An Intelligent Locally Sensitive Hashing Based Algorithm for Data Searching

Haiying Shen, Felix Ching, Ting Li, Ze Li
Department of Computer Science and Computer Engineering
University of Arkansas, AR 72701
{hshen, ching, txl005, zxl008}@uark.edu

Abstract

The rapid growth of information nowadays makes efficient information searching increasingly important for a massive database with tremendous volume of information. Locally Sensitive Hashing (LSH) is an efficient method for searching similar records. This paper analyzes the strengths and weaknesses of LSH in a massive database and Smith-Waterman algorithm. It reveals the strengths of LSH and Smith-Waterman algorithm in the field of database searching and querying. More importantly, this paper presents an intelligent searching algorithm called LSH-SmithWaterman that intelligently integrates LSH and Smith-Waterman algorithm to utilize their strengths and exploit their fullest capacities. Simulation results show the superiority of LSH-SmithWaterman algorithm compared to LSH in information searching. It dramatically reduces the memory and time consumption and performs accurate searching.

1. Introduction

Computer and digital information have dominated modern day's information system; major corporations and small business have adopted digital information systems to keep track of their information.

With the database growing, it becomes harder and harder to find desired data. An analogy will be imagining a large library that holds million of books, without a way like call number to search for a desired book. Therefore, an efficient way is increasingly needed to search and query a large database driven by tremendous advances in information field. This paper analyzes two potential methods that will give insight and shed lights to an efficient and fast querying system: Locally Sensitive Hashing (LSH) and Smith-Waterman algorithm.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative methods for information searching. Section 3 introduces LSH, analyzes the strengths and weaknesses of LSH, and its performance evaluation. Section 4 describes the Smith-Waterman algorithm, its underlying idea, and its performance evaluation. Section 5 proposes LSH-SmithWaterman algorithm to integrate the strengths of LSH and Smith-Waterman algorithm. Section 6 draws conclusions and summarizes the propositions of the SmithWaterman algorithm.

2. Related Works

Numerous methods have been proposed for information searching in a massive database. The methods can be classified into four categories: linear search method, categorized data search method, VP-Tree indexing method, and LSH method.

2.1 Linear Search Method

The simplest approach for querying and searching database is Linear Search Method [1]. Linear Search Method computes the distances between a query record and all the other source records. During the computation, Linear Search keeps track of the record of the shortest distance. Although Linear Search Method returns the best record, i.e. the record with the shortest distance to the query record, it needs to go through the entire source records that has *O(n)* searching efficiency for *n* source records.

2.2 Categorized Data Search Method

One noticeable attempt to invent an efficient querying method is called "Categorized Data Search Method (CDS)" [2]. It requires database to be classified and ordered before query searching. For example, data that relate to city and country names will be grouped into city database and country database correspondingly. CDS uses previously gained knowledge to discern the meaningful words within the query. After CDS extracts the meaningful information from the query, it will go to the corresponding database to search for the desired data. Therefore, if query analysis returns a result indicating that the query is related to a city, CDS will go directly to the city database and perform a search. CDS reduces query response time significantly, while maintaining a high success data query ratio. However, the accuracy of the information extraction plays an important role on the success data query.

2.3 Dynamic VP-Tree Indexing Method

VP-Tree indexing [3, 4, 5, 6] is another approach that attempts to solve Nearest Neighbor Searching problem using tree methods. In this approach, VP-Tree sets a node as the vantage point, and uses the point to partition data subsets. Once the first vantage point is used, VP-Tree will iterate to the next node, setting it as the vantage point, and partition data subsets again. Eventually, the entire datasets will be organized into a balanced tree. VP-Tree N-Nearest Neighbor Search algorithm requires a balanced VP-Tree and a threshold value σ, the algorithm uses depth first [7] to search through the tree. Once the algorithm comes to a leaf, it will search every element within the leaf to see if the element is within the threshold distance. If the element is indeed within the threshold distance, the element will be grouped into the Nearest Neighbor set W. If not, the element will be discarded. If the node is not a leaf, the algorithm will perform a calculation to see if the node is within the threshold. If not, then there's no need to search through the node since the sub-dataset under that node is guaranteed to be outside the threshold. VP-Tree improves searching efficiency, but it is not flexible enough to handle rapid data volume growth. Data deletion and insertion require tree structure updates and hence lead to high overhead for tree maintenance.

