1.00	0.00	theoretical
8	3.08	2.62	average of 128
64	5.16	5.11	average of 16
512	7.24	7.47	average of 2
4096	9.32	9.57	1
32768	11.40	11.67	1
262144	13.48	13.32	1

Table 4.1. Measured degree in [0, 1]

Although this suggests a relationship, it is by no means proof of O(log n) degree. Our worst case analysis has shown only that the degree is at least 2(n - 1)/n or O(1) (a tree) and at most n - 1 or O(n) (fully connected). This allows us to reject quadratic and higher order polynomial functions as candidates. (In any case, fitting a quadratic function to the data would result in a negative coefficient for the n^2 term). Our analysis of *regular* graphs with shortcut edges allowing binary search in one dimension has shown that an O(log n) relationship would be reasonable. However, we cannot conclude, based on the data, that the actual relationship does not have higher complexity. For instance, we could conceivably have within the worst case limits:

$$\text{degree} = \ln n + 1 + \varepsilon n^k = O(n^k),\ k \le 1$$

or based on the data alone,

$$\text{degree} = \ln n + 1 + \varepsilon e^n = O(e^n)$$

for some very small constant ε.

When we repeat the experiment for vertices mapped to higher dimensional space, $[0,1]^d$, where d = 64 or 512, and plot the average degree as before (fig. 4.1), we find that the line curves upward away from the d = 1 line, suggesting a growth rate greater than O(log n). The interesting curve is for d = 8, which curves upward at first, then straightens out around the middle of the graph (at about $n = 2^8 = 256$), again suggesting O(log n) growth. In this case, we can fit the following approximation:

n	degree	$8^{1/2}$ ln n - 5.6	Trials
1	0.00	-5.60	theoretical
8	2.69	0.28	average of 16
64	6.73	6.16	average of 2
512	12.06	12.04	1
4096	17.70	17.93	1
32768	23.55	23.81	1

Table 4.2. Measured degree in $[0, 1]^8$

In other words, we find that when $d < \log_2 n$, that

$$\text{degree} \approx d^{1/2} \ln n + C$$

for some constant C. When d = 1, we have C = 1, or degree = ln n + 1. When d = 8, we have C = -5.6, or degree = 2.828 ln n - 5.6. Due to the long simulation times for large n and large d (up to one day on a 100 Mhz 486 PC), it was not feasible to collect enough data points to find a plausible equation for the constant C.

The results suggest that when $d < \log_2 n$, space complexity is O(n log n), since the number of edges is (n)(degree) = O(n)O(log n).

Time Complexity

We now wish to estimate search time complexity, which is:

(comparisons per vertex)(edges per search) / smoothness

When using deterministic hill climbing, comparisons per vertex is the same as the average degree, which we believe to be O(log n) for $[0, 1]^d$, $d \leq \log_2 n$. The number of edges traversed per search should be O(log n) as long as degree > 2, because we know that the number of vertices reachable in m steps through vertices of degree k is at most $(k - 1)^m$. Smoothness should be greater than 1/2 because BUILD-SRG runs until 1/2 of the randomly picked searches succeeds.

Nevertheless, we will measure these quantities. When running BUILD-SRG, we count the number of edges traversed by hill climbing, including any newly inserted edges, and divide by the number of hill climbs performed (hits + misses). This data is taken from the same experiment used to construct fig. 4.1. We have plotted the average number of edge traversals per hill climb (Y axis) against n (X axis) in the 1, 8, 64, and 512 dimensional spaces $[0, 1]^d$.

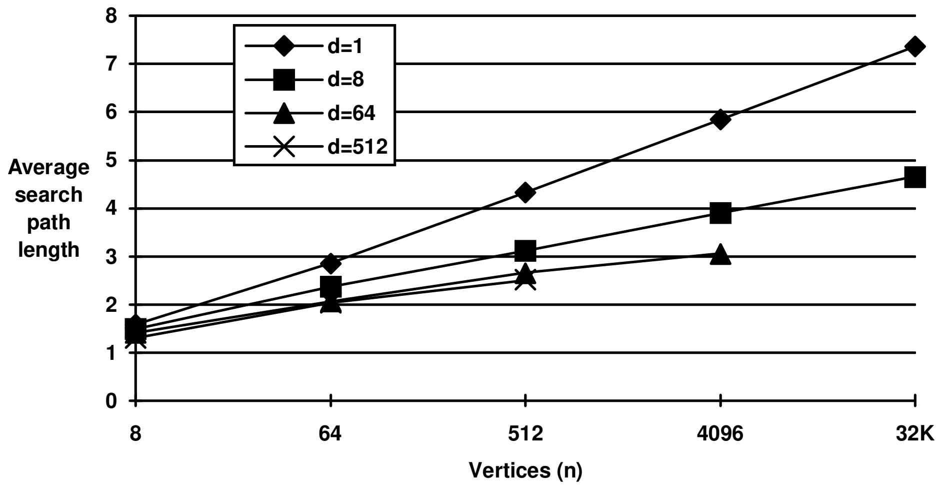

Fig. 4.2. Average number of steps for deterministic hill climbing.

We observe a logarithmic growth rate. In one dimension, average path length is approximately ln n - 2, and less in higher dimensions. This is to be expected because the average degree is higher.

