THE AREA CODE TREE FOR APPROXIMATE NEAREST NEIGHBOUR SEARCH IN DENSE POINT SETS

FATEMA RAHMAN
Bachelor of Science, Jahangirnagar University, 2007

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

THE AREA CODE TREE FOR APPROXIMATE NEAREST NEIGHBOUR SEARCH IN DENSE POINT SETS

FATEMA RAHMAN

Date of Defence: December 21, 2017

Dr. W. Osborn Supervisor	Associate Professor	Ph.D.
Dr. J. Rice Committee Member	Professor	Ph.D.
Dr. D. P. O'Donnell Committee Member	Professor	Ph.D.
Dr. H. Kharaghani Chair, Thesis Examination Committee	Professor	Ph.D.

Dedication

I dedicate this thesis to my baby girl Sarah Zaman.

Abstract

Location based Services (LBSs) have become very popular due to the rapid development of wireless technology and mobile devices. A LBS provides results to a user of a mobile device (e.g.smart phone, tablet) based on their location, interests, and the type of query being performed. For example, a user may want to know the location of the closest restaurant to them. Sometimes the user may also be happy with another suggestion that may not be the closest but close enough to satisfy them. This is an example of an approximate nearest neighbour search. In this thesis, we propose a spatial data structure the Area Code Tree which is a trie-type structure. The Area Code Tree stores Points of Interest (POIs) that are represented in area code format. We also present the algorithms for mapping the area code of a POI, inserting and building an Area Code Tree, and approximate nearest neighbour search. Next we evaluate the Area Code Tree for accuracy, tree construction time, and compare its search performance with the Brute Force Method. We find that the average search time for Area Code Tree in locating nearest neighbour is very low and constant regardless of the number of POIs in the index. In addition, the Area Code Tree can achieve up to 60% accuracy for locating the nearest neighbour in dense point sets. This makes the Area Code tree an excellent candidate for continuous approximate nearest neighbour search for location-based services.

Acknowledgments

I would like to give a special thanks to my supervisor, Dr. Wendy Osborn, for all her ideas, advices, and research direction throughout the journey of pursuing my Masters (Thesis) degree in Computer Science from the Department of Mathematics and Computer Science at the University of Lethbridge. I also would like to thank my committee members, Dr. Jackie Rice and Dr. Daniel Paul O'Donnell, for their valuable advices. I am greatly thankful to the School of Graduate Studies of the University of Lethbridge for all the financial support they provided during my research. I want to thank my spouse Md. Rishad Zaman, my parents, and my sisters for their encouragement and moral support during my study at the University of Lethbridge. Last but not least, I am extremely grateful to almighty Allah for giving me the opportunity to be a part of this wonderful project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to the rapid development of wireless technology and mobile devices, Location based Services (LBSs) have become very popular. LBSs are services that return results based on certain location information and query types. The environment of a LBS is classified into three types according to the mobility of the clients and the data objects: 1) Static client and static data objects, 2) moving client and static data objects, and 3) moving client and moving data objects. In a moving client and static data objects environment, the user continuously changes their location and submits a new query to the server for obtaining updated static data information. Consider two scenarios: (i) A driver in a GPS-equipped car issues a continuous query to find the nearest gas station while driving in a city, and (ii) a tourist walking through a city center uses a location-aware mobile device to issue a continuous query for the nearest restaurant. These queries are examples of exact continuous nearest neighbour queries.

Sometimes users may also be happy to find points of interest (POIs) that may not be the closest but close enough to satisfy them. This is an example of an approximate nearest neighbour search. These queries are sent to the server through a wireless network to be processed with the result returned to the user. As the user moves around, their nearby POIs change, so the result must be updated continuously on their device. The conventional approach of providing up-to-date results to a mobile user is to submit a new query to the server after every location update. This could cause high network overhead and extra processing effort. Although repeated searching is not desirable, but it may be the only option available

when mobile devices have very limited storage capacity. Therefore, it is very important to use an efficient query processing strategy in the server that can be re-executed repeatedly.

Many strategies have been proposed for continuous nearest neighbour processing for location based services [20, 22, 21, 27, 17, 13, 15]. Some limitations of these strategies include repeated searching, caching a significant amount of data on the mobile device in order to avoid query re-submission, and the requirement to know the query trajectory in advance.

In this thesis, a spatial data structure the Area Code Tree[1] is proposed for storing and managing points of interest (POI) data for the purpose of continuous nearest neighbour processing. The Area Code Tree is a Trie-type structure, in which all POIs are represented by area codes. An area code is a sequence of digits that represents the relative location of a point in space. We index the area code data of a POI instead of actual spatial values (i.e. longitude and latitude) to reduce the search for a nearest neighbour to a sequence of simple numeric comparison. We also present the algorithms for mapping the area codes of a POI, inserting and building an Area Code Tree, and approximate nearest neighbour search. Finally, the Area Code Tree is evaluated for accuracy, tree construction time and its search performance is compared with the Brute Force Method which is a basic benchmark for processing nearest neighbour queries.

Twenty-one data sets that represent collections of different POIs across New Zealand are used for our evaluation. Tree construction is performed by inserting one POI area code at a time into the Area Code Tree. The accuracy of the Area Code Tree is measured by calculating the percentage of times that an accurate nearest neighbour is found. We compare the nearest neighbour search performance of the Area Code Tree with the Brute Force Method. Our evaluation results show that the Area Code Tree significantly outperforms the Brute Force Method. Regardless of the number of POIs in the index, the average search time for Area Code Tree is very low and constantly less than 10ms. Our evaluation results

[1]The paper [19] based on this work won the Best Paper Award at SEDE, 2016.

also show that the Area Code Tree achieves higher accuracy as the data set increases in size. In particular, for dense POI sets, the Area Code Tree can achieve up to 60% accuracy for locating the nearest neighbour.

The remainder of this thesis proceeds as follows. Chapter 2 summarizes related works in the area of continuous nearest neighbour processing and other background that are necessary for our work. Chapter 3 presents the Area Code Tree, the algorithms for mapping area code of a POI, inserting and building the Area Code Tree, and approximate nearest neighbour search. Chapter 4 presents the strategy and results of performance evaluation of the Area Code Tree. Chapter 5 concludes the thesis and presents future research directions.

Chapter 2

Related Work and Background

In this chapter, we summarize related work in continuous nearest neighbour searching for location-based services. Although nearest neighbour strategies have been proposed in other contexts, they are considered outside of the scope of this work. In addition, we summarize other background that is necessary for our work.

The k-Nearest Neighbour (kNN) query is one of the most popular query types in location-based services, where a user based on their location issues a query to the service provider for the k-nearest objects of interest to user's current location [1]. A moving k-Nearest Neighbour (MkNN) query is a continuous kNN query issued by a moving object. There are two general approaches for processing MkNN queries:

i. The Sampling-based method, which processes the MkNN query as a kNN query at sampled locations. This method does not provide answers between sampled locations. In order to provide a continuous answer to the query, a high sampling rate is required, which makes the method inefficient due to the frequent processing of kNN queries.

ii. The safe region based-method provides an answer with a *safe region*. As long as the query point stays in the safe region, the answer remains the same. When the query point moves out of the safe region, another answer with its associated safe region is returned.

2.1 Early Techniques for Processing M*k*NN Queries

Song and Roussopoulos [20] introduce four static-branch-and-bound methods that use sampling to solve the continuous k-nearest neighbour problem. These methods retrieve redundant data entries and utilize caching, which greatly reduces the cost of query re-evaluation. To represent the movement of the query point q, the authors use a periodical sampling technique, where the total time period is divided into equal intervals, and each break between time intervals is called a "*sampled position*". Using the information from two consecutive sampled positions, the location of the query point q is calculated using splines. At sampled position t, q_t is the location of q, n is the total number of sites and S is site set. At any time t, the sites are sorted in S in ascending order based on their distance to q_t. The search upper bound is ε_t, which means the distance of kth nearest neighbour to query point q will be no more than ε_t, (i.e. $D_t(k) \leq \varepsilon_t$).

Song and Roussopoulos propose the following algorithms [20]: Fixed Upper Bound, Lazy Search, and Pre-fetching Search, which are best for unpredictable query object movement, and Dual Buffer Search, which is suitable for calculating the next position of the moving query object when the speed and time interval length of the object are available.

2.1.1 Fixed Upper Bound (FUB) Algorithm

The Fixed Upper Bound (FUB) algorithm sets the search bound to infinity if no history of previous sampled positions is available. Otherwise it can set a smaller search bound initially from the search result at the previous sampled position. At sampled position t, the location of query point is q_t. The maximum distance to the k sites to q_t is $D_t(k)$, which is represented in Figure 2.1 by the solid (i.e. inner) circle [20]. After the search the values of q_t and $D_t(k)$ are recorded and used to calculate the search bound at the next sampled position. At sampled position $t+1$, the new position of the query point is q_{t+1}, the distance between locations q_t and q_{t+1} is δ, and the search bound is $\varepsilon_{t+1} = D_t(k) + \delta$ is shown by a dashed (i.e. outer) circle in Figure 2.1.

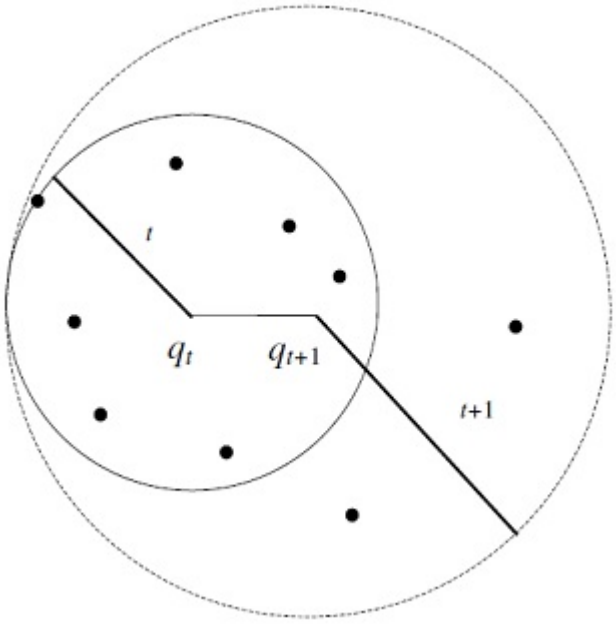


Figure 2.1: ε_{t+1} does not need to be infinite [20].

2.1.2 Lazy Search (LS) Algorithm

At sampled position t, the Lazy Search (LS) algorithm stores the query result set, location of the query point q_t, and the maximum distance of k sites to query point $D_t(k)$, and the minimum distance of sites that outside of k sites set to query point $D_t(k+1)$ in the buffer. At sampled position $t+1$, the query point moves to q_{t+1}. Next, the algorithm checks the validity of the previous query result for the new query. If the distance between q_t and q_{t+1} is no more than $[\{D_t(k+1)-D_t(k)\}/2]$, the previous query result is still valid for the new query. Otherwise, the algorithm finds the sites from the previous query result set that are still valid for the new query. Since, the new query result contains some sites p_i with $D_t(i) < D_t(k+1) - 2\delta$, where δ is the distance between q_t and q_{t+1}. Next, it stores the new query result set, the values of $D_{t+1}(k)$, $D_{t+1}(k+1)$, and q_{t+1} in the buffer and waits for the next query.

2.1.3 Pre-Fetching Search (PFS) Algorithm

According to the Pre-Fetching Search (PFS) algorithm, at sampled position $t = 1$, a search is performed for the m nearest neighbours, where $m > k$. The location of the query point q_1 along with maximum distances $D_1(k)$ and $D_1(m)$ are stored. From this result set, k nearest neighbours are identified. Next at sampled position $t > 1$, the algorithm checks the requirement for updating the result set for q_t. If the result is valid, it does not need to update the result set in the buffer. In Figure 2.2 the sites which are in the dashed circle is already stored in the buffer. Otherwise, a new static m nearest neighbour search is executed and the new result set, the location of the query point, and distance $D_t(k)$ and $D_t(m)$ are stored in the buffer.

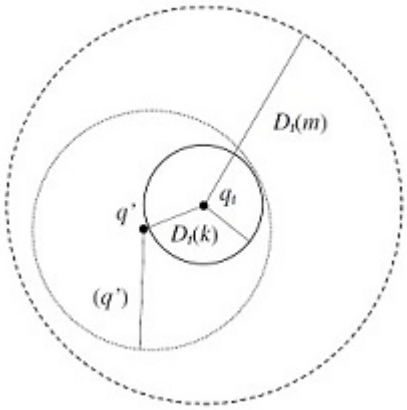


Figure 2.2: In some cases, there is no need to update the buffer [20].

2.1.4 Dual Buffer Search (DBS) Algorithm

According to the Dual Buffer Search (DBS) algorithm, two buffers of the same size k are maintained. At sampled position 1, a static k nearest neighbour search algorithm is executed, and the result set is stored in the first buffer. In the meantime, the next possible position of the query point $q\prime$ is calculated, and the set of k nearest neighbours of $q\prime$ are stored in the second buffer. At sampled position $t > 1$, the contents of the second buffer is moved to the first buffer, and the distance of the k-th neighbour to q_t is used as the next search bound. At the same time, according to the next possible position of moving query

point q'_{t+1}, the k nearest neighbours are sorted and stored in the second buffer. After that, a new k nearest neighbour search is executed and both buffers are updated.

2.1.5 Discussion and Experimental Results

The authors compare the four algorithms and perform an experimental evaluation of them [20]. FUB always executes a new search at new sampled positions with a better initial search bound, while DBS follows the same idea with a lower initial search bound. The LS and PFS are relatively expensive, since they need to collect extra information such as the distance of the $k+1$-th neighbour to a query point. However, by using these two algorithms at some sampled positions, new searches do not need to be executed as the stored result set in the buffers can be used. The authors refer to these sampled positions as no search positions. The algorithms are evaluated by varying four parameters: uncertainty factor (r), number of sites (i.e. points of interest), speed of the moving object (i.e. query point), and k.

Experiment results show that the FUB, DBS, and LS algorithms perform better than the pure static method by a factor of 3 for a large number of sites. When the moving pattern of a query point is predictable, DBS outperforms FUB. However, when the number of sites is smaller and the speed of the query point is slower, LS performs better than FUB. PFS performs better when the percentage of *no search position* is large.

2.2 Time-Parameterized (TP) *k*NN Queries

Tao and Papadias [22] propose a general framework for time-parameterized queries in spatio-temporal databases, and their application in *k*NN searching. In a dynamic environment, the results of conventional spatial queries may become invalid due to the movements of objects and queries. A time-parameterized (TP) query returns the current query result, the validity period of that result and the change of the result at the end of the validity period. It is represented as $< R, T, C >$, where R is the set of objects satisfying the corresponding

spatial query, T is the validity time of R and C is the result change at T. The client can incrementally compute the next result from the sets R, and C. The authors give the following example of a TP nearest neighbour query in Figure 2.3 for the static object set. In Figure 2.3 the query point q is moving east with speed 1, and the point d is the current nearest neighbour of q. For a nearest neighbour query, the influence time of object o, $T_{INF}(o,q)$ is time that it becomes closer to query q than the current nearest neighbour. For example, the influence time of point g is 3, because at time unit 3, g will be closer to q than d, if the result does not change due to another object, as shown in Figure 2.4. At time unit 3, f is the nearest neighbour to q. Therefore, f (in the time-parameterized component, C) is invalidating the current query result, since it is the point with the smallest influence time.

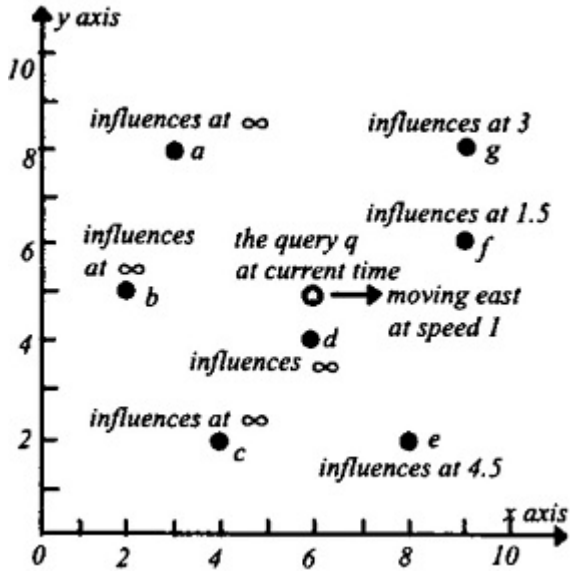


Figure 2.3: TP nearest neighbour query [22].

2.2.1 The TP Nearest Neighbour Query

Tao and Papadias [22] also propose a k-nearest neighbour search algorithm, which uses the R-tree [2] and is based on the branch-and-bound framework. The algorithm uses two distance metrics: (i) *mindist* represents the minimum distance between the query q and any other object in the subtree of entry E, and (ii) *minmaxdist* is the minimum distance from q within which an object in the subtree of E must be found. E is an entry in an R-tree node,

which has a subtree that contains objects.

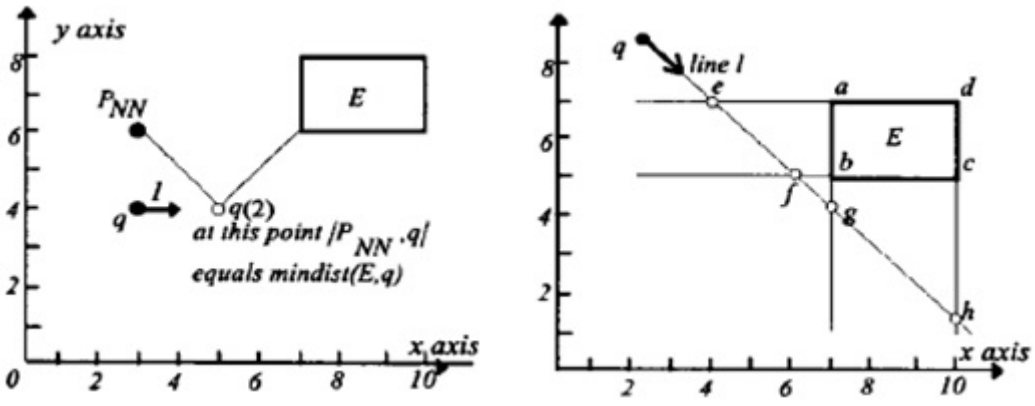
$T_{INF}(O,q)$ is the influence time of an object o with respect to a query q. The expiry time of the current result is the minimum influence time of all objects. Therefore, the time-parameterized component of a TP query can be reduced to a nearest neighbour problem by treating $T_{INF}(O,q)$ as the distance metric. The aim is to find the objects (C) with the minimum $T_{INF}(T)$. These are the candidates that may generate the change of the result at the expiry time (by adding to or deleting from the previous answer set). T_{INF} for intermediate entries E is defined in a way similar to mindist in NN search. $T_{INF}(E,q)$ is the minimum influence time $T_{INF}(O,q)$ of any object o that may lie in the subtree of E. The authors define P_{NN} as the current nearest neighbour of a query point q, and $o(t)$, $q(t)$, and $P_{NN}(t)$ are the positions of object o, query q and P_{NN} at time t respectively. $T_{INF}(o,q)$ has the minimum time t if it satisfies the conditions:

$$dist(o(t),q(t)) \leq dist(P_{NN},q(t)) \text{ and } t \geq 0.$$

If no time t satisfies the inequality, $T_{INF}(o,q)$ is set to ∞.

In case of non-leaf entries, $T_{INF}(E,q)$ is the earliest time when one or more objects of subtree E will get closer to q than P_{NN}. $T_{INF}(E,q)$ is the minimum time t if it satisfies the conditions:

$$dist(E(t),q(t)) \leq dist(P_{NN},q(t)) \text{ and } t \geq 0.$$



(a) Example of $T_{INF}(E,q)$ (b) Different cases of mindist

Figure 2.4: An example of the approximate mindist between q and E [22].

Figure 2.4 shows an example of the approximate mindist between q and E. To calculate the mindist between an edge of E and q the following rules are applied [21]:

i. If the mindist is computed with respect to a corner point, the closest edges are chosen and their connected corner point would be the expected corner point.

ii. If the mindist is computed with respect to an edge, the closest edge is chosen. The mindist is the distance between the query point and the edge.

Figure 2.4(a) shows $T_{INF}(E,q)$ for a non-leaf entry E; $mindist(E,q) = dist(q,P_{NN})$ and Figure 2.4(b) shows the Minimum Bounding Rectangle (MBR) of E with corner points a, b, c, and d, with query point q moving on line l. When q passes the point e, $mindist(E,q)$ is computed with respect to corner point a. When q is on line segment between e and f, then $mindist(E,q)$ is the distance between q and the edge ab of E.

2.2.2 Extension to *k*NN Queries

The authors extend their strategy to handle a to kNN query. Given P_{NN1}, P_{NN2},....., P_{NNk}, the current k nearest neighbours to q, the influence time T_{INFi} of object o is calculated with respect to each of the k nearest neighbours. $T_{INF}(o,q)$ is the minimum of T_{INF1}, T_{INF2},....., T_{INFk}.

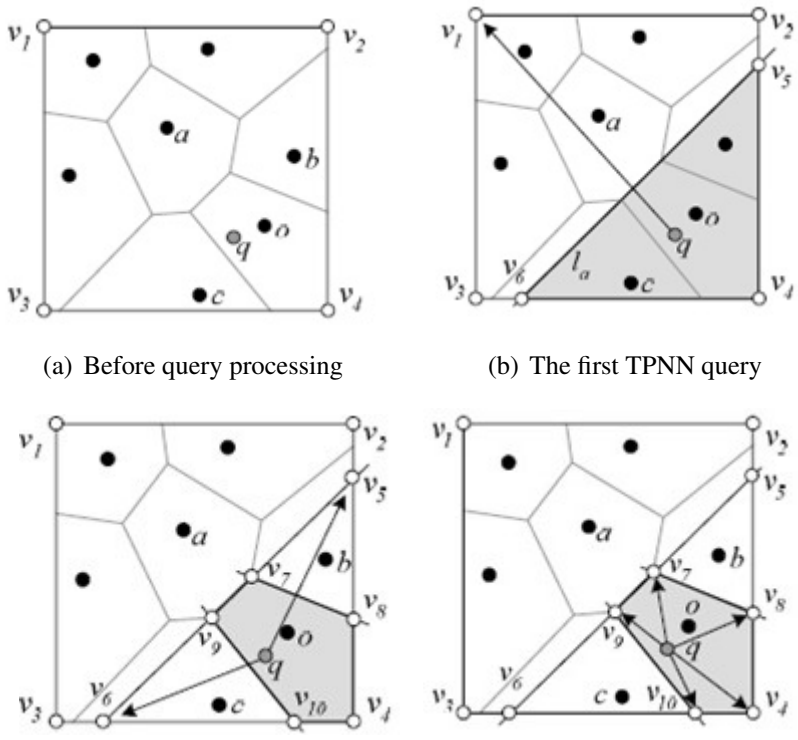
2.2.3 Experimental Result

The authors evaluate the number of disk accesses for different numbers of buffer pages. The experiment results show that, when a buffer is used, TP-kNN query significantly outperforms ordinary kNN query.

