2017 IEEE 31st International Conference on Advanced Information Networking and Applications

A Compact-Trie-Based Structure for k-Nearest Neighbour Searching

Peng Gong
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta
T1K 3M4 Canada
Email: peng.gong@uleth.ca

Wendy Osborn
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta
T1K 3M4 Canada
Email: wendy.osborn@uleth.ca

***Abstract*—This paper proposes a k-nearest neighbour search method inspired by grid space partitioning and the compact-trie structure. A compact trie structure, and a k-nearest neighbour search strategy are presented. Then, a k-nearest neighbour search performance comparison is carried out against two well-known methods, using one million two-dimensional spatial points and finding up to 1000 nearest neighbours. The results of the comparison shows that the proposed compact-trie method can perform up to 300 times better when k is small, and up to 25 times better for larger k values, suggesting that the proposed method is suitable for low dimensions and location-dependent spatial queries in applications such as mobile computing.**

***Keywords*-nearest neighbour search; location-based services; compact trie; performance**

I. Introduction

Along with the popularity of smartphones is the rise of mobile computing. Location-aware mobile devices demand location-dependent services. The location-dependent spatial query is one such service that draws intensive research [1].

A spatial query is a query supported by a spatial database [2]. A spatial database is a database organized to store spatial data and optimized to facilitate spatial query processing. Spatial data, which represent objects in space, can be as simple as spatial points in arbitrary dimensions. For example, spatial data can be ($Longitude$, $Latitude$) pairs, representing points of interest (POIs) on a two-dimensional map. All POIs can be organized in a spatial database, and used to answer spatial queries.

Different types of spatial queries exist. One type of spatial query is the nearest neighbour search. The nearest neighbour search is defined as: given a set of points P in an n-dimensional space S and a metric to determine the distance between any two points in S, how to efficiently find the point in P which is nearest to an arbitrarily given query point q in S [3]. For example, a user may use their mobile device to locate the nearest POI, such as a restaurant or pub, based on their current location which serves as the query point.

A generalization of nearest neighbour search is the k-nearest neighbour search in which k points in set P nearest to an arbitrarily given query point q in S are found, where k can be 1, 2, ... up to P.

Several approaches for efficiently finding nearest neighbours to a query point have been proposed [4]–[11]. However, all require the creation of some type of spatial access method (e.g. k-d tree or the R-tree), which adds significant overhead in space requirements. The compact trie [12] is a more compact structure for storing information. Although traditionally used for indexing strings, they can be applied to a string representation of ($Longitude$, $Latitude$) coordinate data as well, which can result in good performance with lower space requirements.

Therefore, we propose a k-nearest neighbour search method, which is inspired by grid space partitioning and the compact trie structure. The current implementation of this method adopts an array-based data structure and a best-first nearest neighbour search scheme. Although it is currently limited to two-dimensional data, the result of the k-nearest neighbour search performance comparison shows significant improvement over both the brute-force based search method, and the benchmark k-d Tree search method. Because many location-dependent mobile services utilize two-dimensional geographic data, the proposed method may be particularly suitable for nearest-neighbour based location-dependent spatial queries in mobile computing.

The remainder of this paper proceeds as follows. Section II summarizes some related work in the area of k-nearest neighbour searching (in particular, approaches that utilize an index), and some background on the compact trie that is utilized in our work. Section III presents our compact-trie approach to k-nearest neighbour searching. Section IV presents the framework and results of the search performance evaluation. Finally, Section V concludes the paper and gives future research directions.

II. Related Work

In this section, we summarize existing strategies for k-nearest neighbour search. We also summarize the compact trie that we utilize in our work.

A. k-nearest neighbour approaches

Nearest neighbour approaches can be classified into structureless and structured approaches. Structureless approaches

1550-445X/17 $31.00 © 2017 IEEE
DOI 10.1109/AINA.2017.30

do not maintain a data structure to be utilized in the search. The most straightforward structureless approach is a Brute-force approach. Here, the distances between query point q and all points in set P are calculated first, before processing P to find the k-nearest neighbours. A list of k-nearest neighbours is kept, with a new nearest neighbour being inserted into the proper place. Although the space complexity is low, inserting into the list of nearest neighbours can be costly for higher values of k.