We next measure a quantity related to smoothness, again as part of the same experiment. We do an exhaustive search for every vertex using hill climbing, and count the number of failures. The technique is to go sequentially through each vertex in the graph and search for it from random starting points until we find it.

```
For each vertex v in G do
        While Hill-Climb(G, v, random vertex in G) ≠ v do
                Count 1 failure
```

This is a more severe test of smoothness (which we know to be at least 1/2 by construction), because we could spend a lot of time on one single hard-to-find vertex. The graph below plots the average number of failures per vertex. This number would be 0 in a perfectly smooth graph, and at least 1 in a graph with smoothness 1/2.

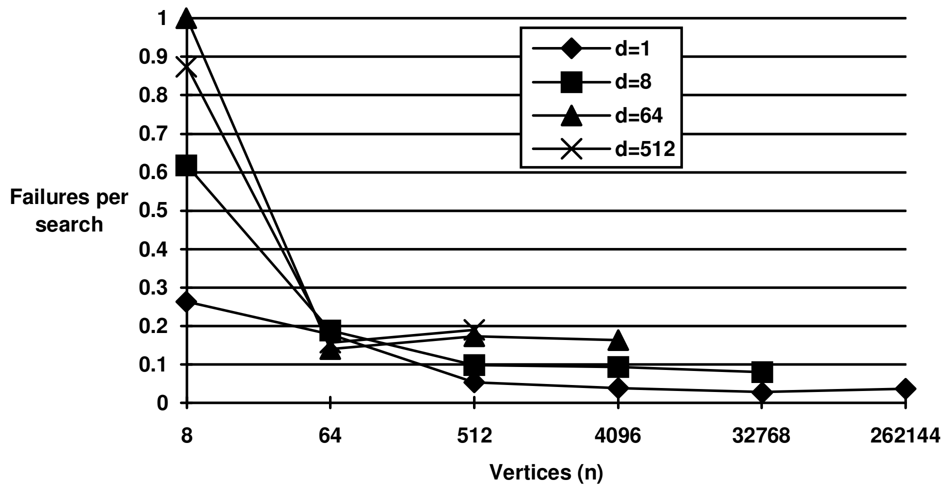

Fig. 4.3. Average number of failures before finding each vertex

The experiment shows that smoothness increases toward 1 as n grows, a reassuring result. The data therefore supports $O(\log^2 n)$ search complexity because edge traversals per hill climb and comparisons per edge traversal are both $O(\log n)$, and 1/smoothness is $O(1)$. If this is so, then update is $O(\log^3 n)$ because we perform $O(\log n)$ hill climbs per update (one hit and one miss per neighbor).

Randomized Hill Climbing

In the next experiment, we replaced hill climbing in BUILD-SRG with randomized hill climbing, as described in section 3. Recall that randomized hill climbing chooses the first neighbor that makes forward progress instead of the neighbor making the most progress, so makes fewer comparisons at each step. The experiment was conducted only on one-dimensional data with vertices randomly distributed in [0, 1] with uniform probability as before. We counted hill climbs and edge traversals as before, and also counted test comparisons in the innermost loop of the randomized hill climbing algorithm (because this number is now less than the average degree). We plotted average degree (neighbors per vertex), average path length (edge traversals per hill climb including edge insertions), and average number of distance comparisons per edge traversal. Each point represents a single trial.

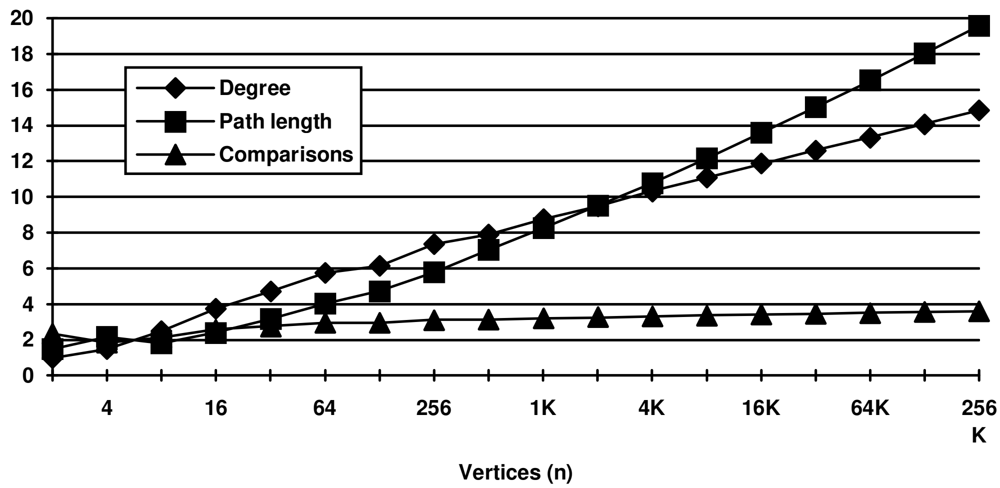

Fig. 4.4. Complexity of nondeterministic hill climbing for 1 dimensional data.