2.3 Retrieve-Influence-Set *k*NN (RIS- *k*NN) strategy

Zhang *et al.* [27] propose a query processing algorithm called the retrieve-influence-set kNN (RIS-kNN) to locally compute k Voronoi Diagram (kVD) cells using a spatial index. RIS-kNN uses the time parameterized kNN (TPkNN) query [22] to find each edge

of a kVD cell that surrounds a query point. Using these techniques fewer queries must be issued to the server, resulting in comparatively less computation and network overhead than regular spatial queries. The authors propose the following strategies to process a location-based nearest neighbour query. The server first executes a TP nearest neighbour query [20] to retrieve the nearest neighbour o of the query point q and then forms a validity region $V(q)$ around q. Figure 2.5 gives an example. The initial validity region is formed from the overall data universe which is defined by four vertices: v1, v2, v3, and v4 in Figure 2.5(a). One of the vertices is selected randomly for issuing a TPNN query with the trajectory from q to the chosen vertex. When the query finds a new influence object, the validity region is updated with the intersection of perpendicular bisector of new influence object. Figure 2.5(b) represents the updated validity region with vertices $v4$, $v5$, and $v6$. Next, another vertex is chosen and a new TPNN query is issued which retrieves a new influence object. The validity region is updated respectively (shown in Figure 2.5(c)).If



(a) Before query processing

(b) The first TPNN query

(c) Subsequent TPNN queries

(d) Confirmed vertices

Figure 2.5: Computation of validity region for nearest neighbour [27].

new influence objects cannot be found with a vertex, the vertex is not considered again and it is confirmed. The same process is repeated until all vertices are confirmed. Finally, when all vertices are confirmed, the influence object set and the query result are returned by the server to the client. Figure 2.6 represents the algorithm to retrieve the influence object set for kNN query [27]. In the RIS-*k*NN algorithm, a TP*k*NN query is issued instead of TPNN query, which returns a pair of influence objects and one of the k nearest neighbours. At the same time, the existence of the pair is checked. If the pair has not been discovered before, then the pair is added to the influence pair set. When all vertices are confirmed, all influence objects are returned and the algorithm terminates.

Algorithm Retrieve_Influence_Set_*k*NN (q, S_o)
/* q is the query point, S_o is the set of nearest neighbors of q */
1. $S_{inf\_p} = \emptyset$ // initialize the influence pair set
2. V = {universe boundary vertices} // initialize the vertex set
3. while (V contains non-confirmed vertex)
4. v = any non-confirmed vertex in V
5. $<o_{inf}, o_r>$ = TP*k*NN(q, v, S_o) // query from q pointing to v
6. if ($o_{inf} = \emptyset$ or $<o_{inf}, o_r> \in S_{inf\_p}$) confirm v
7. else // a new influence object o_{inf} is discovered
8. $S_{inf\_p} = S_{inf\_p} \cup \{<o_{inf}, o_r>\}$
9. update V
10. S_{inf} = all o_{inf} in S_{inf_p}
11. return S_{inf}
End Retrieve_Influence_Set_*k*NN

Figure 2.6: Algorithm for retrieving the influence set kNN query [27].

2.4 Incremental Search Grid

Park *et al.* [17] introduce a spatial index structure called the Incremental Search Grid (ISGrid) to efficiently process continuous *k*NN queries for static objects and moving queries. ISGrid uses a grid structure to access the leaf nodes directly. Therefore, unnecessary non-leaf nodes visiting can be avoided. In addition, this index structure uses minimum bounding rectangles (MBR) as leaf nodes to minimize the dead space that is covered by the grid. Each leaf node is linked to neighbouring leaf nodes. Figure 2.7 represents the configuration of

ISGrid which consists of 3 layers [17]:

1. The grid structure provides direct access to MBRs.
2. A set of leaf node MBRs that are connected to neighbouring nodes.
3. A set of objects for each MBR.

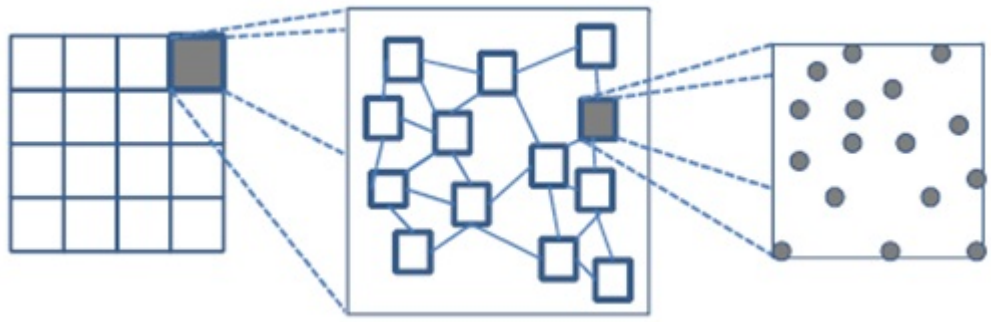


Figure 2.7: Configuration of ISGrid [17].

Therefore, ISGrid has no internal nodes. For kNN query processing using the R-tree index structure [2], visits to internal nodes are required in order to access the leaf nodes. The authors give an example of how ISGrid outperforms the R-tree in Figure 2.8. In their example, the query point Q needs to access the leaf nodes $M10$, $M16$, and $M22$. In the R-tree, Q would also need to access the parent nodes $M4$, $M6$, and $M8$. However, in ISGrid, $M10$ contains the pointers to $M16$ and $M22$, which are used to access these nodes. In order to avoid unnecessary visits of the neighbouring leaf nodes, a leaf node only contains pointers to leaf nodes, which are the nearest neighbours in each direction. The authors use the Voronoi regions of the neighbouring nodes adjacent the Voronoi region of the leaf node to determine the nearest neighbours of a leaf node.

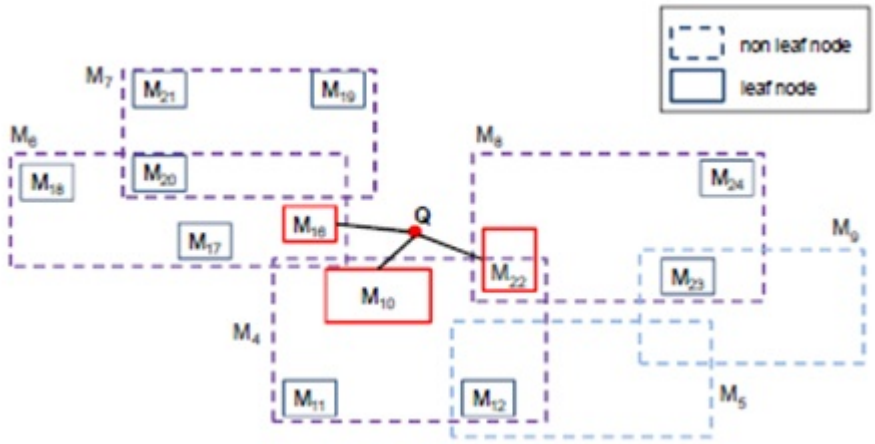


Figure 2.8: Structural problem of R-tree on processing kNN queries [17].

```
00   k-NN query(qid, k, q_point, G)
01   input – qid : identifier of query, q_point : query point,
02              k : number of required objects, G : ISGird
03   output – resultset : k-NN objects
04   {
05      Heap h;
06      LeafNode n;
07      n = FindNearestNeighborNode(G, p_point, h);
08      h.push(n);
09      Until {
10         n = h.pop();
11         resultset = resultset □ FindkNNs(n, q_point);
12         h.push(n.NNs);
13      } While (( NumOf(resultset) < k) or
                (dist(point, k-dist >= dist(point, n));
14      Return resultset;
15   }
```

Figure 2.9: kNN query processing algorithm [17].

2.4.1 *k* Nearest Neighbour Query Processing Algorithm

Park *et al.* [17] propose the following *k*NN query processing algorithm, which is illustrated in Figure 2.9. The algorithm first locates the cell that contains the query point, before locating the nearest overlapping leaf node for the query point. Next, the algorithm finds the next nearest leaf nodes using the information of the neighbour nodes. If the number of leaf

node is less than k, the leaf node is added to the query result. The given steps are repeated until the k nearest leaf nodes to the query point are found.

2.4.2 Performance Evaluation

Park *et al.* [17] evaluate the number of disk I/Os for different values of k. The experiment result shows that ISGrid performs better than the R-tree [2], ST2B-tree [3], and ISR-tree [16] by 25%, 35%, and 10% respectively.

2.5 V*-Diagram and V*-*k*NN

Nutanong *et al.* [13] introduce an integrated safe-region (ISR) technique called the V*-Diagram, and its associated algorithm, V*-*k*NN, to efficiently process moving k nearest neighbour queries (M*k*NN). To compute the safe-region, the V*-Diagram uses not only the knowledge of the data objects, but also the knowledge of the query location and the current search space. According to the authors, both the computation and the data retrieval of the V*-Diagram are more economical than other existing methods.

The authors define the ISR that contains two types of regions. The first is a Fixed Rank Region, which is an order-sensitive Voronoi diagram cell, where the ranking of all objects based on their distances is fixed.

$$F(L) \cap (\bigcap_{i=1}^{k} S(q_b, z, p_i))$$

Here, $F(L)$ is the fixed rank region of L. L is a list of $(k+x)$ objects sorted in ascending order by their distance to the query point. k is the number of requested nearest neighbours and x is the additional objects being fetched. q_b represents the current location of the moving query point. $S(q_b, z, p_i)$ is the safe region with regard to data object p_i, and z is the $(k+x)$th NN to q_b.

$F(L)$ is represented by a list B of the $(k+x-1)$ rank-adjacent bisectors, $B_{(pipi+1)}$, in the order corresponding to L, (for $i = 1, 2, \ldots, k+x-1$). To determine the correct half plane of

each bisector in B, q_b is used as a reference point. Point q_b also indicates where the last call to BF-*k*NN was made to retrieve the $(k+x)$NNs to q_b.

The authors define the safe region for a data object p as

$$S(q_b, z, p) = \{q' : Dist(q', p) + Dist(q_b, q') \leq Dist(q_b, z)\}$$

where $q\prime$ is the next position of the moving query point after q_b.

The authors also define the concepts of known and reliable regions. A known region $W(q_b, z)$ is a sphere centered at q_b with radius $Dist(q_b, z)$. Point object p is one of the $(k+x-1)$NNs of q_b. The sphere with center $q\prime$ and radius $Dist(q\prime, x)$ is a reliable region of $q\prime$. If an object is in reliable region, it is known as reliable object. As long as p is in the $sphere(q\prime, Dist(q\prime, x))$, it is a reliable object and one of the $(k+x)$NNs of $q\prime$. Figure 2.10 depicts the known, reliable, and safe regions. The authors give an example of the V*-diagram to compute an ISR (shown in Figure 2.11) for $k = 2$ and $x = 2$.

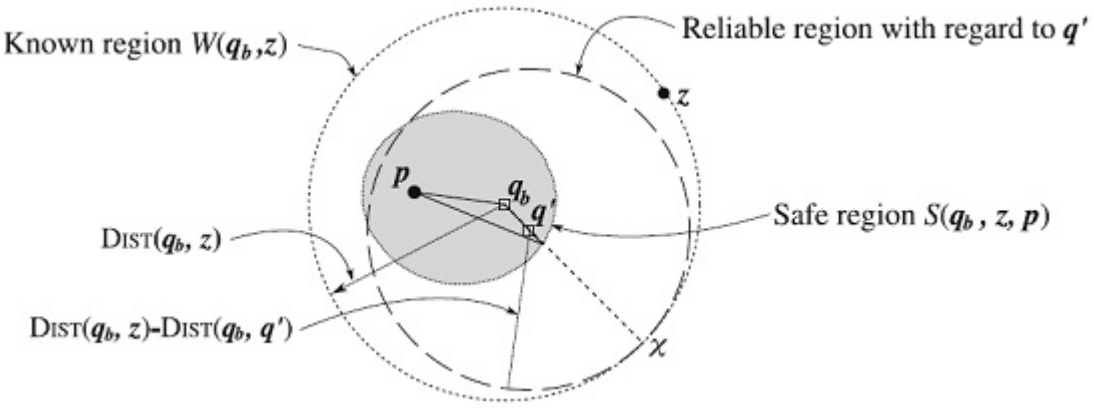


Figure 2.10: The known, reliable, and safe region [13].

Initially, the BF-kNN algorithm [6] is called to retrieve 4 (i.e. $2+2$) nearest neighbours to the query point, q_1. The retrieved objects are a, c, b, and f, with f being the farthest object from q_1. Based on the rank adjacent bisectors in the order corresponding to (a, c, b, f), the fixed rank region, $F < a, c, b, f >$ is computed. Then, the safe region $S(q_1, f, c)$ with

regard to c is computed. Finally, the integrated safe region is computed by

$$F < a, c, b, f > \cap S(q_1, f, c)$$

which is the grey region shown in Figure 2.11. As long as q_1 remains in this ISR, no objects outside of the known region, $W(q_1, f)$ are nearer to the objects a and c, and the ranking of $F < a, c, b, f >$ remains the same. Figure 2.11 shows the algorithm Compute-V* to compute the initial ISR [13]. Figure 2.13 illustrates the V*-kNN algorithm for processing MkNN queries based on the V*-Diagram.

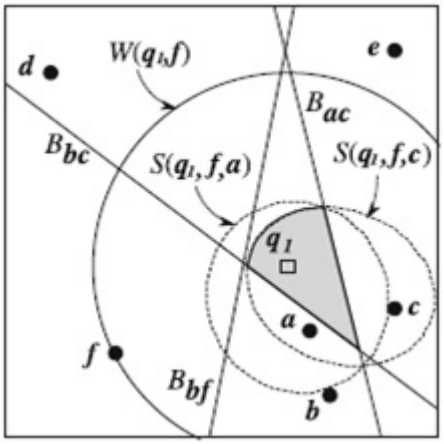


Figure 2.11: Integrated safe region example ($k = 2$, $x = 2$) [13].

According to the algorithm, the initial ISR is computed by the algorithm Compute-V*, where the starting location of the query point is q_0 (line 2). Then, the algorithm maintains the current ISR with the movement of the query point. When the query point exits the current ISR one of two events is triggered: *RankUpdate* and *ReliabilityUpdate*. The event generation is associated with a timestamp and the corresponding query position.

A *RankUpdate* event is triggered when the query point q crosses a rank-adjacent bisector or exits the current fixed-rank region, $F(L)$. For this event, the ranks of the two objects corresponding to the bisector are swapped and the bisector list B is updated accordingly. If the event affects the rank of any of the kNNs, then the new kNNs are reported. In addition,

if the rank update changes p_k, S_k also needs to be updated, where p_k is the k^{th} object in L and S_k is the safe region with regard to p_k.

Algorithm 2: Compute-V*(q_b, k, x)

```
1 L ← BF-kNN(q_b, k + x)
2 z ← L.Item(k + x)
3 p_k ← L.Item(k)
4 S_k ← S(q_b, z, p_k)
5 B ← CreateBisectorList(L)
6 ISR ← ConstructISR(S_k, B, q_b)
7 return (L, z, S_k, B, ISR)
```

Figure 2.12: The algorithm Compute-V* [13].

Algorithm 1: V*-kNN(q_0, k, x)

```
1  q_b ← q_0
2  (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
3  ReportResult(L.Head(k))
4  while (Event ← GetEvent()) do
5      q ← Event.Position
6      switch Event.Type do
7          case RankUpdate
8              Bisector ← Event.Bisector
9              L.OrderSwap(Bisector.Index)
10             B.Update(L,Bisector.Index)
11             if Bisector.Index ≤ k then
12                 ReportResult(L.Head(k))
13             if Bisector.Index ∈ [k − 1, k] then
14                 p_k ← L.Item(k)
15                 S_k ← S(q_b, z, p_k)
16             ISR ← ConstructISR(S_k, B, q)
17         case ReliabilityUpdate
18             q_b ← q
19             (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
```

Figure 2.13: The V*-kNN algorithm [13].

A *ReliabilityUpdate* event is triggered when the query point is leaving safe region S_k. This means that the number of reliable objects is about to become less than k. Compute-V* is called again using the current position q, to obtain x new auxiliary objects and to

make sure all $(k+x)$ objects are reliable again. The new ISR is constructed accordingly. However, neither the current k nearest neighbours nor their ordering changes.

2.5.1 Insertion and Deletion of Objects

Nutanong *et al.* [13] also propose DatasetUpdate to insert objects into, or delete objects from the dataset for V*-*k*NN. Figure 2.14 depicts the algorithm. In this algorithm, q is the position of the query point and p is the object to be inserted or deleted. Initially, the algorithm checks whether to see p is in the known region, $W(q_b, z)$. If it is not, the update can be ignored because it cannot affect the current ISR. Otherwise, an insertion or deletion of p into or from L is performed and the list B is updated accordingly. The insertion of p needs q for computing distances between q and the objects of L, in order to find the correct insertion slot in L. The deletion of p from L requires a simple lookup operation. After the bisector update, the ISR and L could be in one of the following three cases:

i. For deletion, the number of objects in L becomes smaller than k. In this case, q_b is set to q and the algorithm Compute-V* is called to retrieve more objects and compute the new *ISR* accordingly.

ii. The number of objects in L is still greater than k but the update affects the kNN set. Updates to p_k and S_k are required. If q is not inside the new S_k, Compute-V* is called. Otherwise, the *ISR* is updated to reflect the changes in B and S_k. Since the kNNs have changed, the new result is reported to the user.

iii. The update has no effect on kNN set, but affects one of the auxiliary objects. Only the *ISR* is updated to reflect the change in B.

2.5.2 M*k*NN with Dynamically Changing k Values

V*-*k*NN can handle changing k values because of its incremental nature. Figure 2.15 represents the algorithm KUpdate to shows how V*-*k*NN handles dynamically changing k values. The algorithm takes two inputs: q as the current location of the query point and the

new k value. At first, it checks that the new k is greater than the length of L. If it is yes, q_b is set to q and Compute-V* is called. Otherwise, p_k and S_k are updated for the new k. If q is not inside the new S_k , q_b is set to q and Compute-V* is called. Otherwise, only the ISR has to be updated to incorporate the new S_k. Finally, the new kNNs are reported.

Algorithm 3: DatasetUpdate(q,p,*Operation*)

```
1  if p ∈ W(q_b, z) then
2      if Operation = Insertion then
3          L ← Insert(L,p,q)
4      else
5          L ← Delete(L,p)
6      B.Update(L)
7      if k > L.Length() then
8          q_b ← q
9          (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
10         ReportResult(L.Head(k))
11     else if DIST(q, p) ≤ DIST(q, p_k) then
12         p_k ← L.Item(k)
13         S_k ← S(q_b, z, p_k)
14         if q ∉ S_k then
15             q_b ← q
16             (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
17         else
18             ISR ← ConstructISR(S_k,B,q)
19         ReportResult(L.Head(k))
20     else
21         ISR ← ConstructISR(S_k,B,q)
```

Figure 2.14: The algorithm DatasetUpdate [13].

Algorithm 4: KUpdate(q,k)

```
1  if k > L.Length() then
2      q_b ← q
3      (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
4  else
5      p_k ← L.Item(k)
6      S_k ← S(q, z, p_k)
7      if q ∉ S_k then
8          q_b ← q
9          (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
10     else
11         ISR ← ConstructISR(S_k,B,q)
12 ReportResult (L.Head(k))
```

Figure 2.15: The algorithm KUpdate [13].

2.5.3 Experimental Result

The V*-*k*NN algorithm is evaluated and compared with the RIS-*k*NN [27]. The authors vary the following parameters in their experiments: (i) the number of auxiliary objects x, (ii) the cache size c, (iii) the trajectory length l, (iv) the cardinality n of the dataset, (v) the value of k, (vi) the step size s, and (vii) the probability δ that the k value is altered by 1. The authors used both synthetic and real datasets in their experiments. The authors evaluate the impact of x on the response time and the number of page accesses. Results show that: (1) the response time starts to increase when x has approximately 15 objects, and (2) the page-access cost increases as x increases. The authors also rate an increase in CPU for increasing t. Next, the impact of c on the performances of V*-*k*NN and RIS-*k*NN is evaluated. The experimental results show that V*-*k*NN outperforms the RIS-*k*NN in terms of both the total response time and the number of page accesses. For both methods, the page access cost decreases and the total response time remains stable as the buffer size increases. In addition, the authors experimented with the effects of (i) trajectory length (l), (ii) the number of objects in the dataset (n), (iii) the value of k, (iv) the step size (s), and (v) the probability that the k value is altered, on the performances of V*-*k*NN and RIS-*k*NN. The experimental results show that V*-*k*NN consistently outperforms RIS-*k*NN in terms of the total response time, the I/O cost and the communication cost.

2.6 The mqr-tree

Moreau and Osborn [10] propose a two-dimensional spatial access method the mqr-tree which allows spatial objects to be organized in a two-dimensional node based on their spatial relationships. The mqr-tree utilizes a node organization, set of spatial relationship rules and insertion strategy in order to gain significant improvements in overlap and over-coverage. In addition, zero overlap is achieved when the mqr-tree is used to index point data.

2.6.1 Node Organization and Validity

In the mqr-tree, a node contains 5 locations - northeast (NE), northwest (NW), southwest (SW), southeast (SE) and centre (EQ). Each location contains a pointer which points to either an object or a subtree. The node layout is shown in Figure 2.16. A node must have at least two locations and not more than five locations. It is possible to have a node containing pointers to both objects and subtrees. The origin of the node is the centroid of the node MBR. The node MBR contains all objects, and any subtrees in the node. As objects are added to and removed from the node, the centre of the node will change.

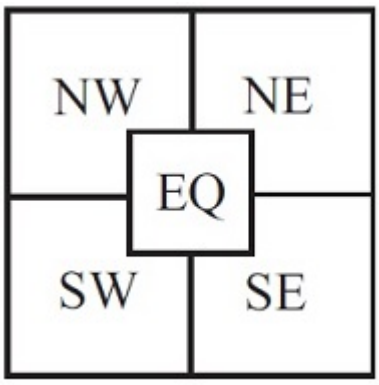


Figure 2.16: Node layout [10].

In every node of the mqr-tree, the relative placement of both objects and subregions are determined by using the centroids of their MBRs. The objects that are referenced from the centre location have the same centroid as the centroid of the node MBR for the node. The placement of all other objects and subregions that are referenced from the other locations (NW, SE, SW, NE) are determined with respect to the centroid of the node MBR. Figure 2.17 depicts the spatial relationships, where A refers to the centroid of a new object, and B refers to the centroid of the node MBR.

2.6.2 Insertion Strategy

First, the root node MBR is adjusted to include a new object and the appropriate location is identified relative to the centroid of the node MBR for inserting the reference of the new object. If the location is empty, the object reference is inserted. Otherwise, if the location contains an object reference, a new leaf node is created for the insertion of the new and existing object reference. Otherwise, a proper location is found in the subtree. Insertion or deletion of an object results in one of the following three events: (i) the centroid of the node remains the same, (ii) the centroid moves and the area of the node MBR expands (shown in Figure 2.18(a)), or (iii) the centroid of the node moves and the area of the node MBR shrinks (shown in Figure 2.18(b)). If the centroid of the node moves, some objects might not be in their proper node location. Therefore, these objects need to be reinserted into the proper relative location.

$A_x = B_x$	$A_x > B_x$	$A_y = B_y$	$A_y > B_y$	Placement
0	0	0	0	SW
0	0	1	0	SW
0	0	0	1	NW
1	0	0	1	NW
0	1	0	0	SE
1	0	0	0	SE
0	1	0	1	NE
0	1	1	0	NE
1	0	1	0	EQ

Figure 2.17: Relative orientation of A with respect to B [10].

2.6.3 Experiment Result

The experiments compare the mqr-tree insertion strategy with the R-tree [10]. The results shows the mqr-tree achieves significant improvements in overlap and overcoverage. In addition, a comparison of region searching shows that the mqr-tree uses a lower number of disk accesses in many cases.