Structured approaches, on the other hand, utilize some type of data structure to improve performance, at the possible expense of both space complexity and the cost of constructing the index [4]–[11]. Burkhard and Keller [4] proposed three data structures for nearest neighbour search, which are equivalent to multi-way trees. Fukunaga [5] employed recursive decomposition of a search space and applied a branch-and-bound methodology to create the resulting search data structure. The core of the branch-and-bound methodology is that while systematically accessing and evaluating all candidate data, subsets of data are eliminated as early and often as possible, according to the continuously optimized bound(s) derived during the evaluation.

The first nearest neighbour search algorithm proposed for the k-d tree was proposed by Friedman *et al.* [6]. Sproull [7] improved upon this approach by noting that since Euclidean distance calculations are invariant under rotation, the planes that partition space along a particular dimension in the k-d tree can actually be arbitrary k-dimensional hyperplanes. Although these may lead to an improved partitioning of space, Sproull identifies the following limitations: 1) the additional cost of computing the distance between a point and an arbitrary hyperplane, and 2) choosing an arbitrary partition hyperplane perpendicular to an axis other than the coordinate axes can incur additional costly computations.

Nearest neighbour approaches using the R-tree are also proposed, including one from Brinkhoff *et al.* [8], which computes k-nearest neighbours via a spatial join operation.

Finally, some approaches can be applied to any hierarchical data structure. Roussopoulos *et al.* [10] propose a branch-and-bound k-nearnest neighbour search that can be applied to any tree-type data structure. It is a depth-first search strategy, which for every node visited, places its children on a queue in order of its distance from the query point. If a node is too far from the query point, it is pruned. Otherwise, it is visited further. This continues until k nearest neighbours have been found. Hjaltason and Samet [11] propose a similar best-first strategy that does not require the number of nearest neighbours to be known in advance.

B. Compact Trie

The trie [14] is a hierarchical data structure traditionally used for indexing strings. In a trie, every leaf node represents a string, while a non-leaf node contains a sequence of one or more characters that make up the prefix (i.e. beginning) of every string that is a descendent of this node. Fig. 1 depicts an example Trie for several strings. Note that the prefix of every node is its immediate ancestor.

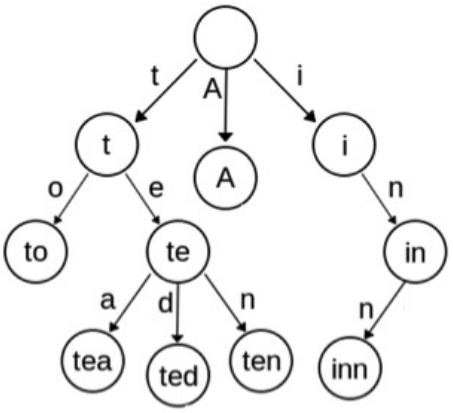

Figure 1. Trie for strings ”A”, ”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, and ”inn” (from [13])

For this work, a compact trie is utilized. A compact trie is a space-optimized trie, which is organized by requiring all non-terminal nodes to have more than one descendant [12]. Therefore, the compact trie will likely have fewer non-terminal nodes than its equivalent full trie, and searching within fewer non-terminal nodes would on average take less time. Although insertion and deletion operations within a compact trie involve additional adding, splitting, and merging operations, insertion is only required to construct the trie once, and deletion will not be needed (assuming a static data set).

Several approaches for implementing a trie have been proposed, including [15], [16]. We chose the double-array implementation of trie, proposed by Aoe [15]. The double array structure has a search time complexity similar to the conceptual trie (see Fig. 1) and a reasonable space complexity of: $number(trie\_nodes) * sizeof(string)$.

Here, a trie is represented by two arrays - BASE and CHECK. When constructing a double-array trie and performing basic trie operations (search, insertion, deletion), all must conform to both of the following rules. When traversing from a parent trie node S to its immediate child trie node T as a result of character C:

$$T = BASE[S] + CODE[C]$$

$$CHECK[T] = S$$

where CODE[C] represents a unique numerical code for character C. All trie nodes are mapped to the index values of BASE and CHECK. Special algorithms are developed to construct the double-array trie and perform basic operations within the trie according to the rule.