We then found equations that approximated the data: The procedure was to start with simple equations predicted from section 3 or from earlier experiments (i.e. guessing degree = ln n), then subtract or divide these predictions from the actual data and make adjustments. The process was done manually with the help of computer generated tables. Errors were not analyzed statistically. One set of closely fitting formulas is:

- degree = $\ln n + \ln 2 \ln^{1/2} n = O(\log n)$
- path length = $5/8 \ln n \ln \ln n = O(\log n \log \log n)$
- comparisons = $(1 + 1/n^{0.288}) \ln \ln n + 1 = O(\log \log n)$

The closeness of the fit as n grows can be seen more easily by examining the actual data and comparing with the formulas.

n	Degree Actual	Formula	Path Len. Actual	Formula	Compares Actual	Formula
2	1.00	1.27	1.50	−0.16	2.33	0.33
4	1.50	2.20	2.17	0.28	1.85	1.55
8	2.50	3.08	1.80	0.95	2.14	2.13
16	3.75	3.93	2.40	1.77	2.58	2.48
32	4.69	4.76	3.16	2.69	2.76	2.70
64	5.75	5.57	4.03	3.70	2.95	2.86
128	6.13	6.38	4.74	4.79	2.96	2.97
256	7.34	7.18	5.79	5.94	3.12	3.06
512	7.90	7.97	7.03	7.14	3.15	3.13
1K	8.77	8.76	8.24	8.39	3.20	3.20
2K	9.46	9.54	9.50	9.68	3.25	3.26
4K	10.32	10.32	10.76	11.01	3.31	3.31
8K	11.09	11.09	12.16	12.38	3.36	3.36
16K	11.85	11.86	13.58	13.78	3.41	3.41
32K	12.61	12.63	15.04	15.22	3.46	3.46
64K	13.34	13.40	16.51	16.68	3.51	3.50
128K	14.08	14.16	18.05	18.17	3.55	3.55
256K	14.85	14.93	19.58	19.68	3.60	3.59

Table 4.3. Experimental and theoretical degree, edge traversals per search, and comparisons per edge traversal using randomized hill climbing to build an SRG.

The product of degree, path, and comparisons yields a search complexity of $O(\log n \log^2 \log n)$. Update still requires $O(\log n)$ searches per new vertex, or $O(\log^2 n \log^2 \log n)$ complexity. Space complexity remains $O(n \log n)$.

Summary

We conducted experiments to estimate space and time complexity of smooth randomized graphs with n uniformly distributed vertices in d-dimensional space using a modified update algorithm. From the data we estimate the following complexities, when $d \le \log_2 n$:

	Deterministic Hill Climbing	Randomized Hill Climbing
Storage	$O(n \log n)$	$O(n \log n)$
Search time	$O(\log^2 n)$	$O(\log n \log^2 \log n)$
Update time	$O(\log^3 n)$	$O(\log^2 n \log^2 \log n)$

Table 4.4. Summary of measured SRG complexities

We did not characterize the cases of $d > \log_2 n$, but complexity is apparently higher.

Section 5

Concurrent Access Protocols

In a distributed system, the data represented by vertices, and the pointers represented by edges, may be stored on different servers in a connected network. Because we can easily simulate a parallel system on a sequential one (but not the other way around), we can, without loss of generality, describe a system in which every vertex is stored on a different server.

A pair of connected vertices:

$$X_1 \text{ --- } X_2$$

on servers S_2 and S_2 is modeled by storing on S_1 both the value Y_1 of key X_1 (indicated $X_1=Y_1$) and the location or network address S_2 of key X_2 (indicated X_2:S_2). The link is bi-directional, so that S_2 stores the corresponding information about S_1.

$$S_1 = \{X_1=Y_1, X_2:S_2\} \text{ --- } S_2 = \{X_2=Y_2, X_1:S_1\}$$

We describe a three layer protocol as one possible distributed implementation of a smooth randomized graph. The protocol makes no distinction between queries (X=?) and updates (X=Y). In both cases, the objective is to create a vertex X on some server and route messages to it from random points in the network. Because these messages (and acknowledgments) carry local information about the graph that can be stored, they effectively add edges between the senders and receivers. Once a query vertex has been inserted into the graph, the answer can be extracted from its neighborhood using a regional traversal. The query can then be deleted, or could be left in place to signify an interest in receiving future updates.

The first level protocol is the one that adds edges to the graph. It also repairs inconsistencies and removes edges from dead or congested servers. The second level protocol implements the update algorithm by routing messages from random points. The third level protocol extracts the answers from the newly updated query.

Both the first and third level protocols can cache data to improve efficiency. Caching is optional with individual servers, but can result in the propagation of the answers to frequently asked questions to many parts of the network, greatly improving response time and reliability. Because caching introduces the possibility of retrieving inconsistent copies, we add timestamps and signature routing to all data.

First Level Protocol: Consistency, Flow Control, Preemptive Caching

Servers can become inconsistent in any number of ways. For instance, a vertex may be updated, inserted, or removed without notifying neighbors. Vertices may also disappear temporarily or permanently due to hardware failure or congestion. If a server does not respond, we don't know if the server has disappeared or if it is merely slow, but in either case we may wish to avoid it in the future.

The first level protocol repairs inconsistencies by removing whatever each server knew about the other (right or wrong) and replacing it with fresh data. Every message and acknowledgment at this level includes the key, X, of the sender. It is assumed that the receiver can identify the sender as well. When server S_1 sends X_1 to S_2, it deletes the outgoing edge, X_2:S_2, if any. When S_2 receives X_1 from S_1, it adds the edge X_1:S_1 to its database, replacing any previous edge to S_1, and sends an acknowledgment X_2 back to S_1. When S_1 receives the acknowledgment from S_2, it adds back the edge X_2:S_2 to its database. We are now guaranteed that S_1 and S_2 will be consistent and be connected by an edge, regardless of their previous states of either.

In the following example, the keys on both servers (Alice and Bob) have each been updated without the knowledge of the other server, producing two inconsistencies.

