# Distance Indexing on Road Networks

Haibo Hu Dik Lun Lee
Dept. of Computer Science and Engineering
Hong Kong Univ. of Science and Technology
Clear Water Bay, Hong Kong SAR, China

{haibo,dlee}@cse.ust.hk

Victor C. S. Lee
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong SAR, China

csvlee@cityu.edu.hk

## ABSTRACT

The processing of kNN and continuous kNN queries on spatial network databases (SNDB) has been intensively studied recently. However, there is a lack of systematic study on the computation of network distances, which is the most fundamental difference between a road network and a Euclidean space. Since the online Dijkstra's algorithm has been shown to be efficient only for short distances, we propose an efficient index, called *distance signature*, for distance computation and query processing over long distances. Distance signature discretizes the distances between objects and network nodes into categories and then encodes these categories. To minimize the storage and search costs, we present the optimal category partition, and the encoding and compression algorithms for the signatures, based on a simplified network topology. By mathematical analysis and experimental study, we showed that the signature index is efficient and robust for various data distributions, query workloads, parameter settings and network updates.

## 1. INTRODUCTION

Spatial database has been intensively studied in the past decade, spanning various fields such as indexing, query optimization, and approximation. Recently, the research focus has been extended to spatial network databases (SNDB) where objects are restricted to move on predefined roads [11, 6, 10, 8]. In SNDB, roads are usually modeled as a simple undirected graph $G$ ($< V, E >$), where a vertex (*node*) of $G$ denotes a road junction, an edge denotes a road segment, and the edge weight denotes the distance along the road. The dataset in an SNDB is a set of *objects* (e.g., hospitals, restaurants) distributed on the road network. Although in reality the objects may lie on the edges (i.e, the roads) or on the nodes (i.e., the road junctions), we consider in this paper only the case where objects are distributed on the nodes. This is because the distance to a point on a road segment is simply the distance to one of the nodes adjacent to the segment plus the road distance from the node to the point. As such, the two cases make little difference for distance measurement.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
*VLDB '06,* September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

Although query processing (in particular the kNN and continuous kNN queries) on SNDB has been intensively investigated [11, 6, 10, 8, 5], no existing work has systematically studied the problem of network distance computation, which is the most fundamental problem that differentiates a road network from a Euclidean space. Whereas the distance in a Euclidean space depends solely on the relative positions of the two points, the distance on the road network depends not only on the relative positions but on the road segments between the two points. When the exact network distance is needed, many works rely on the Dijkstra's algorithm [3], which has been shown to be efficient only for short distances. Although specially-designed precomputed data structures for speeding up query processing have been proposed (e.g., the Network Voronoi Diagram ($NVD$) [8] and the precomputed NN list of condensed nodes [1] for kNN queries), they do not support distance computation or query types other than what they are built for. As an extreme example, since the NN list does not store the path to the NN objects, it does not even support kNN queries with path information returned. Furthermore, since the cost of the Dijkstra's algorithm depends on the distance, not on the number of input objects whose distances are to be computed, object pruning during query processing is not helpful at all if it fails to reduce the distance to be computed by the Dijkstra's algorithm in the refinement step. For example, suppose that a range query needs to return all objects within 1 kilometer and 10 objects are about this distance. Even if some specialized index manages to prune (confirm) 9 of them as non-results (results), to compute the exact distance for the last object in the refinement step costs as much as it does to compute the exact distances of all the 10 objects.

To remove the limitation on the application of these data structures to only specific query types, this paper aims to develop a general-purpose index to support a broader set of queries, which may be considered a counterpart of R-tree in SNDB. More specifically, we expect the index to have the following advantages: (1) it supports efficient distance computation between nodes and objects; (2) it accelerates the processing of common types of queries; (3) it incurs reasonable storage overhead; (4) the index construction and maintenance are efficient; and (5) it works with other query optimization techniques. It is noteworthy that this general-purpose index is targeted at non-dense datasets. As for

dense datasets, since most spatial queries are interested in local areas, the Dijkstra's algorithm is efficient enough for distance computation and the queries.

