

A Multi-Resolution Compression Scheme for Efficient Window Queries over Road Network Databases[†]

Ali Khoshgozaran, Ali Khodaei, Mehdi Sharifzadeh and Cyrus Shahabi*
Computer Science Department
University of Southern California Los Angeles, CA 90089-0781
{jafkhosh,khodaei,sharifza,shahabi}@usc.edu

Abstract

Vector data and in particular road networks are being queried, hosted, and processed by many application domains such as mobile computing. However, many hosting/processing clients such as PDAs cannot afford this bulky data due to their storage and transmission limitations. In particular, the result of a typical spatial query such as window query is too huge for a transfer-and-store scenario. While several general vector data compression schemes have been studied by different communities, we propose a novel approach in vector data compression which is easily integrated within a geospatial query processing system. It uses line aggregation to reduce the number of relevant tuples and Huffman compression to achieve a multi-resolution compressed representation of a road network database. Our empirical results verify that our approach exhibits both a high compression ratio and fast query processing.

1 Introduction

Current advances in GIS applications and online mapping tools have enabled smooth navigation and browsing of geographic maps. With an online geospatial service, a remote user issues a *window query* asking for the visual representation (e.g., map) of a limited geographical neighborhood. The requested geospatial information about the queried area (e.g., road network) must be transferred and stored in the user's machine (client). Consisting of the geometrical representation of various geographical features, this bulky data requires high communication bandwidth to be transmitted to the client and significant amount

of memory storage to be stored at the client side. The storage and transmission requirements become even more crucial when the user queries from a wireless handheld device such as a PDA. The common practice to address the storage/transmission issue in GIS applications is to send a raster image, which is a rendition of requested geospatial data at the server side, to the client. Google Maps¹ and Microsoft Live Local² are examples of such an approach. However, the transmitted raster image is only a visual representation of the geospatial data and hence does not include the geometric objects and corresponding metadata. Instead, the objects are rendered in the image and the corresponding metadata is superimposed as limited labels. Any consecutive query on this data (e.g., nearest neighbor query) in the client results in another handshake with online server which hosts the original dataset. That is, no client application can access data for further processing. A solution to this problem is the multi-resolution compression of geospatial data so that only the corresponding level of abstraction (i.e., zoom level) desired by the user could be sent to the client. An ideal compression technique for vector data, while being fast, must respect the spatial and topological aspects of the data. For instance, the boundary of a hexagon-shaped park must be invariant to the compression.

The problem of vector data compression has been addressed by two independent communities: 1) data compression community that takes numerical approaches embedded in compression schemes to focus on the problem of compressing road segments [1–3]. The advantages of these methods lie in their ability to compress a given vector data up to a certain level. However, they do not address the need to represent geospatial data at different levels of abstraction. 2) GIS community that uses hierarchical data structures such as trees and multi resolution databases to represent vector data at different levels of abstraction [5–10]. With most of these methods, displaying data in a certain zoom level re-

*[†] This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), IIS-0324955 (ITR), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

¹<http://maps.google.com>

²<http://local.live.com>

quires sending all the coarser levels. Another issue is the complication of choosing which objects to display at each level of abstraction. While both approaches have their own merits, they do not blend compression schemes with hierarchical representation of vector data.

In this paper, we propose a novel approach that integrates the application of a data compression technique (i.e., Huffman coding) and a line aggregation operator on vector data within a geospatial query processing system. With our approach, we compress the entire query result that is to be sent to the client and do not remove any roads due to their relative unimportance to the desired level of detail. Hence our approach is orthogonal to those other approaches where only selected portions (e.g. freeways) of the original data are compressed. Using a road network database as our running example, our two-step approach uses spatial aggregation to group spatial and non-spatial attributes of related road segments together, followed by a multi-resolution compression technique which is capable of retrieving data up to any desired accuracy level. The ultimate result is a highly compressed data; the aggregation reduces the number of relevant tuples in the data and the compression decreases the number of bits by which these tuples are represented. We propose two variants of our method for different query scenarios: *cONa* offers high compression (more than 80%) by performing the line aggregation operator **online** over the query result and *cOFa* minimizes query response time (less than 230 milliseconds even for large window queries) by pre-aggregating the entire database **offline**. We argue that depending on the application and query size a geospatial service can choose the appropriate variant.

