

GeoTree: a data structure for constant time geospatial search enabling a real-time mix-adjusted median property price index

Robert Miller*, Phil Maguire†
Department of Computer Science,
National University of Ireland, Maynooth,
Kildare, Ireland.
Email: *robert.miller@mu.ie, †phil.maguire@mu.ie

Abstract—A common problem appearing across the field of data science is k -NN (k -nearest neighbours), particularly within the context of Geographic Information Systems. In this article, we present a novel data structure, the GeoTree, which holds a collection of geohashes (string encodings of GPS co-ordinates). This enables a constant $O(1)$ time search algorithm that returns a set of geohashes surrounding a given geohash in the GeoTree, representing the approximate k -nearest neighbours of that geohash. Furthermore, the GeoTree data structure retains an $O(n)$ memory requirement. We apply the data structure to a property price index algorithm focused on price comparison with historical neighbouring sales, demonstrating an enhanced performance. The results show that this data structure allows for the development of a real-time property price index, and can be scaled to larger datasets with ease.

I. INTRODUCTION

Large scale datasets are a hot topic in computer science. Each one tends to present its own problems and intricacies [1]. The Nearest Neighbour (NN) problem is a well known and vital facet of many data mining research topics. This involves finding the nearest data point to a given point under some metric which measures the *distance* between data points. In the context of geospatial data, the NN problem often emerges in the form of geographical proximity search [2].

Real world geographic data is usually represented by a pair of GPS co-ordinates, which pinpoint any location on Earth with unlimited precision. As a result of their structure, computing the distance between pairs of points in order to find the *nearest neighbour* can be extremely slow on large datasets.

The problem often requires expansion to finding the k nearest neighbours (k -NN), which further increases the complexity by requiring a sorting of the distance matrix in order to extract a ranking of points by proximity. It is extremely computationally expensive to compute and rank these distances on large datasets [3]. A computationally cheap method of solving this problem would vastly improve the scalability of proximity based algorithms [2]. We propose a data structure which enables such cheap computation, the GeoTree, and explore its potential when applied to a real-world geospatial task.

II. BACKGROUND

A. Naive geospatial search

The distance between two pieces of geospatial data defined using the GPS co-ordinate system is computed using the *haversine* formula [4]. If we wish to find the closest point in a dataset to any given point in a naive fashion, we must loop over the dataset and compute the haversine distance between each point and the given, fixed point. This is an $O(n)$ computation. If the distances are to be stored for later use, this also requires $O(n)$ memory consumption. Thus, if the closest point to every point in the dataset must be found, this requires an additional nested loop over the dataset, resulting in $O(n^2)$ memory and time complexity overall (assuming the distance matrix is stored). If such a computation is applied to a large dataset, such as the 147,635 property transactions used in the house price index developed by [5], an $O(n^2)$ algorithm can run extremely slowly even on powerful modern machines.

As GPS co-ordinates are multi-dimensional objects, it is difficult to prune and cut data from the search space without performing the haversine computation. With a considerable portion of big data being geospatial in nature, geospatial algorithms and data structures are coming under increased research attention, with the amount of personal location data available growing by approximately 20% year-on-year according to the *McKinsey Global Institute* [6]. As such, exploring alternative methods of representing GPS co-ordinates is necessary to make algorithmic improvements.

B. GeoHash

A geohash is a string encoding for GPS co-ordinates, allowing co-ordinate pairs to be represented by a single string of characters. The publicly-released encoding method was invented by Niemeyer in 2008 [7]. The algorithm works by assigning a geohash string to a square area on the earth, usually referred to as a *bucket*. Every GPS co-ordinate which falls inside that bucket will be assigned that geohash. The number of characters in a geohash is user-specified and determines the size of the bucket. The more characters in the geohash, the smaller the bucket becomes, and the greater precision the

geohash can resolve to. While geohashes thus do not represent points on the globe, as there is no limit to the number of characters in a geohash, they can represent an arbitrarily small square on the globe and thus can be reduced to an exact point for practical purposes. Figure 1 demonstrates parts of the geohash grid on a section of map.