Based on these criteria, we have devised the *distance signature*, which is the road network equivalence of a coordinate in the Euclidean space. On each node, the signature stores the distance information of all objects. More specifically, the distance spectrum is partitioned into a sequence of uneven categories with distant categories spanning wider ranges. For example, the ranges of the categories may increase exponentially, i.e., a category spans a range that is some $c$ times wider than the range of the category that is prior to it in the sequence. Thus, the distance information is a categorical value. In this way, the signature stores much coarser distance information for remote objects than close objects, so that the processing of spatial queries can be accelerated because most of them are interested in local areas. With additional backtracking links, the signature can support both exact and approximate distance computation at low cost. As for query processing, the signature is efficient in pruning objects during the search or retrieval of the initial results. Furthermore, it is also useful in the refinement step where exact distance retrieval or comparison must be performed. To address the storage and construction costs, which may appear to be high at first glance, we propose optimal algorithms for both category encoding and signature compression so that the cost of signature index is no more than the existing specialized index structures. As for the update cost, since distant objects are coarsely represented, changes on local edges or nodes have little impacts on them. In other words, the impact of updates on the index is limited to a small scope. As for index transparency, we designed the index schema and separated it from the adjacency list of the road network. This way, the index not only is transparent to other query optimization techniques but can work together with them to further boost the search performance.

The rest of the paper is organized as follows. Section 2 reviews existing work on road networks, especially in the field of query processing and indexing. Section 3 introduces the signature index, storage schema, and the basic operations such as distance retrieval and comparison. Section 4 presents and generalizes the query processing algorithms for common types of spatial queries. Section 5 proposes the encoding, compression, and update algorithms, and in particular, it shows the optimal signature for a simplified grid topology. The experimental results are shown and analyzed in Section 6, followed by the conclusion.

## 2. RELATED WORK

Processing spatial queries on road network is an emerging research topic in recent years [11, 6, 10, 8, 7, 1, 5, 9]. The modeling of a spatial road network is the fundamental problem in SNDB. Besides the conventional approach which models the network as a directed or undirected weighted graph, Papadias et al. incorporated the Euclidean space into the road network and applied traditional spatial access methods to speed up query processing [10]. Assuming that Euclidean distance is the lower bound of network distance, they proposed *incremental Euclidean restriction* (IER) to first process a query in the Euclidean space and obtain the results as candidates, and then compute network distances of these candidates for the actual results. However, IER cannot be applied to road networks where the lower bound assumption does not hold, e.g., the network whose edge weights are the time cost for transportation. So they proposed an alternative approach *incremental network expansion* (INE), which essentially expands the network from the query point.

The network expansion is a common search paradigm, which gradually expands the search from the query point through the edges and reports the accessed object during the expansion. In order to avoid re-expanding the same node, this approach usually employs a single-source shortest path (SSSP) algorithm. The most well known SSSP algorithm is the Dijkstra's algorithm [3]. In the Dijkstra's algorithm, a priority queue stores the current shortest paths (SPs) of all the nodes whose SPs are yet to be finalized. The algorithm repeatedly chooses the node with the shortest SP in the queue, finalizes it, and updates any SP in the queue that is affected by this SP. Obviously, if the network expansion always chooses the same node as the the Dijkstra's algorithm does to expand, no re-expansion will occur. Besides the Dijkstra algorithm, $A^*$ algorithm with various expansion heuristics [4] was also employed to choose the next appropriate node to expand. The network expansion paradigm has been employed by many research projects for query processing on SNDB. For example, Jensen et al. proposed a general spatio-temporal framework for NN queries on road networks with both graph representation and detailed search algorithms [6]. To compute network distances, they adapted the Dijkstra's algorithm for online evaluation of the shortest path.