2.7 The Trie Data Structure

A Trie is a special data structure used to store groups of strings that can be visualized like a graph [24]. It consists of nodes and edges. Strings are stored in a top to bottom manner on the basis of their prefix. If two string have a common prefix, they will have the same ancestor in the tree. All prefixes of length 1 are stored at level 1, all prefixes of length 2 are sorted at level 2 and so on. A Trie consists of a special node called the root node. This node has no incoming edges. The Trie is used to do prefix based search.

In Figure 2.19, every character is a Trie node. For example, the root node has children a, b and t. Similarly, ‘a’ at the next level is having only one child ‘n’.

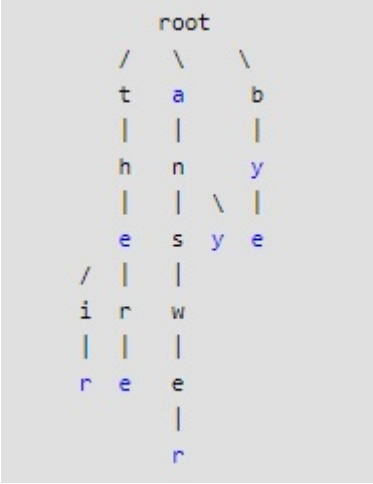
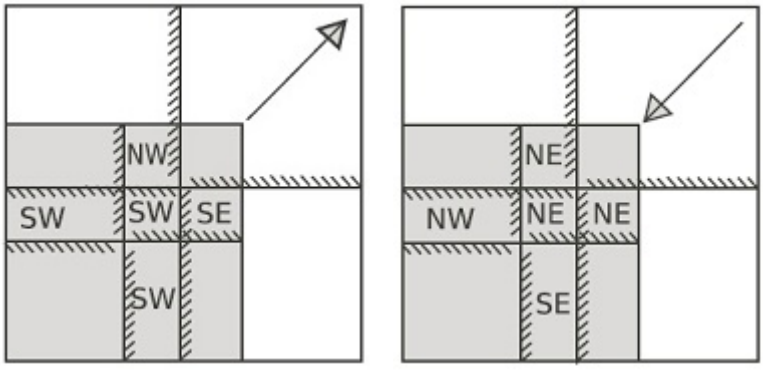


Figure 2.19: Trie data structure [24].



(a) NorthEast node MBR expansion (b) SouthWest node MBR contraction

Figure 2.18: Insertion Strategy [10].

2.7.1 Prefix

The prefix of a string is n letters, where $n <| String |$ that makes the beginning of a string [24]. For example, the word 'abacaba' has the following prefixes: a, ab, aba, abac, abaca, abacab.

2.7.2 Insertion and Search in Trie

The insertion of any string into a Trie starts from the root node. All prefixes of length one are direct children of the root node. In addition, all prefixes of length 2 become children of the nodes existing at level one. The pseudo code for insertion of a string into a Trie is shown in Figure 2.20.

While searching, start from the root node. We keep on reading characters of given key and according to indices of characters, travel from root node to leaf node. The search can terminate due to the end of string. If the value of last node is non-zero then the key exists in Trie. The key refers to the word that exists in Trie. The search can also terminate due to the lack of key nodes in trie without examining all of the characters. The pseudo code to check whether a single word exists in a dictionary of words or not is shown in Figure 2.21.

```
void insert(String s)
{
    for(every char in string s)
    {
        if(child node belonging to current char is null)
        {
            child node=new Node();
        }
        current_node=child_node;
    }
}
```

Figure 2.20: Insertion of a string into Trie [24].

```
boolean check(String s)
{
    for(every char in String s)
    {
        if(child node is null)
        {
            return false;
        }
    }
    return true;
}
```

Figure 2.21: Searching a single word in Trie [24].

2.8 Limitations of Related Works

To solve the continuous kNN query, Song and Roussopoulos [20] extend the search for mNN objects $(m > k)$. According to their strategy, user can be assured that kNN is contained by the mNN result if the distance it moved from query point q is less than $\{Dist(m,q) - Dist(k,q)\}/2$, i.e. the half distance difference between k-th NN and m-th NN with respect to new query point q'. This attempts to keep the result current while the query point moves around and a new query call is not necessary to the server. However, this is ineffective to reduce query reevaluation. For example in Figure 2.22 based on q, a 2NN is executed as a 4NN and δ is the distance between the second NN and the fourth NN. A new query point q' is located more than a distance of $\delta/2$ away from q and hence needs reevaluation. Although the 4NN result still covers 2NN objects to q', this implies that their strategy cannot effectively reduce the number of queries. Choosing a proper value of m is a significant problem of this strategy. A high value of m will increase the network overhead and the storage requirements at the client, while a low value does not reduce the number of queries.

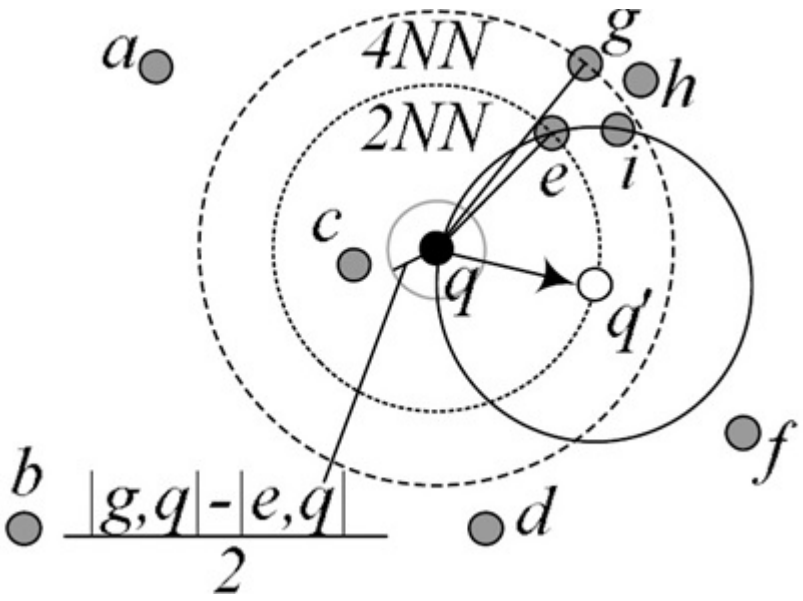


Figure 2.22: mNN query.

Tao et al. [22] utilizes the R-tree [2] to speed up the repeated searching needed for their continuous kNN strategy. Their strategy process a continuous NN query on R-tree according to the movement of a client. The method estimates the future location of the client and provides the continuous query result by pre-computing the result on the estimated path where the velocity of the client is known and constant during the lifespan of the query. This assumption may not work for many applications where the query velocities are continuously updated as the users change their speed or direction of movement.

Zhang *et al.* [27] derive the retrieve-influence-set *k*NN (RIS-*k*NN) algorithm in order to reduce the number of repeated searches and the cost of precomputation. However, it is still expensive because RIS-*k*NN performs multiple TP*k*NN queries [22] and each TP*k*NN query involves a costly tree traversal. In addition, RIS-*k*NN only works for a static number of nearest neighbours. Changing *k* to a larger value requires a recalculation of the kVD cell. If an update affects the *k*NN result or the influence set, the current kVD cell must be recalculated.

Our Area Code Tree provides the ability to perform repeated searching that is efficient, but at the cost of accuracy. The Area Code tree stores the POI area codes instead of co-

ordinates of Minimum Bounding Rectangles. This provides a simple numeric comparison for quickly identifying a nearest neighbour to the query point. The Area Code Tree is a Trie type structure. In the Trie, insert and search algorithms both cost O(K), where k is length of key. The key refers to the string that is inserted or searched in trie. The memory requirement of the trie is O(k*N) where N is number of keys in trie. From our experiment results we find that the average search time for the Area Code Tree is very low and constant-time regardless of the number of POIs in the index. The Area Code Tree needs to be constructed only once for searching many times in static data sets. In addition, for an insertion the cost is also very low and a complete rebuild of the tree is not required. For denser data sets, the Area Code Tree achieves higher accuracy up to 60% for locating the approximate nearest neighbour. This makes the Area code Tree an excellent candidate for continuous approximate nearest neighbour search for location based services. We present the Area Code Tree in the following chapter.

Chapter 3

Area Code Tree

In this chapter, we introduce the Area Code Tree[2] for continuous approximate nearest neighbour search for location based services. The Area Code Tree is a Trie-type structure that stores points of interest (POIs) that are represented in area code format. An area code is a sequence of digits that represent the relative location of a point, such as a POI, in space. Therefore, indexing area code data of a POI instead of actual spatial values such as longitude and latitude, reduces the nearest neighbour search to a sequence of simple numeric comparison. In addition, we also present the algorithms for mapping the area code of a POI, inserting and building an Area Code Tree, and approximate nearest neighbour search. The corresponding pseudocode for each algorithm can be found in Appendix A.

3.1 Mapping Area Code of a Point of Interest (POI)

A POI is represented as (P_x, P_y). A Minimum Bounding Rectangle (MBR) defines a region of space that is occupied by POIs. It is represented with different coordinates: 1) the lower left coordinate $(MINP_x, MINP_y)$, 2) the upper right coordinate $(MAXP_x, MAXP_y)$, and 3) its center (CM_x, CM_y). The area code is obtained by recursively partitioning the space into quadrants. At each level of partitioning, a POI obtains a digit depending on which quadrant it resides in. The quadrant are numbered as follows: SW(1), SE(2), NW(3), and NE(4). The space partitioning continues until either: the POI (P_x, P_y) and the center of the region (CM_x, CM_y) are equal, or the lower left $(MINP_x, MINP_y)$ and upper right

[2]The paper [19] based on this work won the Best Paper Award at SEDE, 2016.

$(MAXP_y, MAXP_y)$ coordinates of the region are the same.

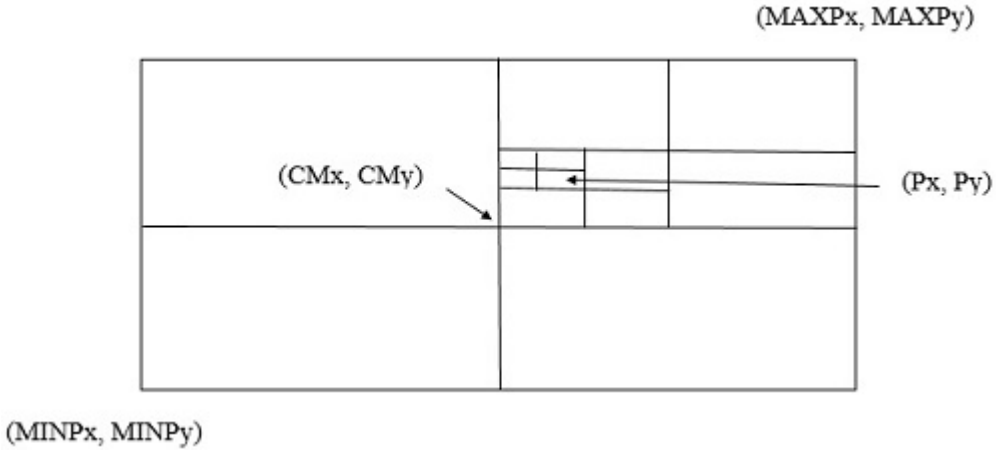


Figure 3.1: Area code mapping of a POI.

- Case 1: $P_x < CM_x$ and $P_y < CM_y$. The POI is located in SW quadrant and 1 is assigned to the POI area code. The coordinates of the region are updated as follows: The lower left coordinate remains $(MINP_x, MINP_y)$, while the upper right coordinate is updated to (CM_x, CM_y), the center point of the previous region (i.e. $MAXP_x = CM_x$ and $MAXP_y = CM_y$.)

- Case 2: $P_x > CM_x$ and $P_y < CM_y$. The POI is located in SE quadrant and 2 is assigned to the POI area code. The coordinates of the region are updated as follows: the new lower left corner of the region is $(CM_x, MINP_y)$ (i.e. $MINP_x = CM_x$ and $MINP_y = MINP_y$), while the new upper right corner of the region is $(MAXP_x, CM_y)$ (i.e. $MAXP_x = MAXP_x$ and $MAXP_y = CM_y$).

- Case 3: $P_x < CM_x$ and $P_y > CM_y$. The POI is located in NW quadrant and 3 is assigned to the POI area code. The coordinates of the region are updated as follows: the new lower left coordinate is $(MINP_x, CM_y)$ (i.e. $MINP_x = MINP_x$ and $MINP_y = CM_y$). The new upper right coordinate of the region is $(CM_x, MAXP_y)$ (i.e. $MAXP_x = CM_x$ and $MAXP_y = MAXP_y$).

- Case 4: $P_x > CM_x$ and $P_y > CM_y$. The POI is located in NE quadrant and 4 is assigned

to the POI area code. The coordinates of the region are updated as follows: the lower left coordinate of the region is (CM_x, CM_y) (i.e. $MINP_x = CM_x$ and $MINP_y = CM_y$), while the upper right coordinate of the updated region remains $(MAXP_x, MAXP_y)$.

Figure 3.1 depicts an example of space partitioning and mapping. Beginning with the top-most partition, the POI resides in NE quadrant. It is followed by the SW quadrant at the next level, then the NW quadrant, and finally the SE quadrant. Therefore, the area code of the POI in the diagram is 4132.

3.2 Area Code Tree and Node Structure

The Area Code Tree is a trie-type structure. It consists of nodes and edges. Each node can consist of a maximum of 4 child nodes. Edges connect each parent node to its child nodes. Each node represents either a prefix or an area code of a POI. The root node does not have any incoming edges. The nodes that are direct children of the root node represent prefixes of length 1, while the nodes that are 2 edges of distance from the root node represent prefixes of length 2, and so on. In other words, a node that is k edges of distance of the root has an associated prefix of length k. The digits of a POI area code are stored in a top to bottom manner on the basis of their prefix in the Area Code Tree. For example: a POI area code 4132 has the prefixes 4, 41, and 413.

3.3 Area Code Tree Construction

Once the area code of a POI is determined, it is inserted into the Area Code Tree in the following manner. Beginning at the root node, we take the most significant digit of the POI area code and check for the existence of a prefix of a child node for that digit. If no such prefix exists, then a new child node is created for that digit and the remainder of area code is placed in the node. If a prefix exists for that digit, then the search for an insertion path continues in the corresponding child node. The search for an insertion location continues with the next digit in the area code of the POI. After a new leaf node is created, there may

exist subsequent area code digits that two or more POIs share in common. In this case, further nodes are created until each POI has its own leaf node.

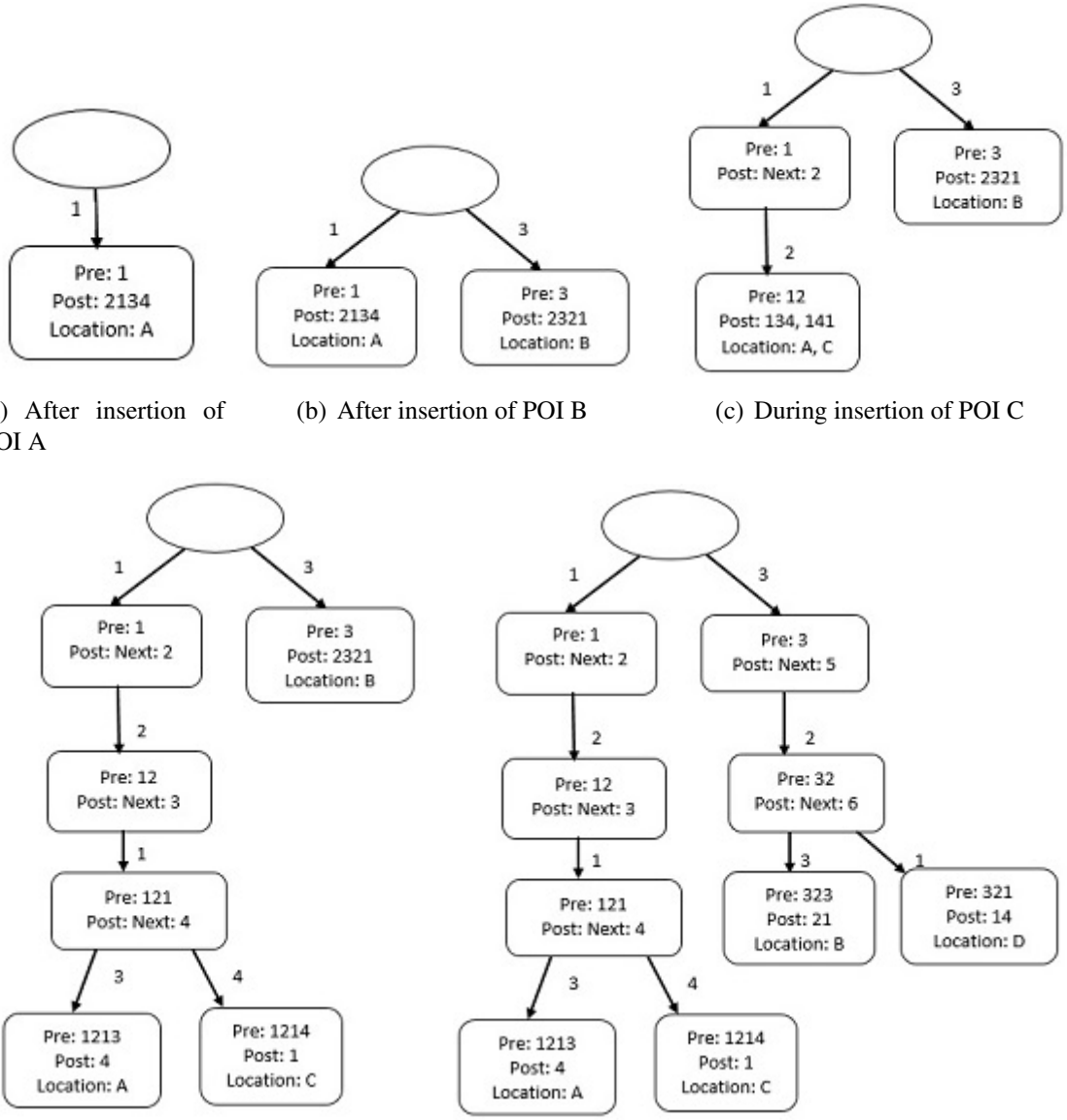
Table 3.1 gives some example area codes that are inserted into the area code tree shown in Figures 3.2 and 3.3. Beginning with POI A, the most significant digit of POI A's area code is 1. The root node has no child node with prefix 1. A new child node is created to store POI A's area code. Next, we insert POI B. The most significant digit of that area code is 3. No child node of the root node has a prefix 3. Therefore, a new child node is created to contain POI B. Figures 3.2(a) and 3.2(b) depicts the Area Code Tree. There are two child nodes of the root: one is POI A (prefix 1, Figure 3.2(b)) and the other is POI B (prefix 3, Figure 3.2(b)).

Table 3.1: Sample Area Codes

POI	Area Code
A	12134
B	32321
C	12141
D	32114
E	21324

Next we insert POI C. This is depicted across the two Figures 3.2(c) and 3.2(d). The most significant digit of the POI C's area code is 1. There is an existing child node with prefix 1 of the root node. Therefore, the search proceeds in that child node. Since the child node is a leaf node, a new leaf node is created that corresponds to the second digit of POI C area code, which is 2. This is shown in Figure 3.2(c). However, both POIs that are now referenced by the new leaf node A and C have the same third area code digit, which is 1. Therefore, another node that corresponds to the common third digit is created. From here, two leaf nodes are created using the fourth area code digits for POIs A and C respectively.

The resulting tree is depicted in Figure 3.2(d). Similarly, POIs D (Figure 3.2(e)) and E (Figure 3.3(a)) are inserted into the Area Code Tree. The final tree is depicted in Figure 3.3(a).



(a) After insertion of POI A

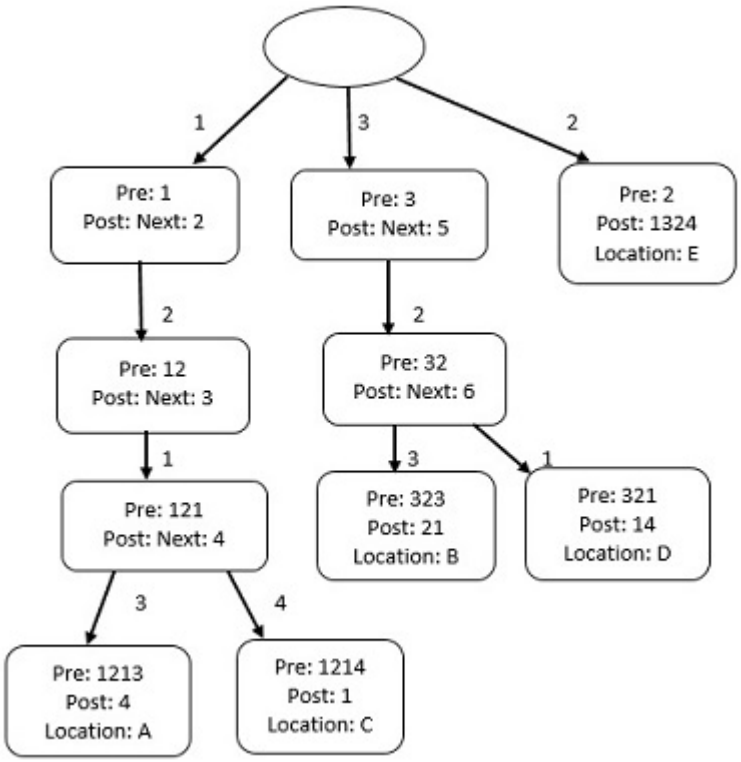
(b) After insertion of POI B

(c) During insertion of POI C

(d) After insertion of POI C

(e) After insertion of POI D

Figure 3.2: Index construction via insertion-Part 1.



(a) After insertion of POI E

Figure 3.3: Part 2 of index construction via insertion.

3.4 Approximate Nearest Neighbour Search

First the area code of the query point needs to be calculated according to the strategy of mapping area code of a POI. Because the POIs in the Area Code Tree are inserted using area code digits, the task of performing a nearest neighbour search is reduced to numeric comparisons. Begin with the most significant digit of the query area code, a path is followed down the area code tree while digits in the query match the digits in the tree nodes. If a match does not exist at a particular level, the closest match is taken. Although this may be possible at any level of the Area Code Tree. However it is very unlikely to occur in an index that is managing many POIs and more likely to happen at the leaf levels. The source code is shown in Appendix A.3. For example, we have query area code of 12134 and an Area Code Tree in Figure 3.3(a), the search proceeds along the path identified by 1, then

2, then 1, then 3, and finally reached the node for location A. if a match does not exist at a particular level, the closest match is taken. For example, for a query area code 32113 the closest match is location D. Since that path matches the first four digits of query area code and the closest last digit is 4.

We introduce the Area Code Tree for continuous approximate nearest neighbour search for location based services in this chapter. We also present its mapping, insertion, and approximate nearest neighbour search algorithms. The strategy and results of performance evaluation of the Area Code Tree will be presented in the next chapter.

Chapter 4

Experimental Evaluation

The methodology and results of our performance evaluation of the Area Code Tree are presented in this chapter. We compare the Area Code Tree's search performance with the Brute Force method. We also evaluate the Area Code Tree for tree construction time and the accuracy for locating an approximate nearest neighbour. By inserting POI area code one at a time, we construct the Area Code Tree. We measure the accuracy by calculating the percentage of times that an exact nearest neighbour is found by the Area Code Tree.

4.1 The Brute Force Search

The Brute Force Search is a very general problem solving technique that consists of finding all possible candidates for the solution and determines whether each candidate satisfies the problem's statement [23]. Here, we compute all the distances between the user location and the POIs. Next we sort the computed distances and select the POI with the smallest distance and compute the nearest neighbour search time. These steps are repeated for all query points (i.e. user locations).