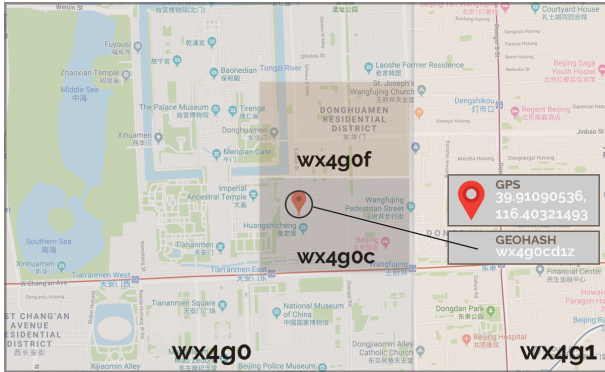


Fig. 1: GeoHash algorithm applied to a map

Geohashes are constructed in such a way that their string similarity signifies something about their proximity on the globe. Take the longest sequential substring of identical characters possible from two geohashes (starting at the first character of each geohash) and call this string x . Then x itself is a geohash (ie. a bucket) with a certain area. The longer the length of x , the smaller the area of this bucket. Thus x gives an upper bound on the distance between the points. We will refer to this substring as the *smallest common bucket* (SCB) of a pair of geohashes. We define the length of the SCB as the length of the substring defining it. This definition can additionally be generalised to a set of geohashes of any size. Furthermore, we define the SCB of a single geohash g to be the set of all geohashes in the dataset which have g as a prefix. We can immediately assert an upper bound of 123,264m for the distance between the geohashes in Figure 2, as per the table of upper bounds in the *pygeohash* package [8].

C. Efficiency improvement attempts

Geohashing algorithms have, over time, improved in efficiency and have been put to use in a wide variety of applications and research contexts [9] [10]. As stated by [2], the efficient execution of nearest neighbour computations requires the use of niche spatial data structures which are

$$\begin{aligned} \text{geohash 1: } & \underbrace{c_1 c_2 c_3}_{\text{SCB}} x_4 \dots x_n \\ \text{geohash 2: } & \underbrace{c_1 c_2 c_3}_{\text{SCB}} y_4 \dots y_n \\ \text{where: } & x_i \neq y_i \forall i \in \{4 \dots n\} \end{aligned}$$

Fig. 2: Geohash precision example

constructed with the proximity of the data points being a key consideration.

The method proposed by Roussopoulos et al. [2] makes use of *R-trees*, a data structure very similar in nature to the geohash [11]. They propose an efficient algorithm for the precise *NN* computation of a spatial point, and extend this to identify the exact k -nearest neighbours using a subtree traversal algorithm which demonstrates improved efficiency over the naive search algorithm. Arya et al. [12] further this research by introducing an approximate k -NN algorithm with time complexity of $O(kd \log n)$ for any given value of k .

A comparison of some data structures for spatial searching and indexing was carried out by [13], with a specific focus on comparison between the aforementioned *R-trees* and *Quadtrees*, including application to large real-world GIS datasets. The results indicate that the Quadtree is superior to the R-tree in terms of build time due to expensive R-tree clustering. As a trade-off, the R-tree has faster query time. Both of these trees are designed to query for a very precise, user-defined area of geospatial data. As a result they are still quite slow when making a very large number of queries to the tree.

Beygelzimer et al. [14] introduce another new data structure, the cover tree. Here, each level of the tree acts as a "cover" for the level directly beneath it, which allows narrowing of the nearest neighbour search space to logarithmic time in n .

Research has also been carried out in reducing the searching overhead when the exact k -NN results are not required, and only a spatial region around each of the nearest neighbours is desired. It is often the case that ranged neighbour queries are performed as traditional k -NN queries repeated multiple times, which results in a large execution time overhead [15]. This is an inefficient method, as the lack of precision required in a ranged query can be exploited in order to optimise the search process and increase performance and efficiency, a key feature of the GeoTree.

Muja et al. provide a detailed overview of more recently proposed data structures such as partitioning trees, hashing based *NN* structures and graph based *NN* structures designed to enable efficient k -NN search algorithms [16]. The *suffix-tree*, a data structure which is designed to rapidly identify substrings in a string, has also had many incarnations and variations in the literature [17]. The GeoTree follows a somewhat similar conceptual idea and applies it to geohashes, allowing very rapid identification of groups of geohashes with shared prefixes.