Another commonly-used search paradigm is *solution-indexing*, which precomputes and stores the solutions to the queries. Kolahdouzan et al. proposed a solution-based approach for kNN queries in SNDB. As they were inspired by the Voronoi Diagram in vector spaces, they called it *Voronoi Network Nearest Neighbor* ($VN^3$) [8]. The Network Voronoi Diagram (NVD) is computed and each Voronoi cell is approximated by a polygon called *network Voronoi polygon* (NVP). By indexing all NVP's with an R-tree, searching for the first nearest neighbor is reduced to a point location problem. To answer kNN queries, they proved that the $k$th NN must be adjacent to some $i$th ($i < k$) NN in NVD, which limits the search area. For distance computation, the distances between border nodes of adjacent NVP's, and even the distances between border nodes and inner nodes in each NVP, are computed and stored. Using these indexes and distances, they showed that $VN^3$ outperforms INE during the search by up to an order of magnitude. However, the performance of $VN^3$ depends on the density and distribution of the dataset: as the dataset gets sparser, the size of each NVP's gets larger, which dramatically increases the size of border-to-border and border-to-inner distances. As such, sparse datasets cause high precomputation overhead and poor search performance. Given that kNN search by network expansion on dense datasets is efficient, $VN^3$ is only suitable for a small range of datasets.

There are other kNN search algorithms that aim to transform a road network into simpler forms. Shahabi et al. applied graph embedding techniques and turned a road network to a high-dimensional Euclidean space so that tradi-

tional kNN search algorithms can be applied [11]. They showed that KNN in the embedding space is a good approximation of the KNN in the road network. However, this technique involves high-dimensional (40-256) spatial indexes. Furthermore, the query result is approximate and the precision depends on the data density and distribution. We also proposed a network reduction approach that reduces a road network to a set of interconnected trees (or more exactly, trees with edges between siblings) [5]. An *nd* index is built on each tree to speed up its own kNN search. This approach was shown to outperform $VN^3$ for medium and dense datasets.

Continuous nearest neighbor (CNN) query was also studied recently. It returns both the kNNs and the valid scopes of the results along a path. In other words, CNN query requires the determination of the positions where the kNNs change. A naive solution is to evaluate a kNN query on each node of the path. Kolahdouzan and Shahabi proposed the Upper Bound Algorithm (UBA) to reduce the number of kNN evaluations by allowing a kNN result to be valid for a distance range [7]. Cho and Chung further proposed a unique continuous search algorithm (UNICONS) to improve the search performance [1]. UNICONS first divides the path into sub-paths by the intersection nodes. For each sub-path, it evaluates two kNN queries at the starting and ending nodes, respectively. The kNNs for this sub-path are thus the union of two kNN sets and the objects along this sub-path. An online algorithm is then used to find the split points on this sub-path where the kNNs change. To speed up conventional kNN evaluation, they also proposed a solution-based index called *NN lists* which precomputes and stores the kNNs for some condensed nodes, i.e., nodes with large degrees.

# 3. DISTANCE SIGNATURE

Existing solution-based indexes suffer from poor generality and insufficient support for distance computation. In this section, we propose *distance signature* as an efficient alternative for distance computation and query processing on road networks. The basic idea is to maintain the approximate network distances between the nodes and the objects. As in other database approaches, these approximate values can be used to prune unqualified objects during query execution. Furthermore, in order to facilitate the refinement step of query processing, the signatures also provide efficient access to the exact distance values. The rest of this section introduces the distance signature, its storage schema, and basic operations on the signatures.

## 3.1 Distance Signature and Storage Schema

At each node $n$, the distance signature stores the approximate distance information of each data object. The information is in the form of a categorical value based on the exact distance. For example, if the whole distance spectrum is partitioned into four categories: 0–100 meters, 100–400 meters, 400–900 meters, and beyond 900 meters, and object $a$ and $b$, respectively, are 75 and 475 meters away from $n$, then $a$ is assigned to category 0, and $b$ is assigned to category 2. Obviously, the number of categories used to partition the spectrum and the partition method greatly affect the performance of object searching and storage cost. We will derive the optimal category partitioning scheme in Section 5.1.