2 Preliminaries

With the advent of techniques to collect geospatial data such as road networks, different vector datasets such as NAVTEQ³ and TIGER/Line⁴ have been developed. Regardless of the accuracy and methods used to obtain such datasets, their spatial information is stored as a line string (to be defined later). In this section, we define terms and notations used throughout the paper and provide the background for compression technique we adopt.

2.1 Vector database

Definition 1. A *linestring* $LS = \langle p_1, \dots, p_n \rangle$ is an ordered set of $n \geq 2$ points $p_i = (x_i, y_i)$ each representing a location at longitude x_i and latitude y_i . Figure 1 illustrates a linestring LS_1 consisting of 4 points representing a road segment in the city of Los Angeles.

Definition 2. A *vector entity* $VE = \langle a_1, \dots, a_m, LS \rangle$ is a combination of one linestring attribute LS associated with

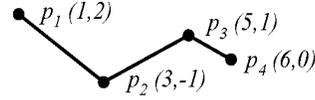


Figure 1. A sample road segment LS_1 .

$m \geq 1$ non-spatial attributes a_1, \dots, a_m . We use vector entity and *entity* interchangeably throughout the paper. In Figure 1, $VE_1 = \langle \text{'Vermont St'}, LS_1 \rangle$ is an entity including LS_1 .

Definition 3. A *vector database* VD is a database storing vector entities VE . It could represent the road network of a city or the rivers of a country. An example of a vector database is $VD_1 = \langle VE_1, \dots, VE_3 \rangle$ where $VE_1 = \langle \text{'Vermont St'}, LS_1 \rangle$, $VE_2 = \langle \text{'Vermont St'}, LS_2 \rangle$ and $VE_3 = \langle \text{'Slausen Ave'}, LS_3 \rangle$. Figure 2 depicts the corresponding linestrings.

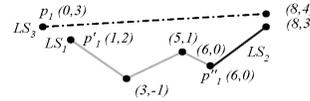


Figure 2. Linestrings of vector database VD .

2.2 Huffman coding

Consider a set S of words each composed of symbols from an alphabet A . Given alphabet $A = \{a_1, \dots, a_n\}$, assume F is the set of numeric values f_i : the frequency of repetition of symbol a_i in the words of set S . The standard Huffman coding finds a function $H_{A,F}(a_i)$ that maps each symbol a_i in A to a binary code (i.e., bit string) b_i . The mapping guarantees that $\sum_k f_k * |a_k|$ is minimized where $|a_k|$ is the size of a_k [4]. The generated mapping function is a lossless prefix-free binary encoding of A (i.e., no bit string b_i is a prefix of $b_j \neq b_i$). The more a symbol a_i is repeated in S , the shorter code b_i is assigned to a_i by Huffman coding. Hence, encoding the words in S using b_i s results into an efficient compression of the original set S . Section 3 describes why this property makes Huffman coding an appropriate method for compressing a vector database.

3 Our approach

In this section, we define two operators performed on the set of vector entities of a query result. Later in Section 4, we describe alternative query processing schemes based on different ways these operators can be assembled together.

3.1 Line aggregation operator $LAgg$

In a vector database VD , the information of each real-world road is stored as a set of vector entities each containing a segment of the road as a linestring attribute associated with some non-spatial attributes such as road name. Given VD a set of vector entities $VE = \langle a_1, a_2, \dots, a_m, LS \rangle$,

³<http://www.navteq.com>

⁴<http://www.census.gov/geo/www/tiger>

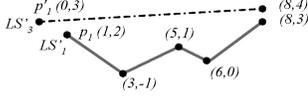


Figure 3. Aggregation of vector database VD.