III. Compact-Trie-Based K-NN Search Method

In this section, a compact-trie-based k-nearest neighbour search method is proposed. The method is divided into three steps: (1) data preparation, (2) trie construction, and (3) search. We first present some preliminaries and motivation behind this approach before presenting the steps.

A. Preliminaries

The method is inspired by the intrinsic relationship between grid partitioning and Cartesian coordinates composed of positive pure decimal numbers.

Intuitively, an n-dimensional space can be partitioned by orthogonal lines into n-dimensional cells. For example, take a simple square-shaped two-dimensional space. Applying the Cartesian coordinate system, the original space before any grid partitioning can be defined by the Cartesian coordinates of its four corners: the lower-left corner $(0, 0)$, the lower-right corner $(1, 0)$, the upper-right corner $(1, 1)$, and the upper-left corner $(0, 1)$. Further, the original space can be assigned a label by using the Cartesian coordinates of its lower-left corner $(0, 0)$. Next, this space is partitioned by orthogonal lines into 10 by 10, 100 square-shaped, non-overlapping regions (i.e. cells) evenly. Each of these regions can be labeled by the Cartesian coordinates of its lower-left corner. For example, $(0.0, 0.1)$ is the label for the region $(0.0, 0.1)$,$(0.0, 0.2)$,$(0.1, 0.1)$ and $(0.1, 0.2)$, while $(0.9, 0.9)$ represents the region $(0.9, 0.9)$,$(0.9, 1.0)$,$(1.0, 0.9)$, and $(1.0, 1.0)$. Similarly, any one region, such as the one labeled as $(0.2, 0.3)$, can be partitioned further into 10 by 10, 100 square-shaped, non-overlapping sub-regions. Each sub-region can be denoted or labeled by the Cartesian coordinates of its lower-left corner, such as $(0.20, 0.30)$, $(0.20, 0.31)$, $(0.20, 0.32)$ $\ldots$ $(0.29, 0.39)$.

Conceptually, such recursive partitioning of a space could be performed infinitely. The only physical limit is the precision of a positive pure decimal number. The resulting region of space can be infinitely small so as to be deemed as a spatial point. Alternatively, any spatial point can be deemed as a certain region. For example, the spatial point $(0.20, 0.31)$ is the region labeled as $(0.20, 0.31)$. So the Cartesian coordinates of a spatial point composed of positive pure decimal numbers can be deemed as the label of a certain region resulted from such a grid partitioning.

The labels of the regions can be re-organized by making some modifications as follows, by removing the leading 0s (before the decimal point) and then interleaving the remaining digits. Using the grid partitioning example above, the labels of the regions at the first level of partitioning: $(0.0, 0.0)$, $(0.0, 0.1)$, $(0.0, 0.2)$, $\ldots$, $(0.9, 0.9)$ can each be modified to become 00, 01, 02, $\ldots$, 99 respectively. Similarly, the labels of the regions at the second level of partitioning: $(0.20, 0.30)$, $(0.20, 0.31)$, $(0.20, 0.32)$, $\ldots$, $(0.29, 0.39)$ can be modified to become 2300, 2301, 2302, $\ldots$, 2399. The common prefix 23 is the label of the first level region encompassing the 100 second level regions which can be uniquely denoted by the rest suffixes 00, 01, 02, $\ldots$, 99, respectively. Similar modifications can be applied to any region at any level of partitioning.

Here are some benefit resulting from these label modifications. The new labels are more succinct. New labels different in length indicate their corresponding regions are at different levels of partitioning. The longer a new label, the higher the partitioning level of the corresponding region, and the smaller the corresponding region. New labels equal in length indicate that their corresponding regions are at the same level of partitioning. New labels sharing a common prefix are within the same region, of which the new label is the common prefix. This approach is used to assemble Cartesian coordinates of spatial points composed of positive pure decimal numbers into the strings in our strategy.

Grid partitioning will not result in any partially overlapped regions. The regions at the same level of partitioning are non-overlapping. A region at a lower level of partitioning would either fully encompass a region at a higher level of partitioning, or not encompass it at all. A tree-type data structure is a natural fit for organizing the regions resulting from such a grid partitioning. The root node represents the entire space. Every other node can represent a certain region. The level of a node indicates the partitioning level of the region represented by the node. There is no direct connection between any pair of nodes at the same level. The direct connection between an upper level node and a lower level node can indicate the region represented by the upper level node encompasses the region represented by the lower level node.