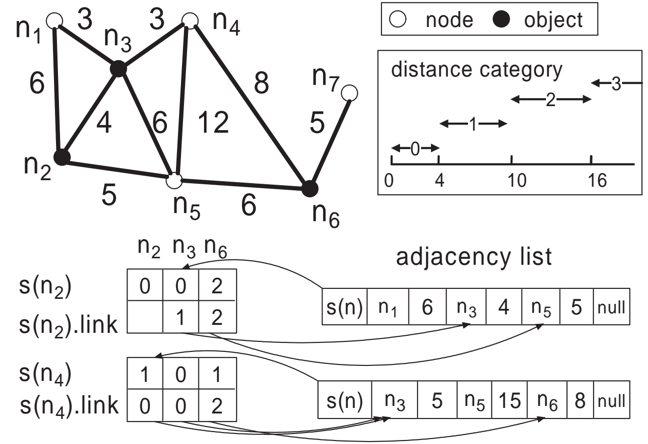



Figure 3.1: An Example of Distance Signature. Distances are partitioned into 4 categories. Each signature is composed of $s[n_2], s[n_3], s[n_6]$, in this order.

The whole set of categorical values for a single node forms a sequence, which is called a *distance signature*, and is denoted by $\mathbf{s(n)}$. Each categorical value of an object is called a *component* of $s(n)$, and is denoted by $s(n)[i]$ (or $s[i]$ if node $n$ is clear in the context). A signature is comparable to a coordinate in the multi-dimensional Euclidean space in that both can be used for locating the nodes' positions and computing distances.

In addition, to provide efficient access to the exact distance from node $n$ to object $i$, the component of $s[i]$ also keeps a link to designate the next node from $n$ along the shortest path to $i$. The link is denoted by the next node's position index in $n$'s adjacency list, and is called the *backtracking link* of $s[i]$ (denoted by $s[i].link$). Figure 3.1 illustrates an example of distance signature on a small road network with 7 nodes and 3 objects.

Let $|s[i]|$ denote the size of the categorical value, $|s[i].link|$ denote the size of the link, and let $D$ denote the cardinality of the dataset. Then the storage requirement for distance signature of each node is $\sum_{i=1}^{D} |s[i]| + |s[i].link|$. At first sight, the storage is no less than that of the existing solution-based indexes. However, it is in general lower because of the following two reasons.

- Since the dataset is not dense, $D$ is moderate. In addition, a fine partition of distance categories does not require a large $|s[i]|$ — 5 bits is enough for 32 categories. Likewise, $|s[i].link|$ is also small, because the degrees of the nodes on a road network are normally small (e.g., a intersection of two roads has a degree of 4).
- Spatial queries, whether in Euclidean spaces or on road networks, are mostly interested in the neighboring areas around the queries. The farther the object is from the query, the less likely that it concerns the query. As such, we can further reduce the size of the signatures by applying variable-length encoding scheme to the categories and compression scheme to the signatures. As shown in Section 5.3, these optimizations

effectively reduce the storage overhead while maintaining the effectiveness of the signatures.

As for storage schema, the distance signature can either be merged with the adjacency list, or stored separately as shown in Figure 3.1. As we will see in Section 3.2, since the signature is usually accessed together with the adjacency list, it is preferable to merge the signature with the adjacency list. However, if the adjacency list alone is accessed more frequently (i.e., the queries are not as many as other road network operations), a separate storage is preferred. In this case, we apply the same greedy approach as in [5] to group the signatures for paging. Moreover, in order for the signature to be randomly accessible, a link physically pointing to the signature is added to the adjacency list (see Figure 3.1).

## 3.2 Basic Operations on Signature

Based on the signature, we define the following operations on the distances: retrieval, comparison, and sorting. Depending on whether only the signature of the node is used, these operations can be either *approximate* or *exact* operations.