the Line Aggregation operator $LAgg$ combines each group of VEs with a common non-spatial attribute value into one single vector entity VE' . The entity VE' consists of a single linestring LS' generated by concatenation of all linestrings LS of the original entities. For example, Figure 3 illustrates the result of applying $LAgg$ on the vector database of Figure 2. Notice that the linestring of the generated entity consists of the same points of the original entities thus preserving the spatial aspect of the data. We use the notation of relational algebra [11] to formally define $LAgg$ as $a_1, a_2, \dots, a_k \subset f_{k+1}a_{k+1}, f_{k+2}a_{k+2}, \dots, f_m a_m, CONCAT LS(VD)$ where a_1, a_2, \dots, a_k ($1 \leq k \leq m$) are non-spatial attributes on which we group the entities. Each f_i is an aggregate function defined on non-spatial attribute a_i . $CONCAT$ is the spatial aggregate function that returns the concatenation of a set of related linestrings as one linestring. Performing the operator $LAgg$ on the vector database VD generates an aggregated vector database VD' . It consists of vector entities $VE' = \langle a_1, a_2, \dots, a_k, a'_{k+1}, a'_{k+2}, \dots, a'_m, LS' \rangle$ where $a'_i = f_i(\langle a_i \rangle)$ and $LS' = CONCAT(\langle LS \rangle)$ are the results of applying functions f_i on the multiset of values for attribute a_i in the group of related entities in VD and $CONCAT$ on their corresponding linestrings LS , respectively. We apply $LAgg$ on VD_1 defined in Section 2.1 to get $VD'_1 = StreetName LAgg CONCAT LS (VD_1)$. VD'_1 will now consist of the following vector entities $VE'_1 = \langle 'Vermont St', LS'_1 \rangle$, $VE'_3 = \langle 'Slausen Ave', LS'_3 \rangle$, where LS_1 and LS_2 are concatenated to generate LS'_1 (see Figure 3).

Notice that aggregating a real-world vector database in which each entity includes only one road segment significantly reduces the number of tuples in the database. For example, applying $LAgg$ on the vector database of 2500 entities in downtown Los Angeles based on street names, types (e.g. road/highway) and directions generates only 260 aggregated entities (about 90% reduction in number of tuples). Notice that although applying $LAgg$ does not lose any spatial data in reducing number of tuples, it increases the size of each tuple. However, we will later show that both the reduction in the number of tuples and the increase in each tuple size make the data more compressible.

3.2 Compression operator $Comp$

The second operator in our scheme is a compression operator $Comp$ which is used to convert the input data to a compressed format that is much smaller than the original data. $Comp$ consists of four different modules to convert the input data to a compressed format sequentially. The

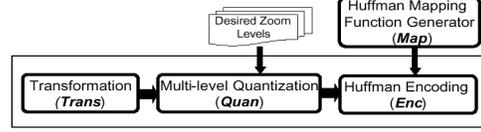


Figure 4. The $Comp$ operator.

modules illustrated in Figure 4 are as follows:

1. The transformation module $Trans$: Given L , a set of linestrings LS , the transformation module $Trans$ computes the differential representation of each LS in L . This is an alternative way of representing a line segment. Given $LS = \langle p_1, \dots, p_n \rangle$, the differential representation of LS is $Trans(LS) = \langle \langle p_1.x, \Delta x_1, \dots, \Delta x_{n-1} \rangle, \langle p_1.y, \Delta y_1, \dots, \Delta y_{n-1} \rangle \rangle$

where $\Delta x_i = p_{i+1}.x - p_i.x$ and $\Delta y_i = p_{i+1}.y - p_i.y$ for $1 \leq i < n$. We refer to point p_1 as the *base point* of linestring LS . The differential representation of the linestrings illustrated in Figure 3 is as follows:

$$Trans(LS'_1) = \langle \langle 1, 2, 2, 1, 2 \rangle, \langle 2, -3, 2, -1, 3 \rangle \rangle$$

$$Trans(LS'_3) = \langle \langle 0, 8 \rangle, \langle 3, 1 \rangle \rangle$$

In Section 5, we show that compressing the differential representation of real-world road segments results into high compression ratios.