S_1={Alice=123, Bill:S_2} ---- S_2={Bob=456, Alicia:S_1}

S_1 sends "Alice, (rest of message)" to S_2
S_1 removes the edge to S_2
S_2 updates edge to S_1 with key "Alice"

S_1={Alice=123} <---- S_2={Bob=456, Alice:S_1}

S_2 replies "Ack:Bob" to S_1
S_1 adds edge to S_2 with key "Bob"

S_1={Alice=123, Bob:S_2} ---- S_2={Bob=456, Alice:S_1}

If server S_2 is down, then S_1 will never receive an acknowledgment, and remove the edge to it. When a message is routed through S_1 as part of a search at a higher level protocol, the message will be routed to some other server. Later when S_2 comes back up, the edge would be added back when a message happens to be routed from S_2 to S_1. If S_2 is simply responding slowly due to being overloaded, then this protocol achieves flow control by routing messages away from S_2 until it is ready.

For example, in fig. 5.1, two messages for X_1 and X_2 are routed through S_1 at the same time. Either message could be routed through either S_2 or S_3, although S_2 is closer (in 2-d space). If S_1 routes X_1 to S_2, then receives X_2 before the acknowledgment from S_2, then it has no choice to route X_2 to S_3 until the edge to S_2 is restored.

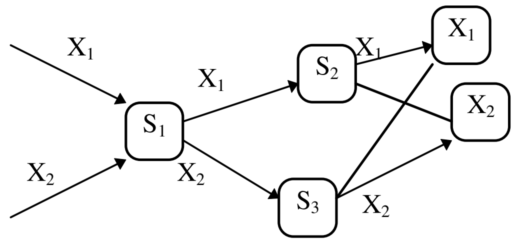

Fig. 5.1. Flow control. After S_1 sends message X_1 through S_2, it must route around it until S_2 returns an acknowledgment.

Note how caching could easily be added to the protocol by including the values along with the keys, i.e. sending "Alice=123" and responding "Ack:Bob=456", then storing these values:

$$S_1=\{Alice=123, Bob=456:S_2\} \text{ ---- } S_2=\{Bob=456, Alice=123:S_1\}$$

This is preemptive caching because the data being cached has not yet been requested.

Second Level Protocol: Updates

We are now ready to describe the query/update protocol. Suppose server S wishes to query X or advertise that it knows X=Y. Then at the second level protocol, S would send a message containing X to several random locations, with instructions to route each messages back to itself, using a directed search if possible. At the minima, the message returns to X, creating an edge. If X is an update (X=Y), then we are done. If X is a query (X=?), then we would obtain the answer from the neighborhood of X using the third level protocol.

> Create a vertex S={X=?} or S={X=Y}
> Send (S, X) to several (O(log n)) random servers.
> At each server $\{X_0=Y_0, X_1:S_1, X_2:S_2, ...\}$ receiving (S, X), do concurrently
>> Find X_c in $\{X_0, X_1, X_2, ...\}$ closest to X
>> If c = 0 then send () to S (a minima)
>> Else send (S, X) to S_c

The example in fig. 5.2 shows the effect of routing one message (S, X) to the new vertex S starting at S_1. Hill climbing toward X proceeds through S_2 and S_3, then reaches a minimum. At this point, the message is routed to S, taking the address from the message. The first level protocol then adds an edge between S_3 and S.

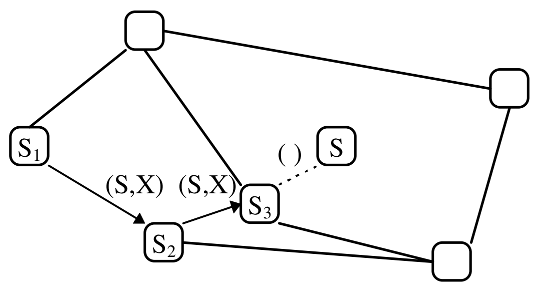

Fig. 5.2. Update protocol. Testing access to S from S_1 results in an edge being added between S and S_3.

The algorithm does concurrent hill climbing, except that it bypasses nodes for which it is waiting for an acknowledgment from a previous message. It could also do a randomized search by substituting at line 4:

> Pick X_c in $\{X_0, X_1, X_2, ...\}$ with probability inversely related to dist(X, X_c)

We should note that picking a random server at line 2 is not trivial; it would require knowledge of every server in the network. One way to implement this step is to start at a known server and traverse several (O(log n)) edges randomly as above, but using a uniform distribution. We could add a hop count to the message, incremented by each server, to decide when to switch from randomized routing to directed routing.

Third Level Protocol: Query Processing and Multi-Level Caching

At this point, we have inserted X into the graph. If X is an update, then we are done. If X is a query (X=?), then we still need to extract the answer Y from the neighborhood of X.

If, during insertion, we encountered a vertex X=Y, then this is surely a minima since dist(X, X) = 0. Therefore, this vertex will be a neighbor of X=?. One way to extract the data is:

For each neighbor Si={X_i=Y_i} of S={X=?} do
    If dist(X_i, X) = 0 then return Y_i

Note that if the first level protocol implements caching, then all of the needed information is already stored on S. If not, then Y_i could be returned when the message is routed back to S in the last step of the second level protocol.

In the case of a neighborhood query (such as for text retrieval where distance determines relevance), we are now interested in all strings X_i such that
dist(X, X_i) ≤ T, where T is some threshold. A key=value mapping is the special case of T = 0. An update is the special case of T < 0.