The common theme within this existing body of work is the sentiment that methods of speeding up k -NN search, particularly upon data of a geospatial nature, require specialised data structures designed specifically for the purpose of proximity searching [2].

III. GEOTREE

The goal of our data structure is to allow efficient approximate ranged proximity search over a set of geohashes. For example, given a database of house data, we wish to retrieve

a collection of houses in a small radius around each house without having to iterate over the entire database. In more general terms, we wish to pool all other strings in a dataset which have a maximal length SCB with respect to any given string.

A. High-level description

A GeoTree is a general tree (a tree which has an arbitrary number of children at each node) with an immutable fixed height h set by the user upon creation. Each level of the tree represents a character in the geohash, with the exception of level zero - the root node. For example, at level one, the tree contains a node for every character that occurs among the first characters of each geohash in the database. For each node in the first level, that node will contain children corresponding to each possible character present in the second position of every geohash string in the dataset sharing the same first character as represented by the parent node. The same principle applies from level three to level h of the GeoTree, using the third to h^{th} characters of the geohash respectively.

At any node, we refer to the path to that node in the tree as the *substring* of that node, and represent it by the string where the i^{th} character corresponds to the letter associated with the node in the path at depth i .

The general structure of a GeoTree is demonstrated in Figure 3. As can be seen, the first level of the tree has a node for each possible letter in the alphabet. Only characters which are actually present in the first letters of the geohashes in our dataset will receive nodes in the constructed tree. We, however, include all characters in this diagram for clarity. In the second level, the a node also has a child for each possible letter. This same principle applies to the other nodes in the tree. Formally, at the i^{th} level, each node has a child for each of the characters present among the $(i + 1)^{th}$ position of the geohash strings which are in the SCB of the current substring of that node. A worked example of a constructed GeoTree follows in Figure 4.

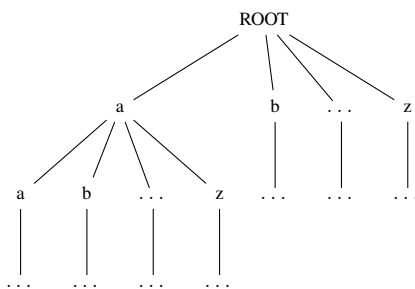


Fig. 3: GeoTree General Structure

Consider the following set of geohashes which has been created for the purpose of demonstration: $\{gc7j98, gc7j98, gd7j98, ac7j98, gc9aa, gc7j9d, ac7j98, gd7jya, gc9aa\}$. The GeoTree generated by the insertion of the geohashes above with a fixed height of six would appear as seen in Figure 4.

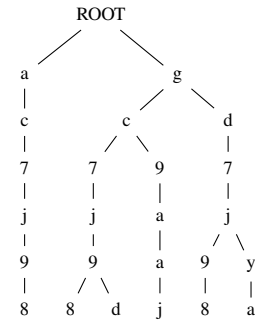


Fig. 4: Sample GeoTree Structure

B. GeoTree Data Nodes

The data attributes associated with a particular geohash are added as a child of the leaf node of the substring corresponding to that geohash in the tree, as shown in Figure 5. In the case where one geohash is associated with multiple data entries, each data entry will have its own node as a child of the geohash substring, as demonstrated in the diagram.

It is now possible to collect all data entries in the SCB of a particular geohash substring without iterating over the entire dataset. Given a particular geohash in the tree, we can move any number of levels up the tree from that geohash's leaf nodes and explore all nearby data entries by traversing the subtree given by taking that node as the root. Thus, to compute the set of geohashes with an SCB of length m or greater with respect to the particular geohash in question, we need only explore the subtree at level m along the path corresponding to that particular geohash. Despite this improvement, we wish to remove the process of traversing the subtree altogether.