4.2 Experimental Setup

For our experiments, we use the XAMPP cross-platform web server which consists of an Apache HTTP Server, a MySQL database, and the PHP programing language. We use Apache as the local web server. We store POI area codes using the MySQL database. We use the PHP programming language to map the POIs' area codes, calculate the construction

time of the Area Code Tree, and calculate the search time of an exact or approximate nearest neighbour query.

4.3 Data Sets

Twenty-one data sets of different POIs from across New Zealand are used for our evaluation. Ten of those data sets of random POIs are taken from different locations of the North Island of New Zealand. The smallest contains 1000 points and the largest contains 10000 points. One of these data sets contains 1000 points is shown in Appendix B.1. Another ten data sets of random POIs are drawn from different locations of the Waikato Region of New Zealand (part of the North Island). The Waikato data sets are denser than the North Island data sets, which allows us to evaluate the Area Code Tree in denser data sets. For our evaluation, the remaining one data set is a trajectory which contains 10 user's locations shown in Appendix B.2. This trajectory contains the sequence of coordinates (i.e. locations) along a user's route in the Waikato region. These locations serve as the nearest neighbour queries. The first twenty data sets were generated by combining some actual locations in New Zealand (North Island and Waikato) with other randomly generated locations. The user location trajectory was generated using tracking S/W[3].

4.4 Experiments

Altogether, 200 nearest neighbour evaluations are performed in the following manner. First an Area Code Tree is created for each of the 20 POI sets above. For each tree, we perform ten nearest neighbour searches using the user location set. We also perform the same searches using the Brute Force method.

We measure the following performance criteria:

1. For each Area Code Tree construction, we measure the overall tree construction time and the average insertion time for each POI area code in Area Code Tree.

[3]These locations were recorded by Dr. Annika Hinze in the Waikato region of New Zealand.

2. For the search comparison, we measure the average nearest neighbour search time in milliseconds.

3. The accuracy of the Area Code Tree is measured by calculating the percentage of POIs found by the Area Code Tree that matched those found by the Brute Force search.

4.4.1 Results

The results of our comparison using the Waikato and North Island data sets are represented by Figures 4.1 and 4.2 respectively. For both figures, the x-axis represents thousands of POIs (i.e. 1 represents 1000 POIs, up to 10 for 10,000 POIs), while the y-axis represents the average search time in seconds. For both groups of POI sets, the search time for the Area Code Tree is significantly less than the Brute Force method. Regardless of the density of the dataset and the number of POI area codes in the index, the average search time for the Area Code Tree is less than 10ms. On the other hand, the average search time for Brute Force method increases steadily from less than 10ms to almost 50ms for searching 10,000 area codes.

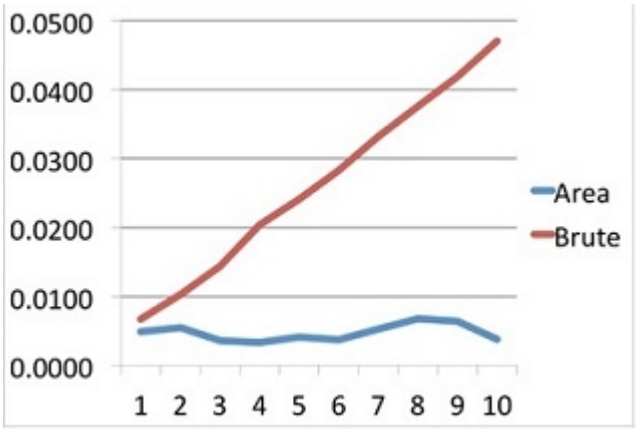


Figure 4.1: Waikato POIs.

The results of the evaluation for accuracy are represented by Figure 4.3. The Area Code Tree can achieve between 40% and 60% accuracy in locating the nearest neighbour for

dense POI sets (i.e. Waikato). For the less dense point sets (i.e. North Island), the accuracy was lower between 0% and 40%.

Finally, Table 4.1 shows the construction times of the Area Code Tree for different POI sets. Both the overall tree construction time (O/A Time) and the average insertion time of a POI (Avg. Time) are recorded. The table shows that as the data set size increases, the construction time of an Area Code Tree increases significantly as well. For the North Island datasets, the construction time is just under a minute for 1000 POIs, and for 10,000 POIs, it is just over 8 minutes. However, for the Waikato Region datasets, the construction times range from less than a minute to over 10 minutes.

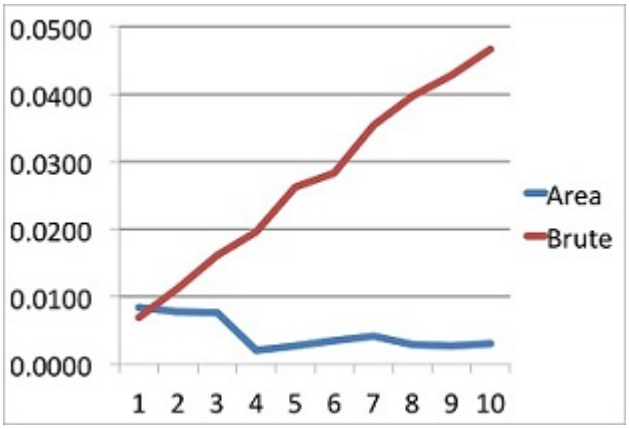


Figure 4.2: North Island POIs.

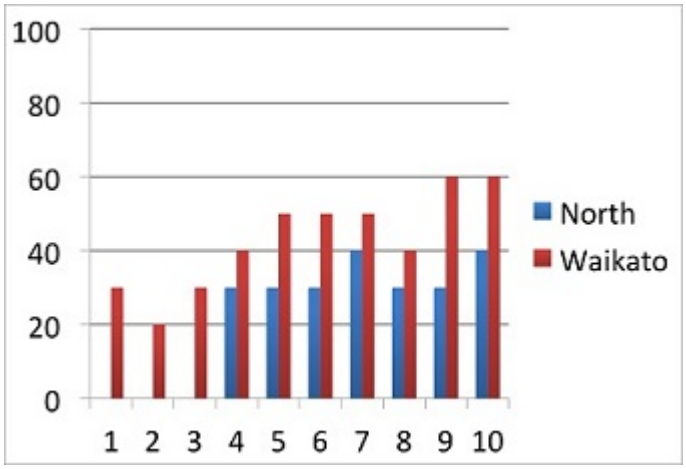


Figure 4.3: Accuracy of Area Code Tree.

Table 4.1: Sample Area Codes

Data Sets	POIs	Tree Construction (Sec.)	Avg. insertion (Sec.)
North Island	1000	46.16	0.046
	2000	79.60	0.040
	3000	106.61	0.036
	4000	128.21	0.032
	5000	166.32	0.033
	6000	219.94	0.037
	7000	281.70	0.040
	8000	334.07	0.042
	9000	437.50	0.049
	10000	490.96	0.049
Waikato	1000	18.63	0.019
	2000	39.71	0.020
	3000	91.74	0.030
	4000	159.69	0.040
	5000	227.17	0.045
	6000	303.42	0.051
	7000	409.23	0.058
	8000	481.41	0.060
	9000	586.86	0.065
	10000	686.63	0.069

4.5 Discussion

As a result, we conclude that the Area Code Tree provides higher accuracy in nearest neighbour searching as the data set size increases in size. Although, it appears that the overall tree construction times are high, there need two things to be noted. First of all, the Area Code Tree needs to be constructed only once for searching many times in static data sets. Secondly, for an insertion, it takes under 50ms on average for less denser data sets and a complete rebuild of the tree is not required.

Chapter 5

Conclusion

In this thesis, we propose a spatial data structure the Area Code Tree for storing and managing points of interest (POIs). We also present the algorithms for mapping the area code of a POI, inserting and building an Area Code Tree, and an approximate nearest neighbour search. In addition, we evaluate the Area Code Tree for accuracy, tree construction time, and compare its search performance with the Brute Force method. Our Area Code Tree provides the ability to perform efficient repeated searching for continuous nearest neighbours but at the cost of accuracy. We observe from our experiment results that Area Code Tree performs significantly better than the Brute Force method for an approximate nearest neighbour search. We also found that the search time for the Area Code Tree is very low and constantantly less than 10ms regardless of the number of POIs in the index. Most importantly, the Area Code Tree can achieve up to 60% accuracy for locating the nearest neighbour in dense point sets. This makes the Area Code Tree an excellent candidate for continuous approximate nearest neighbour search for location based services.

5.1 Future Work

In the future, we will evaluate the search performance of the Area Code Tree by comparing to other spatial access methods and strategies for continuous nearest neighbour search such as Time-Parameterized *k*NN (TP-*k*NN) [22] and Retrieve-Influence-Set *k*NN (RIS-*k*NN) [27] strategies. Time-Parameterized *k*NN strategy (TP-*k*NN) utilizes the R-tree to stores spacial objects using their coordinates of Minimum Bounding Rectangles. There-

fore, the nearest neighbour search using this strategy involves repeated access of R-tree index which results a large number of disk accesses. Although Retrieve-Influence-Set *k*NN strategy (RIS-*k*NN) reduces the number of repeated searches and the cost of precomputation for continuous nearest neighbours searching but it is still expensive. Because RIS-*k*NN strategy performs multiple TP-*k*NN queries to estimate all possible client's movement path. On the other hand, our Area Code Tree stores spacial objects (i.e. POI) using their area codes instead of their coordinates of Minimum Bounding Rectangles. This provides the Area Code Tree the ability to quickly identify nearest neighbours to the query point by doing simple numeric comparisons. We also extend the Area Code Tree for K nearest neighbour search and find that approximate k-nearest neighbour searching is accomplished in very low and constant time. In future, we will evaluate the search performance of the Area Code Tree for locating k-nearest neighbours with other special access methods. Furthermore, we will use other types of area codes, such as telephone area codes, and postal/zip codes in the Area Code Tree. Our experiment results show that the Area Code Tree has promise for different applications in continuous query processing.

Bibliography

[1] Jie Bao, Chi-Yin Chow, Mohamed F Mokbel, and Wei-Shinn Ku. Efficient evaluation of k-range nearest neighbour queries in road networks. In *2010 Eleventh International Conference on Mobile Data Management*, pages 115–124. IEEE, 2010.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 322–331, 1990.

[3] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. St2b-tree: A self-tunable spatio-temporal b+-tree index for moving objects. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 29–42, 2008.

[4] Chi-Yin Chow, Mohamed Mokbel, Joe Naps, and Suman Nath. Approximate evaluation of range nearest neighbour queries with quality guarantee. *Advances in Spatial and Temporal Databases*, pages 283–301, 2009.

[5] Victor Teixeira de Almeida. Towards optimal continuous nearest neighbour queries in spatial databases. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, pages 227–234, 2006.

[6] Gisli R Hjaltason and Hanan Samet. Ranking in spatial databases. In *International Symposium on Spatial Databases*, pages 83–95. Springer, 1995.

[7] Gísli R Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[8] Jiun-Long Huang and Chen-Che Huang. A proxy-based approach to continuous location-based spatial queries in mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 25(2):260–273, 2013.

[9] Weihuang Huang, Guoliang Li, Kian-Lee Tan, and Jianhua Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, pages 932–941. ACM, 2012.

[10] Marc Moreau and Wendy Osborn. mqr-tree: A 2-dimensional spatial access method. *J. Comput. Sci. Eng.*, 15:1–12, 2012.

[11] Marc Moreau, Wendy Osborn, and Brad Anderson. The mqr-tree: improving upon a 2-dimensional spatial access method. In *Fourth International Conference on Digital Information Management*, pages 1–6. IEEE, 2009.

[12] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbour monitoring. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of data*, pages 634–645. ACM, 2005.

[13] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. Analysis and evaluation of v*-knn: an efficient algorithm for moving knn queries. *The International Journal on Very Large Data Bases*, 19(3):307–332, 2010.

[14] Wendy Osborn and Annika Hinze. Tip-tree: A spatial index for traversing locations in context-aware mobile access to digital libraries. *Pervasive and Mobile Computing*, 15:26–47, 2014.

[15] Kwangjin Park and Patrick Valduriez. A hierarchical grid index (hgi), spatial queries in wireless data broadcasting. *Distributed and Parallel Databases*, 31(3):413–446, 2013.

[16] Yonghun Park, Hyoungsoon Park, Dongmin Seo, and Jaesoo Yoo. An index structure for efficient k-nn query processing in location based services. In *Proceedings of the 4th International Conference*, pages 1–6. IEEE, 2009.

[17] Yonghun Park, Dongmin Seo, Jongtae Lim, Jinju Lee, Mikyoung Kim, Weiwei Bao, Christopher T Ryu, and Jaesoo Yoo. A new spatial index structure for efficient query processing in location based services. In *Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC), 2010 IEEE International Conference*, pages 434–441. IEEE, 2010.

[18] Fatema Rahman and Wendy Osborn. The area code tree for nearest neighbour searching. In *Proc. 2009 IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, 2015.

[19] Fatema Rahman and Wendy Osborn. The area code tree for approximate nearest neighbour searching in dense point sets. In *ISCA 25th Int't Conf. on Software Engineering and Data Engineering*, 2016.

[20] Zhexuan Song and Nick Roussopoulos. K-nearest neighbour search for moving query point. In *International Symposium on Spatial and Temporal Databases*, pages 79–96. Springer, 2001.

[21] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 334–345. ACM, 2002.

[22] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbour search. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 287–298. VLDB Endowment, 2002.

[23] Pandian Vasant. *Handbook of research on novel soft computing intelligent algorithms: Theory and practical applications*. IGI Global, 2013.

[24] Venki. Trie: Insert and search. `https://www.geeksforgeeks.org/trie-insert-and-search`, 2015.

[25] Hui Xiao, Qingquan Li, and Qinghong Sheng. Continuous k-nearest neighbour queries for moving objects. In *International Symposium on Intelligence Computation and Applications*, pages 444–453. Springer, 2007.

[26] Kefeng Xuan, Geng Zhao, David Taniar, Maytham Safar, and Bala Srinivasan. Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools and Applications*, 53(2):459–479, 2011.

[27] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 443–454. ACM, 2003.

[28] Baihua Zheng, Jianliang Xu, Wang-Chien Lee, and Lun Lee. Grid-partition index: a hybrid method for nearest-neighbour queries in wireless location-based services. *The International Journal on Very Large Data Bases*, 15(1):21–39, 2006.

Appendix A

Source Code

A.1 Area Code Mapping of a POI

```
<?php
require("config.inc.php");
set_time_limit(5000);

$x_maximum=179.0;
$x_minimum=0;
$y_maximum=9.0;
$y_minimum=0.0;
$x_maximum=($x_maximum*100000)/2;
$x_minimum=($x_minimum*100000)/2;
$y_maximum=($y_maximum * 100000)/2;
$y_minimum=($y_minimum * 100000)/2;
//reterieve points fom database
$query = "Select place,x_1,x_2,y_1,y_2 FROM databasetable";

if($rows)
  foreach ($rows as $row) {
 $place_id=$row["place"];

$point_x1=($row["x_1"]*100000)/2;
$point_x2=($row["x_2"]*100000)/2;
$point_y1=($row["y_1"]*100000)/2;
$point_y2=($row["y_2"]*100000)/2;

$main_lx=$x_minimum;
$main_hx=$x_maximum;
$main_ly=$y_minimum;
$main_hy=$y_maximum;
$center_x=floor(($point_x1+$point_x2)/2);
$center_y=floor(($point_y1+$point_y2)/2);

$direction="";
```

```
while(true){
$main_center_x=floor(($main_lx+$main_hx)/2);
$main_center_y=floor(($main_ly+$main_hy)/2);
// area code calculation
if( $center_x >= $main_center_x && $center_y >= $main_center_y){
    $direction.="4";
        $new_main_lx=$main_center_x;
        $new_main_ly=$main_center_y;
        $new_main_hx=$main_hx;
        $new_main_hy=$main_hy;
}

else if($center_x <= $main_center_x && $center_y <= $main_center_y){
    $direction.="1";
    $new_main_lx=$main_lx;
        $new_main_ly=$main_ly;
        $new_main_hx=$main_center_x;
        $new_main_hy=$main_center_y;
}
else if($center_x >= $main_center_x && $center_y <= $main_center_y){
    $direction.="2";
    $new_main_lx=$main_center_x;
        $new_main_ly=$main_ly;
        $new_main_hx=$main_hx;
        $new_main_hy=$main_center_y;
}
else if($center_x <= $main_center_x && $center_y >= $main_center_y){
    $direction.="3";
    $new_main_lx=$main_lx;
        $new_main_ly=$main_center_y;
        $new_main_hx=$main_center_x;
        $new_main_hy=$main_hy;
}
else if($main_center_x==$center_x && $main_center_y==$center_y){
    $direction.="5";
        break;
}
if(floor($main_hx)==floor($new_main_hx) &&
    floor($main_hy)==floor($new_main_hy) &&
        floor($main_lx)==floor($new_main_lx) &&
        floor($main_ly)==floor($new_main_ly)){
  break;
}
```

```
$main_hx=floor($new_main_hx);
$main_hy=floor($new_main_hy);
$main_lx=floor($new_main_lx);
$main_ly=floor($new_main_ly);

$query="UPDATE databasetable area_code=:area
WHERE place=:id ";
$query_params=array(
     ':area'=>$direction,
     ':id'=>$place_id
     );
try{
     $stmt=$db->prepare($query);
     $result=$stmt->execute($query_params);
    }
catch (PDOException $ex) {

}
}
?>
```

A.2 Inserting and Building an Area Code Tree

```
<?php
require("config.inc.php");
set_time_limit(50000);
$child_data_format="\"1\"-{}\"2\"-{}\"3\"-{}\"4\"-{}\"5\"-{}";
//get child node value for corresponding row
function child_value_calculation($row_id,$db){
$table=$GLOBALS['table_name'];
$query="select child from " .$table ."
where row=:row_id";
$query_params=array(':row_id'=>$row_id);
try {
     $stmt=$db->prepare($query);
     $result=$stmt->execute($query_params);
}
catch (PDOException $ex) {
     echo $ex->getMessage();
}
          $row = $stmt->fetch();
          return $row["child"];
}
//get what is inside any {} for a specific $pre<-prefix value
function get_value_in_string($pre,$value){
```

```
    $count=0;
        $index=0;
        $double_quote='"';
        //checking the number of parenthesis
    for($i=0;$i<strlen($value);$i++){
                if($value[$i]=='"'){
                   $count++;
                }
                if($count==((2*$pre)-1))
                        break;
        }
      $position="";
           for($i=$i+5;$i<strlen($value);$i++){
             if($value[$i]=='}')
                   break;
             else  $position.= $value[$i];
           }
         return $position;
   }
//get the postfix value
function postfix_value($value){
    $post_value="";
        if(strlen($value)==0)
          return $post_value;
        if($value[0]=='P')
          for($i=2;$value[$i]!=',';$i++)
             $post_value.=$value[$i];
        else return $post_value;
    return $post_value;
  }
//get location information
function locaiton_value($value){
$location="";
if(strlen($value)==0)
return $location;
if($value[0]=='P'){
//go through the string  until , is found
for($i=2;$value[$i]!=',';$i++);
//next character  is L, so next all the values after :
if($value[$i+1]=='L'){
        for($i=$i+3;$i<strlen($value);$i++)
        $location.=$value[$i];
        }
   return $location;
        }
```

```
else return  $location;
if($value[0]=='N'){
     $location_exist=0;
     for($i=2;$i<strlen($value);$i++)
     if($value[$i]==','){
          $location_exist=1;
          break;
          }
     if($location_exist==1)
     for($i=$i+3;$i<strlen($value);$i++)
     $location.=$value[$i];
     }
return  $location;
}
//check the next row
function next_row_value($value){
$next_row="";
if(strlen($value)==0)
return $next_row;
if($value[0]=='N'){
     for($i=2;$i<strlen($value);$i++){
     if($value[$i]<48 && $value[$i]>57)
     break;
     $next_row.=$value[$i];
          }
}
else return $next_row;
return $next_row;
}
// build the child node value
//this function appends a location after row if necessary
function location_append_after_Row($pre,$value,$data){
$prefix=chr($pre+48);
$child_data="";
for($i=0;$i<strlen($value);$i++){
          if(($value[$i]==$prefix)&&($value[$i-1]=='"'))
          break;
          else $child_data=$child_data.$value[$i];
     }
for(;$i<strlen($value);$i++){
     if($value[$i]=='}')
     break;
          else $child_data=$child_data.$value[$i];
}
$child_data.=$data;
```

```
for(;$i<strlen($value);$i++)
$child_data=$child_data.$value[$i];
return $child_data;
}
function child_data_value($pre,$value,$data){
$prefix=chr($pre+48);
$child_data="";
for($i=0;$i<strlen($value);$i++){
if(($value[$i]==$prefix)&&($value[$i-1]=='"'))
    break;
else $child_data=$child_data.$value[$i];
}
$child_data=$child_data.$prefix."\"-{".$data;
for(;$i<strlen($value);$i++)
if($value[$i]=='}')
break;
for(;$i<strlen($value);$i++)
    $child_data=$child_data.$value[$i];
return $child_data;
}
//insert new row
function insert_new_row($db,$row_id,$prefix_data){
$child_data_format ="\"1\"-{}\"2\"-{}\"3\"-{}\"4\"-{}\"5\"-{}";
$table= $GLOBALS['table_name'];
$query="insert into ".$table." (parent,child,pre)
values (:parent,:child,:prefix)" ;
$query_params=array(':parent'=>$row_id ,
':child'=>$child_data_format,':prefix'=>$prefix_data);
try {
    $stmt=$db->prepare($query);
    $result=$stmt->execute($query_params);
    }
catch (PDOException $ex) {
    echo $ex->getMessage();
        }
}
function get_latest_id($db){
$table=$GLOBALS['table_name'];
$query="SELECT LAST_INSERT_ID() as row  from  ".$table;
try{
    $stmt=$db->prepare($query);
    $result=$stmt->execute();
    }
    catch(PDOException $ex){
    echo $ex->getMessage();
```

```
    }
        $row=$stmt->fetch();
return $row["row"];
}
function Update_with_nextRow_location_value
($db,$row_id,$pre,$value,$data){
$node_info=location_append_after_Row($pre,$value,$data);
$table=$GLOBALS['table_name'];
$query="update ". $table ."set child=:data where row=:row_id ";
$query_params=array(':data'=>$node_info,':row_id'=>$row_id);
try {
    $stmt= $db->prepare($query);
    $result = $stmt->execute($query_params);
        }
    catch (PDOException $ex) {
    echo $ex->getMessage();
        }
}

//update when any field is empty
  // $db-> database link
  // $row_id -> row link
  // $pre -> prefix
  // $value -> child value
function Update_with_prefix_location_value
($db,$row_id,$pre,$value,$post_with_location){
$node_info=child_data_value($pre,$value,$post_with_location);
$table=$GLOBALS['table_name'];
$query="update  ".$table." set child=:data
where row=:row_id ";
$query_params=array(':data'=>$node_info,':row_id'=>$row_id);
try{
    $stmt=$db->prepare($query);
    $result=$stmt->execute($query_params);
    }
    catch (PDOException $ex){
    echo $ex->getMessage();

        }
}
// first id => starting row number
//$area => area code
function process_data($first_id,$Area,$location,$db){
//last  created row number
$row_number=get_latest_id($db);
```

```
//first cahracter of area code
if(strlen($Area)>0)
$pre = ord($Area[0])-48;
else $pre='';
//remaining character of area code
if(strlen($Area)>1)
    $post=substr($Area,1);//new_value
else $post="";
        $row_id=intval($first_id,0);
        $prefix_data="";
while(1){
    //child value in database
        $value=child_value_calculation($row_id,$db);
    $inner_value="";
        $postfix="";
        $location_data="";
        $next_row="";
    //getting the value for coresponding  prefix
    switch($pre){
        case 1:
            $inner_value=get_value_in_string(1,$value);
            break;
            case 2:
            $inner_value=get_value_in_string(2,$value);
            break;
            case 3:
            $inner_value=get_value_in_string(3,$value);
            break;
            case 4:
            $inner_value=get_value_in_string(4,$value);
            break;
            case 5:
            $inner_value=get_value_in_string(5,$value);
            break;
        }
$postfix=postfix_value($inner_value);
$location_data=locaiton_value($inner_value);
$next_row=next_row_value($inner_value);
if(strlen($inner_value)==0){
    $post_with_location = "P:".$post.",L:".$location;
        Update_with_prefix_location_value
        ($db,$row_id,$pre,$value,$post_with_location);
        break;
        }
else{
```

```
if(strlen($postfix)>0&&strlen( $location_data)>0){
//create new row and get the row number
        $prefix_data.=$pre;
        insert_new_row($db,$row_id,$prefix_data);
        $row_number=get_latest_id($db);
        $data ="N:".$row_number;
        //update previous node data
        Update_with_prefix_location_value
         ($db,$row_id,$pre,$value,$data);
    //get info about new row
        $row_id=intval($row_number,0);
        //collect two location info
        $location_1_prefix=$post[0];
        $location_2_prefix=$postfix[0];
        $location_1_place=$location;
        $location_2_place= $location_data;
        $location_1_postfix=substr($post,1);
        $location_2_postfix=substr($postfix,1);
    $both_empty=0;
//if prefix of new location and old location are same
//create a new row
while($location_1_prefix==$location_2_prefix){
    $prefix_data.=$location_1_prefix;
        insert_new_row($db,$row_id,$prefix_data);
        $row_number=get_latest_id($db);
        $value=child_value_calculation($row_id,$db);
        $data ="N:".$row_number;
        Update_with_prefix_location_value
        ($db,$row_id,$location_1_prefix,$value,$data);
        $row_id=intval($row_number,0);
if(strlen($location_1_postfix)==1&&
        strlen($location_2_postfix)==1){
        $both_empty=1;
    $location_1_prefix=$location_1_postfix[0];
        $location_2_prefix=$location_2_postfix[0];
        $location_1_postfix="";
        $location_2_postfix="";
    break;}
        $location_1_prefix=$location_1_postfix[0];
        $location_2_prefix=$location_2_postfix[0];
        $location_1_postfix=substr($location_1_postfix,1);
        $location_2_postfix=substr($location_2_postfix,1);
}
if($both_empty==0){
        $value=child_value_calculation($row_id,$db);
```

```
        $data ="P:".$location_1_postfix.",L:".$location_1_place;
        Update_with_prefix_location_value
        ($db,$row_id,$location_1_prefix,$value,$data);
        $value=child_value_calculation($row_id,$db);
        $data ="P:".$location_2_postfix.",L:".$location_2_place;
        Update_with_prefix_location_value
        ($db,$row_id,$location_2_prefix,$value,$data);
}
else {
    $value=child_value_calculation($row_id,$db);
    $data="P:".$location_1_postfix.",
        L:".$location_1_place.",
        L:".$location_2_place;
        Update_with_prefix_location_value
        ($db,$row_id,$location_2_prefix,$value,$data);
}
break;
}
else if(strlen($next_row)>0){
$prefix_data.=$pre;
$prevous_prefix=$pre;
if(strlen($post)>0)
    $pre = ord($post[0])-48;
        else $pre='';
                if(strlen($post)>1)
                $post = substr($post,1);
                        else{$post="";
                        $data=",L:".$location;
                        Update_with_nextRow_location_value(
                        $db,$row_id,$pre,$value,$data);
                }
                $row_id= intval($next_row,0);
        }
}
}
}
function get_PlaceAreaCode($id,$db){
        $table=$GLOBALS['area_table_name'];
        $query="select place,area_code from " .$table .
        " where Place_id = :row_id";
        $query_params = array(':row_id' => $id);
        try {
        $stmt=$db->prepare($query);
        $result=$stmt->execute($query_params);
    }
```

```
    catch (PDOException $ex) {
        echo $ex->getMessage();
    }
        $row=$stmt->fetch();
        $place_name=$row['place'];
        $area=$row['area_code'];
return array($place_name,$area);
}
if (isset($_POST["mySubmit"])){
        $time_start = microtime(true);
        insert_new_row($db,0,"");
        $first_id=get_latest_id($db);
        $query="select place,area_code FROM ".$area_table_name;
        try {
        $stmt   = $db->prepare($query);
            $result = $stmt->execute();
    }
    catch (PDOException $ex) {
    }
    $rows=$stmt->fetchall();
    if($rows)
        foreach ($rows as $row) {
            $new_area_code=$row['area_code'];
                $location=$row['place'];
                process_data
                ($first_id,$new_area_code,$location,$db);
        }
        $time_end = microtime(true);
        $time = $time_end - $time_start;
        echo "Times required to build area code for Table
        ".$area_table_name." are ".$time." seconds";
}
 ?>
```