2. Multi-level quantization module $Quan$: Depending on the level of detail (LOD) desired by the application, $Quan$ defines the maximum number of digits up to which the data can be rounded without losing the accuracy in its visual representation. For instance, if the user intends to view the data in an LOD where the points $p = (1.4, 2.3)$ and $p' = (1.0, 2.0)$ are displayed as a single point, p' can be used to visualize p (we round p by one digit). As a result, $Quan$ increases the frequencies of the symbols used by the Huffman mapping function generator (defined later in this section) which results in high compression. Figure 6 illustrates the effect of quantization on the distribution of symbols.

Suppose that the application requires the data in several LOD's which are known a priori. For each level L_i , $Quan$ processes the output of $Trans$ (i.e., differential values of linestrings) to *quantize* the data with respect to L_i . This representation of the quantized data is called zoom level Z_i . The quantization process is done as follows: Based on the error tolerance for each level, $Quan$ first partitions the domain of data into intervals (bins) of size $\frac{10}{K}$ units where K is a constant. Subsequently, it distributes any numeric data among these bins. That is, the number N in interval $i_q = [r_q, r_{q+1})$ will then be represented by r_q (r_{q+1}) if $|N - r_{q+1}|$ is greater (less) than $|N - r_q|$. Figure 5 shows the result of quantizing five numbers in interval $[-5, 15)$ to 4 bins. As seen -3, -1, 1, 2, and 3 are quantized to -5, 0, 0, 0, and 5, respectively.

Quantizing the roads data for multiple LOD's enables the application to view the data in arbitrary levels of detail

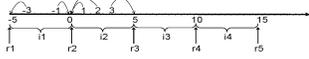


Figure 5. Quantization of the transformed VD.

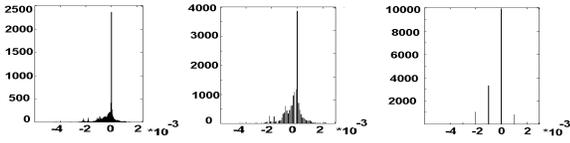


Figure 6. Symbol distribution in the output of *Quan* for 3 levels of detail.

with a hierarchical representation. The choice of the number of required LOD's depends on the application. For each level L_i , the number of bins used by *Quan* for quantization is determined by the smoothness of transition desired from viewing the data in level L_i to the next finer/coarser level.

The hierarchical nature of the output of *Quan* allows a gradual decrease in data precision as long as the error level (i.e., the displacement of points from their original locations) is tolerable to be displayed properly for that LOD. Notice that each zoom level uses the original data for quantization instead of the data from the previous levels and hence the errors will not propagate through all levels. Also in order to preserve the finest granularity of data, in level L_0 we do not quantize the data and thus perform a lossless compression of query results.

3. Huffman mapping function generator *Map*: Our ultimate goal is to compress the query result set before sending it to the user. Discrete acquisition scheme of road vector data and its frequency suggest that the value of differences in latitude and longitude of consecutive recorded points should have high repetition and correlation (see Figure 6.a). Moreover applying our transformation and quantization modules further increases this correlation. Consequently as a compression technique that significantly compresses such highly correlated data, we use Huffman coding (see Section 4). Having sets A, F and S (defined in section 2.2), the *Map* operator first computes f_i , the frequency of occurrence of symbol a_i in S and constructs Huffman mapping function $H_{A,F}(a_i)$ where $F=\{f_1, \dots, f_n\}$. We use *Dictionary* to refer to the mapping function $H_{A,F}()$. In our approach, S is the set of geocoordinates of points representing linestrings in VD. As S includes only numeric values, we have $A=\{0, \dots, 9, -, blank, eof\}$ (in the worst case). Table 1 illustrates a part of the mapping constructed by applying *Map* on our original vector database. The Matlab code we used ⁵ has a fixed small alphabet which protects us from facing the problem of gradual growth of the dictionary size which is common in the applications of Huffman coding. This is an important property in evaluating the effectiveness

⁵<http://www.mathworks.com>

of our compression scheme and measuring its compression ratio. After performing the line aggregation, transformation and quantization modules on the vector database, a bulky dataset of differential values is generated. Applying the *Map* module on this data on-the-fly can sometimes take significant amount of time. Therefore, for each zoom level, in an off-line process we apply *Map* on our differential values and store the constructed dictionary once for each zoom level.