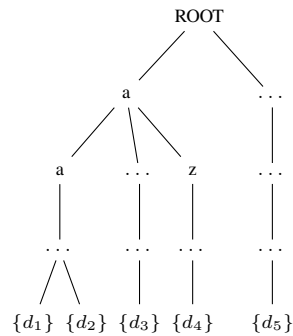


Fig. 5: GeoTree Structure with Data Nodes

C. Subtree Data Caching

In order to eliminate traversal of the subtree we must cache all data entries in the subtree at each level. To cache the subtree traversal, each non-leaf node receives an additional child node which we will refer to as the *list* (ls) node. The list node holds references to every data entry that has a leaf node within the same subtree as the list node itself. As a result, the list node offers an instant enumeration of every leaf node within the subtree structure in which it sits, removing the need to traverse

the subtree and collect the data at the leaf nodes. The structure of the tree with list nodes added is demonstrated in Figure 6 (some nodes and list nodes are omitted for the sake of brevity and clarity).

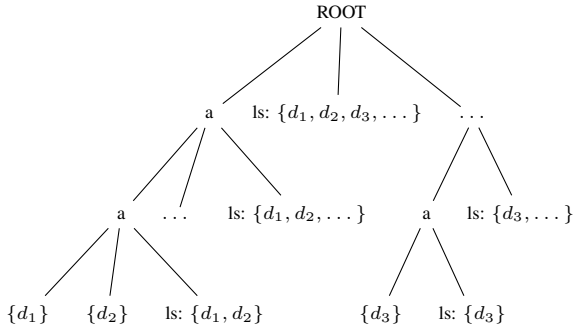


Fig. 6: GeoTree Structure with List Nodes

D. Retrieval of the Subtree Data

Given any geohash, we can query the tree for a set of nearby neighbouring geohashes by traversing down the GeoTree along some substring of that geohash. A longer length substring will correspond to a smaller radius in which neighbours will be returned. When the desired level is reached, the cached list node at that level can be queried for instant retrieval of the set of approximate k -NN of the geohash in question.

As a result of this structure’s design, the GeoTree does not produce a distance measure for the items in the GeoTree. Rather, it clusters groups of nearby data points. While this does not allow for fine tuning of the search radius, it allows a set of data points which are geospatially close to the specified geohash to be retrieved in constant time.

E. Memory Requirement of the Data Structure

As each geohash is associated with only one character at each level of the GeoTree, only one node on each level will hold that geohash’s data entry in its list node. Thus, each data entry is inserted into one single list node at every level of the tree. Given a tree of height h , this means that the data will be stored in h different list nodes in addition to the one leaf node which the data receives. If the dataset is of size n , then there will be $(h + 1) * n$ data entries stored in the tree. However, as the height of the tree is fixed and specified prior to the building of the tree, the overall memory requirement of the GeoTree is $O(n)$. This can be further improved to only n data entries stored by collecting a set of the data once in memory and filling the list nodes with a list of pointers to the data entries, if necessary.

F. Technical Implementation

To touch briefly on the implementation of GeoTree [18], a nested hash map structure is used in order to store the tree. The root node is the root hash map of the nest, with the hash keys at this level corresponding to the letters of the level one nodes. Each of these keys point to a value which is

another hash map containing keys corresponding to the level two letters of geohashes which have matching first letters with the parent key. The nesting process continues down to the leaf nodes (or terminal hash values in this case) in the same fashion described in subsection III-A. The final hash key (representing the last character of the geohash) points to the list of data entries associated with that geohash.

G. Time Complexity

1) *Building (Insertion)*: As hash maps offer $O(1)$ insertion, insertion of data at each level of the GeoTree is $O(1)$. Furthermore, due to the height of the tree, h , being constant and fixed, insertion of entries to the GeoTree is an $O(1)$ operation overall.

2) *SCB Lookup*: The $O(1)$ lookup of hash maps also means that the tree can be traversed in steps of $O(1)$ time. As the *list* nodes hold the SCB of every geohash substring possible from those in the dataset, and a maximum of h SCBs will need to be queried, it follows that any SCB lookup is also $O(1)$.