2.4 Time-Space Efficient Locality Sensitive Hashing Method

LSH shows promising capacity on accurate data searching, but its drawback is the requirement of large memory space to store hash tables and slow searching. Two variations of LSH include "Point-perturbation based LSH" and "hash-perturbation based LSH." [15, 16].

Point-perturbation based LSH algorithm takes a query object and creates several random perturbed objects that are R distance away from the query object. Then, the algorithm unions the search results of the query object with the search results of the perturbed objects. This algorithm relies on the fact that since perturbed objects are R distance close to the query object, the result hash value should be relatively within the distance as well. As a result, the bucket that contains the hash value of the perturbed objects should have the query object's close neighbors, since perturbed objects are near the query object. Point-perturbation's goal is to trade time with space. It uses less amount of hash table compared to the basic LSH, but it has to check more hash buckets.

Hash perturbation based LSH (HPB LSH) differ from point-perturbation based LSH (PPB LSH) on that HPB LSH perturbs directly on the query object's hash value instead of on random perturbed objects. In basic LSH method, hash values of a query objects identify which bucket the query objects will be hashed to. Based on this property, objects near q but not hashed to the same bucket as q will be likely to be hashed into the nearby buckets. Therefore, when HPB LSH perturbs on a query object's hash value, it will generate other hash values that will point to the near-buckets that are likely to contain the query object's near neighbors. This allows HPB LSH to avoid the overhead of perturbing objects and the computation associated with the hash functions [16], hence achieves a faster perturbation.

3. Locally Sensitive Hashing

Locally Sensitive Hashing (LSH) [8, 9] is a searching and querying scheme that is used to find data that are identical or similar to the query data. LSH utilizes hash function family, hash bucket, and hash table to hash and calculate the data [13, 14]. Figure 1 shows the steps of LSH to identify similar records.

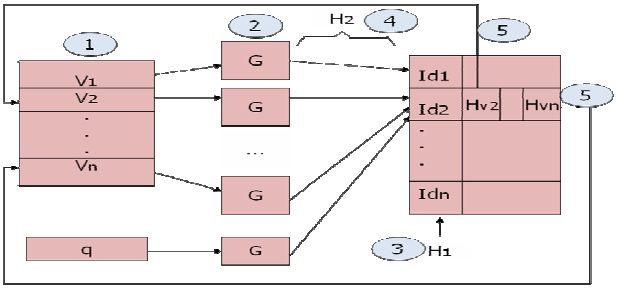

Figure 1. Steps of LSH data searching.

1. Translate records to number vectors (IDs). LSH extracts keyword tokens from all records, and builds a token list consisting of the tokens. The token list determines the dimension of the number vector; a source record is mapped to the token list, if both the token list and the source record have the token, the number vector will print a 1 in the corresponding dimension, otherwise 0.
2. Use n groups of g hash function family (each g has m hash functions) to generate G consisting of a number of hash buckets (i.e. rows).
3. Use hash function H1 to compute an index Id in the final hash table.
4. Use hash function H2 to compute the value Hv which is stored in the row indicated by the index in the final hash table.
5. For a query record (q), the same operations are conducted. The records whose Ids are the same as q's Id should be the similar records to q. To further refine the results, the distance between q and every record is calculated. Records whose distances are within range R are the returned records.

The hash functions in g is generated by $h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor$, where w is a specified value, a and b are randomly generated value by g(x), and $g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. LSH has an important parameter: k, that is the number of projections per hash value. It represents a tradeoff between the time spent in computing hash values and time spent in the refinement stage. To improve operation efficiency, LSH divides g evenly into two groups of hash function denoted by u. Instead of the normal g hash function family, two u families are used simultaneously to generate bucket G. Since a record has more than Id, before the refinement, the record group discovered may contain more than one the same record. To avoid redundant distance calculation for efficiency, for each same point encountered, LSH skips repeating calculation.

LSH holds high accuracy in regard of finding identical and similar data within range R specified, but the greatest drawback with LSH is its significant resource consumption. LSH requires dramatic amount of memory and needs long time to execute.

We conducted experiments on E^2LSH 0.1 of MIT [8]. The dimension of the dataset is 20,591. The number of source records was 10,000. 97 query records were selected from source records. In the hash function $h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor$, w was set to 4 as an optimized value [9]. The distance threshold of R was set to 3.