Table 1. Symbol mapping for alphabet A

Symbol a_i	-	0	1	Blank	5	...
Bit string b_i	00	0111	010	10	11011	...

4. Huffman encoding module *Enc*: This module performs the actual compression of the vector data. *Enc* uses the dictionary constructed by *Map* to encode the output of *Quan* and generate the compressed bit string of linestrings for each zoom level. After going through phases mentioned above each original road segment of query result set (i.e., linestring LS) will be transformed into its base point (i.e., $LS.p_1$) followed by encoded values of its quantized latitude/longitude differences corresponding to each zoom level. For each quantized difference δ , we have $Enc(\delta) = \{H_{A,F}(a_i) | a_i \in \delta\}$ where $H_{A,F}()$ is the Huffman mapping function generated by *Map*. For example, *Enc* compresses the quantized linestrings of Figure 5 as $Enc(LS'_1) = 01111001 \dots 011011$ which in combination with the base point $p_1(1,2)$ constructs the compressed data.

4 Query processing

Suppose that user issues a window query from his/her PDA. A *Naive* approach retrieves the relevant vector entities (query result) and sends the *uncompressed* result back to the user. Depending on different layouts of the operators in our system, we propose two variants of our compression scheme;

Compression of Off-line aggregated data (cOFa): This approach uses the aggregation operator *LAgg* off-line (once) to generate the aggregated vector database VD' from the original vector database VD. It then submits the user's spatial queries to VD' instead of VD. The retrieved query result is then fed into *Comp* whose output is a highly compressed encoded copy of the original query result which is transferred more efficiently to the user. The bottom query path in Figure 7 shows this approach.

cOFa offers two advantages over the *Naive* approach: 1) It highly reduces the size of the data that must be transferred to the querying device. 2) It always outperforms the *Naive* approach in terms of query response time.

Compression of On-Line aggregated data (cONa): This approach is similar to *cOFa* except that it first issues the window query against the original vector database VD. It

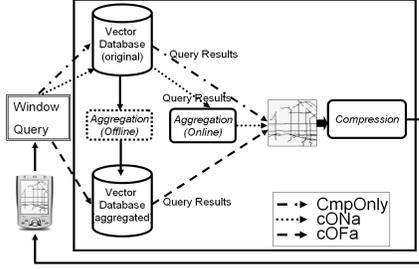


Figure 7. Alternative query flows.

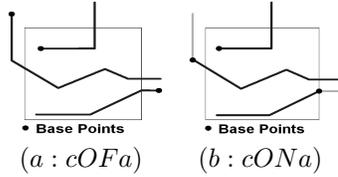


Figure 8. Base points in *cOFa* vs. *cONa*

then applies the *LAgg* operator to the query results. Finally, *cONa* sends the aggregated output to the *Comp* operator similar to *cOFa*. Notice that here the aggregation operator is performed once for every received query in contrast to *cOFa* in which the aggregation operator is performed once for the entire VD. The middle query path in Figure 7 illustrates *cONa*. This approach achieves an even higher compression than that of *cOFa* while imposing an increase on the query processing time.

One important difference between *cOFa* and *cONa* is the way they treat base points. With *cOFa*, the entire vector database is aggregated offline and thus (assuming the aggregation is made on street name) each road consisting of several road segments will be represented as a single linestring with a fixed base point (i.e., the first point of its first linestring before aggregation). Therefore, while compressing the road segments that overlap with our window query, we use those fixed base points and the entire linestring representing each road as an input to our *Comp* operator. In contrast, *cONa* first finds all the road segments that overlap with the window query and performs line aggregation only on these line segments. Consequently, for each road in the query window, *cONa* treats the first point of the results' first line segment as the base point for that road. Figure 8 illustrates this difference in assigning base points to the resulting linestrings in *cOFa* and *cONa*. Although the query result retrieved in *cOFa* contains more information than what the user actually requires, its overall compression ratio is far better than the *Naive* approach and is comparable to *cONa* (see Section 5). This property suits the applications where the user is most likely to navigate to the adjacent areas right after his query result is displayed.

