Adaptive Spatial Indexing for Robust Distributed Data

Matthew V. Mahoney
Computer Science Dept., Florida Institute of Technology
415 Rutgers Ave., Melbourne FL 32901, USA
407-724-1582, matmahoney@aol.com

Abstract

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. 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).

1. Introduction

The problem of providing an efficient, robust, location-transparent distributed data service is unsolved. Services such as Web search engines (Steinberg) and indexed USENET news (Krol) archival servers are inefficient, requiring $O(n^2)$ space to store n data items, because while the number of servers must grow in proportion to the number of users, each server also fully replicates the database (for USENET) or index (for search engines). The complexity (growth rate) is a concern because the Internet has been doubling in size (n) about once per year since 1980 (Kantor).

Efficient systems, such as the Domain Name Service (DNS) (Hunt) or distributed relational databases (Bell), lack robustness, even when the data is replicated on redundant servers. These systems are usually organized as trees, which are susceptible to human error when a single person or organization has access to all copies of the root node.

A spatial index can be used to quickly locate data when there is a mapping between the keys and d-dimensional Euclidean space, $\mathbf{R}^d$. For relational databases (Korth), each of the d keys is mapped into one dimension by lexical ordering, and we are interested in finding the data at a single point. For text retrieval (Harman), each word or term in a document or query is hashed to a vector in highly dimensioned space using a technique such as surrogate coding (Stanfill). The document mapping is the sum of these vectors. We are interested in retrieving all documents within a region "close" to some query, i.e., having many words in common.

The most efficient centralized spatial index known is the k-d tree (Bentley), which has O(n) storage, O(log n) search time and O(log n) update time. In distributed systems, the most efficient are trees such as the DNS. Storage increases to O(n log n) because of the need to replicate servers near the root to balance the load between levels. These data structures are deterministic but not robust.

In section 2, we describe the *smooth randomized graph*, a robust but nondeterministic data structure, and we describe distributable query and update algorithms. In section 3, we analyze the worst case space and time complexities, and experimentally determine the average case complexities. In section 4, we conclude that these algorithms could provide distributed data services more efficiently and more robustly than those currently available on the Internet.

2. The Smooth Randomized Graph

In this section we show how to implement a database using a *spatial index*, where there is a mapping between the database keys and $\mathbf{R}^d$ -- d-dimensional Euclidean space. The index allows us to quickly find keys near a given point in space. The index is *adaptive*; updated as needed to repair inefficiencies regardless of the previous state of the index. The same adaptive process repairs physical damage when the index is distributed across an unreliable network.

The data structure is called a *smooth randomized graph*. A graph is *smooth* if it can be traversed efficiently (specifically, hill climbing toward any given point proceeds without encountering local minima). The graph is *randomized*, lacking organized structure, which has the effect of removing single points of failure. The query and update algorithms are nondeterministic, with poor worst case performance, but efficient on average.

The vertices of the graph, each representing one datum, can be distributed over servers on a network. We can assume, without loss of generality, that each vertex/datum is stored on a separate server. Storing multiple vertices on a single server does not change our analysis of complexity.

Each server has only local knowledge of the graph, represented by edges. A phone directory might be distributed across several servers as follows:

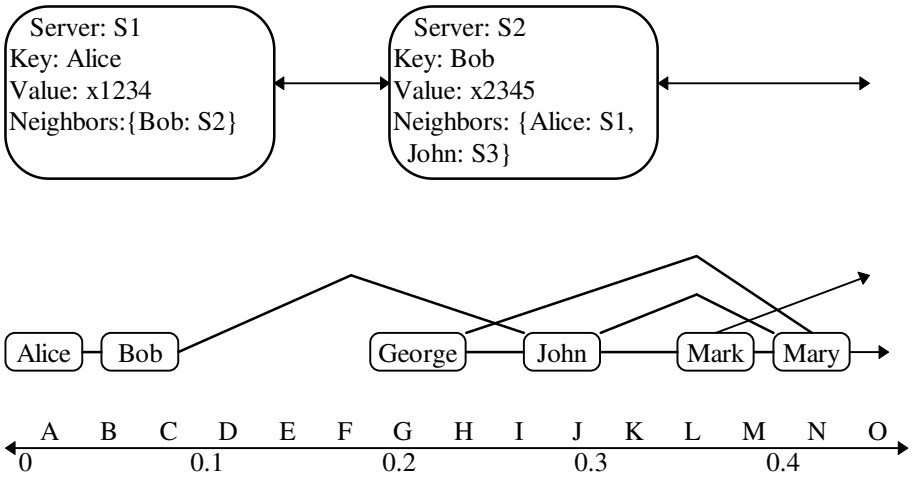


Fig. 2.1. A distributed phone directory mapped into 1-dimensiional space.