A.3 Approximate Nearest Neighbour Search

```
<?php
require("config.inc.php");
set_time_limit(50000);
$table_name="databasetable";
function area_code_calculation($latitude,$longitude){
$main_lx=(0*100000)/2;
$main_hx=(179.0*100000)/2;
$main_ly=(0.0*100000)/2;
$main_hy=(9.0*100000)/2;
$center_x=($latitude*100000)/2;
```

```
$center_y=($longitude*100000)/2;
$direction="";
while(true){
    $main_center_x=floor(($main_lx+$main_hx)/2);
    $main_center_y=floor(($main_ly+$main_hy)/2);
    if($center_x>=$main_center_x && $center_y>=$main_center_y){
    $direction.="4";
    $new_main_lx=$main_center_x;
    $new_main_ly=$main_center_y;
        $new_main_hx=$main_hx;
                $new_main_hy=$main_hy;
            }
        // area code calculation
    else if($center_x<=$main_center_x && $center_y<=$main_center_y){
    $direction.="1";
    $new_main_lx=$main_lx;
        $new_main_ly=$main_ly;
    $new_main_hx=$main_center_x;
    $new_main_hy=$main_center_y;
    }
    else if($center_x>=$main_center_x && $center_y<=$main_center_y){
    $direction.="2";
    $new_main_lx=$main_center_x;
    $new_main_ly=$main_ly;
    $new_main_hx=$main_hx;
    $new_main_hy=$main_center_y;
    }
    else if($center_x<=$main_center_x && $center_y>=$main_center_y){
    $direction.="3";
    $new_main_lx=$main_lx;
        $new_main_ly=$main_center_y;
    $new_main_hx=$main_center_x;
    $new_main_hy=$main_hy;
    }
    else if($main_center_x==$center_x && $main_center_y==$center_y){
    $direction.="5";
        break;
    }
    if(floor($main_hx)==floor($new_main_hx) &&
    floor($main_hy)==floor($new_main_hy) &&
        floor($main_lx)==floor($new_main_lx) &&
        floor($main_ly) == floor($new_main_ly))
    {
    break;
        }
```

```
$main_hx=floor($new_main_hx);
$main_hy=floor($new_main_hy);
$main_lx=floor($new_main_lx);
$main_ly=floor($new_main_ly);
}
return $direction;
}
function get_value_in_string($pre,$value){
$count=0;
$index=0;
$double_quote='"';
//checking the number of parenthesis
for($i=0;$i<strlen($value);$i++){
if($value[$i]=='"'){
   $count++;
}
if($count==((2*$pre)-1))
break;
}
$position="";
for($i=$i+5;$i<strlen($value);$i++){
if($value[$i]=='}')
break;
else $position.=$value[$i];
}
return $position;
}
function postfix_value($value){
$post_value="";
if(strlen($value)==0)
return $post_value;
if($value[0]=='P')
for($i=2;$value[$i]!=',';$i++)
$post_value.=$value[$i];
else return $post_value;
return $post_value;
}
//get location information
function locaiton_value($value){
$location="";
if(strlen($value)==0)
return $location;
for($i=0;$i<strlen($value);$i++)
if($value[$i]=='L')
break;
```

```
$count_start=0;
for($i=$i+2;$i<strlen($value);$i++)
if($value[$i]>='0'&& $value[$i]<='9'){
        $location.=$value[$i];
        $count_start=1;
        }
else {
    if($count_start==1)
    break;}

return $location;
}
//check the next row
function next_row_value($value){
$next_row="";
if(strlen($value)==0)
return $next_row;
if($value[0]=='N'){
for($i=2;$i<strlen($value);$i++){
   if($value[$i]==',')
    break;
        $next_row.=$value[$i];
        }
        }
        else return $next_row;
return $next_row;
}
function child_value_calculation($row_id,$db){
$table=$GLOBALS['table_name'];
$query="select child,parent,pre from " .$table ."
where row=:row_id";
$query_params=array(':row_id'=>$row_id);
try {
    $stmt=$db->prepare($query);
    $result=$stmt->execute($query_params);
    }
    catch (PDOException $ex) {
    echo $ex->getMessage();
    }
$row=$stmt->fetch();
return array($row["child"],$row["parent"],$row["pre"]);
}
$file=fopen("path-points.txt","r");
$path_no=0;
echo "File Name: ".$table_name."<br/><br/>";
```

```
//Output a line of the file until the end is reached
while(! feof($file)){
$path_no++;
fscanf($file,"%lf%lf",$latitude,$longitude);
echo "Path ".$path_no." : ". $latitude."      ".$longitude."<br/>";
$time_start=microtime(true);
$direction=area_code_calculation($latitude,$longitude);
$location_found=0;
$prefix=$direction[0];
$postfix_value=substr($direction,1);
$row_id=1;
echo "Nearby Path Name:";
while(1){
$row_info=child_value_calculation($row_id,$db);
$value=$row_info[0];
$parent=$row_info[1];
$pref=$row_info[2];
$inner_value="";
$postfix="";
$location="";
$next_row="";
$inner_value=get_value_in_string($prefix,$value);
if(strlen($inner_value)==0){
for($i=1;$i<=5;$i++){
        if($prefix==$i)
        continue;
        else {
    $inner_value=get_value_in_string($i,$value);
        if(strlen($inner_value)>0){
                break;
                }
    }
}
}
$postfix=postfix_value($inner_value);//what is in database
$location=locaiton_value($inner_value);//what is in database
$next_row=next_row_value($inner_value);
if(strlen($location)>0){
$location_found++;
echo $location."     ";
for($i=1;$i<=5;$i++){
if(intval($prefix,0)==$i)
continue;
else {
$inner_value=get_value_in_string($i,$value);
```

```
if(strlen($inner_value)>0){
$prefix=$i;
break;
                                }
        }
        }
}
if($location_found==0){
$prefix=$postfix_value[0];
$postfix_value=substr($postfix_value,1); }
if(strlen($next_row)>0){
$row_id=intval($next_row,0);
continue;
        }
if($location_found >=2||strlen($postfix_value)==0)
break;
}
$time_end = microtime(true);
$time = $time_end - $time_start;
echo "Total time for Path ".$path_no." is ".$time." seconds";
}
fclose($file);
?>
```

Appendix B

Sample Data Sets

B.1 Static Data set of 1000 POIs across New Zealand

```
174.78000 : 0.00000 : 174.78000 : 0.00000 : 0 : 1
174.06000 : 0.02000 : 174.06000 : 0.02000 : 0 : 2
176.63000 : 0.02000 : 176.63000 : 0.02000 : 0 : 3
177.43000 : 0.02000 : 177.43000 : 0.02000 : 0 : 4
173.71000 : 0.03000 : 173.71000 : 0.03000 : 0 : 5
176.89000 : 0.03000 : 176.89000 : 0.03000 : 0 : 6
177.28000 : 0.03000 : 177.28000 : 0.03000 : 0 : 7
175.51000 : 0.04000 : 175.51000 : 0.04000 : 0 : 8
176.29000 : 0.04000 : 176.29000 : 0.04000 : 0 : 9
176.45000 : 0.04000 : 176.45000 : 0.04000 : 0 : 10
173.54000 : 0.06000 : 173.54000 : 0.06000 : 0 : 11
175.47000 : 0.06000 : 175.47000 : 0.06000 : 0 : 12
174.12000 : 0.07000 : 174.12000 : 0.07000 : 0 : 13
175.13000 : 0.07000 : 175.13000 : 0.07000 : 0 : 14
174.77000 : 0.08000 : 174.77000 : 0.08000 : 0 : 15
176.85000 : 0.09000 : 176.85000 : 0.09000 : 0 : 16
174.43000 : 0.10000 : 174.43000 : 0.10000 : 0 : 17
175.72000 : 0.10000 : 175.72000 : 0.10000 : 0 : 18
177.34000 : 0.10000 : 177.34000 : 0.10000 : 0 : 19
174.64000 : 0.11000 : 174.64000 : 0.11000 : 0 : 20
175.22000 : 0.11000 : 175.22000 : 0.11000 : 0 : 21
175.35000 : 0.11000 : 175.35000 : 0.11000 : 0 : 22
173.72000 : 0.12000 : 173.72000 : 0.12000 : 0 : 23
176.42000 : 0.13000 : 176.42000 : 0.13000 : 0 : 24
177.62000 : 0.13000 : 177.62000 : 0.13000 : 0 : 25
174.15000 : 0.14000 : 174.15000 : 0.14000 : 0 : 26
174.85000 : 0.14000 : 174.85000 : 0.14000 : 0 : 27
175.03000 : 0.14000 : 175.03000 : 0.14000 : 0 : 28
175.96000 : 0.15000 : 175.96000 : 0.15000 : 0 : 29
175.32000 : 0.16000 : 175.32000 : 0.16000 : 0 : 30
177.74000 : 0.16000 : 177.74000 : 0.16000 : 0 : 31
177.96000 : 0.19000 : 177.96000 : 0.19000 : 0 : 32
```

178.14000 : 0.19000 : 178.14000 : 0.19000 : 0 : 33
175.45000 : 0.20000 : 175.45000 : 0.20000 : 0 : 34
178.04000 : 0.22000 : 178.04000 : 0.22000 : 0 : 35
174.57000 : 0.23000 : 174.57000 : 0.23000 : 0 : 36
176.72000 : 0.24000 : 176.72000 : 0.24000 : 0 : 37
175.53000 : 0.26000 : 175.53000 : 0.26000 : 0 : 38
178.31000 : 0.26000 : 178.31000 : 0.26000 : 0 : 39
173.28000 : 0.28000 : 173.28000 : 0.28000 : 0 : 40
175.75000 : 0.28000 : 175.75000 : 0.28000 : 0 : 41
177.24000 : 0.28000 : 177.24000 : 0.28000 : 0 : 42
174.37000 : 0.29000 : 174.37000 : 0.29000 : 0 : 43
176.25000 : 0.30000 : 176.25000 : 0.30000 : 0 : 44
174.82000 : 0.31000 : 174.82000 : 0.31000 : 0 : 45
177.62000 : 0.32000 : 177.62000 : 0.32000 : 0 : 46
174.11000 : 0.33000 : 174.11000 : 0.33000 : 0 : 47
175.65000 : 0.33000 : 175.65000 : 0.33000 : 0 : 48
176.68000 : 0.33000 : 176.68000 : 0.33000 : 0 : 49
177.52000 : 0.33000 : 177.52000 : 0.33000 : 0 : 50
174.08000 : 0.34000 : 174.08000 : 0.34000 : 0 : 51
173.68000 : 0.35000 : 173.68000 : 0.35000 : 0 : 52
177.18000 : 0.35000 : 177.18000 : 0.35000 : 0 : 53
177.46000 : 0.35000 : 177.46000 : 0.35000 : 0 : 54
175.47000 : 0.36000 : 175.47000 : 0.36000 : 0 : 55
175.02000 : 0.37000 : 175.02000 : 0.37000 : 0 : 56
176.93000 : 0.37000 : 176.93000 : 0.37000 : 0 : 57
174.96000 : 0.38000 : 174.96000 : 0.38000 : 0 : 58
176.79000 : 0.38000 : 176.79000 : 0.38000 : 0 : 59
177.03000 : 0.38000 : 177.03000 : 0.38000 : 0 : 60
174.78000 : 0.39000 : 174.78000 : 0.39000 : 0 : 61
174.81000 : 0.39000 : 174.81000 : 0.39000 : 0 : 62
175.12000 : 0.40000 : 175.12000 : 0.40000 : 0 : 63
173.72000 : 0.41000 : 173.72000 : 0.41000 : 0 : 64
178.00000 : 0.42000 : 178.00000 : 0.42000 : 0 : 65
176.62000 : 0.44000 : 176.62000 : 0.44000 : 0 : 66
177.21000 : 0.44000 : 177.21000 : 0.44000 : 0 : 67
176.96000 : 0.45000 : 176.96000 : 0.45000 : 0 : 68
177.70000 : 0.45000 : 177.70000 : 0.45000 : 0 : 69
175.10000 : 0.46000 : 175.10000 : 0.46000 : 0 : 70
174.24000 : 0.47000 : 174.24000 : 0.47000 : 0 : 71
175.84000 : 0.47000 : 175.84000 : 0.47000 : 0 : 72
173.79000 : 0.48000 : 173.79000 : 0.48000 : 0 : 73
175.10000 : 0.48000 : 175.10000 : 0.48000 : 0 : 74
176.80000 : 0.48000 : 176.80000 : 0.48000 : 0 : 75
173.59000 : 0.49000 : 173.59000 : 0.49000 : 0 : 76
173.89000 : 0.49000 : 173.89000 : 0.49000 : 0 : 77

177.57000 : 0.50000 : 177.57000 : 0.50000 : 0 : 78
178.00000 : 0.50000 : 178.00000 : 0.50000 : 0 : 79
173.84000 : 0.53000 : 173.84000 : 0.53000 : 0 : 80
174.52000 : 0.53000 : 174.52000 : 0.53000 : 0 : 81
175.95000 : 0.53000 : 175.95000 : 0.53000 : 0 : 82
177.60000 : 0.53000 : 177.60000 : 0.53000 : 0 : 83
177.52000 : 0.55000 : 177.52000 : 0.55000 : 0 : 84
177.64000 : 0.56000 : 177.64000 : 0.56000 : 0 : 85
174.02000 : 0.57000 : 174.02000 : 0.57000 : 0 : 86
177.68000 : 0.57000 : 177.68000 : 0.57000 : 0 : 87
175.63000 : 0.59000 : 175.63000 : 0.59000 : 0 : 88
175.89000 : 0.59000 : 175.89000 : 0.59000 : 0 : 89
174.97000 : 0.61000 : 174.97000 : 0.61000 : 0 : 90
176.65000 : 0.61000 : 176.65000 : 0.61000 : 0 : 91
177.12000 : 0.61000 : 177.12000 : 0.61000 : 0 : 92
177.99000 : 0.61000 : 177.99000 : 0.61000 : 0 : 93
176.44000 : 0.62000 : 176.44000 : 0.62000 : 0 : 94
178.26000 : 0.62000 : 178.26000 : 0.62000 : 0 : 95
175.70000 : 0.63000 : 175.70000 : 0.63000 : 0 : 96
173.58000 : 0.66000 : 173.58000 : 0.66000 : 0 : 97
175.86000 : 0.66000 : 175.86000 : 0.66000 : 0 : 98
177.53000 : 0.66000 : 177.53000 : 0.66000 : 0 : 99
174.17000 : 0.67000 : 174.17000 : 0.67000 : 0 : 100
175.29000 : 0.67000 : 175.29000 : 0.67000 : 0 : 101
173.67000 : 0.69000 : 173.67000 : 0.69000 : 0 : 102
177.92000 : 0.69000 : 177.92000 : 0.69000 : 0 : 103
177.56000 : 0.70000 : 177.56000 : 0.70000 : 0 : 104
174.14000 : 0.71000 : 174.14000 : 0.71000 : 0 : 105
178.26000 : 0.71000 : 178.26000 : 0.71000 : 0 : 106
173.91000 : 0.73000 : 173.91000 : 0.73000 : 0 : 107
175.42000 : 0.73000 : 175.42000 : 0.73000 : 0 : 108
175.74000 : 0.73000 : 175.74000 : 0.73000 : 0 : 109
173.29000 : 0.76000 : 173.29000 : 0.76000 : 0 : 110
178.08000 : 0.76000 : 178.08000 : 0.76000 : 0 : 111
173.27000 : 0.77000 : 173.27000 : 0.77000 : 0 : 112
174.39000 : 0.77000 : 174.39000 : 0.77000 : 0 : 113
175.03000 : 0.77000 : 175.03000 : 0.77000 : 0 : 114
178.08000 : 0.77000 : 178.08000 : 0.77000 : 0 : 115
174.33000 : 0.78000 : 174.33000 : 0.78000 : 0 : 116
177.09000 : 0.78000 : 177.09000 : 0.78000 : 0 : 117
176.85000 : 0.79000 : 176.85000 : 0.79000 : 0 : 118
177.51000 : 0.80000 : 177.51000 : 0.80000 : 0 : 119
177.75000 : 0.80000 : 177.75000 : 0.80000 : 0 : 120
175.50000 : 0.81000 : 175.50000 : 0.81000 : 0 : 121
176.73000 : 0.81000 : 176.73000 : 0.81000 : 0 : 122

176.95000 : 0.81000 : 176.95000 : 0.81000 : 0 : 123
174.36000 : 0.82000 : 174.36000 : 0.82000 : 0 : 124
177.52000 : 0.82000 : 177.52000 : 0.82000 : 0 : 125
175.83000 : 0.83000 : 175.83000 : 0.83000 : 0 : 126
176.45000 : 0.83000 : 176.45000 : 0.83000 : 0 : 127
178.09000 : 0.83000 : 178.09000 : 0.83000 : 0 : 128
174.01000 : 0.84000 : 174.01000 : 0.84000 : 0 : 129
177.03000 : 0.84000 : 177.03000 : 0.84000 : 0 : 130
178.21000 : 0.85000 : 178.21000 : 0.85000 : 0 : 131
173.65000 : 0.86000 : 173.65000 : 0.86000 : 0 : 132
177.13000 : 0.86000 : 177.13000 : 0.86000 : 0 : 133
174.39000 : 0.87000 : 174.39000 : 0.87000 : 0 : 134
174.46000 : 0.87000 : 174.46000 : 0.87000 : 0 : 135
174.18000 : 0.88000 : 174.18000 : 0.88000 : 0 : 136
175.32000 : 0.88000 : 175.32000 : 0.88000 : 0 : 137
175.68000 : 0.88000 : 175.68000 : 0.88000 : 0 : 138
176.21000 : 0.88000 : 176.21000 : 0.88000 : 0 : 139
177.88000 : 0.88000 : 177.88000 : 0.88000 : 0 : 140
174.86000 : 0.89000 : 174.86000 : 0.89000 : 0 : 141
176.09000 : 0.89000 : 176.09000 : 0.89000 : 0 : 142
177.29000 : 0.89000 : 177.29000 : 0.89000 : 0 : 143
173.63000 : 0.90000 : 173.63000 : 0.90000 : 0 : 144
175.85000 : 0.90000 : 175.85000 : 0.90000 : 0 : 145
176.78000 : 0.90000 : 176.78000 : 0.90000 : 0 : 146
177.82000 : 0.90000 : 177.82000 : 0.90000 : 0 : 147
175.61000 : 0.93000 : 175.61000 : 0.93000 : 0 : 148
176.63000 : 0.93000 : 176.63000 : 0.93000 : 0 : 149
176.17000 : 0.94000 : 176.17000 : 0.94000 : 0 : 150
177.23000 : 0.94000 : 177.23000 : 0.94000 : 0 : 151
177.38000 : 0.94000 : 177.38000 : 0.94000 : 0 : 152
175.87000 : 0.95000 : 175.87000 : 0.95000 : 0 : 153
175.88000 : 0.95000 : 175.88000 : 0.95000 : 0 : 154
173.18000 : 0.96000 : 173.18000 : 0.96000 : 0 : 155
173.52000 : 0.97000 : 173.52000 : 0.97000 : 0 : 156
176.37000 : 0.97000 : 176.37000 : 0.97000 : 0 : 157
175.42000 : 0.98000 : 175.42000 : 0.98000 : 0 : 158
176.73000 : 0.98000 : 176.73000 : 0.98000 : 0 : 159
174.76000 : 0.99000 : 174.76000 : 0.99000 : 0 : 160
176.78000 : 0.99000 : 176.78000 : 0.99000 : 0 : 161
174.02000 : 1.00000 : 174.02000 : 1.00000 : 0 : 162
176.77000 : 1.01000 : 176.77000 : 1.01000 : 0 : 163
177.61000 : 1.01000 : 177.61000 : 1.01000 : 0 : 164
175.29000 : 1.02000 : 175.29000 : 1.02000 : 0 : 165
175.83000 : 1.03000 : 175.83000 : 1.03000 : 0 : 166
177.13000 : 1.03000 : 177.13000 : 1.03000 : 0 : 167