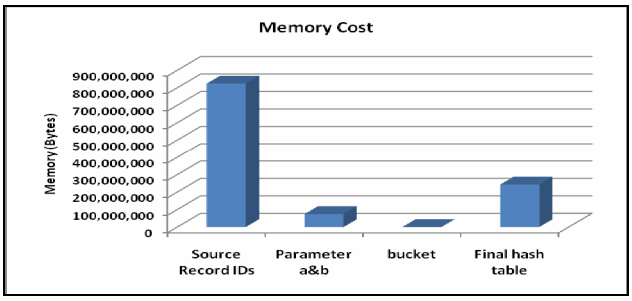

Figure 2. Memory cost of LSH.

Figure 2 shows the memory used for different objects in LSH. As the figure shows, the IDs for source records occupy the most memory, followed by hash table, and the total memory added up consumes approximately 120 million bytes for data searching.

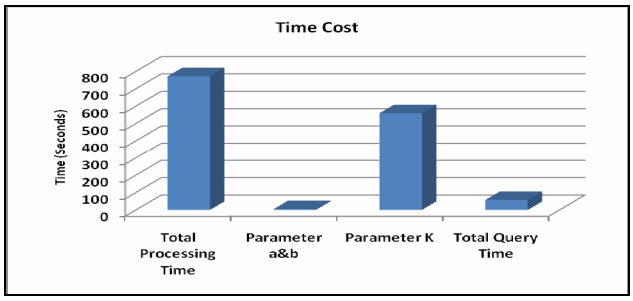

Figure 3. Time cost of LSH.

Figure 3 demonstrates the time used in each phase in LSH. As the figure indicates, the time to compute parameter k takes up most of the computation time, with a total time adding to approximately 1400 seconds. This time measured is the system's time, which runs in a different measure from the real world time. The real world time needed for computation is approximately 45 minutes to 1 hour. The results show that LSH's costs are so great that it is impossible to be used for business purpose. But despite the weaknesses, LSH performs an unmatchable accuracy in data searching.

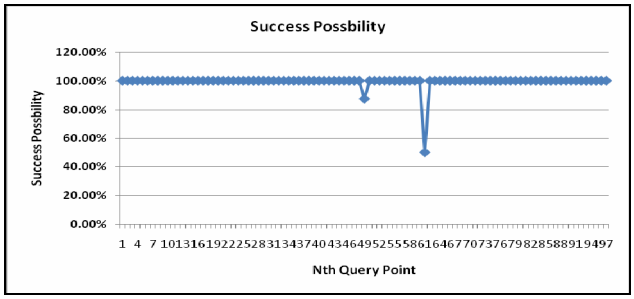

Figure 4. Success possibility of LSH.

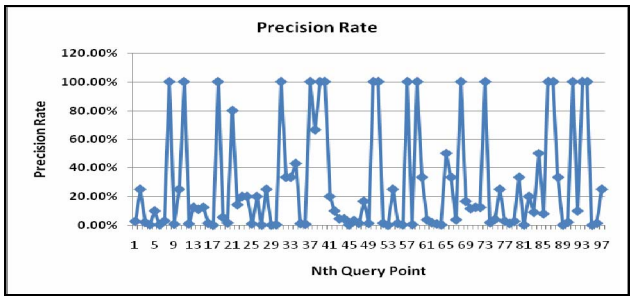

Figure 5. Precision rate of LSH.

Success possibility represents the possibility that a searching scheme finds the desired data. Figure 4 plots the success possibility of LSH. We can see that LSH find all the data and its similar points except for two queries. We use precision rate to denote the percentage of similar records in the returned records. Figure 5 demonstrates that most of LSH's precision rates are very low, which means the returned records have a lot of undesired records. It implies that the refinement stage of LSH is not effectiveness on pruning.

The experiment results confirm that LSH performs a high accuracy searching, but costs significant amount of time and memory. They imply that by reducing the length of both source record and query record, the amount of memory and time will be reduced significantly. One solution is to change the binary IDs of records to decimal ID by compressing 10 bits to one value; that is, to translate the binary strings into decimal strings. By applying this method to the dataset in our simulation, the length of records is reduced by 10 folds, resulting in a string with approximately 2000 characters long. Figure 6 and 7 depict the memory and time consumption with this compression method. From the experiment results, we can conclude that by compressing the source records and query records, we can obtain much less memory and time consumption.