5 Performance evaluation

We have conducted several experiments to evaluate and compare the performance of our proposed approaches. The effectiveness of *cOFa* and *cONa* is determined in terms of 1) response time which is the time it takes to prepare the query result and transfer it to the client and 2) size of the encoded query result and 3) the displacement error resulting from lossy compression at different zoom levels. The response time consists of three times based on the type of operations performed. For each window query, the first time value, T_q is the time it takes to issue the window query against our vector database VD and retrieve the results. If line aggregation is needed, it is accounted for in T_q . The second time, T_t is the time the *Trans* and *Quan* modules take to compute the quantized differential values and finally T_c is the time the *Enc* operator takes to encode its input using the dictionary. In the following sets of experiments, we compared *cOFa* and *cONa* from different perspectives against two other methods: 1) *Naive* approach which simply sends the query result to the client and 2) *CmpOnly* approach (the top query path in Figure 7) which only compresses the query result using our *Comp* operator without applying aggregation (*LAgg*). Our experiments are performed on a real-world dataset obtained from NAVTEQ covering a 5 mile by 4 mile area in the city of Log Angeles containing more than 5000 road segments. In our experiments we selected 7 different rounds R_0 to R_6 for our window query and randomly generated a total of 300^6 query windows and compressed the results for 5 different zoom levels Z_0 (i.e., finest, lossless) to Z_4 (coarsest). Results are then averaged for each round. R_0, R_1, \dots, R_6 in our experiments represent areas with sizes 10%, 20%, ..., 70% of the entire area in our dataset, respectively. Experiments were run on an Intel P4 3.20 GHz with 2 GB of RAM.

Our first set of experiments focuses on the effect of increasing the size of our query window on the size of resulting compressed data. We define *Rate Of Compression* (ROC) as the average size of the data compressed by our schemes in each round over the size of original data being sent by the *Naive* approach. Figure 9.a shows the ROC as the query window size increases from R_0 to R_6 for an intermediate zoom level Z_2 . As expected, for all query window sizes larger than the first lossless round R_0 , *cOFa* compresses the original data well and it outperforms the *Naive* approach. It also shows that *cONa* achieves about 80% reduction in query result size. Our experiments show that the data behaves similarly for all other zoom levels. As the figure illustrates, at R_0 , *cOFa* transmits almost 50% more than the original size of the data. The reason is that the amount of excessive data being compressed is relatively large for very small window sizes (see Figure 8). However, as the window

⁶Due to overlaps between large query windows in the last few rounds, we gradually decrease number of random query windows as R increases.

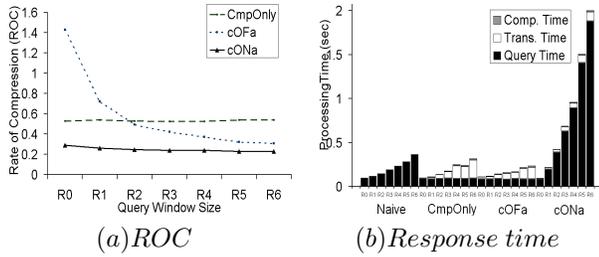


Figure 9. Impact of query window size

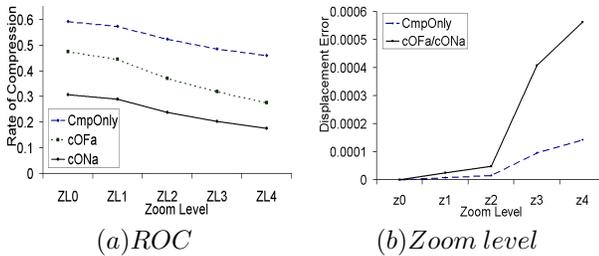


Figure 10. Impact of zoom level

size increases, *cOFa* reduces size very fast and approaches *cONa* for the larger areas being queried.