The compact trie is adopted to store, index, and search spatial points. The spatial points are deemed as regions resulting from the grid partitioning described earlier. Every node of the compact trie represents a certain region and is associated with the new label of the region. The new label associated with any non-terminal node, except for the root node, must be the longest possible common prefix of the new labels associated with its immediate child nodes, and the length of the common prefix must be an integral multiple of the dimensionality of the spatial points. The region denoted by such a new label is actually the smallest possible region resulted from the same grid partitioning that encompasses all regions represented by the child nodes of the non-terminal node. So, constructing a compact trie from spatial points naturally groups certain spatial points and/or certain regions together, and naturally partitions the space, which is appealing to a search for a certain spatial point or region. It is worth mentioning that the level of a compact trie node does not indicate the partitioning level of the region represented by the node.

Last, referring back to the grid partitioning example, it is worth noting that the length of the new label of any region at the first partitioning level is 2, or the dimensionality 2

multiplied by the level of partitioning 1. The length of the new label of any region at the second partitioning level is 4, or the dimensionality 2 multiplied by the level of partitioning 2. To generalize, the length of the new label of any region at the nth partitioning level is the dimensionality multiplied by the level of partitioning n. That is why the length of the unique string associated with any compact trie node, except for the root node, must be an integral multiple of the dimensionality. So, the length of the longest possible common prefix referred in the previous paragraph must also be an integral multiple of the dimensionality, rather than the length of the longest common prefix. Therefore, the compact trie constructed in this proposed method is not literally following the compact trie definition because the string associated with a non-terminal node (except for the root node) in the compact trie is not the longest common prefix of all strings associated with its immediate child nodes.

B. Sample Point Set

For illustration purposes, and inspired by the ($Longitude$, $Latitude$) pair of the Global Position System (GPS) data format (e.g. (47.644548, -122.326897)), we generate a synthetic data set consisting of 15 two-dimensional spatial points, whose coordinates are represented as positive pure decimal numbers. Table I shows the data set. The number of digits after the decimal point is 6 in the present illustration, but can be larger if higher spatial resolution is needed for the application. In addition, we assume that all points in the data set are unique.

ID	Longitude	Latitude	String Format
1	0.001251	0.563585	050613255815
2	0.193304	0.808741	189038370441
3	0.585009	0.479873	548759080793
4	0.350291	0.895962	385905299612
5	0.822840	0.746605	872426864005
6	0.174108	0.858943	187548190483
7	0.710501	0.513535	751103550315
8	0.303995	0.014985	300134999855
9	0.091403	0.364452	039614440532
10	0.147313	0.165899	114675381939
11	0.147313	0.165890	114675381930
12	0.091403	0.374452	039714440532
13	0.091403	0.374552	039714450532
14	0.710501	0.514535	751104550315
15	0.091403	0.365452	039615440532

Table I
SAMPLE POINT SET

039614440532
039615440532
039714440532
039714450532
050613255815
114675381930
114675381939
187548190483
189038370441
300134999855
385905299612
548759080793
751103550315
751104550315
872426864005

Table II
SORTED STRING FORMATS

C. Data Preparation

We now describe our strategy, by first assembling the strings for each co-ordinate. We first normalize all co-ordinate values to positive pure decimal numbers between 0 and 1, and to as many digits after the decimal as required. Table I shows the co-ordinate values normalized to 6 digits after the decimal.

Next, the core step of data preparation is to, for each spatial point in the data set, assemble a string composed of digits only, based on its preprocessed positive pure decimal number coordinates, by interleaving the digits after the decimal point of all coordinates in an orderly fashion. The complete numeric information of the multi-dimensional coordinates of the spatial point will be preserved. For example, let us take spatial point 1 (0.001251, 0.563585) from Table I to illustrate the assembling process. The first digits after the decimal point of the two coordinates are 0 and 5. Interleave them to form a string 05. Next, the second digits after the decimal point of the two coordinates are 0 and 6. Interleave them in the same order to form a string 06. Append the string 06 to the string 05 to form a string 0506. Repeat the same interleaving and appending procedures for the remaining digits after the decimal point of the two coordinates, ultimately producing the final string 050613255815 for spatial point 1. Similarly, the remaining spatial points are processed.