Suppose we wish to find all data within some threshold distance T of point Q in graph G. One sequential algorithm is:

```
Update(G, Q)
Query(G, Q, Q, T)
Remove Q from G
```

where

```
Query(G, Q, V, T)
    If V is not tagged and dist(Q, V) < T then
        Output V
        Tag V
        For each neighbor X of V do
            Query(G, Q, X, T)
```

We treat a query as a temporary update where we store only the key. Once *update* connects the query to the graph, we can retrieve the relevant data from its neighborhood. We can then remove the query vertex and its edges (or leave them in place to signify an interest in being notified of future updates).

The *query* algorithm does a recursive depth-first traversal of the neighborhood of Q within the hypersphere of radius T. In this example (where d = 2 dimensions), the neighbors of Q might be traversed in the order shown.

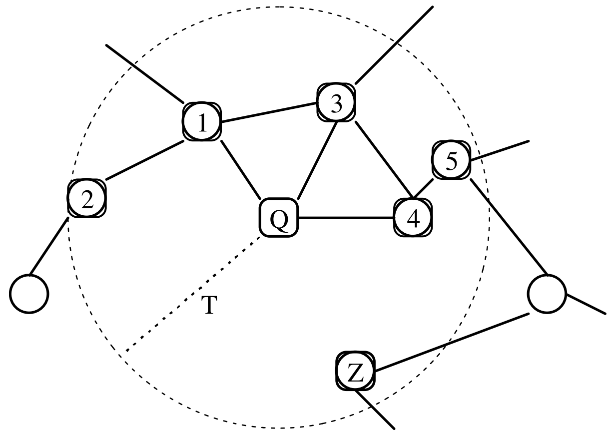

Fig. 5.3. Illustration of the query algorithm. A depth-first traversal bounded by T visits every vertex within T provided that the graph is smooth. This graph is not smooth and vertex Z is missed.

If the graph is smooth, a query is guaranteed to find every node X within T of Q, because by the definition of smoothness, there must be a path between Q and X such that every node on the path is closer to Q than X, and therefore within T as well. The graph above misses Z, but is not smooth because a search for Q through Z would encounter a local minimum.

A concurrent implementation of the query algorithm uses the database key X, the server address S, and the time, t, to uniquely define a tag. We can combine the second and third level protocols so that as messages are routed back to S from the minima, they also propagate requests for data closer to X than T.

Create a vertex S={X=?,now} or S={X=Y,now}
Send (S,X,T,now) to several random servers.
At each server $\{(X_0=Y_0,t_0), (X_1:t_1,S_1), (X_2:t_2,S_2), ...\}$ receiving (S,X,T,t),
        do concurrently
    Find X_c in $\{X_0, X_1, X_2, ...\}$ closest to X
    If c = 0 then (a minima)
        If dist(X, X_0) <= T and (X:t,S) is not in database then
            Add (X:t,S) to database   -- *tag*
            Send $(X_0=Y_0,t_0)$ to S
            Send (S,X,T,t) to all S_1, S_2, ...
        Else send () to S
    Else send (S,X,T,t) to S_c

In the following example, we add query node S, then route one of the messages to it from node 1 using the normal update protocol. Once the message is within T of the goal S (node 3), we recursively propagate the message to all neighbors. All nodes within T also send a response (X=Y,t) back to S, establishing a new edge (dotted line). We prevent looping (i.e. 3 to 4a to 5a to 3) by storing a tag that uniquely identifies the message and discarding duplicates.

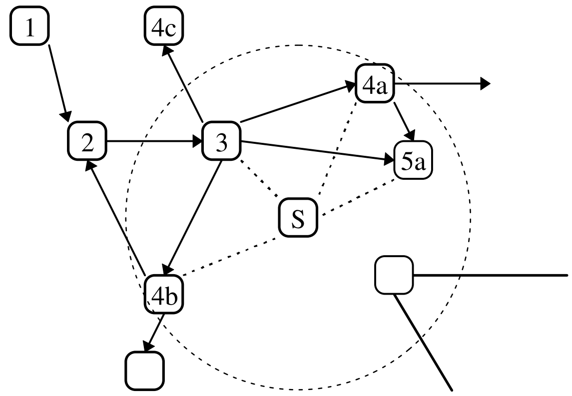

Fig. 5.4. Combining update and query protocol. The update test message initiates a local graph traversal when it enters the circle.

It is important that all servers agree on the distance function. If not, then it should be included in the message (S,X,T,t,dist). This could lead to another difficulty: a graph may be smooth under one distance function but not under another. A server might have to anticipate what version of dist other clients might use by sampling past requests.

Multi-level Caching

The caching mechanism introduced in level 1 could be generalized, such that we send and store messages of the form

$(X_1=Y_1:t,S_1S_2...S_n)$

where S_1 is the server storing the original value of X_1, last updated at time t, and the S_2, S_3, ... traces the order in which it was copied to each server. There is no reason why we cannot return a cached copy of (X=Y:t,S) in response to a query (X=?), and have the receiver save those copies. Then if a particular datum proves to be popular, then many copies will be propagated through the network, making the data available even if the original server is overloaded with requests. A client would use the signature list (S_1S_2:...) and timestamp (t) to resolve multiple copies of a datum from different sources, preferring newer data and data from trusted servers.

Servers may delete cached data at any time. Caching is not required for the protocol to work.

Section 6

Conclusion

A spatial index is useful not just for mapping keys to values, but for any type of database where we are interested in retrieving data "close" to some key, for example, an associative memory or a text retrieval system. One example might be a mailing list where we wish to identify duplicate entries with slightly different spellings.