H. Comparison with set enumeration trees (SE-trees)

The SE-tree, or *set enumeration tree*, is a power set data structure which creates a branching tree of all possible subsets of a set of variables [19]. The set enumeration tree shares some basic similarities with the GeoTree. There are, however, fundamental differences between these data structures. The *set enumeration tree* is a structure defined on sets which, by definition, do not consider the ordering of variables. While the SE-tree contains all possible subsets of a set of variables, it does not contain all possible ordered collections of those variables. For example, $\{A, B\}$ will be contained in the SE-tree of variables $\{A, B, C\}$, yet $\{B, A\}$ will not appear in the tree.

In the case of the GeoTree, all possible combinations of characters must be considered, as geohashes are sensitive to ordering. The geohash $gh1992a$, for example, corresponds to an entirely different geographical location than $hg1992a$, despite both containing the same characters in slightly different ordering. The GeoTree is designed to support this sensitivity to ordering, whereas the *set enumeration tree* is not. Furthermore, the *set enumeration tree* has no provision for the cached list nodes of data, which is perhaps the most crucial feature of the GeoTree. Although many interesting algorithms for traversing the SE-tree are explored in [19], they are irrelevant in the present context, as the data structure in question is not designed for proximity search but for the purpose of classification.

IV. REAL-WORLD PERFORMANCE

A. Application: House Price Index Algorithm

In order to test the performance of GeoTree in practice, we applied it to the computation of an Irish house price index. House price indexes and forecasting models have come under increased attention from a data mining context, with a view to improve the current methods of calculating and forecasting property price changes. Such algorithms could

help identify price bubbles, facilitating preemptive measures to avoid another market collapse [20], [21], [22].

Many of these algorithms are based around the mix-adjusted median or central price tendency model, which requires a geospatial k -NN search [5], [23]. This approach is based on the principle that large amounts of aggregated data will cancel noise and result in a stable, smooth signal. It also offers the benefit of being less complex than the highly-theoretical hedonic regression model. It also requires less data than the repeat-sales model, in the sense of both quantity and time period spread [5], [23], [24].

Maguire et al. [5] introduced an enhanced central-price tendency model which outperformed the robustness of the hedonic regression method used by the Irish Central Statistics Office [25]. The primary limitation of this method is the algorithmic complexity and brute-force nature of the geospatial search, which impinges on its scalability to larger datasets, and restricts the introduction of further parameters. Our aim was to apply the GeoTree data structure to improve the execution time, scalability and robustness of this method. We re-implemented the algorithm used by [5] (described below), running the algorithm on the same data set (Irish Property Price Register) used in the original article as a control test for performance before introducing the GeoTree. For the purposes of algorithmic complexity calculation, we let n be the average number of house sales present in one month of the dataset, and let t be the number of months of data in the dataset.

Stage two (voting) of the **original** algorithm is executed as follows:

- ⇒ Iterate over each month, m , of the dataset (t operations)
 - ⇒ Iterate over each house, h , sold during m (n operations)
 - ⇒ Iterate over houses sold in m to find the nearest to h (n operations*)

Stage four (stratification) of the **original** algorithm is executed as follows:

- ⇒ Iterate over each month, m , of the dataset (t operations)
 - ⇒ Iterate over each house, h , soldHHP during m (n operations)
 - ⇒ Iterate over each month prior to m , m_p ($\frac{t-1}{2}$ operations)
 - ⇒ Iterate over houses sold in m_p to find the nearest to h (n operations*)

By introducing the GeoTree to the algorithm, the steps which formerly required an $O(n)$ iteration over all houses in the dataset to identify the nearest house (marked by an asterisk) now become an $O(1)$ GeoTree ranged proximity search operation. There is, however, a mild trade-off. Rather

than returning the closest property to the house in question, the GeoTree structure instead returns everything in a small area around the house (formally, it returns the maximal length non-empty SCB for that house’s geohash). The bucket can then be iterated over to find the true closest property, or an alternative strategy can be employed, such as taking the median price of all houses within the small area.

B. Performance Results

Table I compares the performance of the algorithms described previously with and without GeoTrees (on a database of 279,474 property sale records), including both single threaded execution time and multi-threaded execution time (running eight threads across eight CPU cores) on our test machine. The results using the GeoTree are marked with a + symbol.