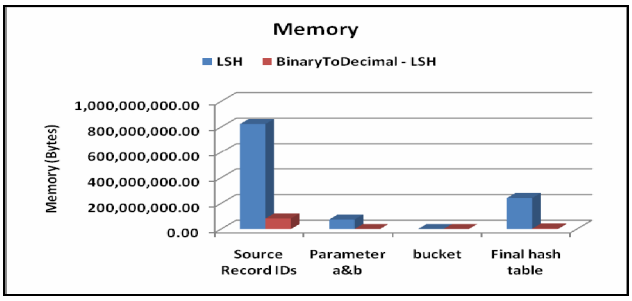

Figure 6. Memory cost: BinaryToDecimal vs. LSH.

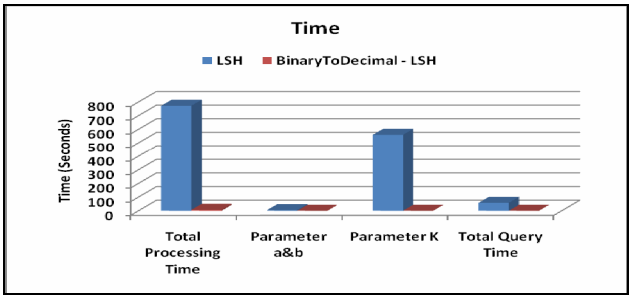

Figure 7. Time cost: BinaryToDecimal vs. LSH.

4. Smith-Waterman Algorithm

The Smith-Waterman algorithm is a well-known algorithm in bioinformatics [10] field for performing local sequence alignment; that is, for determining similar regions between two nucleotide or protein sequences. Instead of a whole sequence, the Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure.

Smith-Waterman algorithm utilizes the power of dynamic programming in order to realize the local alignment [11]. It first generates a distance matrix as shown in Table 1 by using gap penalty and substitution matrix. Once the distance matrix is generated, Smith-Waterman will find the maximum value within the matrix, trace the maximum value back until the value drops to 0. Algorithm 1 shows the pseudocode for Smith-Waterman algorithm.

Algorithm 1: Pseudo-code of Smith-Waterman algorithm

---

1. Assigns a score to each pair of bases
   Uses similarity scores
   Uses positive scores for related residues
   Uses negative scores for substitutions and gaps
2. Initializes edges of the matrix with zeros
3. Scores are summed in matrix. Summation is accomplished using the following algorithm:

$$F(i,j)=\max\begin{cases}0\\ F(i-1,j-1)+\sigma(x_i,y_j)\\ F(i-1,j)-A\\ F(i,j-1)-A\end{cases}$$

where A is the gap penalty, and $\sigma(x_i, y_j)$ represents the corresponding cell within the substitution matrix.

4. Begins the trace back at the maximum value found anywhere in the matrix
5. Continues until the score falls to 0[12].

Let's take an example to see how Smith-Waterman algorithm works. For instance, there are two strings:

String1: A **T C A G** A G **T C**
String2: G **T C A G** ------ **T C** A

Table 1. An example of a distance matrix of 11x10.

T[j]											
S[i]			A	T	C	A	G	A	G	T	C
		0	0	0	0	0	0	0	0	0	0
	G	0	0	0	0	0	1	0	1	0	0
	T	0	0	1	0	0	0	0	0	2	0
	C	0	0	0	2	0	0	0	0	0	3
	A	0	1	0	0	3	1	1	1	1	1
	G	0	0	0	0	1	4	2	2	2	2
	T	0	0	1	0	1	2	3	1	3	1
	C	0	0	0	2	1	2	1	2	1	4
	A	0	1	0	0	3	2	3	1	1	2

Gap penalty -2
Substitution Matrix:

	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

Even though Smith-Waterman algorithm has fast running time and consumes almost no extra memory, one major weakness it exhibits is the slight inaccuracy on data searching compared to LSH. As shown in Figure 8, the success possibility of Smith-Waterman algorithm is lower than LSH; there are few more query records that deviate from 100% success rate.

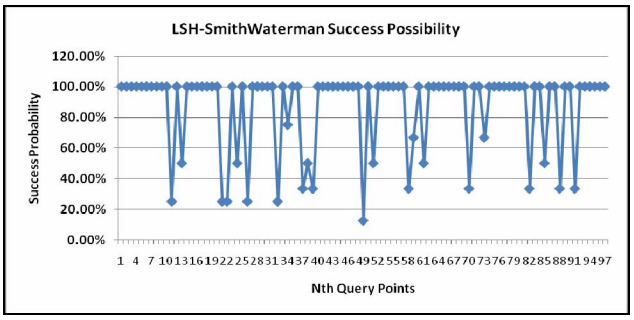

Figure 8. Success possibility of Smith-Waterman alg.