2.1. Query Algorithm

Knowing the mapping of the key we seek, we can move toward it from any vertex along the edges of the graph by hill climbing. *Hill climbing* (Mitchell) is a *greedy* (Cormen) algorithm that always takes the step that makes the most forward progress. Specifically, let G be a graph. Let GOAL and START be vertices in G. Let dist(x, y) be the distance between the mappings of x and y in $\mathbf{R}^d$.

```
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 n^2 hill climbing episodes (from all n^2 pairs of n vertices) that succeed in finding the goal. A graph is *smooth* if it has a smoothness of 1. The following graph is not smooth:

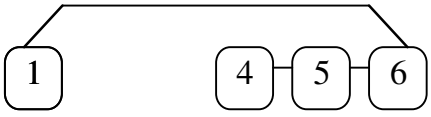


Fig. 2.2. A non-smooth graph

Suppose that the goal is 1 and we start at 5. Hill climbing fails because it reaches a minimum at 4. If we go the other way, hill climbing succeeds by going through 6 (the neighbor of 1 that is closest to 5).

Even if a graph is not smooth, we can eventually find any vertex by repeating the hill climb from random starting points. In a parallel implementation, we might search simultaneously from several points. To pick a random starting point without global knowledge, we would have to start at a known point and traverse several edges randomly.

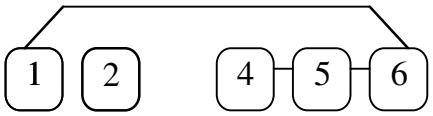
2.2. Update Algorithm

There is a simple way to build a graph with high smoothness. After inserting a new vertex, test whether it is reachable by hill climbing and add edges as needed to remove minima.

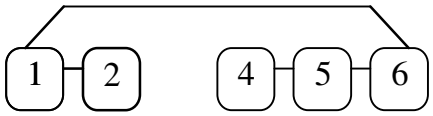
Let G be a graph. Let GOAL be the vertex to add:

```
Update(G, GOAL)
      Add GOAL to G with no connecting edges
      Repeat until hits > misses  -- or smoothness > 1/2
            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
```

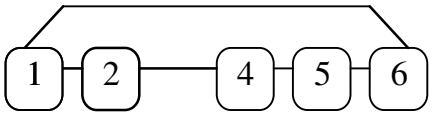
Suppose that we add a vertex *2* to fig. 2, and we randomly pick starting vertices 1 and 5 for our first two hill-climbing attempts. Before the first attempt, the new vertex is disconnected:

Fig. 2.3a. 

Hill climbing from 1 fails immediately, so we add an edge between 1 and 2:

Fig. 2.3b. 

Hill climbing from 5 fails when we reach a minimum at 4, so we add an edge between 2 and 4:

Fig. 2.3c. 

We now have two misses and no hits. The next three hill climbing attempts succeed no matter where we start. After that we stop.

The *update* algorithm is adaptive. No matter what the previous state of the graph, according to the update algorithm, smoothness of at least 1/2 is achieved.

We can speed up the algorithm by testing the update from several points simultaneously, instead of waiting for the result from each test before starting the next. A test which fails because of server or network failure or excessive delays is counted as a miss.

The update algorithm describes how edges are added from a predefined set of vertices. The same update algorithm can be used after new vertices are added. Deleting a vertex can simply be achieved by removing the edges from the vertex's neighbors and then removing the vertex itself. Deletion may reduce the smoothness of the graph.

2.3. Randomized Hill Climbing

There are more efficient query and update algorithms than the ones based on hill climbing. Suppose that instead of comparing each neighbor to find the closest to the goal at each step, we test neighbors in random order and 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 make less progress at each step by choosing a sub-optimal neighbor, but we do fewer comparisons at each step.

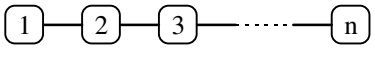
```
Let G be a graph.  Let GOAL and START be vertices in G.