C. Correlation

Despite the algorithmic alteration of taking the median price of a group of geohashed nearest neighbours, as opposed to the nearest neighbour per se, the house price indexes produced by the original algorithm and the GeoTree-enhanced version are very similar. Figure 7 shows both versions of the Residential Property Price Index (RPPI) superimposed. The two different versions yielded highly correlated outputs (Pearson’s $r = 0.999$, Spearman’s $\rho = 0.997$, Kendall’s $\tau = 0.966$), revealing that GeoTree succeeded in delivering an almost identical index to the original, though with major performance gains in execution time.

D. Scalability Testing

In order to test the scalability of the GeoTree, we obtained a dataset comprising 2,857,669 property sale records for California, and evaluated both the build and query time of the data structure. Table II shows mean build time and mean query time on both 10% (~285,000 records) and 100% (~2.85 million records) of the dataset. In this context, query time refers to the total time to perform **100 sequential queries**, as a single query was too fast to accurately measure.

The results demonstrate that the height of the tree has a modest effect on the build time, while dataset size has a linear effect on build time, thus supporting the claimed $O(n)$ build time with $O(1)$ insertion. Furthermore, query time is shown to remain constant regardless of both tree height and dataset size, with negligible differences in all instances.

V. CONCLUSION

We have shown that the GeoTree data structure introduced in this article offers an efficient $O(1)$ method for geospatial approximate k -NN search over a collection of geohashes. The application to a real-world property price index algorithm revealed significant reductions in execution time, and potentially opens the door for a real-time property price index. The data structure also performed well when applied to a much larger dataset, demonstrating its scalability. In conclusion, any data science problem which requires geospatial sampling around

TABLE II: Scalability Performance of GeoTree

Height h	4	5	6	7	8
Build Time (10%) ^a	17.63s (0.08s)	18.10s (0.10s)	18.46s (0.22s)	18.84s (0.08s)	19.39s (0.09s)
Build Time (100%) ^b	179.67s (0.58s)	183.80s (0.57s)	183.99s (0.52s)	192.06s (0.60s)	194.31s (0.94s)
Query Time (10%) ^c	5.1ms (0.3ms)	5.2ms (0.4ms)	5.3ms (0.9ms)	5.3ms (0.4ms)	5.3ms (0.5ms)
Query Time (100%) ^c	5.4ms (1.0ms)	5.3ms (0.9ms)	5.5ms (1.0ms)	5.7ms (1.3ms)	5.6ms (1.2ms)

^a Build Time (10%) is the total time to insert 10% of dataset (~285,000 records)

^b Build Time (100%) is the total time to insert 100% of dataset (~2.85m records)

^c Query Time consists of total time to execute 100 sequential neighbour queries on 10% and 100% of the dataset respectively

^d Times reported are in the format $\mu(\sigma)$ calculated over ten trials

a particular area can employ the GeoTree for $O(1)$ retrieval of approximate neighbours, potentially enabling, for example, fast retrieval of locations of interest to map users, or geo-targeted advertisement and social networking updates.