### 3.2.1 Distance retrieval

Distance retrieval is to obtain the distance from a node $n$ to an object $a$, denoted by $d(n, a)$. Thanks to the distance signature, the exact value of $d(n, a)$ can be gradually approached and finally retrieved by recursively following the backtracking link ($s[a].link$) until it reaches $a$. Approximate distance retrieval, on the other hand, returns the distance in the form of a range. Denoted by $\tilde{d}(n, a, \Delta)$ (where $\Delta$ is an input distance range), this operation returns a distance range which comprises $d(n, a)$ and does not partially intersect with $\Delta$ (however, it may be fully contained in $\Delta$). The algorithm for approximate retrieval is the same as exact retrieval, except that it terminates once the distance range does not partially intersect with $\Delta$. The approximate distance retrieval is useful when we need to get an unambiguous comparison result on two distances (see Section 3.2.2). Algorithm 1 lists the pseudo-code of this operation.

**Algorithm 1** Exact and Approximate Distance Retrieval

**Input:** node $n$, object $a$, and approximate distance $\Delta$ (optional)
**Output:** result $c$ as $d(n, a)$ or $\tilde{d}(n, a, \Delta)$
**Procedure:**
1: set $c$ to node $n$'s signature component of object $a$, $s(n)[a]$
2: pointer $p = s(n)[a].link$
3: **while** $p$ has not reached $a$ **do**
4: $c$ = distance category of $s(p)[a]$ + d(n,p)
5: **if** $\Delta$ exists and $c$ does not partially intersect $\Delta$ **then**
6: return $c$
7: $p = s(p)[a].link$

### 3.2.2 Distance comparison

Distance comparison is to compare the distances from a node $n$ to two objects $a$ and $b$. This is the atomic operation for distance sorting (see Section 3.2.3) and kNN search (see Section 4.2).

The exact comparison is based on distance retrieval in Section 3.2.1. Let $\tilde{d}(a)$ and $\tilde{d}(b)$ denote the current approximate distances and initially let $\tilde{d}(a) = s[a]$ and $\tilde{d}(b) = s[b]$. Then the comparison algorithm recursively retrieves a finer approximate distance for $\tilde{d}(a)$ or $\tilde{d}(b)$ until there is no ambiguity between them. More specifically, it retrieves $\tilde{d}(a) = \tilde{d}(n, a, \tilde{d}(b))$, then $\tilde{d}(b) = \tilde{d}(n, b, \tilde{d}(a))$, and then all over again. In essence, this algorithm backtracks the shortest path from $n$ to $a$ and $b$ alternately. However, it does not switch $a, b$ at each step of the backtracking, because both the signatures and the adjacency lists are stored in pages, and it is I/O efficient to backtrack $a$ or $b$ in a batch. Algorithm 2 shows the pseudo-code for this operation.

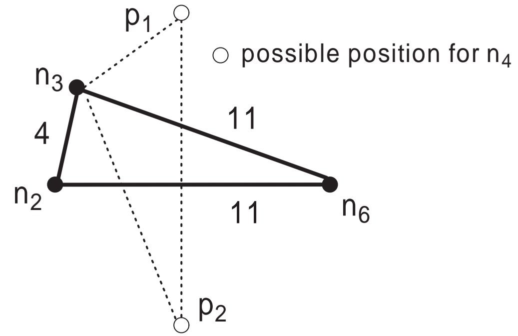



Figure 3.2: An Example of Approximate Distance Compare. $n_4$ is the node, $n_2, n_6$ are the objects, and $n_3$ is the observer.

**Algorithm 2** Exact Distance Comparison

**Input:** node $n$, object $a$, and $b$
**Output:** comparison result ">", "<", or "="
**Procedure:**
1: **if** $s[a] <> s[b]$ **then**
2: compare $s[a]$ and $s[b]$ and return
3: set $\tilde{d}(a) = s[a], \tilde{d}(b) = s[b]$
4: **while** $\tilde{d}(a)$ and $\tilde{d}(b)$ are ambiguous **do**
5: let $\tilde{d}(a) = \tilde{d}(n, a, \tilde{d}(b))$ or alternately
6: compare $\tilde{d}(a)$ and $\tilde{d}(b)$ and return

In order to reduce accesses to adjacency lists and distance signatures, we also devise an approximate comparison algorithm, which is based on signature $s(n)$ only. Since only an approximate result is needed and only $s(n)$ is available, we embed the nodes $a$, $b$, and $n$ in a 2D Euclidean space. Figure 3.2 shows an example of approximate comparison for the road network in Figure 3.1, where $d(n_4, n_2)$ is compared with $d(n_4, n_6)$. Our idea behind the approximate comparison is to let another object $c$ ($n_3$ in this example), called the *observer*, "search for" node $n$ ($n_4$ in this example) in the embedded space. More specifically, the observer makes a decision on which side of the perpendicular bisector of $n_2n_6$ ($p_1p_2$ in this example) $n$ is located at.