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
```

We next examine the space complexity of smooth randomized graphs and the time complexities of the query and update algorithms.

3. Complexity

Analyzing average case storage requirements and running times of the query and update algorithms is difficult. Consider the cases of the ordered list in $\mathbf{R}^1$ and the fully connected graph in $\mathbf{R}^n$.



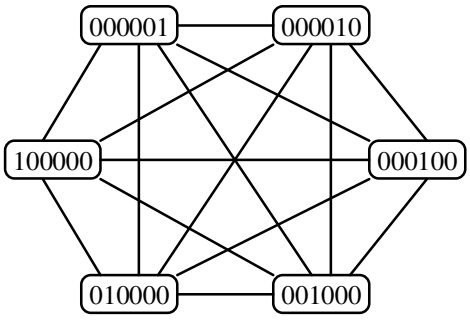


Fig. 3.1. a. A smooth list. b. A smooth, fully connected graph.

Both graphs are smooth. Storage complexity is O(n) for the first graph, $O(n^2)$ for the second, since the edges require space. Query complexity is O(n) in both cases. In the list, hill climbing traverses O(n) edges. In the complete graph, hill climbing traverses one edge, but O(n) comparisons are required to find it.

Note that neither graph would be smooth if any edge were to be removed. In the complete graph, if the i'th key is mapped into 1 in the i'th dimension and 0 in all other n - 1 dimensions, then all edges have the same length, $2^{1/2}$ (where the *length* of an edge connecting vertices x and y is dist(x, y).

The update algorithm does not guarantee a smooth graph, so a query could take arbitrarily long because of a series of unlucky guesses for the starting point.

3.1. Empirical Average Case Complexity

We now construct smooth randomized graphs and estimate complexity empirically. For our first test, we first insert all of the vertices to build a 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, $[0, 1]^d$.

```
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 count a hit
      Else count a miss and add an edge between FOUND and START
```

The following graph shows the average degree (number of neighbors) of each vertex for various numbers of vertices (n) and dimensions (d).

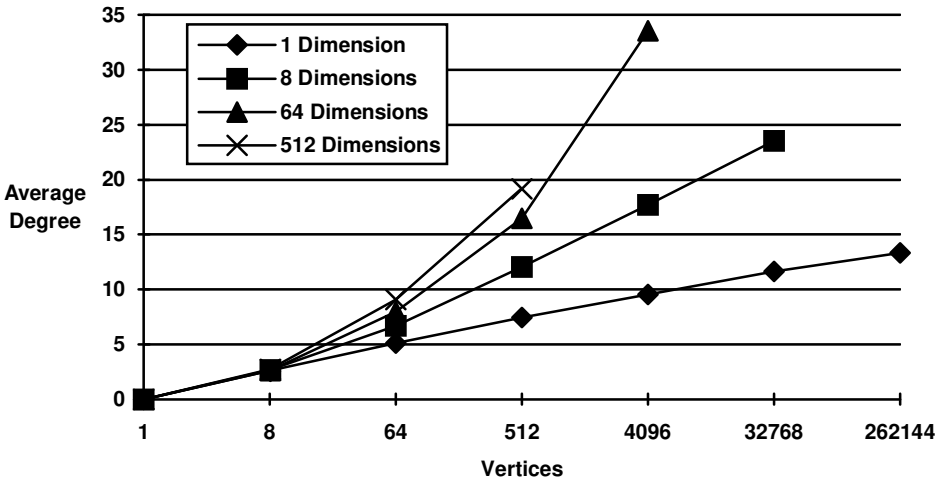


Fig. 3.2. Average degree per vertex.

Empirically we find that when $d < \log_2 n$, that

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

for some constant C. When d = 1, we have C = 1, or degree = ln n + 1. When d = 8, degree = 2.8 ln n - 5.6. It was not practical to test this relationship for d > 8. The equation for C is not known. When $d > \log_2 n$, the rate of growth is unknown, but faster than O(log n), as seen in the graph. Note how the line for d = 8 curves upward until n > 256, then straightens out.

The results confirm that space complexity is O(n log n), since the number of edges is n•degree = O(n) O(log n).

Query time complexity is the number of comparisons per vertex (O(log n)) times the average path length, divided by smoothness. Empirically we find the average hill climbing path to be ln n - 2 for d = 1, and less for higher dimensions (and higher degree). This is expected because the number of reachable vertices increases exponentially as the path length increases. The actual values found are as follows.

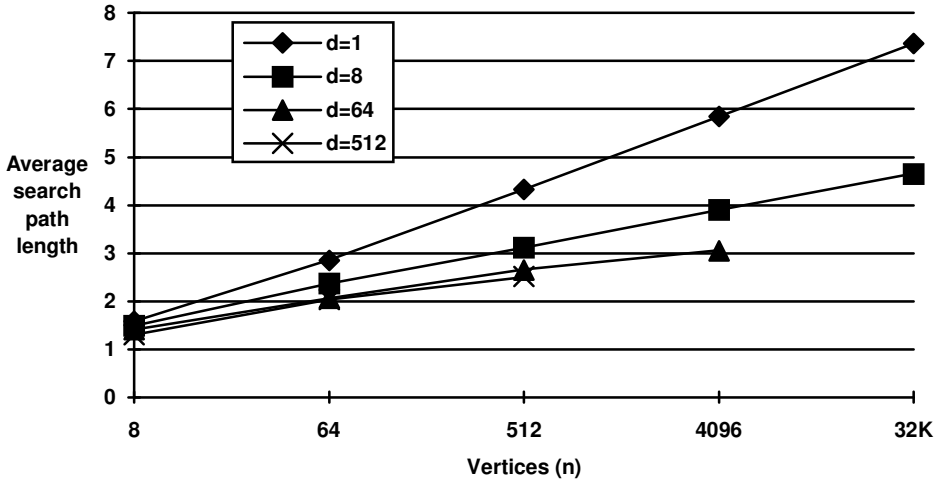


Fig. 3.3. Average number of steps for deterministic hill climbing.

The update algorithm guarantees average smoothness of at least 1/2. The actual smoothness approaches 1 as n grows because most of the edges are added early during the update testing. The following graph shows the average number of additional tests (after the first) needed to query each vertex. The queries are exhaustive; we count tests until every vertex is retrieved. The test procedure is:

```
For each vertex V in G do
      While Hill-Climb(G, V, random vertex in G) ≠ V do
            Count 1 search test
```

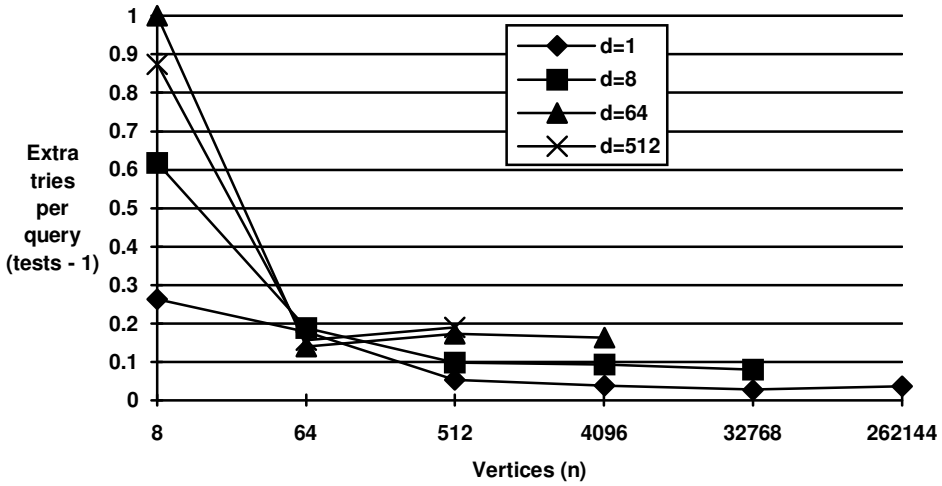


Fig. 3.4. Average number of extra hill climbs (after the first) per query.

These two results confirm that query complexity is $O(\log^2 n)$. Update complexity is $O(\log^3 n)$ because each update is O(log n) times (one hit and one miss per neighbor).

3.2. Randomized Hill Climbing

The following results were obtained when the hill climbing algorithm was replaced with randomized hill climbing for d = 1. 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.

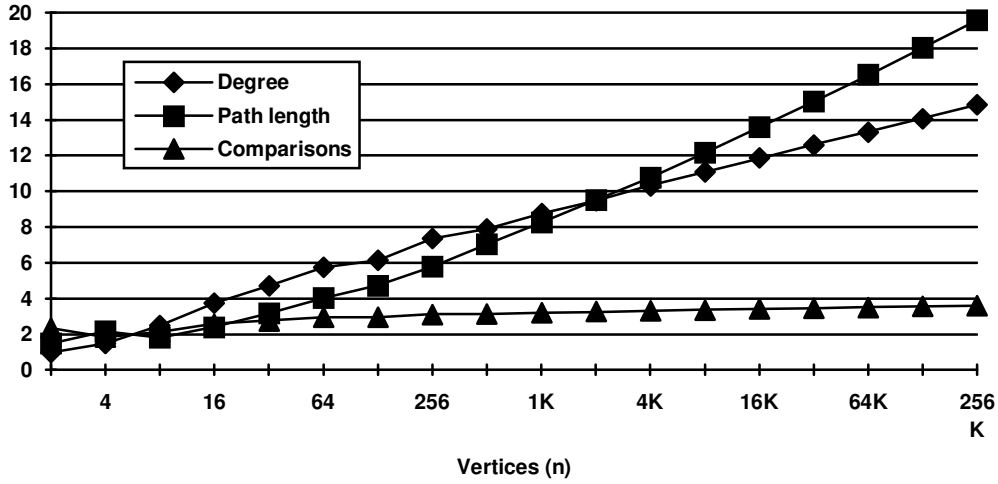


Fig. 3.5. Complexity of nondeterministic hill climbing for 1 dimensional data.

The three lines indicate the average number of edges at each vertex (degree), the average number of edges that are traversed during each update test including the newly added edge (path length), and the average number of neighbors that need to be tested at each step in order to find one that makes forward progress (comparisons). Storage cost is the product of n and degree. Query cost is the path length times comparisons divided by smoothness (not plotted, but greater than 1/2). Update cost is query cost times degree.

With deterministic hill climbing, the number of distance comparisons would equal the degree. Using nondeterministic hill climbing, the comparisons are greatly reduced at a small cost in degree and path length. We find that the following equations closely fit the empirical data:

- 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 resulting complexities are:

- Query: $O(\log n \log^2 \log n)$
- Update: $O(\log^2 n \log^2 \log n)$

- Storage: O(n log n)

4. Conclusion

Spatial indexes have applications in nearest-neighbor classification problems (Mitchell), and text retrieval, as well as simple relational queries. Regional queries (requesting all data within a certain distance) can be answered quickly by localized graph traversal.

A smooth randomized graph could be used to implement a decentralized, distributed index on an unreliable network such as the Internet. We described concurrent query and update algorithms and showed empirically that they have reasonably efficient (logarithmic) run times and storage costs. (Mahoney) describes a detailed message-passing protocol for the query and update algorithms.