174.49000 : 1.04000 : 174.49000 : 1.04000 : 0 : 168
176.34000 : 1.04000 : 176.34000 : 1.04000 : 0 : 169
177.73000 : 1.04000 : 177.73000 : 1.04000 : 0 : 170
176.86000 : 1.05000 : 176.86000 : 1.05000 : 0 : 171
173.71000 : 1.06000 : 173.71000 : 1.06000 : 0 : 172
175.42000 : 1.06000 : 175.42000 : 1.06000 : 0 : 173
176.50000 : 1.06000 : 176.50000 : 1.06000 : 0 : 174
177.33000 : 1.06000 : 177.33000 : 1.06000 : 0 : 175
177.79000 : 1.06000 : 177.79000 : 1.06000 : 0 : 176
176.09000 : 1.07000 : 176.09000 : 1.07000 : 0 : 177
176.34000 : 1.07000 : 176.34000 : 1.07000 : 0 : 178
177.25000 : 1.07000 : 177.25000 : 1.07000 : 0 : 179
174.95000 : 1.08000 : 174.95000 : 1.08000 : 0 : 180
176.20000 : 1.09000 : 176.20000 : 1.09000 : 0 : 181
176.67000 : 1.11000 : 176.67000 : 1.11000 : 0 : 182
176.72000 : 1.11000 : 176.72000 : 1.11000 : 0 : 183
177.22000 : 1.11000 : 177.22000 : 1.11000 : 0 : 184
177.71000 : 1.11000 : 177.71000 : 1.11000 : 0 : 185
174.04000 : 1.12000 : 174.04000 : 1.12000 : 0 : 186
175.88000 : 1.12000 : 175.88000 : 1.12000 : 0 : 187
178.20000 : 1.12000 : 178.20000 : 1.12000 : 0 : 188
175.50000 : 1.13000 : 175.50000 : 1.13000 : 0 : 189
174.57000 : 1.14000 : 174.57000 : 1.14000 : 0 : 190
175.71000 : 1.14000 : 175.71000 : 1.14000 : 0 : 191
177.68000 : 1.14000 : 177.68000 : 1.14000 : 0 : 192
173.27000 : 1.15000 : 173.27000 : 1.15000 : 0 : 193
176.63000 : 1.15000 : 176.63000 : 1.15000 : 0 : 194
176.84000 : 1.15000 : 176.84000 : 1.15000 : 0 : 195
173.28000 : 1.17000 : 173.28000 : 1.17000 : 0 : 196
174.30000 : 1.17000 : 174.30000 : 1.17000 : 0 : 197
175.65000 : 1.17000 : 175.65000 : 1.17000 : 0 : 198
175.26000 : 1.18000 : 175.26000 : 1.18000 : 0 : 199
177.86000 : 1.18000 : 177.86000 : 1.18000 : 0 : 200
175.77000 : 1.19000 : 175.77000 : 1.19000 : 0 : 201
176.03000 : 1.19000 : 176.03000 : 1.19000 : 0 : 202
177.02000 : 1.19000 : 177.02000 : 1.19000 : 0 : 203
174.74000 : 1.20000 : 174.74000 : 1.20000 : 0 : 204
175.38000 : 1.20000 : 175.38000 : 1.20000 : 0 : 205
173.20000 : 1.22000 : 173.20000 : 1.22000 : 0 : 206
177.34000 : 1.22000 : 177.34000 : 1.22000 : 0 : 207
173.63000 : 1.23000 : 173.63000 : 1.23000 : 0 : 208
175.18000 : 1.23000 : 175.18000 : 1.23000 : 0 : 209
176.30000 : 1.25000 : 176.30000 : 1.25000 : 0 : 210
176.35000 : 1.25000 : 176.35000 : 1.25000 : 0 : 211
177.44000 : 1.25000 : 177.44000 : 1.25000 : 0 : 212

```
173.24000 : 1.26000 : 173.24000 : 1.26000 : 0 : 213
176.20000 : 1.26000 : 176.20000 : 1.26000 : 0 : 214
174.46000 : 1.27000 : 174.46000 : 1.27000 : 0 : 215
174.97000 : 1.27000 : 174.97000 : 1.27000 : 0 : 216
177.46000 : 1.27000 : 177.46000 : 1.27000 : 0 : 217
174.92000 : 1.28000 : 174.92000 : 1.28000 : 0 : 218
176.90000 : 1.28000 : 176.90000 : 1.28000 : 0 : 219
175.27000 : 1.29000 : 175.27000 : 1.29000 : 0 : 220
175.29000 : 1.30000 : 175.29000 : 1.30000 : 0 : 221
176.15000 : 1.30000 : 176.15000 : 1.30000 : 0 : 222
176.11000 : 1.33000 : 176.11000 : 1.33000 : 0 : 223
176.27000 : 1.34000 : 176.27000 : 1.34000 : 0 : 224
176.74000 : 1.34000 : 176.74000 : 1.34000 : 0 : 225
177.08000 : 1.34000 : 177.08000 : 1.34000 : 0 : 226
175.03000 : 1.35000 : 175.03000 : 1.35000 : 0 : 227
175.57000 : 1.35000 : 175.57000 : 1.35000 : 0 : 228
176.08000 : 1.35000 : 176.08000 : 1.35000 : 0 : 229
177.38000 : 1.35000 : 177.38000 : 1.35000 : 0 : 230
174.87000 : 1.36000 : 174.87000 : 1.36000 : 0 : 231
173.73000 : 1.38000 : 173.73000 : 1.38000 : 0 : 232
177.42000 : 1.38000 : 177.42000 : 1.38000 : 0 : 233
176.63000 : 1.40000 : 176.63000 : 1.40000 : 0 : 234
173.41000 : 1.41000 : 173.41000 : 1.41000 : 0 : 235
175.15000 : 1.42000 : 175.15000 : 1.42000 : 0 : 236
176.64000 : 1.42000 : 176.64000 : 1.42000 : 0 : 237
176.98000 : 1.44000 : 176.98000 : 1.44000 : 0 : 238
178.18000 : 1.45000 : 178.18000 : 1.45000 : 0 : 239
175.71000 : 1.46000 : 175.71000 : 1.46000 : 0 : 240
174.34000 : 1.49000 : 174.34000 : 1.49000 : 0 : 241
175.84000 : 1.49000 : 175.84000 : 1.49000 : 0 : 242
176.17000 : 1.49000 : 176.17000 : 1.49000 : 0 : 243
177.55000 : 1.49000 : 177.55000 : 1.49000 : 0 : 244
174.63000 : 1.51000 : 174.63000 : 1.51000 : 0 : 245
174.48000 : 1.52000 : 174.48000 : 1.52000 : 0 : 246
175.64000 : 1.52000 : 175.64000 : 1.52000 : 0 : 247
177.64000 : 1.52000 : 177.64000 : 1.52000 : 0 : 248
174.62000 : 1.53000 : 174.62000 : 1.53000 : 0 : 249
173.58000 : 1.54000 : 173.58000 : 1.54000 : 0 : 250
175.68000 : 1.54000 : 175.68000 : 1.54000 : 0 : 251
177.15000 : 1.57000 : 177.15000 : 1.57000 : 0 : 252
173.45000 : 1.58000 : 173.45000 : 1.58000 : 0 : 253
178.01000 : 1.58000 : 178.01000 : 1.58000 : 0 : 254
175.80000 : 1.60000 : 175.80000 : 1.60000 : 0 : 255
175.07000 : 1.61000 : 175.07000 : 1.61000 : 0 : 256
175.72000 : 1.61000 : 175.72000 : 1.61000 : 0 : 257
```

176.81000 : 1.61000 : 176.81000 : 1.61000 : 0 : 258
173.29000 : 1.63000 : 173.29000 : 1.63000 : 0 : 259
173.43000 : 1.63000 : 173.43000 : 1.63000 : 0 : 260
174.36000 : 1.63000 : 174.36000 : 1.63000 : 0 : 261
175.29000 : 1.63000 : 175.29000 : 1.63000 : 0 : 262
175.03000 : 1.64000 : 175.03000 : 1.64000 : 0 : 263
176.84000 : 1.64000 : 176.84000 : 1.64000 : 0 : 264
177.86000 : 1.64000 : 177.86000 : 1.64000 : 0 : 265
178.03000 : 1.64000 : 178.03000 : 1.64000 : 0 : 266
175.45000 : 1.67000 : 175.45000 : 1.67000 : 0 : 267
177.35000 : 1.67000 : 177.35000 : 1.67000 : 0 : 268
177.98000 : 1.68000 : 177.98000 : 1.68000 : 0 : 269
173.47000 : 1.70000 : 173.47000 : 1.70000 : 0 : 270
174.28000 : 1.70000 : 174.28000 : 1.70000 : 0 : 271
174.38000 : 1.70000 : 174.38000 : 1.70000 : 0 : 272
175.39000 : 1.70000 : 175.39000 : 1.70000 : 0 : 273
175.04000 : 1.71000 : 175.04000 : 1.71000 : 0 : 274
175.75000 : 1.71000 : 175.75000 : 1.71000 : 0 : 275
175.91000 : 1.71000 : 175.91000 : 1.71000 : 0 : 276
175.32000 : 1.72000 : 175.32000 : 1.72000 : 0 : 277
176.85000 : 1.72000 : 176.85000 : 1.72000 : 0 : 278
176.99000 : 1.72000 : 176.99000 : 1.72000 : 0 : 279
174.13000 : 1.73000 : 174.13000 : 1.73000 : 0 : 280
176.67000 : 1.73000 : 176.67000 : 1.73000 : 0 : 281
177.64000 : 1.73000 : 177.64000 : 1.73000 : 0 : 282
178.21000 : 1.73000 : 178.21000 : 1.73000 : 0 : 283
175.55000 : 1.75000 : 175.55000 : 1.75000 : 0 : 284
177.57000 : 1.77000 : 177.57000 : 1.77000 : 0 : 285
178.03000 : 1.77000 : 178.03000 : 1.77000 : 0 : 286
174.87000 : 1.78000 : 174.87000 : 1.78000 : 0 : 287
176.02000 : 1.79000 : 176.02000 : 1.79000 : 0 : 288
176.90000 : 1.79000 : 176.90000 : 1.79000 : 0 : 289
175.67000 : 1.80000 : 175.67000 : 1.80000 : 0 : 290
173.86000 : 1.82000 : 173.86000 : 1.82000 : 0 : 291
174.45000 : 1.83000 : 174.45000 : 1.83000 : 0 : 292
174.15000 : 1.85000 : 174.15000 : 1.85000 : 0 : 293
174.30000 : 1.85000 : 174.30000 : 1.85000 : 0 : 294
175.28000 : 1.85000 : 175.28000 : 1.85000 : 0 : 295
175.42000 : 1.86000 : 175.42000 : 1.86000 : 0 : 296
178.01000 : 1.87000 : 178.01000 : 1.87000 : 0 : 297
174.08000 : 1.88000 : 174.08000 : 1.88000 : 0 : 298
175.08000 : 1.88000 : 175.08000 : 1.88000 : 0 : 299
175.19000 : 1.88000 : 175.19000 : 1.88000 : 0 : 300
176.50000 : 1.88000 : 176.50000 : 1.88000 : 0 : 301
177.01000 : 1.89000 : 177.01000 : 1.89000 : 0 : 302

177.20000 : 1.89000 : 177.20000 : 1.89000 : 0 : 303
173.72000 : 1.90000 : 173.72000 : 1.90000 : 0 : 304
175.02000 : 1.90000 : 175.02000 : 1.90000 : 0 : 305
176.98000 : 1.90000 : 176.98000 : 1.90000 : 0 : 306
178.28000 : 1.90000 : 178.28000 : 1.90000 : 0 : 307
175.52000 : 1.91000 : 175.52000 : 1.91000 : 0 : 308
174.80000 : 1.92000 : 174.80000 : 1.92000 : 0 : 309
175.36000 : 1.93000 : 175.36000 : 1.93000 : 0 : 310
176.82000 : 1.93000 : 176.82000 : 1.93000 : 0 : 311
173.95000 : 1.94000 : 173.95000 : 1.94000 : 0 : 312
174.26000 : 1.94000 : 174.26000 : 1.94000 : 0 : 313
175.18000 : 1.94000 : 175.18000 : 1.94000 : 0 : 314
173.80000 : 1.95000 : 173.80000 : 1.95000 : 0 : 315
177.57000 : 1.95000 : 177.57000 : 1.95000 : 0 : 316
178.16000 : 1.95000 : 178.16000 : 1.95000 : 0 : 317
176.04000 : 1.96000 : 176.04000 : 1.96000 : 0 : 318
176.37000 : 1.98000 : 176.37000 : 1.98000 : 0 : 319
176.87000 : 1.98000 : 176.87000 : 1.98000 : 0 : 320
174.91000 : 1.99000 : 174.91000 : 1.99000 : 0 : 321
175.81000 : 1.99000 : 175.81000 : 1.99000 : 0 : 322
173.66000 : 2.01000 : 173.66000 : 2.01000 : 0 : 323
177.43000 : 2.01000 : 177.43000 : 2.01000 : 0 : 324
177.58000 : 2.01000 : 177.58000 : 2.01000 : 0 : 325
174.53000 : 2.03000 : 174.53000 : 2.03000 : 0 : 326
176.93000 : 2.03000 : 176.93000 : 2.03000 : 0 : 327
177.33000 : 2.04000 : 177.33000 : 2.04000 : 0 : 328
174.44000 : 2.06000 : 174.44000 : 2.06000 : 0 : 329
174.75000 : 2.06000 : 174.75000 : 2.06000 : 0 : 330
173.89000 : 2.07000 : 173.89000 : 2.07000 : 0 : 331
173.88000 : 2.08000 : 173.88000 : 2.08000 : 0 : 332
175.02000 : 2.08000 : 175.02000 : 2.08000 : 0 : 333
175.31000 : 2.11000 : 175.31000 : 2.11000 : 0 : 334
174.25000 : 2.12000 : 174.25000 : 2.12000 : 0 : 335
174.43000 : 2.12000 : 174.43000 : 2.12000 : 0 : 336
173.99000 : 2.13000 : 173.99000 : 2.13000 : 0 : 337
174.15000 : 2.13000 : 174.15000 : 2.13000 : 0 : 338
177.99000 : 2.15000 : 177.99000 : 2.15000 : 0 : 339
176.63000 : 2.16000 : 176.63000 : 2.16000 : 0 : 340
177.93000 : 2.16000 : 177.93000 : 2.16000 : 0 : 341
175.16000 : 2.19000 : 175.16000 : 2.19000 : 0 : 342
174.36000 : 2.21000 : 174.36000 : 2.21000 : 0 : 343
174.08000 : 2.22000 : 174.08000 : 2.22000 : 0 : 344
174.79000 : 2.22000 : 174.79000 : 2.22000 : 0 : 345
177.75000 : 2.23000 : 177.75000 : 2.23000 : 0 : 346
176.07000 : 2.24000 : 176.07000 : 2.24000 : 0 : 347

178.06000 : 2.24000 : 178.06000 : 2.24000 : 0 : 348
175.80000 : 2.25000 : 175.80000 : 2.25000 : 0 : 349
174.26000 : 2.26000 : 174.26000 : 2.26000 : 0 : 350
177.13000 : 2.26000 : 177.13000 : 2.26000 : 0 : 351
177.37000 : 2.26000 : 177.37000 : 2.26000 : 0 : 352
173.97000 : 2.27000 : 173.97000 : 2.27000 : 0 : 353
173.56000 : 2.28000 : 173.56000 : 2.28000 : 0 : 354
174.38000 : 2.28000 : 174.38000 : 2.28000 : 0 : 355
175.16000 : 2.28000 : 175.16000 : 2.28000 : 0 : 356
175.93000 : 2.28000 : 175.93000 : 2.28000 : 0 : 357
175.55000 : 2.29000 : 175.55000 : 2.29000 : 0 : 358
177.45000 : 2.29000 : 177.45000 : 2.29000 : 0 : 359
175.39000 : 2.31000 : 175.39000 : 2.31000 : 0 : 360
177.40000 : 2.31000 : 177.40000 : 2.31000 : 0 : 361
174.25000 : 2.35000 : 174.25000 : 2.35000 : 0 : 362
174.39000 : 2.37000 : 174.39000 : 2.37000 : 0 : 363
175.26000 : 2.39000 : 175.26000 : 2.39000 : 0 : 364
176.22000 : 2.39000 : 176.22000 : 2.39000 : 0 : 365
175.77000 : 2.40000 : 175.77000 : 2.40000 : 0 : 366
173.66000 : 2.41000 : 173.66000 : 2.41000 : 0 : 367
174.44000 : 2.41000 : 174.44000 : 2.41000 : 0 : 368
176.63000 : 2.41000 : 176.63000 : 2.41000 : 0 : 369
177.17000 : 2.41000 : 177.17000 : 2.41000 : 0 : 370
175.80000 : 2.42000 : 175.80000 : 2.42000 : 0 : 371
173.22000 : 2.43000 : 173.22000 : 2.43000 : 0 : 372
173.25000 : 2.43000 : 173.25000 : 2.43000 : 0 : 373
177.41000 : 2.43000 : 177.41000 : 2.43000 : 0 : 374
177.88000 : 2.43000 : 177.88000 : 2.43000 : 0 : 375
175.60000 : 2.44000 : 175.60000 : 2.44000 : 0 : 376
176.55000 : 2.44000 : 176.55000 : 2.44000 : 0 : 377
178.18000 : 2.44000 : 178.18000 : 2.44000 : 0 : 378
174.98000 : 2.45000 : 174.98000 : 2.45000 : 0 : 379
175.66000 : 2.45000 : 175.66000 : 2.45000 : 0 : 380
177.17000 : 2.46000 : 177.17000 : 2.46000 : 0 : 381
177.93000 : 2.46000 : 177.93000 : 2.46000 : 0 : 382
174.75000 : 2.49000 : 174.75000 : 2.49000 : 0 : 383
176.57000 : 2.49000 : 176.57000 : 2.49000 : 0 : 384
177.43000 : 2.49000 : 177.43000 : 2.49000 : 0 : 385
175.97000 : 2.52000 : 175.97000 : 2.52000 : 0 : 386
175.69000 : 2.53000 : 175.69000 : 2.53000 : 0 : 387
176.77000 : 2.53000 : 176.77000 : 2.53000 : 0 : 388
176.35000 : 2.55000 : 176.35000 : 2.55000 : 0 : 389
173.95000 : 2.57000 : 173.95000 : 2.57000 : 0 : 390
176.02000 : 2.57000 : 176.02000 : 2.57000 : 0 : 391
173.17000 : 2.58000 : 173.17000 : 2.58000 : 0 : 392

174.39000 : 2.58000 : 174.39000 : 2.58000 : 0 : 393
174.80000 : 2.58000 : 174.80000 : 2.58000 : 0 : 394
173.90000 : 2.59000 : 173.90000 : 2.59000 : 0 : 395
176.08000 : 2.59000 : 176.08000 : 2.59000 : 0 : 396
177.94000 : 2.59000 : 177.94000 : 2.59000 : 0 : 397
174.13000 : 2.60000 : 174.13000 : 2.60000 : 0 : 398
177.92000 : 2.60000 : 177.92000 : 2.60000 : 0 : 399
175.11000 : 2.62000 : 175.11000 : 2.62000 : 0 : 400
176.02000 : 2.62000 : 176.02000 : 2.62000 : 0 : 401
178.02000 : 2.62000 : 178.02000 : 2.62000 : 0 : 402
174.59000 : 2.63000 : 174.59000 : 2.63000 : 0 : 403
176.72000 : 2.63000 : 176.72000 : 2.63000 : 0 : 404
177.97000 : 2.63000 : 177.97000 : 2.63000 : 0 : 405
173.35000 : 2.64000 : 173.35000 : 2.64000 : 0 : 406
175.96000 : 2.65000 : 175.96000 : 2.65000 : 0 : 407
176.28000 : 2.65000 : 176.28000 : 2.65000 : 0 : 408
178.09000 : 2.65000 : 178.09000 : 2.65000 : 0 : 409
178.22000 : 2.65000 : 178.22000 : 2.65000 : 0 : 410
176.48000 : 2.66000 : 176.48000 : 2.66000 : 0 : 411
177.90000 : 2.66000 : 177.90000 : 2.66000 : 0 : 412
174.08000 : 2.67000 : 174.08000 : 2.67000 : 0 : 413
174.46000 : 2.69000 : 174.46000 : 2.69000 : 0 : 414
175.44000 : 2.69000 : 175.44000 : 2.69000 : 0 : 415
177.70000 : 2.69000 : 177.70000 : 2.69000 : 0 : 416
176.65000 : 2.70000 : 176.65000 : 2.70000 : 0 : 417
176.53000 : 2.71000 : 176.53000 : 2.71000 : 0 : 418
176.69000 : 2.71000 : 176.69000 : 2.71000 : 0 : 419
177.82000 : 2.71000 : 177.82000 : 2.71000 : 0 : 420
178.29000 : 2.71000 : 178.29000 : 2.71000 : 0 : 421
177.64000 : 2.72000 : 177.64000 : 2.72000 : 0 : 422
173.36000 : 2.73000 : 173.36000 : 2.73000 : 0 : 423
174.00000 : 2.73000 : 174.00000 : 2.73000 : 0 : 424
174.93000 : 2.73000 : 174.93000 : 2.73000 : 0 : 425
176.78000 : 2.73000 : 176.78000 : 2.73000 : 0 : 426
176.97000 : 2.74000 : 176.97000 : 2.74000 : 0 : 427
175.69000 : 2.75000 : 175.69000 : 2.75000 : 0 : 428
173.54000 : 2.76000 : 173.54000 : 2.76000 : 0 : 429
174.81000 : 2.76000 : 174.81000 : 2.76000 : 0 : 430
176.20000 : 2.77000 : 176.20000 : 2.77000 : 0 : 431
173.38000 : 2.78000 : 173.38000 : 2.78000 : 0 : 432
174.02000 : 2.79000 : 174.02000 : 2.79000 : 0 : 433
174.66000 : 2.79000 : 174.66000 : 2.79000 : 0 : 434
177.16000 : 2.79000 : 177.16000 : 2.79000 : 0 : 435
175.71000 : 2.80000 : 175.71000 : 2.80000 : 0 : 436
175.02000 : 2.81000 : 175.02000 : 2.81000 : 0 : 437