In order to make a quick decision, the approximate comparison is based on a simple heuristic, i.e., "if $n_4$ is located exactly on the perpendicular bisector (and thus $d(n_4, n_2) = d(n_4, n_6)$), is it still possible for $n_3$ to find $n_4$ within the distance range $s(n_4)[n_3]$?" More specifically, if all *possible locations* for $n_4$ on $p_1p_2$ make $d(n_4, n_3)$ smaller than the lower bound of $s(n_4)[n_3]$, then according to Figure 3.2, $n_4$ can never be located on $p_1p_2$ and the search should move to-

wards $n_2$, i.e., $d(n_4, n_2) < d(n_4, n_6)$; likewise, if all possible locations on $p_1p_2$ for $n_4$ makes $d(n_4, n_3)$ greater than the upper bound of $s(n_4)[n_3]$, then the search for $n_4$ should move towards $n_6$, i.e., $d(n_4, n_2) > d(n_4, n_6)$; otherwise, $n_3$ cannot make a decision. The possible locations of $n_4$ on $p_1p_2$ are the segment(s) where distance ranges $s(n_4)[n_2]$ and $s(n_4)[n_6]$ still hold in the embedded space; since $s(n_4)[n_2] = s(n_4)[n_6]$, the possible locations are two symmetric line segments mirrored by $n_2n_6$. In Figure 3.2, since two segments share a single end, they are merged as $p_1p_2$. Since $d(n_4, n_3)$ increases/decreases monotonously on the line segments, to check all possible locations on $p_1p_2$ is equivalent to checking the end points only, and there are four end points at most.

A single observer may fail to make a decision. As such, the algorithm chooses several observers and each of them votes for the final decision. To choose the observers, we select those objects that are closer to $n$ than $a$ and $b$ in the signature. This is based on the fact that a closer object has a more accurate distance range and less distortion during the embedding. The final comparison result is then the simple majority of the votes. Algorithm 3 lists the pseudo-code of this algorithm.

**Algorithm 3** Approximate Distance Comparison

**Input:** node $n$, object $a$, and $b$
**Output:** comparison result ">", "<", or "="
**Procedure:**
1: **if** $s[a] <> s[b]$ **then**
2: compare $s[a]$ and $s[b]$ and return
3: **for** each object $i$ that $s[i] < s[a]$ **do**
4: vote for $a$ or $b$
5: count votes and return

Approximate distance comparison can be used for getting the initial result of distance sorting. It is also noteworthy that the algorithm requires the distances between two objects during the embedding. However, this additional storage is not costly, since the cardinality of the dataset is not large and those distances that fall in the last distance category do not need to be stored (since these objects are never used as the observer for one another). In addition, to eliminate the I/O cost for these frequently accessed distances, they are stored in memory as a table.

### 3.2.3 Distance Sorting

Distance sorting is to impose an ordering on a set of objects $O = \{o_1, o_2, \cdots, o_m\}$ based on their distances to a node $n$. It is the basic operation for kNN search.

Distance sorting consists of two steps, initial sorting and refinement. The initial sorting quickly obtains an approximate order based on the approximate distance comparison in Section 3.2.2. To apply the approximate comparison, we can use any existing comparison-based sorting algorithm such as *fast sort*. The refinement step confirms the initial order by exactly comparing any two consecutive distances, starting from the beginning of the order. If by comparison, $d(n, o_i) > d(n, o_{i+1})$, i.e., the exact comparison result contradicts the initial approximate result, $o_i$ and $o_{i+1}$ will be switched. Like the *bubble sort* algorithm, the newly switched upfront object (i.e., $o_{i+1}$) must compare with the object immediately in front of it (i.e., $o_{i-1}$) to see if the switch should be further propagated upfront. Algorithm 4 lists the pseudo-code for the sorting algorithm.