After all co-ordinates are mapped to strings, the strings are sorted in ascending order according to their numeric values. Such an ordering naturally groups together strings sharing the longest common prefix. Table II shows the ordered strings for our example. Strings 039614440532 and 039615440532 are grouped together because they share the prefix 03961, while 039714440532 and 039714450532 are grouped together because they share the prefix 0397144. Similarly, strings 039614440532, 039615440532, 039714440532 and 039714450532 are grouped together because they share the prefix 039; while 114675381930 and 114675381939 are grouped together because they share the prefix 11467538193; 187548190483 and 189038370441 are grouped together because they share the prefix 18; and finally, 751103550315 and 751104550315 are grouped together because they share the prefix 75110.

The purpose of such an ordered output is to facilitate the construction of the compact trie, which is presented in the

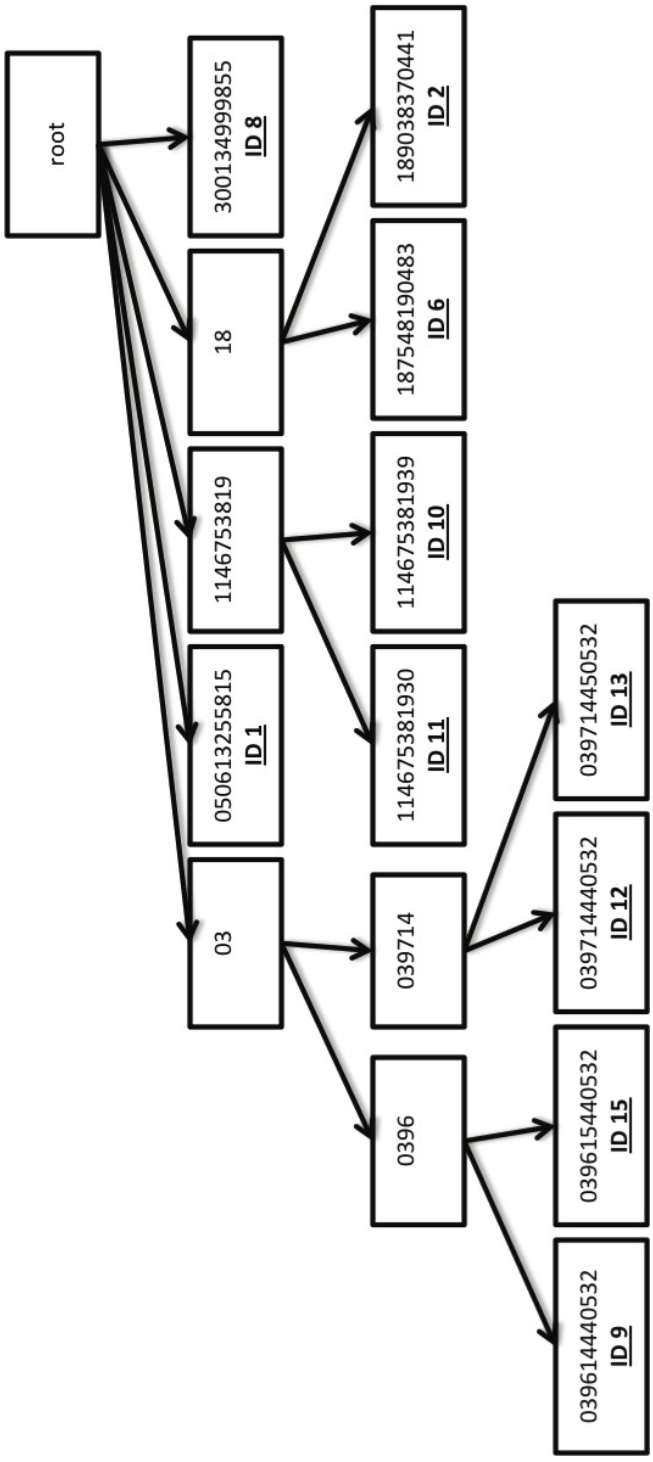

Figure 2. Trie Structure after Inserting the First Eight Strings from TABLE II

next section. It is worth mentioning that sorting strings in some way is frequently employed in the initial construction of a trie, when indexing a large number of strings [14].