A distributed spatial index could create a powerful system. Anyone could insert data for all to read, but only the owner would be able to update or delete the data. We described USENET, the Web, and DNS, all of which could be implemented this way, but it is even more powerful than that. For example, email could be thought of as a mapping where the key is the recipient's name, and the value is the message, encrypted for privacy. A chat room is a mapping from the name of the room or channel to the messages posted to it, but we reverse the usual roles of the query and update, expiring updates immediately while letting the query persist until we leave the room. Imagine if the Web used key strings instead of location-specific URL's to retrieve Web pages. Frequently accessed pages could be cached on other servers for faster access, and would still be available even if the original server were down.

Although there are ways to index data more efficiently than with a random graph, none are suitable for a large scale distributed implementation. The SRG employs adaptive techniques to recover from network congestion, hardware failure, and human failure, where one person could bring down the system. We have shown evidence strongly suggesting that the smooth randomized graph does have reasonably good complexity, better than current systems on the Internet, and we described a protocol that would allow a distributed implementation, error recovery, and decentralized management. Still, a number of issues need to be addressed:

- How to allocate vertices among servers. If every user has his or her own server, vertices would be allocated locally. Service providers could also rent server space. A system that gives users unlimited free space is subject to abuse.
- Security. If data is stored locally, then the owner of the data would be responsible for protecting it. The issue is in how to identify the owners of data when a query gives different answers from different sources. Clients need a secure way of identifying the source of the data.
- Distance function. The SRG query and update algorithm depend on all servers using the same distance function. What function is best? If the client decides, then a graph that is smooth under one function may not be smooth under another.

We also need to decide just how these servers should be implemented. What would be the details of the message representation? What protocol should they use, HTTP/CGI, telnet, or something new? TCP virtual circuits or UDP datagrams?

All of these issues can probably be solved to some level of satisfaction, though it is not the purpose of this thesis to do so. We focused on the complexity issue: is it feasible to build such a system for a network that doubles in size each year? We analyzed the complexity, and compared it with distributed data services now on the Internet.

	Storage	**Query**	**Update**	**Robust?**
USENET	$O(n^2)$	O(1)	O(n)	Yes
Web	$O(n^2)$	O(1)	O(n)	Yes
DNS	O(n log n)	O(log n)	O(log n)	No
SRG	O(n log n)	~O(log n)	$\sim O(\log^2 n)$	Yes
Sequential (Hash table)	O(n)	O(1)	O(1)	No

Fig. 5.1. Comparison of distributed data systems
("~" means "about". Actual complexity is $O(\log^2 \log n)$ higher).

Although the SRG compares favorably with other distributed systems, we wonder if we could do better. An SRG compares poorly with widely used sequential data structures running on single computers, for example, hash tables [Cormen, 1990]. Why is this? Nearly all commonly used data structures, even those with less efficient access times, such as trees, linked lists, arrays, etc., are O(n) in storage. Is there a fundamental law that demands O(n log n) storage in distributed systems (and therefore at least O(log n) search or update time)? We know no proof of such a law, nor do we know of any counter example.

The relatively low cost of an SRG depends on the number of dimensions in the spatial index being O(log n), which for the Internet, would be about 30. This would seem to preclude text retrieval, where there are thousands of dimensions, one per term. But would it? The complexity results are based on uniformly distributed data, but we know that text vectors are not uniformly distributed. In all natural languages, words have a Zipf distribution, such that the n'th most common word occurs with frequency about k/n, where k = 0.1 in English [Zipf, 1935]. Surprisingly, the Zipf distribution is common to many other types of data, not just text. Cache memory designs are based on measurements showing a Zipf distribution among address accesses during typical program execution, for example [Stone, 1993]. [Kauffman, 1996], states that all complex systems, such as the DNA-gene regulatory system, evolve toward the boundary between stability and chaos. The attractors (state cycles), and thus the outputs, of such systems have a Zipf distribution.

We did not explore the effects of a non uniform distribution on complexity, but it would be reasonable to expect that a higher number of dimensions could be supported. If we look at the range of distances between all possible pairs of n vertices in d dimensions, we find that the distribution becomes narrow as d increases. In other words, if d is small, then there are some pairs of points that are close together and other pairs that are far apart, but if d is very large, then all pairs of points are about the same distance apart. We believe that it is this effect which increases the storage cost of an SRG as d increases.

Our experiments were conducted with uniformly distributed random data, but we know that real data is not like that. If data tends to cluster together, then we would observe a distribution of distances that mimics a smaller number of dimensions. This might result in a lower storage cost.

We saw earlier that we could trade off the number of edges vs. path length to reduce communication costs at the expense of memory and computation. It is very tempting to find the optimal tradeoffs in a prototype system and use them in the final design. Unfortunately, in a small system, the optimal tradeoff may very well be a fully connected graph and a path length of 1, i.e., full replication of the index on every server. Unfortunately, that is exactly what we have with USENET and the Web search engines, and it is only when n becomes very large that we realize that there is a better way.

References

Bentley, Jon Lewis, "Multidimensional Binary Search Trees Used for Associative Searching", Communications of the ACM, 18(1975), pp. 509-517.

Bell, D., and J. Grimson, *Distributed Database Systems*, Addison-Wesley, 1992.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

Grossman, David. A., and Ophir Frieder, “Information Retrieval: Algorithms and Heuristics”, Kluwer Academic Publishers: to be published (1998).

Harman, D. (ed), "Overview of the Third Text REtrieval Conference (TREC-3)", National Institute of Standards and Technology Special Publication 500-225, Gaitersberg MD 20879, 1995.

Hunt, Craig, *TCP/IP Network Administration*, O'Reilly & Assoc., 1992.