5. Combination of LSH and Smith-Waterman Algorithm

The strength of Smith-Waterman lies in the fact that it can discover the similarity within any type of string. That is, it can discover the similarity between two binary strings, two alphanumeric strings, or any other kind of strings [10, 11, 12]. With this strength in hand, Smith-Waterman algorithm will be able to simplify the first step of LSH.

The first step of LSH is to translate records into number vectors. For totally 20000 tokens in the simulation dataset, each record has a number vector that is 20000 characters in length, with 10000 number vectors in the source record file of 10000 records. Apparently this step takes a significant amount of memory and time in order to store and process the source records.

With Smith-Waterman algorithm at disposal, it is possible to combine it with LSH to create a searching scheme that is fast and efficient. We call the combination LSH-SmithWaterman. Instead of translating the source records into binary strings, LSH-Smith Waterman translates the records to IDs based on the corresponding positions of their tokens in the token list. For example, one record is:

Li Ze |23 | male| UofA| CSCE|JBHT|345|,

and each of its token's position in the token list are:

Li Ze: 1678; 23:23; male:223; UofA: 897; CSCE: 9543 JBHT:9599; and 345: 345;

With the parameters above, it will generate a string of

1678|23|223|897|9543|9599|345

This string is much shorter than the binary string, and will save substantial amount of memory and time. Once the entire source file and the query file are translated, LSH-SmithWaterman will begin the comparison and determine similar records. Let's say the query record string is

Li Ting|23| female| American| UofA | CSCE |JBHT| 345|

and each of its token's position in the token list are:

Li Ting: 1680; 23:23; female: 224; American: 874 UofA: 897; CSCE: 9543; JBHT: 9599; 345: 345;

Then, the ID of the record is:

1680|23|224|874|897|9543|9599|345

Using Smith-Waterman algorithm as the comparison scheme, we can get

1678|23|223|-----|897|9543|9599|345

1680|23|224|847|897|9543|9599|345

Similarity (5/8=62.5%)

If the desired similarity of a query is set to be less than 62.5%, then the query record will be considered as similar data and returned to the user.

We also take advantage of Smith-Waterman algorithm in LSH regional alignment comparison step. Specifically, we replace the distance calculation refinement step with the algorithm to further improve LSH's efficiency. Figures 9, 10, 11 and 12 show the memory, time, success probability and precision rate of LSH-SmithWaterman in the same simulation as LSH. As the results indicate, the LSH-SmithWaterman gains a significant improvement over LSH in terms of memory and time reduction, and it achieves much higher precision rate than LSH, at the cost of a slightly lower success possibility.

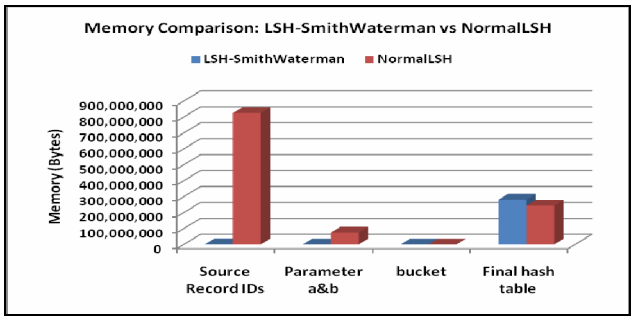

Figure 9. Memory cost: LSH-SmithWaterman vs. LSH.

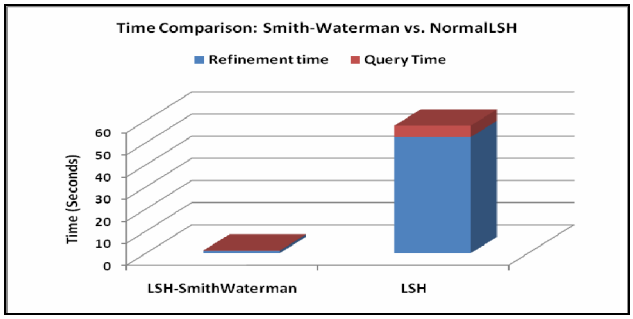

Figure 10. Time cost: LSH-SmithWaterman vs. LSH.