REFERENCES

- [1] D. J. Hand, *Data Mining Based in part on the article Data mining by David Hand, which appeared in the Encyclopedia of Environmetrics*. American Cancer Society, 2013.
- [2] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *SIGMOD Rec.*, vol. 24, no. 2, pp. 71–79, 5 1995. [Online]. Available: <http://doi.acm.org/10.1145/568271.223794>
- [3] M. Safar, "K nearest neighbor search in navigation systems," *Mobile Information Systems*, vol. 1, no. 3, pp. 207–224, 2005.
- [4] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957. [Online]. Available: <http://www.jstor.org/stable/2309088>
- [5] P. Maguire, R. Miller, P. Moser, and R. Maguire, "A robust house price index using sparse and frugal data," *Journal of Property Research*, vol. 33, no. 4, pp. 293–308, 2016. [Online]. Available: <https://doi.org/10.1080/09599916.2016.1258718>
- [6] J.-G. Lee and M. Kang, "Geospatial big data: Challenges and opportunities," *Big Data Research*, vol. 2, no. 2, pp. 74 – 81, 2015, visions on Big Data. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S2214579615000040>
- [7] G. Niemeyer. (2008) geohash.org is public! Accessed: 2019-05-02. [Online]. Available: <https://blog.labix.org/2008/02/26/geohashorg-is-public>
- [8] W. McGinnis. (2017) Pygeohash. [Python]. [Online]. Available: <https://github.com/wdm0006/pygeohash>
- [9] R. Moussalli, M. Srivatsa, and S. Asaad, "Fast and flexible conversion of geohash codes to and from latitude/longitude coordinates," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 5 2015, pp. 179–186.
- [10] R. Moussalli, S. W. Asaad, and M. Srivatsa, "Enhanced conversion between geohash codes and corresponding longitude/latitude coordinates," US Patent US20160283515A1, 2015. [Online]. Available: <https://patents.google.com/patent/US20160283515>
- [11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984. [Online]. Available: <http://doi.acm.org/10.1145/971697.602266>
- [12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, Nov. 1998. [Online]. Available: <http://doi.acm.org/10.1145/293347.293348>
- [13] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: a comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 546–557.
- [14] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 97–104. [Online]. Available: <http://doi.acm.org/10.1145/1143844.1143857>
- [15] J. Bao, C. Chow, M. F. Mokbel, and W. Ku, "Efficient evaluation of k-range nearest neighbor queries in road networks," in *2010 Eleventh International Conference on Mobile Data Management*, 5 2010, pp. 115–124.
- [16] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 11 2014.
- [17] A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, and S. Muthukrishnan, "40 years of suffix trees," *Communications of the ACM*, vol. 59, no. 4, pp. 66–73, 2016.
- [18] R. Miller. (2020) Geotree data structure code implementation. [Python]. [Online]. Available: <https://github.com/robertmiller72/GeoTree>
- [19] R. Rymon, "Search through systematic set enumeration," 1992.
- [20] P. Klotz, T. C. Lin, and S.-H. Hsu, "Modeling property bubble dynamics in greece, ireland, portugal and spain," *Journal of European Real Estate Research*, vol. 9, no. 1, pp. 52–75, 2016. [Online]. Available: <https://doi.org/10.1108/JERER-11-2014-0038>
- [21] W. E. Diewert, J. de Haan, and R. Hendriks, "Hedonic regressions and the decomposition of a house price index into land and structure components," *Econometric Reviews*, vol. 34, no. 1-2, pp. 106–126, 2015. [Online]. Available: <https://doi.org/10.1080/07474938.2014.944791>
- [22] A. Jadevicius and S. Huston, "Arima modelling of lithuanian house price index," *International Journal of Housing Markets and Analysis*, vol. 8, no. 1, pp. 135–147, 2015. [Online]. Available: <https://doi.org/10.1108/IJHMA-04-2014-0010>
- [23] Y. M. Goh, G. Costello, and G. Schwann, "Accuracy and robustness of house price index methods," *Housing Studies*, vol. 27, no. 5, pp. 643–666, 2012. [Online]. Available: <https://doi.org/10.1080/02673037.2012.697551>
- [24] N. Prasad and A. Richards, "Improving median housing price indexes through stratification," *Journal of Real Estate Research*, vol. 30, no. 1, pp. 45–72, 2008. [Online]. Available: <https://ideas.repec.org/a/jre/issued/v30n12008p45-72.html>
- [25] N. O'Hanlon, "Constructing a national house price index for ireland," *Journal of the Statistical and Social Inquiry Society of Ireland*, vol. 40, pp. 167–196, 2011. [Online]. Available: <http://hdl.handle.net/2262/62349>

TABLE I: Complexity and performance of the algorithms

Algorithm	Complexity	μ (1 core) ^a	σ ^b	μ (8 cores) ^a	σ ^b
Voting	$O(n^2t)$	233.54 seconds ^c	2.37%	46.73 seconds ^c	1.69%
Voting⁺	$O(nt)$	12.78 seconds ^c	1.68%	3.02 seconds ^c	0.69%
Stratify	$O\left(\frac{n^2t(t-1)}{2}\right)$	29.03 hours	2.41%	4.19 hours	1.89%
Stratify⁺	$O\left(\frac{nt(t-1)}{2}\right)$	\sim 0.05 hours (163.89s)	1.71%	\sim 0.01 hours (39.63s)	0.85%
Overall	$O\left(\frac{n^2t(t+1)}{2}\right)$	29.11 hours	2.43%	4.21 hours	1.90%
Overall⁺	$O\left(\frac{nt(t+1)}{2}\right)$	\sim 0.05 hours (177.73s)	1.67%	\sim 0.01 hours (43.71s)	0.79%

^a Execution times reported are the mean (μ) of ten trials.