D. Compact Trie Construction

Once the strings corresponding to the co-ordinates of each spatial point have been formed, they are used to create a compact trie.

Fig. 2 depicts a partially constructed compact trie after the first 8 strings from TABLE II have been inserted. Each node is represented by a rectangular box. There is always one and only one depth-0 node, the root node, which is associated with an empty string according to the definition of the trie. Every other node of the compact trie is associated with one unique string. The unique string associated with any non-terminal node (except for the root node) is the longest common prefix of all strings associated with its immediate child nodes. Every terminal node contains one unique string corresponding to one unique spatial point. Each also contains the unique ID associated with the spatial point uniquely corresponding to that terminal node.

After all 15 strings have been inserted, the resulting compact trie is composed of 22 nodes. There are 15 terminal nodes in total.

With respect to the arrows in Fig. 2, an arrow in the compact trie points from a parent node to its immediate child node. The directional arrow is not intended to suggest that the traverse between a parent node and its immediate child node is only in one direction. The directional arrow is used to indicate the parental-child relationship between the two nodes. The traversal between a parent node and its immediate child node is bi-directional. In other words, a tree traversal algorithm would be able to traverse either from a parent node to its immediate child node or from a child node to its immediate parent node. A node can be accessed both from its immediate parent node and from its immediate child node, if they do exist. If there is no arrow between two nodes, accessing one node from the other node must be via existing arrows or paths.

E. Searching the Compact Trie

For k-nearest neighbour searching the compact trie, we utilized the best-first nearest neighbor search (BFNNS) algorithm [17]. The core of the BFNNS algorithm is to build and maintain a priority queue. Applicable to all nodes, the shorter the minimum distance of a node to the query point, the higher the priority of the node in the priority queue.

Since a terminal node represents one unique spatial point, the minimum distance of a terminal node to the query point is certain and fixed. A non-terminal node represents one region, as explained earlier. Therefore, the minimum distance of a non-terminal node to the query point is the shortest distance from the boundary of the region to the query point. This minimum distance is also the minimum distance possible between any point within the region and the query point. The distance is calculated by re-forming the original co-ordinates from the string representation of the node.

There is no need to differentiate their priority in the priority queue when a terminal node and a non-terminal node have the same minimum distance to the query point; either one can be processed first because both would have to be processed anyway in order to find all k-nearest neighbours.

After the priority queue is initialized with the root node, the k-nearest neighbour search starts by always removing the first item of the priority queue until either all k-nearest neighbours have been found or the priority queue is depleted completely. If the item removed from the priority queue is a non-terminal node, all of its immediate child nodes are inserted into the priority queue first before any item is removed from the priority queue. The insertion will follow the same principles described above. If an item removed from the priority queue is a terminal node, the next nearest neighbour is found as the unique spatial point represented by the terminal node.

One advantage of the BFNNS algorithm is that the next best or the next nearest neighbour will always be the next terminal node (i.e. spatial point) identified from the priority queue. So, if a k-nearest neighbour search needs to identify more nearest neighbours, the search can resume from where it stops last time rather than start over again, which can be significant in continuous k-nearest neighbour search. This is significant in structures that have more than two children from a given node. In contrast, the depth-first nearest neighbour search (DFNNS) algorithm [17] is largely based on branch-and-bound and keeps pruning the branch determined to be outside the most updated bound. Any additional nearest neighbour would likely require the DFNNS to start over again because the branch containing the next nearest neighbour may have been pruned and cannot be recovered.

IV. Evaluation

In this section, we preliminarily evaluate the k-nearest-neighbour search performance using the proposed compact-trie structure. The experimental environment and tests are presented first, followed by the results of the evaluation.