**Algorithm 4** Distance Sorting

**Input:** node $n$, object set $O = \{o_1, o_2, \cdots, o_m\}$
**Operator:** $sort(n, O)$
**Output:** the sorted object set $O$
**Procedure:**
1: fast sort $O$ by approximate distance comparison
2: **for** each i from 1 to $m - 1$ **do**
3: **if** $d(n, o_i) > d(n, o_{i+1})$ **then**
4: switch $o_i$ and $o_{i+1}$
5: $i = i - 1$

# 4. QUERY PROCESSING ON DISTANCE SIGNATURES

Distance signature is superior to the existing indexes in terms of the diversity of the kinds of queries supported. Since it indexes the underlying distances, rather than the solution for a particular type of queries, it can be applied to virtually any queries relating to distances. In this section, we present the algorithms to process common spatial queries based on the distance signatures. We discuss range and kNN queries, and generalize the processing paradigm to other query types such as aggregation queries and network joins.

## 4.1 Range Query Processing

To process a range query on node $n$ with distance threshold $\epsilon$, the signature of $n$ is first accessed. For each object $o$ in the signature, if the upper bound of distance category $s(n)[o]$ (denoted by $s(n)[o].ub$) is smaller than $\epsilon$, the object clearly belongs to the result. Likewise, if the lower bound of $s(n)[o]$ (denoted by $s(n)[o].lb$) exceeds $\epsilon$, the object clearly does not belong to the result. However, if $s(n)[o]$ covers $\epsilon$, a more accurate approximate distance is needed. As such, the approximate distance retrieval is invoked with parameter $\Delta$ set to $[\epsilon, \epsilon]$. Algorithm 5 shows the pseudo-code of this procedure.

**Algorithm 5** Range Query Processing Algorithm

**Input:** query node $n$ and distance threshold $\epsilon$
**Output:** the result set $C$
**Procedure:**
1: **for** each object $o$ **do**
2: **if** $\epsilon > s(n)[o].ub$ **then**
3: insert $o$ into $C$
4: **else if** $\epsilon < s(n)[o].lb$ **then**
5: continue;
6: **else**
7: $\tilde{d} = \tilde{d}(n, o, [\epsilon, \epsilon])$
8: **if** $\epsilon > \tilde{d}.ub$ **then**
9: insert $o$ into $C$

The range query processing on distance signatures is more efficient than the network expansion method since: (1) the expansion is unguided, whereas the backtracking in approximate distance retrieval is guided; (2) the search terminates as soon as there is no ambiguity on the results.

## 4.2 K Nearest Neighbor Query

In this paper, we differentiate three types of kNN queries with regard to whether the distance information of the results needs to be returned.

- **Type 1**: the exact distance of every kNN to the query node $n$ must be returned.
- **Type 2**: the order of the distances of kNN objects must be reserved.
- **Type 3**: no distance or ordering information needs to be returned.

Our general kNN algorithm first solves a kNN query as a type 3 query, and then refines the results for type 2 and type 1. At first, the algorithm reads the signature of node $n$, which gives a rough kNN ordering. Let $B_i$ be the set of objects in category $i$, and all objects in $B_1$, $B_2$, $\cdots$, and $B_{m-1}$ can be confirmed as results, where $\sum_{i=1}^{m-1} |B_i| \leq k < \sum_{i=1}^{m} |B_i|$. Then the algorithm sorts the objects in $B_m$ and chooses the top $k - \sum_{i=1}^{m-1} |B_i|$ objects as results. Now that the query is completed as a type 3 query, if the query is type 2, the algorithm continues to sort the objects in each category $B_i$ $(1 \leq i < m)$. If the query is type 1, the algorithm first retrieves the exact distances of the results and then sorts them. Algorithm 6 lists the pseudo-code of kNN algorithm.