It is rather surprising that no widely used distributed systems lacking a single point of failure has ever been implemented with better than $O(n^2)$ space complexity. To summarize:

	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)	~$O(\log^2 n)$	Yes

Fig. 4.1. Comparison of distributed data systems
("~" means "about". Actual complexity is $O(\log^2 \log n)$ higher).

USENET and the Web search engines have $O(n^2)$ storage complexity because each of the O(n) servers fully replicates the data.. They are robust because if any server fails, the data is available on other servers. DNS is highly reliable because every datum is stored on at least two independent servers (Hunt), but is not robust because accessibility of the entire tree depends on the integrity of the root, where a single organization has access to all redundant copies. In July 1997, an administrator uploaded a corrupted file to all of the DNS root servers making large sections of the .com and .net domains temporarily unavailable (Internet World).

There are two mechanisms that ensure robustness. First, the query algorithm makes use of redundant paths to data, and second, the update algorithm ensures accessibility by repairing damage. We have removed the potential single point of failure that occurs at the root of a search tree without sacrificing much of the tree's efficiency.

Before a distributed database service can be designed and implemented, a number of issues still need to be addressed, such as protocols, security, allocation, caching, resolution of inconsistencies, and so on. These issues are beyond the scope of this paper, although some are discussed briefly in (Mahoney). We focused on the issue of complexity, although our results are based on empirical evidence and are presented without proof. We also have not adequately

characterized complexity in high dimensions, which is a concern for text retrieval. Although the algorithms are efficient, it is not clear that there are not better ones.

Acknowledgments

I wish to thank my thesis advisor at Florida Tech, Dr. Philip K. Chan, for his help in preparing this paper.

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.

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.

Internet World, Jan. 1998, p. 58.

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

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.

Mahoney, Matthew V., “Complexity of Adaptive Spatial Indexing for Robust Distributed Data”, MSCS Thesis, Florida Institute of Technology, 1998.

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