Kantor, A., and M. Newbarth, "Off the Charts", Internet World, Dec. 1996, p. 44.

Kauffman, Stuart A., "Antichaos and Adaptation", Scientific American web site, http://www.sciam.com/explorations/062496kaufman.html, 1996 (Oct. 27, 1997)

Korth, H. and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1991.

Krol, E., *The Whole Internet User's Guide and Catalog*, 2nd Ed., O'Reilly & Assoc., 1992.

Mitchell, Tom M., *Machine Learning*, McGraw-Hill, 1997.

Schäuble, Peter, “Multimedia Information Retrieval”, Kluwer Academic Publishers, 1997.

Sparck Jones, Karen, and Peter Willett, eds., “Readings in Information Retrieval”, Morgan Kaufmann Publishers, 1997.

Stanfill, Craig, and Brewster Kahle, "Parallel Free-Text Search on the Connection Machine System", Communications of the ACM, 29(1986) pp. 1229-1239.

Steinberg, S., "Seek and Ye Shall Find (Maybe)", Wired, May 1996, p. 108.

Stone, Harold S., *High Performance Computer Architecture*, 3rd. Ed., Addison Wesley, 1993.

Tansel, et. al., *Temporal Databases*, Benjamin/Cummings Publishing Co., 1993.

Zipf, George Kingley, *The Psycho-Biology of Language, an Introduction to Dynamic Philology*, M.I.T. Press, 1935, 1965.

Appendix

Test Results: Deterministic Hill Climbing

Test results for deterministic hill climbing (Fig. 4.1, 4.2, 4.3). Columns are:

- Trials: Number of independent experiments. Remaining data is averaged over the trials.
- n: Number of vertices.
- dim: Number of dimensions (d).
- ln n: Natural log of n (for reference only).
- Order: Average number of neighbors per vertex, weighting each vertex equally
- Fanout: Average number of neighbors tested at each vertex during hill climbing. This number may be larger because high-order vertices are traversed more frequently, or smaller because the neighbors are counted before the graph is complete.
- Path: Average number of edges traversed during hill climbing, including the added edge if any.
- Finds: Average number of hill climbs needed before a vertex is found after completion of the build phase. Approximate inverse of smoothness.

Trials	n	dim	ln n	Order	Fanout	Path	Finds
128	8	1	2.08	2.56	2.85	1.61	1.3340
16	8	8	2.08	2.88	2.98	1.57	1.1953
2	8	64	2.08	4.25	4.07	1.40	1.0625
16	64	1	4.16	5.05	5.15	2.86	1.1563
2	64	8	4.16	7.00	6.96	2.38	1.1328
1	64	64	4.16	7.75	7.75	2.16	1.3281
2	512	1	6.24	7.47	7.46	4.30	1.1895
1	512	8	6.24	12.08	11.99	3.12	1.1094
1	512	64	6.24	17.04	19.02	2.68	1.2051
1	4096	1	8.32	9.58	9.47	5.81	1.0483
1	4096	8	8.32	17.65	17.50	3.90	1.1003
1	4096	64	8.32	33.69	43.37	3.06	1.1926
1	32768	1	10.40	11.67	11.39	7.36	1.0277
1	32768	8	10.40	23.55	22.99	4.66	1.0802

Trials	n	dim	ln n	Order	Fanout	Path	Finds
128	8	1	2.08	2.62	2.88	1.58	1.2627
16	8	8	2.08	2.69	2.93	1.49	1.6172
2	8	64	2.08	2.63	3.36	1.41	2.0000
1	8	512	2.08	2.75	2.97	1.30	1.8750
16	64	1	4.16	5.11	5.19	2.86	1.1777
2	64	8	4.16	6.73	6.88	2.37	1.1875
1	64	64	4.16	8.00	9.47	2.07	1.1406
1	64	512	4.16	9.09	8.77	2.04	1.1563
2	512	1	6.24	7.47	7.40	4.33	1.0537
1	512	8	6.24	12.06	12.09	3.12	1.0977
1	512	64	6.24	16.46	20.08	2.66	1.1738
1	512	512	6.24	19.20	22.85	2.51	1.1895
1	4096	1	8.32	9.57	9.41	5.84	1.0378
1	4096	8	8.32	17.70	17.51	3.90	1.0930
1	4096	64	8.32	33.56	42.39	3.06	1.1626
1	32768	1	10.40	11.67	11.39	7.36	1.0277
1	32768	8	10.40	23.55	22.99	4.66	1.0802

Trials	n	dim	ln n	Order	Fanout	Path	Finds
4	256	1	5.55	6.65	6.66	3.80	1.1465
2	256	2	5.55	7.58	7.49	3.44	1.1309
1	256	4	5.55	8.59	8.46	3.15	1.1016
1	256	8	5.55	10.37	10.45	2.85	1.1250
1	256	16	5.55	11.55	12.29	2.68	1.1602
1	256	32	5.55	12.70	13.14	2.58	1.1719
1	256	64	5.55	13.84	14.26	2.50	1.1133
1	256	128	5.55	13.73	15.47	2.49	1.2188
1	256	256	5.55	14.69	15.53	2.40	1.1445
1	256	512	5.55	14.20	16.72	2.39	1.2227
1	256	1024	5.55	14.75	16.12	2.38	1.2070