176.33000 : 2.81000 : 176.33000 : 2.81000 : 0 : 438
176.70000 : 2.81000 : 176.70000 : 2.81000 : 0 : 439
177.25000 : 2.81000 : 177.25000 : 2.81000 : 0 : 440
177.87000 : 2.81000 : 177.87000 : 2.81000 : 0 : 441
178.17000 : 2.81000 : 178.17000 : 2.81000 : 0 : 442
176.26000 : 2.82000 : 176.26000 : 2.82000 : 0 : 443
177.45000 : 2.82000 : 177.45000 : 2.82000 : 0 : 444
178.25000 : 2.82000 : 178.25000 : 2.82000 : 0 : 445
173.50000 : 2.85000 : 173.50000 : 2.85000 : 0 : 446
176.35000 : 2.85000 : 176.35000 : 2.85000 : 0 : 447
176.06000 : 2.86000 : 176.06000 : 2.86000 : 0 : 448
175.65000 : 2.88000 : 175.65000 : 2.88000 : 0 : 449
175.77000 : 2.88000 : 175.77000 : 2.88000 : 0 : 450
174.18000 : 2.89000 : 174.18000 : 2.89000 : 0 : 451
177.30000 : 2.89000 : 177.30000 : 2.89000 : 0 : 452
173.63000 : 2.91000 : 173.63000 : 2.91000 : 0 : 453
178.21000 : 2.91000 : 178.21000 : 2.91000 : 0 : 454
178.30000 : 2.91000 : 178.30000 : 2.91000 : 0 : 455
175.25000 : 2.92000 : 175.25000 : 2.92000 : 0 : 456
175.93000 : 2.93000 : 175.93000 : 2.93000 : 0 : 457
175.41000 : 2.94000 : 175.41000 : 2.94000 : 0 : 458
176.70000 : 2.94000 : 176.70000 : 2.94000 : 0 : 459
176.89000 : 2.94000 : 176.89000 : 2.94000 : 0 : 460
178.11000 : 2.94000 : 178.11000 : 2.94000 : 0 : 461
173.48000 : 2.95000 : 173.48000 : 2.95000 : 0 : 462
174.36000 : 2.96000 : 174.36000 : 2.96000 : 0 : 463
175.17000 : 2.96000 : 175.17000 : 2.96000 : 0 : 464
174.75000 : 2.97000 : 174.75000 : 2.97000 : 0 : 465
173.78000 : 2.98000 : 173.78000 : 2.98000 : 0 : 466
176.84000 : 2.98000 : 176.84000 : 2.98000 : 0 : 467
177.76000 : 2.98000 : 177.76000 : 2.98000 : 0 : 468
176.00000 : 2.99000 : 176.00000 : 2.99000 : 0 : 469
176.28000 : 3.00000 : 176.28000 : 3.00000 : 0 : 470
173.18000 : 3.01000 : 173.18000 : 3.01000 : 0 : 471
176.95000 : 3.01000 : 176.95000 : 3.01000 : 0 : 472
173.45000 : 3.02000 : 173.45000 : 3.02000 : 0 : 473
173.20000 : 3.03000 : 173.20000 : 3.03000 : 0 : 474
176.07000 : 3.03000 : 176.07000 : 3.03000 : 0 : 475
178.31000 : 3.03000 : 178.31000 : 3.03000 : 0 : 476
174.32000 : 3.04000 : 174.32000 : 3.04000 : 0 : 477
176.41000 : 3.04000 : 176.41000 : 3.04000 : 0 : 478
173.29000 : 3.05000 : 173.29000 : 3.05000 : 0 : 479
173.40000 : 3.05000 : 173.40000 : 3.05000 : 0 : 480
175.60000 : 3.05000 : 175.60000 : 3.05000 : 0 : 481
175.82000 : 3.06000 : 175.82000 : 3.06000 : 0 : 482

175.87000 : 3.06000 : 175.87000 : 3.06000 : 0 : 483
178.21000 : 3.06000 : 178.21000 : 3.06000 : 0 : 484
175.42000 : 3.07000 : 175.42000 : 3.07000 : 0 : 485
178.12000 : 3.07000 : 178.12000 : 3.07000 : 0 : 486
175.64000 : 3.08000 : 175.64000 : 3.08000 : 0 : 487
174.60000 : 3.09000 : 174.60000 : 3.09000 : 0 : 488
175.50000 : 3.10000 : 175.50000 : 3.10000 : 0 : 489
177.88000 : 3.10000 : 177.88000 : 3.10000 : 0 : 490
173.87000 : 3.11000 : 173.87000 : 3.11000 : 0 : 491
175.20000 : 3.11000 : 175.20000 : 3.11000 : 0 : 492
176.14000 : 3.11000 : 176.14000 : 3.11000 : 0 : 493
173.93000 : 3.12000 : 173.93000 : 3.12000 : 0 : 494
175.14000 : 3.13000 : 175.14000 : 3.13000 : 0 : 495
175.69000 : 3.13000 : 175.69000 : 3.13000 : 0 : 496
177.08000 : 3.13000 : 177.08000 : 3.13000 : 0 : 497
177.36000 : 3.13000 : 177.36000 : 3.13000 : 0 : 498
177.94000 : 3.13000 : 177.94000 : 3.13000 : 0 : 499
175.73000 : 3.14000 : 175.73000 : 3.14000 : 0 : 500
176.24000 : 3.14000 : 176.24000 : 3.14000 : 0 : 501
178.11000 : 3.15000 : 178.11000 : 3.15000 : 0 : 502
178.30000 : 3.15000 : 178.30000 : 3.15000 : 0 : 503
174.71000 : 3.16000 : 174.71000 : 3.16000 : 0 : 504
175.03000 : 3.16000 : 175.03000 : 3.16000 : 0 : 505
176.08000 : 3.18000 : 176.08000 : 3.18000 : 0 : 506
176.42000 : 3.18000 : 176.42000 : 3.18000 : 0 : 507
174.80000 : 3.19000 : 174.80000 : 3.19000 : 0 : 508
175.82000 : 3.19000 : 175.82000 : 3.19000 : 0 : 509
176.70000 : 3.19000 : 176.70000 : 3.19000 : 0 : 510
173.67000 : 3.21000 : 173.67000 : 3.21000 : 0 : 511
174.82000 : 3.21000 : 174.82000 : 3.21000 : 0 : 512
177.02000 : 3.21000 : 177.02000 : 3.21000 : 0 : 513
177.28000 : 3.21000 : 177.28000 : 3.21000 : 0 : 514
177.55000 : 3.21000 : 177.55000 : 3.21000 : 0 : 515
175.78000 : 3.23000 : 175.78000 : 3.23000 : 0 : 516
177.16000 : 3.23000 : 177.16000 : 3.23000 : 0 : 517
173.38000 : 3.25000 : 173.38000 : 3.25000 : 0 : 518
175.60000 : 3.25000 : 175.60000 : 3.25000 : 0 : 519
177.21000 : 3.25000 : 177.21000 : 3.25000 : 0 : 520
173.27000 : 3.27000 : 173.27000 : 3.27000 : 0 : 521
175.33000 : 3.27000 : 175.33000 : 3.27000 : 0 : 522
176.82000 : 3.27000 : 176.82000 : 3.27000 : 0 : 523
174.65000 : 3.28000 : 174.65000 : 3.28000 : 0 : 524
176.80000 : 3.28000 : 176.80000 : 3.28000 : 0 : 525
173.60000 : 3.29000 : 173.60000 : 3.29000 : 0 : 526
174.50000 : 3.29000 : 174.50000 : 3.29000 : 0 : 527

175.49000 : 3.29000 : 175.49000 : 3.29000 : 0 : 528
175.54000 : 3.29000 : 175.54000 : 3.29000 : 0 : 529
175.75000 : 3.30000 : 175.75000 : 3.30000 : 0 : 530
176.83000 : 3.30000 : 176.83000 : 3.30000 : 0 : 531
176.99000 : 3.31000 : 176.99000 : 3.31000 : 0 : 532
173.34000 : 3.32000 : 173.34000 : 3.32000 : 0 : 533
174.57000 : 3.32000 : 174.57000 : 3.32000 : 0 : 534
176.09000 : 3.32000 : 176.09000 : 3.32000 : 0 : 535
176.26000 : 3.35000 : 176.26000 : 3.35000 : 0 : 536
176.94000 : 3.35000 : 176.94000 : 3.35000 : 0 : 537
174.45000 : 3.38000 : 174.45000 : 3.38000 : 0 : 538
176.95000 : 3.38000 : 176.95000 : 3.38000 : 0 : 539
174.00000 : 3.39000 : 174.00000 : 3.39000 : 0 : 540
175.40000 : 3.40000 : 175.40000 : 3.40000 : 0 : 541
175.44000 : 3.40000 : 175.44000 : 3.40000 : 0 : 542
175.67000 : 3.40000 : 175.67000 : 3.40000 : 0 : 543
176.06000 : 3.40000 : 176.06000 : 3.40000 : 0 : 544
176.75000 : 3.40000 : 176.75000 : 3.40000 : 0 : 545
178.33000 : 3.40000 : 178.33000 : 3.40000 : 0 : 546
176.26000 : 3.41000 : 176.26000 : 3.41000 : 0 : 547
176.82000 : 3.42000 : 176.82000 : 3.42000 : 0 : 548
173.81000 : 3.43000 : 173.81000 : 3.43000 : 0 : 549
176.52000 : 3.43000 : 176.52000 : 3.43000 : 0 : 550
177.77000 : 3.43000 : 177.77000 : 3.43000 : 0 : 551
177.86000 : 3.43000 : 177.86000 : 3.43000 : 0 : 552
174.10000 : 3.44000 : 174.10000 : 3.44000 : 0 : 553
178.15000 : 3.45000 : 178.15000 : 3.45000 : 0 : 554
175.77000 : 3.46000 : 175.77000 : 3.46000 : 0 : 555
175.81000 : 3.46000 : 175.81000 : 3.46000 : 0 : 556
176.42000 : 3.46000 : 176.42000 : 3.46000 : 0 : 557
174.88000 : 3.48000 : 174.88000 : 3.48000 : 0 : 558
174.51000 : 3.49000 : 174.51000 : 3.49000 : 0 : 559
174.59000 : 3.49000 : 174.59000 : 3.49000 : 0 : 560
176.41000 : 3.49000 : 176.41000 : 3.49000 : 0 : 561
177.24000 : 3.49000 : 177.24000 : 3.49000 : 0 : 562
175.28000 : 3.50000 : 175.28000 : 3.50000 : 0 : 563
176.32000 : 3.50000 : 176.32000 : 3.50000 : 0 : 564
175.77000 : 3.51000 : 175.77000 : 3.51000 : 0 : 565
176.45000 : 3.51000 : 176.45000 : 3.51000 : 0 : 566
174.38000 : 3.53000 : 174.38000 : 3.53000 : 0 : 567
178.05000 : 3.53000 : 178.05000 : 3.53000 : 0 : 568
178.13000 : 3.53000 : 178.13000 : 3.53000 : 0 : 569
178.21000 : 3.53000 : 178.21000 : 3.53000 : 0 : 570
177.68000 : 3.55000 : 177.68000 : 3.55000 : 0 : 571
176.74000 : 3.56000 : 176.74000 : 3.56000 : 0 : 572

176.25000 : 3.57000 : 176.25000 : 3.57000 : 0 : 573
173.65000 : 3.58000 : 173.65000 : 3.58000 : 0 : 574
176.15000 : 3.58000 : 176.15000 : 3.58000 : 0 : 575
176.40000 : 3.58000 : 176.40000 : 3.58000 : 0 : 576
173.25000 : 3.60000 : 173.25000 : 3.60000 : 0 : 577
177.76000 : 3.60000 : 177.76000 : 3.60000 : 0 : 578
175.12000 : 3.61000 : 175.12000 : 3.61000 : 0 : 579
175.72000 : 3.61000 : 175.72000 : 3.61000 : 0 : 580
173.35000 : 3.62000 : 173.35000 : 3.62000 : 0 : 581
175.53000 : 3.63000 : 175.53000 : 3.63000 : 0 : 582
176.65000 : 3.63000 : 176.65000 : 3.63000 : 0 : 583
173.48000 : 3.64000 : 173.48000 : 3.64000 : 0 : 584
173.75000 : 3.64000 : 173.75000 : 3.64000 : 0 : 585
174.69000 : 3.64000 : 174.69000 : 3.64000 : 0 : 586
175.60000 : 3.64000 : 175.60000 : 3.64000 : 0 : 587
174.24000 : 3.65000 : 174.24000 : 3.65000 : 0 : 588
178.29000 : 3.65000 : 178.29000 : 3.65000 : 0 : 589
175.63000 : 3.66000 : 175.63000 : 3.66000 : 0 : 590
173.94000 : 3.69000 : 173.94000 : 3.69000 : 0 : 591
177.38000 : 3.69000 : 177.38000 : 3.69000 : 0 : 592
177.57000 : 3.69000 : 177.57000 : 3.69000 : 0 : 593
176.04000 : 3.70000 : 176.04000 : 3.70000 : 0 : 594
173.47000 : 3.71000 : 173.47000 : 3.71000 : 0 : 595
177.49000 : 3.71000 : 177.49000 : 3.71000 : 0 : 596
175.17000 : 3.73000 : 175.17000 : 3.73000 : 0 : 597
175.92000 : 3.73000 : 175.92000 : 3.73000 : 0 : 598
175.01000 : 3.74000 : 175.01000 : 3.74000 : 0 : 599
175.71000 : 3.75000 : 175.71000 : 3.75000 : 0 : 600
177.33000 : 3.76000 : 177.33000 : 3.76000 : 0 : 601
173.67000 : 3.78000 : 173.67000 : 3.78000 : 0 : 602
175.80000 : 3.78000 : 175.80000 : 3.78000 : 0 : 603
175.93000 : 3.78000 : 175.93000 : 3.78000 : 0 : 604
177.74000 : 3.78000 : 177.74000 : 3.78000 : 0 : 605
174.42000 : 3.80000 : 174.42000 : 3.80000 : 0 : 606
175.92000 : 3.81000 : 175.92000 : 3.81000 : 0 : 607
175.93000 : 3.83000 : 175.93000 : 3.83000 : 0 : 608
173.65000 : 3.85000 : 173.65000 : 3.85000 : 0 : 609
175.04000 : 3.86000 : 175.04000 : 3.86000 : 0 : 610
175.67000 : 3.86000 : 175.67000 : 3.86000 : 0 : 611
177.27000 : 3.86000 : 177.27000 : 3.86000 : 0 : 612
177.04000 : 3.87000 : 177.04000 : 3.87000 : 0 : 613
177.83000 : 3.87000 : 177.83000 : 3.87000 : 0 : 614
175.15000 : 3.88000 : 175.15000 : 3.88000 : 0 : 615
175.93000 : 3.88000 : 175.93000 : 3.88000 : 0 : 616
177.56000 : 3.88000 : 177.56000 : 3.88000 : 0 : 617

177.64000 : 3.88000 : 177.64000 : 3.88000 : 0 : 618
174.68000 : 3.89000 : 174.68000 : 3.89000 : 0 : 619
177.48000 : 3.89000 : 177.48000 : 3.89000 : 0 : 620
173.29000 : 3.90000 : 173.29000 : 3.90000 : 0 : 621
173.94000 : 3.90000 : 173.94000 : 3.90000 : 0 : 622
175.83000 : 3.90000 : 175.83000 : 3.90000 : 0 : 623
174.89000 : 3.91000 : 174.89000 : 3.91000 : 0 : 624
174.63000 : 3.92000 : 174.63000 : 3.92000 : 0 : 625
175.67000 : 3.92000 : 175.67000 : 3.92000 : 0 : 626
177.88000 : 3.92000 : 177.88000 : 3.92000 : 0 : 627
173.50000 : 3.93000 : 173.50000 : 3.93000 : 0 : 628
175.60000 : 3.95000 : 175.60000 : 3.95000 : 0 : 629
176.16000 : 3.95000 : 176.16000 : 3.95000 : 0 : 630
176.99000 : 3.96000 : 176.99000 : 3.96000 : 0 : 631
178.04000 : 3.96000 : 178.04000 : 3.96000 : 0 : 632
173.21000 : 3.97000 : 173.21000 : 3.97000 : 0 : 633
175.72000 : 3.97000 : 175.72000 : 3.97000 : 0 : 634
176.66000 : 3.97000 : 176.66000 : 3.97000 : 0 : 635
177.80000 : 3.97000 : 177.80000 : 3.97000 : 0 : 636
175.53000 : 3.98000 : 175.53000 : 3.98000 : 0 : 637
175.75000 : 3.98000 : 175.75000 : 3.98000 : 0 : 638
178.06000 : 3.99000 : 178.06000 : 3.99000 : 0 : 639
173.83000 : 4.00000 : 173.83000 : 4.00000 : 0 : 640
175.50000 : 4.00000 : 175.50000 : 4.00000 : 0 : 641
178.23000 : 4.00000 : 178.23000 : 4.00000 : 0 : 642
175.71000 : 4.01000 : 175.71000 : 4.01000 : 0 : 643
177.55000 : 4.02000 : 177.55000 : 4.02000 : 0 : 644
174.75000 : 4.03000 : 174.75000 : 4.03000 : 0 : 645
176.40000 : 4.04000 : 176.40000 : 4.04000 : 0 : 646
177.89000 : 4.04000 : 177.89000 : 4.04000 : 0 : 647
176.63000 : 4.07000 : 176.63000 : 4.07000 : 0 : 648
173.56000 : 4.08000 : 173.56000 : 4.08000 : 0 : 649
175.87000 : 4.08000 : 175.87000 : 4.08000 : 0 : 650
176.76000 : 4.08000 : 176.76000 : 4.08000 : 0 : 651
174.95000 : 4.09000 : 174.95000 : 4.09000 : 0 : 652
176.81000 : 4.09000 : 176.81000 : 4.09000 : 0 : 653
177.56000 : 4.09000 : 177.56000 : 4.09000 : 0 : 654
173.21000 : 4.11000 : 173.21000 : 4.11000 : 0 : 655
173.47000 : 4.11000 : 173.47000 : 4.11000 : 0 : 656
174.98000 : 4.11000 : 174.98000 : 4.11000 : 0 : 657
178.25000 : 4.11000 : 178.25000 : 4.11000 : 0 : 658
173.85000 : 4.12000 : 173.85000 : 4.12000 : 0 : 659
175.49000 : 4.12000 : 175.49000 : 4.12000 : 0 : 660
177.02000 : 4.12000 : 177.02000 : 4.12000 : 0 : 661
174.65000 : 4.13000 : 174.65000 : 4.13000 : 0 : 662

174.47000 : 4.14000 : 174.47000 : 4.14000 : 0 : 663
173.98000 : 4.15000 : 173.98000 : 4.15000 : 0 : 664
177.48000 : 4.15000 : 177.48000 : 4.15000 : 0 : 665
175.53000 : 4.16000 : 175.53000 : 4.16000 : 0 : 666
176.36000 : 4.16000 : 176.36000 : 4.16000 : 0 : 667
177.93000 : 4.16000 : 177.93000 : 4.16000 : 0 : 668
173.77000 : 4.17000 : 173.77000 : 4.17000 : 0 : 669
176.19000 : 4.17000 : 176.19000 : 4.17000 : 0 : 670
173.36000 : 4.18000 : 173.36000 : 4.18000 : 0 : 671
174.30000 : 4.19000 : 174.30000 : 4.19000 : 0 : 672
174.58000 : 4.19000 : 174.58000 : 4.19000 : 0 : 673
175.57000 : 4.19000 : 175.57000 : 4.19000 : 0 : 674
177.76000 : 4.19000 : 177.76000 : 4.19000 : 0 : 675
175.00000 : 4.22000 : 175.00000 : 4.22000 : 0 : 676
175.93000 : 4.22000 : 175.93000 : 4.22000 : 0 : 677
173.29000 : 4.24000 : 173.29000 : 4.24000 : 0 : 678
173.42000 : 4.24000 : 173.42000 : 4.24000 : 0 : 679
177.43000 : 4.24000 : 177.43000 : 4.24000 : 0 : 680
177.51000 : 4.24000 : 177.51000 : 4.24000 : 0 : 681
177.59000 : 4.25000 : 177.59000 : 4.25000 : 0 : 682
178.02000 : 4.25000 : 178.02000 : 4.25000 : 0 : 683
173.53000 : 4.26000 : 173.53000 : 4.26000 : 0 : 684
175.85000 : 4.26000 : 175.85000 : 4.26000 : 0 : 685
175.87000 : 4.26000 : 175.87000 : 4.26000 : 0 : 686
177.09000 : 4.26000 : 177.09000 : 4.26000 : 0 : 687
174.88000 : 4.28000 : 174.88000 : 4.28000 : 0 : 688
176.02000 : 4.28000 : 176.02000 : 4.28000 : 0 : 689
177.96000 : 4.28000 : 177.96000 : 4.28000 : 0 : 690
178.01000 : 4.28000 : 178.01000 : 4.28000 : 0 : 691
174.37000 : 4.30000 : 174.37000 : 4.30000 : 0 : 692
174.47000 : 4.30000 : 174.47000 : 4.30000 : 0 : 693
176.07000 : 4.30000 : 176.07000 : 4.30000 : 0 : 694
176.60000 : 4.30000 : 176.60000 : 4.30000 : 0 : 695
176.73000 : 4.31000 : 176.73000 : 4.31000 : 0 : 696
174.45000 : 4.32000 : 174.45000 : 4.32000 : 0 : 697
178.15000 : 4.32000 : 178.15000 : 4.32000 : 0 : 698
174.19000 : 4.34000 : 174.19000 : 4.34000 : 0 : 699
174.33000 : 4.34000 : 174.33000 : 4.34000 : 0 : 700
176.58000 : 4.34000 : 176.58000 : 4.34000 : 0 : 701
174.56000 : 4.35000 : 174.56000 : 4.35000 : 0 : 702
177.45000 : 4.35000 : 177.45000 : 4.35000 : 0 : 703
177.21000 : 4.36000 : 177.21000 : 4.36000 : 0 : 704
173.32000 : 4.38000 : 173.32000 : 4.38000 : 0 : 705
174.33000 : 4.38000 : 174.33000 : 4.38000 : 0 : 706
178.26000 : 4.40000 : 178.26000 : 4.40000 : 0 : 707

```
174.63000 : 4.41000 : 174.63000 : 4.41000 : 0 : 708
174.25000 : 4.42000 : 174.25000 : 4.42000 : 0 : 709
174.54000 : 4.43000 : 174.54000 : 4.43000 : 0 : 710
174.76000 : 4.43000 : 174.76000 : 4.43000 : 0 : 711
178.02000 : 4.43000 : 178.02000 : 4.43000 : 0 : 712
174.81000 : 4.44000 : 174.81000 : 4.44000 : 0 : 713
176.37000 : 4.44000 : 176.37000 : 4.44000 : 0 : 714
175.70000 : 4.45000 : 175.70000 : 4.45000 : 0 : 715
174.45000 : 4.46000 : 174.45000 : 4.46000 : 0 : 716
177.10000 : 4.46000 : 177.10000 : 4.46000 : 0 : 717
174.95000 : 4.47000 : 174.95000 : 4.47000 : 0 : 718
174.75000 : 4.48000 : 174.75000 : 4.48000 : 0 : 719
175.10000 : 4.48000 : 175.10000 : 4.48000 : 0 : 720
175.10000 : 4.48000 : 175.10000 : 4.48000 : 0 : 721
175.86000 : 4.48000 : 175.86000 : 4.48000 : 0 : 722
174.29000 : 4.49000 : 174.29000 : 4.49000 : 0 : 723
175.45000 : 4.49000 : 175.45000 : 4.49000 : 0 : 724
177.03000 : 4.50000 : 177.03000 : 4.50000 : 0 : 725
174.25000 : 4.51000 : 174.25000 : 4.51000 : 0 : 726
175.50000 : 4.51000 : 175.50000 : 4.51000 : 0 : 727
176.56000 : 4.51000 : 176.56000 : 4.51000 : 0 : 728
178.31000 : 4.53000 : 178.31000 : 4.53000 : 0 : 729
176.32000 : 4.54000 : 176.32000 : 4.54000 : 0 : 730
177.90000 : 4.54000 : 177.90000 : 4.54000 : 0 : 731
176.48000 : 4.55000 : 176.48000 : 4.55000 : 0 : 732
173.98000 : 4.56000 : 173.98000 : 4.56000 : 0 : 733
176.67000 : 4.56000 : 176.67000 : 4.56000 : 0 : 734
174.05000 : 4.57000 : 174.05000 : 4.57000 : 0 : 735
177.15000 : 4.57000 : 177.15000 : 4.57000 : 0 : 736
178.16000 : 4.58000 : 178.16000 : 4.58000 : 0 : 737
173.64000 : 4.59000 : 173.64000 : 4.59000 : 0 : 738
174.47000 : 4.60000 : 174.47000 : 4.60000 : 0 : 739
177.58000 : 4.61000 : 177.58000 : 4.61000 : 0 : 740
178.02000 : 4.61000 : 178.02000 : 4.61000 : 0 : 741
174.43000 : 4.63000 : 174.43000 : 4.63000 : 0 : 742
176.33000 : 4.63000 : 176.33000 : 4.63000 : 0 : 743
175.23000 : 4.64000 : 175.23000 : 4.64000 : 0 : 744
174.16000 : 4.65000 : 174.16000 : 4.65000 : 0 : 745
176.79000 : 4.66000 : 176.79000 : 4.66000 : 0 : 746
175.00000 : 4.68000 : 175.00000 : 4.68000 : 0 : 747
175.79000 : 4.68000 : 175.79000 : 4.68000 : 0 : 748
177.64000 : 4.68000 : 177.64000 : 4.68000 : 0 : 749
173.42000 : 4.69000 : 173.42000 : 4.69000 : 0 : 750
176.83000 : 4.69000 : 176.83000 : 4.69000 : 0 : 751
174.63000 : 4.70000 : 174.63000 : 4.70000 : 0 : 752
```