The second set of experiments focuses on the effect of the query window size on performance (i.e., response time) for each round. Figure 9.b shows that for all rounds and with all three approaches *cONa*, *cOFa* and *CmpOnly* the compression time T_c is negligible (as compared to T_q and T_t). In contrast, T_q for *cONa* increases with query window size because the *Lagg* operator's implementation in Oracle (the DBMS used in our experiments) takes a significant amount of time to concatenate related linestrings for bulky vector databases. However, notice that in the worst case, *cONa* takes only 2 seconds to compress a query result. It is also important to note that both *cOFa* and *cONa* take less transformation/quantization time T_t compared to *CmpOnly* approach because they operate on less number of linestrings (and hence less number of base points).

The third set of experiments measures the effect of different zoom levels Z_i at the rate of compression (ROC) for round 4. As shown in Figure 10.a, *cONa* achieves the highest rate of compression. The reason is that *cONa* requires to compress less number of linestrings (compared to *CmpOnly*) and each linestring has less excessive data (compared to *cOFa*). For example, *cONa* achieves more than 80% reduction in data size (ROC of 0.2) for Z_4 . Also the figure suggests that all three methods reach higher ROC as query window size grows. This is promising because real-world vector datasets are typically by far larger than our sample dataset.

Our last set of experiments show the effectiveness of our scheme by measuring the displacement error as zoom level increases. We have also constructed a part of our results at zoom level Z_3 . As Figure 10.b illustrates, the average error

resulted because of quantization is 0.00056 for the coarsest zoom level which corresponds to a 0.04 mile displacement. It also shows that the displacement error in both *cOFa* and *cONa* is more than *CmpOnly*. This is because the error propagates into longer sequence of differential values after applying *Lagg* on query results.

6 Conclusion and future work

Due to the bulky nature of vector data, many clients cannot afford transmission and storage of road networks for a relatively large area. General data compression schemes or hierarchical representation of roads at different levels of abstraction has its own disadvantages. We propose a novel approach which easily integrates with a spatial query processing system and compresses vector data significantly and fast. Our experiments show that depending on processing time/storage requirements, variants of our method can be used to guarantee high compression ratios (*cONa*) or fast response times (*cOFa*). We plan to address the problem of compressing other types of geometries such as polygons. Furthermore we are taking into consideration the client time for decoding and displaying the query results. We also aim at enhancing our approach to dynamically choose between the two variants at the query time.

References

- [1] S. Shekhar et al., "Vector Map Compression: A Clustering Approach", ACM-GIS'02, pp 74-80, 2002.
- [2] A. Akimov et al., "Reference Line Approach for Vector Data Compression", ICIP'02 2002.
- [3] Q. Zhu et al., "An Efficient Data Management Approach for Large Cybercity GIS", ISPRS, 2002.
- [4] D.A.Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc.IRE'40, pp. 1098-1101, 1952.
- [5] Q. Han et al., "A Multi-level Data Structure for Vector Maps", Proc. ACM-GIS'04, Washington DC, 2004.
- [6] T. Ai et al., "Progressive Transmission of Vector Data Based on Changes Accumulation Model", SDH, Leicester, Springer-Verlag,Berlin, pp. 85-96, 2003.
- [7] M. Bertolotto et al., "Progressive Transmission of Vector Map Data Over the World Wide Web", GeoInformatica, Springer, 5(4), pp. 345-373, Dec. 2001.
- [8] A. C. Paiva et al., "A Multiresolution Approach for Internet GIS Applications", Proc. DEXA 2004.
- [9] E. Puppo et al., "Towards a Formal Model for Multiresolution Spatial Maps", SSD'95, Portland, ME, pp152-169.
- [10] J. Persson, "Streaming of Compressed Multi-Resolution Geographic Vector Data", Proc. GEOINFORMATICS, Sweden, 2004.
- [11] A. Silberschatz, H. Korth, S. Sudarshan, Database System Concepts, McGraw Hill, 1998.