Trials	n	dim	ln n	Order	Fanout	Path	Finds
1	1024	1	6.93	8.05	8.03	4.87	1.0811
1	1024	2	6.93	9.41	9.13	4.30	1.0645
1	1024	4	6.93	11.29	11.06	3.76	1.0889
1	1024	8	6.93	13.83	13.95	3.39	1.1191
1	1024	16	6.93	17.06	17.85	3.10	1.1426
1	1024	32	6.93	19.60	21.91	2.91	1.1699
1	1024	64	6.93	21.31	24.97	2.82	1.1846
1	1024	128	6.93	23.64	27.74	2.71	1.1738
1	1024	256	6.93	25.53	29.21	2.64	1.1660
1	1024	512	6.93	24.62	32.01	2.64	1.2168
1	1024	1024	6.93	25.11	33.50	2.61	1.1963

Trials	n	dim	ln n	Order	Fanout	Path	Finds
1	4096	1	8.32	9.58	9.41	5.85	1.0457
1	4096	2	8.32	11.32	10.91	5.12	1.0459
1	4096	4	8.32	13.82	13.29	4.44	1.0728
1	4096	8	8.32	17.75	17.63	3.89	1.0984
1	4096	16	8.32	22.90	24.67	3.51	1.1270
1	4096	32	8.32	29.08	32.74	3.24	1.1497
1	4096	64	8.32	34.24	41.95	3.06	1.1523

Trials	n	dim	ln n	Order	Fanout	Path	Finds
1	16384	1	9.70	10.96	10.74	6.85	1.0359
1	16384	2	9.70	13.09	12.60	5.94	1.0422
1	16384	4	9.70	16.24	15.50	5.12	1.0587
1	16384	8	9.70	21.51	21.11	4.42	1.0867

Trials	n	dim	ln n	Order	Fanout	Path	Finds
1	65536	1	11.09	12.37	12.04	7.87	1.0336
1	65536	2	11.09	14.83	14.17	6.79	1.0345
1	65536	4	11.09	18.70	17.72	5.77	1.0505
1	65536	8	11.09	25.55	24.81	4.91	1.0768

Test Results: Randomized Hill Climbing

Test results for randomized hill climbing (Fig. 4.4). Each line represents one trial. Key:

- n: number of vertices.
- d: number of dimensions (second table. d = 1 in the first table).
- deg: Average number of neighbors (same as "order" above).
- path: as above.
- tests: "fanout" as above.
- length: Average edge length. Vertices are uniformly distributed in $[0, 2^{15}/d^{1/2})^d$. dist(a, b) ≡ $|a - b|^2$ (Euclidean distance squared). The space was chosen so that the maximum distance between diagonally opposite corners is 2^{30}.

```
n=     2 deg=  1.00 path=  1.50 tests=  2.33 length=400118230
n=     4 deg=  1.50 path=  2.17 tests=  1.85 length=135625335
n=     8 deg=  2.50 path=  1.80 tests=  2.14 length=146632048
n=    16 deg=  3.75 path=  2.40 tests=  2.58 length=210729953
n=    32 deg=  4.69 path=  3.16 tests=  2.76 length=150764551
n=    64 deg=  5.75 path=  4.03 tests=  2.95 length=135981257
n=   128 deg=  6.13 path=  4.74 tests=  2.96 length=112885719
n=   256 deg=  7.34 path=  5.79 tests=  3.12 length=102618208
n=   512 deg=  7.90 path=  7.03 tests=  3.15 length= 86481803
n=  1024 deg=  8.77 path=  8.24 tests=  3.20 length= 86274867
n=  2048 deg=  9.46 path=  9.50 tests=  3.25 length= 77062958
n=  4096 deg= 10.32 path= 10.76 tests=  3.31 length= 71470495
n=  8192 deg= 11.09 path= 12.16 tests=  3.36 length= 65274216
n= 16384 deg= 11.85 path= 13.58 tests=  3.41 length= 61596182
n= 32768 deg= 12.61 path= 15.04 tests=  3.46 length= 57251611
n= 65536 deg= 13.34 path= 16.51 tests=  3.51 length= 54378841
n=131072 deg= 14.08 path= 18.05 tests=  3.55 length= 51458284
n=262144 deg= 14.85 path= 19.58 tests=  3.60 length= 48878647
```

```
n=    4 d=   2 deg=  1.00 path=  0.75 tests=  1.33 len=115892273
n=    8 d=   3 deg=  3.25 path=  1.73 tests=  2.56 len=149059743
n=   16 d=   4 deg=  3.75 path=  2.30 tests=  2.51 len= 95908214
n=   32 d=   5 deg=  5.50 path=  2.63 tests=  3.22 len=112717106
n=   64 d=   6 deg=  6.88 path=  3.08 tests=  3.74 len= 90161646
n=  128 d=   7 deg=  8.95 path=  3.40 tests=  4.46 len= 89725966
n=  256 d=   8 deg= 11.30 path=  3.96 tests=  5.07 len= 85027810
```

```
n=  512 d=   9 deg= 13.89 path=  4.44 tests=  5.78 len= 83748440
n= 1024 d=  10 deg= 16.70 path=  4.89 tests=  6.50 len= 79138167
n= 2048 d=  11 deg= 19.86 path=  5.38 tests=  7.22 len= 76455514
n= 4096 d=  12 deg= 23.79 path=  5.81 tests=  8.21 len= 74114184
n= 8192 d=  13 deg= 27.82 path=  6.27 tests=  9.09 len= 71483939
n=16384 d=  14 deg= 32.53 path=  6.73 tests= 10.14 len= 69748360
n=32768 d=  15 deg= 37.94 path=  7.19 tests= 11.28 len= 67660581
n=65536 d=  16 deg= 43.92 path=  7.64 tests= 12.52 len= 65862563
```