175.45000 : 4.73000 : 175.45000 : 4.73000 : 0 : 753
175.81000 : 4.74000 : 175.81000 : 4.74000 : 0 : 754
177.11000 : 4.74000 : 177.11000 : 4.74000 : 0 : 755
173.45000 : 4.75000 : 173.45000 : 4.75000 : 0 : 756
175.78000 : 4.75000 : 175.78000 : 4.75000 : 0 : 757
173.42000 : 4.76000 : 173.42000 : 4.76000 : 0 : 758
174.18000 : 4.76000 : 174.18000 : 4.76000 : 0 : 759
174.47000 : 4.77000 : 174.47000 : 4.77000 : 0 : 760
173.79000 : 4.78000 : 173.79000 : 4.78000 : 0 : 761
173.46000 : 4.79000 : 173.46000 : 4.79000 : 0 : 762
175.93000 : 4.81000 : 175.93000 : 4.81000 : 0 : 763
177.29000 : 4.81000 : 177.29000 : 4.81000 : 0 : 764
178.04000 : 4.81000 : 178.04000 : 4.81000 : 0 : 765
173.98000 : 4.82000 : 173.98000 : 4.82000 : 0 : 766
174.61000 : 4.82000 : 174.61000 : 4.82000 : 0 : 767
174.97000 : 4.82000 : 174.97000 : 4.82000 : 0 : 768
175.77000 : 4.82000 : 175.77000 : 4.82000 : 0 : 769
175.96000 : 4.83000 : 175.96000 : 4.83000 : 0 : 770
178.09000 : 4.83000 : 178.09000 : 4.83000 : 0 : 771
174.74000 : 4.84000 : 174.74000 : 4.84000 : 0 : 772
173.30000 : 4.85000 : 173.30000 : 4.85000 : 0 : 773
174.06000 : 4.85000 : 174.06000 : 4.85000 : 0 : 774
177.28000 : 4.85000 : 177.28000 : 4.85000 : 0 : 775
174.53000 : 4.87000 : 174.53000 : 4.87000 : 0 : 776
177.68000 : 4.87000 : 177.68000 : 4.87000 : 0 : 777
174.67000 : 4.88000 : 174.67000 : 4.88000 : 0 : 778
177.36000 : 4.88000 : 177.36000 : 4.88000 : 0 : 779
175.68000 : 4.92000 : 175.68000 : 4.92000 : 0 : 780
176.36000 : 4.92000 : 176.36000 : 4.92000 : 0 : 781
174.53000 : 4.93000 : 174.53000 : 4.93000 : 0 : 782
175.14000 : 4.93000 : 175.14000 : 4.93000 : 0 : 783
175.25000 : 4.93000 : 175.25000 : 4.93000 : 0 : 784
176.38000 : 4.93000 : 176.38000 : 4.93000 : 0 : 785
174.87000 : 4.95000 : 174.87000 : 4.95000 : 0 : 786
176.73000 : 4.95000 : 176.73000 : 4.95000 : 0 : 787
176.95000 : 4.96000 : 176.95000 : 4.96000 : 0 : 788
178.23000 : 4.97000 : 178.23000 : 4.97000 : 0 : 789
175.73000 : 4.99000 : 175.73000 : 4.99000 : 0 : 790
176.68000 : 4.99000 : 176.68000 : 4.99000 : 0 : 791
174.52000 : 5.00000 : 174.52000 : 5.00000 : 0 : 792
174.82000 : 5.00000 : 174.82000 : 5.00000 : 0 : 793
175.76000 : 5.00000 : 175.76000 : 5.00000 : 0 : 794
178.11000 : 5.00000 : 178.11000 : 5.00000 : 0 : 795
173.26000 : 5.01000 : 173.26000 : 5.01000 : 0 : 796
174.38000 : 5.01000 : 174.38000 : 5.01000 : 0 : 797

```
174.75000 : 5.01000 : 174.75000 : 5.01000 : 0 : 798
176.40000 : 5.02000 : 176.40000 : 5.02000 : 0 : 799
175.45000 : 5.03000 : 175.45000 : 5.03000 : 0 : 800
174.44000 : 5.04000 : 174.44000 : 5.04000 : 0 : 801
177.46000 : 5.04000 : 177.46000 : 5.04000 : 0 : 802
177.74000 : 5.04000 : 177.74000 : 5.04000 : 0 : 803
178.20000 : 5.04000 : 178.20000 : 5.04000 : 0 : 804
175.89000 : 5.05000 : 175.89000 : 5.05000 : 0 : 805
173.48000 : 5.06000 : 173.48000 : 5.06000 : 0 : 806
176.30000 : 5.07000 : 176.30000 : 5.07000 : 0 : 807
175.67000 : 5.08000 : 175.67000 : 5.08000 : 0 : 808
176.83000 : 5.08000 : 176.83000 : 5.08000 : 0 : 809
178.01000 : 5.08000 : 178.01000 : 5.08000 : 0 : 810
174.91000 : 5.10000 : 174.91000 : 5.10000 : 0 : 811
174.45000 : 5.11000 : 174.45000 : 5.11000 : 0 : 812
173.23000 : 5.12000 : 173.23000 : 5.12000 : 0 : 813
173.27000 : 5.12000 : 173.27000 : 5.12000 : 0 : 814
174.59000 : 5.13000 : 174.59000 : 5.13000 : 0 : 815
175.79000 : 5.13000 : 175.79000 : 5.13000 : 0 : 816
174.31000 : 5.14000 : 174.31000 : 5.14000 : 0 : 817
174.03000 : 5.15000 : 174.03000 : 5.15000 : 0 : 818
175.01000 : 5.15000 : 175.01000 : 5.15000 : 0 : 819
174.57000 : 5.16000 : 174.57000 : 5.16000 : 0 : 820
174.91000 : 5.16000 : 174.91000 : 5.16000 : 0 : 821
176.05000 : 5.16000 : 176.05000 : 5.16000 : 0 : 822
176.75000 : 5.16000 : 176.75000 : 5.16000 : 0 : 823
176.47000 : 5.17000 : 176.47000 : 5.17000 : 0 : 824
174.37000 : 5.18000 : 174.37000 : 5.18000 : 0 : 825
173.96000 : 5.20000 : 173.96000 : 5.20000 : 0 : 826
175.77000 : 5.21000 : 175.77000 : 5.21000 : 0 : 827
176.38000 : 5.21000 : 176.38000 : 5.21000 : 0 : 828
177.50000 : 5.21000 : 177.50000 : 5.21000 : 0 : 829
178.14000 : 5.21000 : 178.14000 : 5.21000 : 0 : 830
173.90000 : 5.22000 : 173.90000 : 5.22000 : 0 : 831
174.97000 : 5.23000 : 174.97000 : 5.23000 : 0 : 832
174.14000 : 5.25000 : 174.14000 : 5.25000 : 0 : 833
174.64000 : 5.25000 : 174.64000 : 5.25000 : 0 : 834
173.23000 : 5.27000 : 173.23000 : 5.27000 : 0 : 835
177.05000 : 5.29000 : 177.05000 : 5.29000 : 0 : 836
174.84000 : 5.30000 : 174.84000 : 5.30000 : 0 : 837
175.30000 : 5.30000 : 175.30000 : 5.30000 : 0 : 838
176.69000 : 5.30000 : 176.69000 : 5.30000 : 0 : 839
177.14000 : 5.30000 : 177.14000 : 5.30000 : 0 : 840
177.98000 : 5.30000 : 177.98000 : 5.30000 : 0 : 841
175.72000 : 5.31000 : 175.72000 : 5.31000 : 0 : 842
```

177.56000 : 5.31000 : 177.56000 : 5.31000 : 0 : 843
175.17000 : 5.32000 : 175.17000 : 5.32000 : 0 : 844
175.83000 : 5.32000 : 175.83000 : 5.32000 : 0 : 845
176.45000 : 5.32000 : 176.45000 : 5.32000 : 0 : 846
176.23000 : 5.33000 : 176.23000 : 5.33000 : 0 : 847
173.62000 : 5.34000 : 173.62000 : 5.34000 : 0 : 848
174.34000 : 5.34000 : 174.34000 : 5.34000 : 0 : 849
174.11000 : 5.35000 : 174.11000 : 5.35000 : 0 : 850
175.12000 : 5.35000 : 175.12000 : 5.35000 : 0 : 851
176.17000 : 5.35000 : 176.17000 : 5.35000 : 0 : 852
177.42000 : 5.35000 : 177.42000 : 5.35000 : 0 : 853
173.88000 : 5.36000 : 173.88000 : 5.36000 : 0 : 854
173.45000 : 5.37000 : 173.45000 : 5.37000 : 0 : 855
174.57000 : 5.37000 : 174.57000 : 5.37000 : 0 : 856
175.66000 : 5.37000 : 175.66000 : 5.37000 : 0 : 857
176.42000 : 5.37000 : 176.42000 : 5.37000 : 0 : 858
177.86000 : 5.37000 : 177.86000 : 5.37000 : 0 : 859
173.48000 : 5.40000 : 173.48000 : 5.40000 : 0 : 860
176.34000 : 5.40000 : 176.34000 : 5.40000 : 0 : 861
174.06000 : 5.41000 : 174.06000 : 5.41000 : 0 : 862
176.31000 : 5.41000 : 176.31000 : 5.41000 : 0 : 863
173.74000 : 5.43000 : 173.74000 : 5.43000 : 0 : 864
173.36000 : 5.44000 : 173.36000 : 5.44000 : 0 : 865
177.51000 : 5.45000 : 177.51000 : 5.45000 : 0 : 866
174.89000 : 5.46000 : 174.89000 : 5.46000 : 0 : 867
175.18000 : 5.46000 : 175.18000 : 5.46000 : 0 : 868
176.09000 : 5.46000 : 176.09000 : 5.46000 : 0 : 869
177.77000 : 5.46000 : 177.77000 : 5.46000 : 0 : 870
175.77000 : 5.47000 : 175.77000 : 5.47000 : 0 : 871
176.91000 : 5.47000 : 176.91000 : 5.47000 : 0 : 872
173.80000 : 5.48000 : 173.80000 : 5.48000 : 0 : 873
174.69000 : 5.49000 : 174.69000 : 5.49000 : 0 : 874
175.75000 : 5.49000 : 175.75000 : 5.49000 : 0 : 875
177.27000 : 5.50000 : 177.27000 : 5.50000 : 0 : 876
175.45000 : 5.51000 : 175.45000 : 5.51000 : 0 : 877
175.70000 : 5.51000 : 175.70000 : 5.51000 : 0 : 878
174.00000 : 5.52000 : 174.00000 : 5.52000 : 0 : 879
175.65000 : 5.52000 : 175.65000 : 5.52000 : 0 : 880
174.20000 : 5.53000 : 174.20000 : 5.53000 : 0 : 881
175.29000 : 5.55000 : 175.29000 : 5.55000 : 0 : 882
174.31000 : 5.56000 : 174.31000 : 5.56000 : 0 : 883
175.45000 : 5.57000 : 175.45000 : 5.57000 : 0 : 884
175.83000 : 5.57000 : 175.83000 : 5.57000 : 0 : 885
173.33000 : 5.60000 : 173.33000 : 5.60000 : 0 : 886
176.39000 : 5.60000 : 176.39000 : 5.60000 : 0 : 887

173.86000 : 5.61000 : 173.86000 : 5.61000 : 0 : 888
175.06000 : 5.61000 : 175.06000 : 5.61000 : 0 : 889
178.05000 : 5.61000 : 178.05000 : 5.61000 : 0 : 890
176.03000 : 5.62000 : 176.03000 : 5.62000 : 0 : 891
175.67000 : 5.63000 : 175.67000 : 5.63000 : 0 : 892
175.87000 : 5.64000 : 175.87000 : 5.64000 : 0 : 893
174.45000 : 5.66000 : 174.45000 : 5.66000 : 0 : 894
174.48000 : 5.66000 : 174.48000 : 5.66000 : 0 : 895
174.50000 : 5.66000 : 174.50000 : 5.66000 : 0 : 896
173.35000 : 5.67000 : 173.35000 : 5.67000 : 0 : 897
173.93000 : 5.69000 : 173.93000 : 5.69000 : 0 : 898
174.99000 : 5.70000 : 174.99000 : 5.70000 : 0 : 899
177.56000 : 5.70000 : 177.56000 : 5.70000 : 0 : 900
174.45000 : 5.71000 : 174.45000 : 5.71000 : 0 : 901
176.48000 : 5.71000 : 176.48000 : 5.71000 : 0 : 902
178.12000 : 5.71000 : 178.12000 : 5.71000 : 0 : 903
175.25000 : 5.73000 : 175.25000 : 5.73000 : 0 : 904
177.17000 : 5.73000 : 177.17000 : 5.73000 : 0 : 905
175.25000 : 5.74000 : 175.25000 : 5.74000 : 0 : 906
177.73000 : 5.74000 : 177.73000 : 5.74000 : 0 : 907
173.38000 : 5.75000 : 173.38000 : 5.75000 : 0 : 908
174.62000 : 5.75000 : 174.62000 : 5.75000 : 0 : 909
178.15000 : 5.76000 : 178.15000 : 5.76000 : 0 : 910
174.03000 : 5.77000 : 174.03000 : 5.77000 : 0 : 911
174.09000 : 5.77000 : 174.09000 : 5.77000 : 0 : 912
175.04000 : 5.77000 : 175.04000 : 5.77000 : 0 : 913
174.50000 : 5.79000 : 174.50000 : 5.79000 : 0 : 914
175.71000 : 5.79000 : 175.71000 : 5.79000 : 0 : 915
173.46000 : 5.80000 : 173.46000 : 5.80000 : 0 : 916
174.47000 : 5.80000 : 174.47000 : 5.80000 : 0 : 917
177.17000 : 5.80000 : 177.17000 : 5.80000 : 0 : 918
178.03000 : 5.80000 : 178.03000 : 5.80000 : 0 : 919
178.18000 : 5.80000 : 178.18000 : 5.80000 : 0 : 920
173.26000 : 5.81000 : 173.26000 : 5.81000 : 0 : 921
175.99000 : 5.81000 : 175.99000 : 5.81000 : 0 : 922
173.87000 : 5.82000 : 173.87000 : 5.82000 : 0 : 923
174.08000 : 5.82000 : 174.08000 : 5.82000 : 0 : 924
177.44000 : 5.83000 : 177.44000 : 5.83000 : 0 : 925
173.88000 : 5.84000 : 173.88000 : 5.84000 : 0 : 926
176.01000 : 5.86000 : 176.01000 : 5.86000 : 0 : 927
176.34000 : 5.87000 : 176.34000 : 5.87000 : 0 : 928
173.50000 : 5.88000 : 173.50000 : 5.88000 : 0 : 929
173.80000 : 5.88000 : 173.80000 : 5.88000 : 0 : 930
173.85000 : 5.89000 : 173.85000 : 5.89000 : 0 : 931
176.61000 : 5.89000 : 176.61000 : 5.89000 : 0 : 932

174.03000 : 5.90000 : 174.03000 : 5.90000 : 0 : 933
174.07000 : 5.90000 : 174.07000 : 5.90000 : 0 : 934
177.88000 : 5.90000 : 177.88000 : 5.90000 : 0 : 935
173.66000 : 5.91000 : 173.66000 : 5.91000 : 0 : 936
175.50000 : 5.91000 : 175.50000 : 5.91000 : 0 : 937
176.24000 : 5.91000 : 176.24000 : 5.91000 : 0 : 938
176.70000 : 5.91000 : 176.70000 : 5.91000 : 0 : 939
177.78000 : 5.91000 : 177.78000 : 5.91000 : 0 : 940
173.24000 : 5.93000 : 173.24000 : 5.93000 : 0 : 941
173.88000 : 5.93000 : 173.88000 : 5.93000 : 0 : 942
175.29000 : 5.94000 : 175.29000 : 5.94000 : 0 : 943
175.31000 : 5.94000 : 175.31000 : 5.94000 : 0 : 944
178.14000 : 5.94000 : 178.14000 : 5.94000 : 0 : 945
178.27000 : 5.94000 : 178.27000 : 5.94000 : 0 : 946
173.78000 : 5.96000 : 173.78000 : 5.96000 : 0 : 947
174.12000 : 5.96000 : 174.12000 : 5.96000 : 0 : 948
175.66000 : 5.98000 : 175.66000 : 5.98000 : 0 : 949
174.63000 : 5.99000 : 174.63000 : 5.99000 : 0 : 950
174.08000 : 6.00000 : 174.08000 : 6.00000 : 0 : 951
174.54000 : 6.00000 : 174.54000 : 6.00000 : 0 : 952
174.05000 : 6.01000 : 174.05000 : 6.01000 : 0 : 953
174.12000 : 6.01000 : 174.12000 : 6.01000 : 0 : 954
174.79000 : 6.01000 : 174.79000 : 6.01000 : 0 : 955
176.99000 : 6.01000 : 176.99000 : 6.01000 : 0 : 956
174.94000 : 6.02000 : 174.94000 : 6.02000 : 0 : 957
175.49000 : 6.02000 : 175.49000 : 6.02000 : 0 : 958
176.37000 : 6.02000 : 176.37000 : 6.02000 : 0 : 959
175.52000 : 6.03000 : 175.52000 : 6.03000 : 0 : 960
177.16000 : 6.04000 : 177.16000 : 6.04000 : 0 : 961
178.28000 : 6.04000 : 178.28000 : 6.04000 : 0 : 962
173.97000 : 6.06000 : 173.97000 : 6.06000 : 0 : 963
176.81000 : 6.07000 : 176.81000 : 6.07000 : 0 : 964
176.36000 : 6.08000 : 176.36000 : 6.08000 : 0 : 965
176.47000 : 6.09000 : 176.47000 : 6.09000 : 0 : 966
176.98000 : 6.09000 : 176.98000 : 6.09000 : 0 : 967
173.17000 : 6.11000 : 173.17000 : 6.11000 : 0 : 968
175.83000 : 6.11000 : 175.83000 : 6.11000 : 0 : 969
177.36000 : 6.12000 : 177.36000 : 6.12000 : 0 : 970
173.55000 : 6.14000 : 173.55000 : 6.14000 : 0 : 971
176.53000 : 6.14000 : 176.53000 : 6.14000 : 0 : 972
174.17000 : 6.15000 : 174.17000 : 6.15000 : 0 : 973
177.73000 : 6.15000 : 177.73000 : 6.15000 : 0 : 974
173.27000 : 6.16000 : 173.27000 : 6.16000 : 0 : 975
175.49000 : 6.16000 : 175.49000 : 6.16000 : 0 : 976
173.19000 : 6.17000 : 173.19000 : 6.17000 : 0 : 977

```
173.78000 : 6.18000 : 173.78000 : 6.18000 : 0 : 978
175.04000 : 6.18000 : 175.04000 : 6.18000 : 0 : 979
177.07000 : 6.18000 : 177.07000 : 6.18000 : 0 : 980
173.72000 : 6.19000 : 173.72000 : 6.19000 : 0 : 981
173.17000 : 6.20000 : 173.17000 : 6.20000 : 0 : 982
174.05000 : 6.20000 : 174.05000 : 6.20000 : 0 : 983
177.30000 : 6.20000 : 177.30000 : 6.20000 : 0 : 984
174.67000 : 6.21000 : 174.67000 : 6.21000 : 0 : 985
175.07000 : 6.21000 : 175.07000 : 6.21000 : 0 : 986
175.70000 : 6.21000 : 175.70000 : 6.21000 : 0 : 987
176.64000 : 6.21000 : 176.64000 : 6.21000 : 0 : 988
173.25000 : 6.23000 : 173.25000 : 6.23000 : 0 : 989
173.70000 : 6.23000 : 173.70000 : 6.23000 : 0 : 990
174.48000 : 6.23000 : 174.48000 : 6.23000 : 0 : 991
175.04000 : 6.25000 : 175.04000 : 6.25000 : 0 : 992
175.89000 : 6.25000 : 175.89000 : 6.25000 : 0 : 993
177.16000 : 6.25000 : 177.16000 : 6.25000 : 0 : 994
174.49000 : 6.26000 : 174.49000 : 6.26000 : 0 : 995
175.51000 : 6.26000 : 175.51000 : 6.26000 : 0 : 996
174.99000 : 6.27000 : 174.99000 : 6.27000 : 0 : 997
177.51000 : 6.27000 : 177.51000 : 6.27000 : 0 : 998
173.43000 : 6.28000 : 173.43000 : 6.28000 : 0 : 999
173.47000 : 6.30000 : 173.47000 : 6.30000 : 0 : 1000
```

B.2 10 User Locations along their Trajectory

```
175.31742 3.48012
175.33814 3.46961
175.35851 3.45078
175.36101 3.44832
175.38889 3.42815
175.39926 3.41810
175.41133 3.40457
175.42336 3.39956
175.42591 3.39762
175.44345 3.39085
```