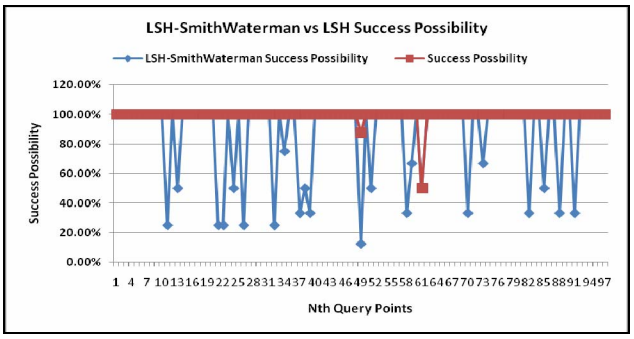

Figure 11. Success possibility of LSH-Smith-Waterman vs LSH.

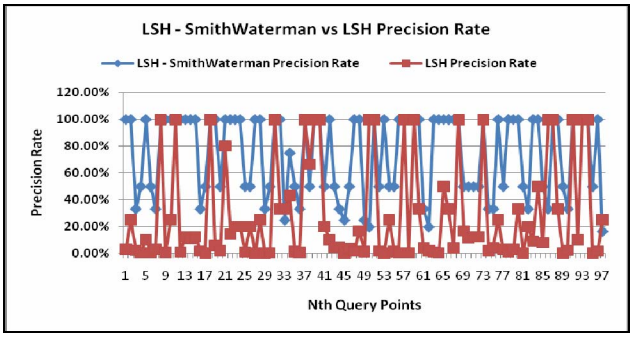

Figure 12. Precision rate of LSH-Smith-Waterman vs LSH.

6. Conclusions

Both LSH and Smith-Waterman algorithm hold their individual ground for similar data searching in a massive database. LSH has a high accuracy searching power, but consumes heavy amount of memory space and time. On the other hand, Smith-Waterman saves significant amount of memory and time, but has a more inaccurate data search. This paper presents LSH-SmithWaterman algorithm that intelligently integrates these two methods, and successfully utilizes the strengths of both. Simulation results show the superiority of LSH-SmithWaterman algorithm compared to LSH in efficient information searching. It dramatically reduces the memory and time consumption and performs accurate searching.

Acknowledgment

This research was supported by U.S. Acxiom Corporation.

References

[1] "Nearest Neighbor Search" http://en.wikipedia.org/wiki/Nearest_neighbor_search

[2] S.-W. Ann, et al. "Categorized Data Search Method for Intelligent Query Processing." In *Proc. of IWCMC,* pages 901-905, 2005.

[3] A. W.-C. Fu, P. M.-S. Chan, et al. "Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances." In *VLDB Journal*, 9(2) 154-173, June 2000.

[4] T. C. Chiueh. Content-based image indexing. In *Proceedings of the 20th VLDB Conference*, pages 582-593, 1994.

[5] J. Uhlmann. "Satisfying general proximity/similarity queries with metric trees." *Information Processing Letters*, 40:4:175-179, November 1991.

[6] P. Yianilos. "Data structures and algorithms for nearest neighbor search in general metric spaces." In *Proc. of SODA*, pages 311-321, 1993.

[7] "Depth-first search", http://www.nist.gov/dads/HTML/depthfirst.html

[8] "LSH Algorithm and Implementation (E2LSH)", http://web.mit.edu/andoni/www/LSH/index.html

[9] A. Andoni, P. Indyk, "E2LSH 0.1 User Manual" June 21, 2005.

[10] P. E. Black, "Smith-Waterman algorithm", *Technical report*, U.S. National Institute of Standards and Technology. March 2006.

[11] S. M. Brown, "The Smith-Waterman Algorithm", *Computers for Molecular Biology*. NYU Medical Center Course G16.2604. http://www.med.nyu.edu/rcr/rcr/course/sim-sw.html

[12] Baylor College of Medicine, "Smith-Waterman Algorithm", Aug., 2002. http://searchlauncher.bcm.tmc.edu/help/SmithWaterman.html

[13] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of SCG*, pages 253 262, 2004.

[14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of STOC*, pages 604 613,1998.

[15] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proc. of SODA*, Jan 2006.

[16] Q. Lv, W. Josephson, et al. A Time-Space Efficient Locality Sensitive Hashing Method for Similarity Search in High Dimensions. *Technical report*, Princeton University, 2006.