A. Environment and Tests

We compare our structure versus both the brute-force method, and the k-d Tree method proposed in [17]. Brute-force provides the absolute base line performance that any other method must surpass, especially in a low number of dimensions, while the k-d Tree method can be considered a benchmark approach for processing k-nearest neighbour queries. The performance comparison is carried out using a Lenovo ThinkPad T420, with Intel(R) Core(TM) i5-2520M

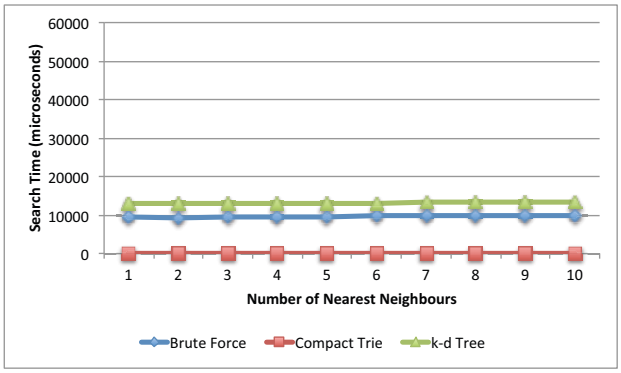

Figure 3. Search Performance, 1 to 10 Nearest Neighbours

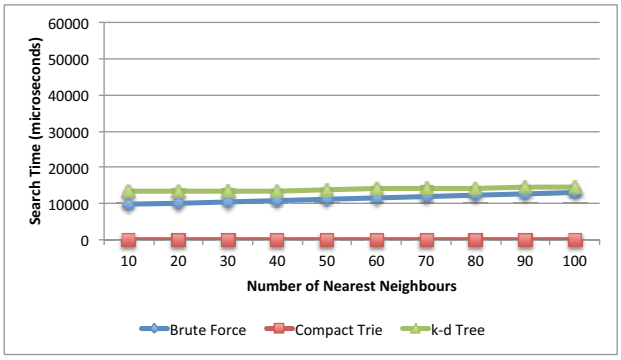

Figure 4. Search Performance, 10 to 100 Nearest Neighbours

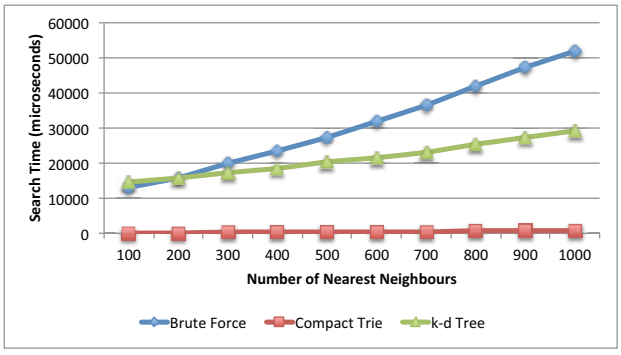

Figure 5. Search Performance, 100 to 1000 Nearest Neighbours

CPU at 2.50 GHz, 8.00 GB RAM, and running 64-bit Windows 7 Professional Service Pack 1. We utilize a synthetic data set of 1,000,000 randomly-generated points of interest (POIs). In addition, point queries are randomly generated for each test.

The test is that for a given two-dimensional query point, search for its k nearest neighbours in the POIs. For several k values ranging from 1 to 1000, both the compact-trie-based search and the brute-voce search are executed one thousand times, each time with a different randomly-generated query point. The average running time (in microseconds) of the program to complete one test for a given k value is used to measure the k-nearest neighbour search performance of the program.

B. Results

Figs. 3,4, and 5 presents the results of our evaluation. As we can see, the compact-trie-based method performs consistently better than both the brute-force based k-d Tree search methods:

- The best improvements are found for between 100 and 1000 nearest neighbours (Fig. 5) The compact-trie-based method performs at least 25 times better than both of the other approaches. We found that the execution times for the compact-trie approach are between approximately 115 microseconds for 100 nearest neighbours, up to 850 microseconds for 1000 nearest neighbours. For brute-force, the execution times are between approximately 13,000 and 52,000 for locating between 100 and 1000 nearest neighbours, respectively. For the k-d Tree search, the execution times are between approximately 14,500 and 29,000 microseconds for locating between 100 and 1000 nearest neighbours.
- For up to 100 nearest neighbours (Fig. 4), the compact-trie-based method performs at least 175 times better. For the compact-trie approach, the range of execution times is between 54 and 115 microseconds for between 10 to 100 nearest neighbours. For the brute-force and k-d Tree approaches, the range for the same k values is between 9,900 and 13,000 microseconds, and 13,200 and 15,000 microseconds, respectively.
- Even at the smaller numbers of nearest neighbours, the improvement is significant. The smaller the value of k, the higher the performance ratio is, up to 300 times better (as seen in Fig. 3. This suggests the compact-trie-based method may still have a significant advantage in k-nearest neighbour search applications where k is less than 100. Executions times for up to 10 nearest neighbours are between 40 and 54 microseconds for the compact-trie method, between 9,300 and 9,900 microseconds for the brute-force approach, and between 13,000 and 13,200 microseconds for the k-d Tree approach.

V. Conclusion

In this paper, a compact-trie-based k-nearest neighbour search method is proposed. Through the k-nearest neighbour search performance comparison against both the brute-force based and the k-d Tree methods, the compact-trie-based method shows consistent performance superiority.

The intrinsic relationship between grid partitioning and the Cartesian coordinates of spatial points composed of positive decimal numbers, and the natural space partitioning by a compact trie constructed from the modified Cartesian coordinates of spatial points composed of positive decimal numbers inspired me to devise the compact-trie-based method.

More theoretical analysis needs to be done to prove the better performance of the compact-trie-based method in theory. And the k-nearest neighbour search performance comparison result needs to be verified with more comparisons versus other strategies, in particular ones that utilize hierarchical data structures in their search strategies.

We found that applying the best-first search scheme to the compact-trie-based method seems relatively not that promising because for each non-terminal node traversed, a list of its child nodes sorted by their minimum distances to the query point may need to be created, which could be computationally expensive. It would be quite worthwhile to implement other approaches for comparison, for even further performance improvements.

So far, the implementation of the compact-trie-based method is limited to positive decimal numbers. Expanding the input to negative decimal numbers may just be a matter of conversion. And other numeral systems might be worthwhile exploring, especially the binary system because any information can be represented by bits or binary numbers, and bit-based storage and/or computing may be more efficient. Last, it is more significant to explore the potential applicability of the compact-trie-based method to high-dimensional data, to tackle the curse of dimensionality.

ACKNOWLEDGMENT

The authors would like to thank reviewers for their helpful comments in the current and previous drafts of this paper.

REFERENCES

[1] S. Ilarri, E. Mena, and A. Illarramendi, "Location-dependent query processing: Where we are and where we are heading," *ACM Computing Surveys*, vol. 42, no. 3, pp. 1–73, March 2010.

[2] S. Shekhar and S. Chawla, *Spatial Databases: A Tour*. Prentice Hall, 2003.

[3] M. L. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, Massachusetts: MIT Press, 1969.

[4] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching," *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[5] K. Fukunage and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, vol. 24, no. 7, pp. 750–753, 1975.

[6] J. H. Friedman, F. Baskett, and L. J. Shustek, "An algorithm for finding nearest neighbors," *IEEE Trans. Computers*, vol. 24, no. 10, pp. 1000–1006, 1975.

[7] R. Sproull, "Refinements to nearest-neighbor searching in k-dimensional trees." *Algorithmica*, vol. 6, no. 4, pp. 579–589, 1991.

[8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '93. New York, NY, USA: ACM, 1993, pp. 237–246.

[9] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[10] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *SIGMOD Record*, vol. 24, no. 2, pp. 71–79, May 1995.

[11] G. R. Hjaltason and H. Samet, "Ranking in spatial databases," in *SSD '95: Proceedings of the 4th International Symposium on Advances in Spatial Databases*. Springer-Verlag, 1995, pp. 83–95.

[12] D. Morrison, "Patricia – practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[13] "Trie," http://en.wikipedia.org/wiki/Trie, 2006, accessed 10-Apr-2016.

[14] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[15] J.-I. Aoe, "An efficient digital search algorithm by using a double-array structure," *IEEE Transactions on Software Engineering*, vol. 15, no. 9, pp. 1066–1077, 1989.

[16] J.-I. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software Practice and Experience*, vol. 22, no. 9, pp. 695–721, 1992.

[17] G. R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 517–580, 2003.