^b Standard deviation (σ) reported as a percentage of the mean (μ).

^c **Includes build time** for the dataset array / GeoTree on the dataset, as applicable.

^d All algorithms computed using an AMD Ryzen 2700X CPU.

^e All algorithms executed on the Irish Residential Property Price Register database of **279,474 property sale records** as of time of execution.

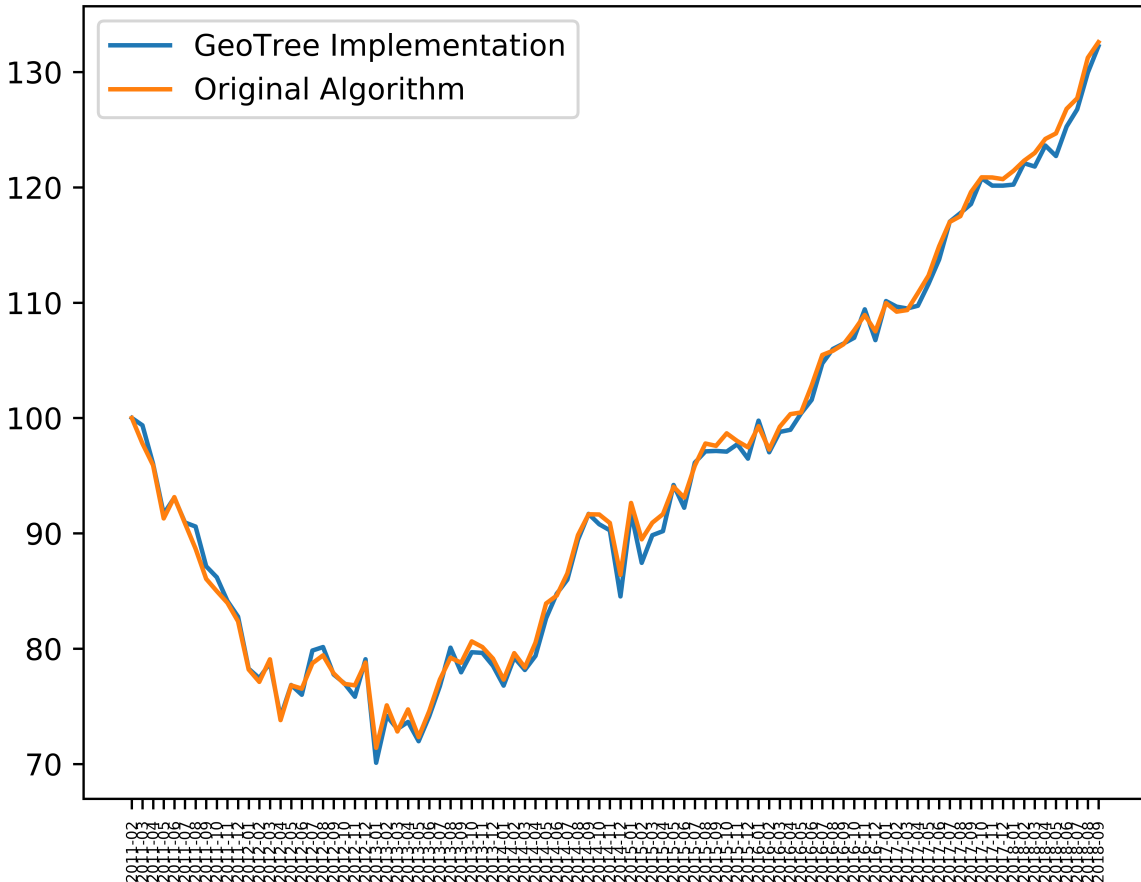


Fig. 7: Irish RPPI (GeoTree vs Original), from 02-2011 to 09-2018