**Algorithm 6** kNN Query Processing Algorithm

**Input:** query node $n$ and $k$
**Output:** the result set $C$
**Procedure:**

```
1: divide objects into B_i by s(n)
2: m = min_j Σ_{i=0}^{j} |B_i| > k
3: sort(n, B_m)
4: insert the first k − Σ_{i=1}^{m−1} |B_i| objects in B_m into C
5: discard the rest objects in B_m and all B_i (i > m)
6: if type 2 then
7:    for each 1 ≤ i < m do
8:       sort(n, B_i)
9: if type 1 then
10:   for each 1 ≤ i < m do
11:      for each o ∈ B_i do
12:         get d(n, o)
13:      sort B_i based on d(n, o)
14: C = ∪_{i=1}^{m} B_i
```

## 4.3 Generalization to Other Queries

As shown in the algorithms above, the advantage of distance signature is that the search algorithm can control the accuracy of distance retrieval as it needs. As such, the paradigm to process a general query on road network is: (1) to read the signature and find all results and candidates; and (2) for each candidate, to gradually retrieve a more accurate distance until the candidate is confirmed to be or not to be a result. This paradigm can be directly applied to queries such as aggregation queries which return the aggregate values, instead of individual objects for range queries. The same paradigm can also be extended to network joins which return pairs of objects from two datasets that satisfy certain spatial relations at a given node. For example, $\epsilon$-join returns pairs of objects whose network distances are within $\epsilon$. This query can be processed by joining the two signatures of the two datasets and gradually retrieving more accurate distances for candidate pairs until they are confirmed as results or non-results.

# 5. SIGNATURE CONSTRUCTION AND MAINTENANCE

In this section, we propose the construction and maintenance algorithms for distance signatures. More specifically, we discuss: (1) how the distance spectrum is partitioned into categories, (2) how the categories are encoded, (3) how the signatures are compressed, and (4) how they are updated.

## 5.1 Distance Spectrum Partition

A good partition of distance spectrum must consider the following factors:

- **Dataset distribution**. The distribution, especially the density of the dataset, determines the object distribution in the distance spectrum. Obviously, a dense dataset requires more categories than a sparse dataset does.
- **Query load**. For example, the distance threshold $\epsilon$ of a range query and the $k$ of a kNN query affect how precisely the distance spectrum should be partitioned. In order to quantify the query load, we define "spreading" (denoted by $sp$) as the distance threshold of those objects that are interesting to the query. For range queries, $sp = \epsilon$, and for type 3 kNN queries, $sp$ is the distance of the $k{+}1^{th}$ nearest neighbor. Obviously, the distribution of $sp$ should affect the partition of distance spectrum so that the signatures can achieve maximum performance.
- **Storage availability**. Accurate partition requires more storage to encode the categories than coarse partition. As such, the availability of disk storage is also a concern.

In what follows, we derive the optimal categories analytically under some simplifications:

- The road network is a uniform grid. More specifically, each node connects to 4 nodes and all edge weights are 1. As for the dataset, the objects are uniformly distributed with density $p$.
- The spreadings ($sp$) of the queries are uniformly distributed over distance range $[0, SP]$.
- Disk storage is unlimited.

Since most queries are interested in local areas only, we propose to partition the distance spectrum exponentially, i.e., at distance $T$, $cT$, $c^2T$, $\cdots$, where $c, T$ are both constants. We will show later that exponential partitioning has some additional benefits in category encoding and compression. Nonetheless, we are yet to determine the exponent $c$ and the distance $T$ of the first partition. The objective is to reduce $Cost$ (in terms of number of bits), i.e., the average I/O accesses to the signatures during query processing[1].

Let $cost(i)$ denote the I/O accesses for queries whose $sp = i$. Then,

$$Cost = (SP)^{-1} \sum_{i=0}^{SP} cost(i) \quad (1)$$

[1]Since we assume that the road network is a uniform grid, the size of the adjacency list is far smaller than the signature and hence it is omitted.

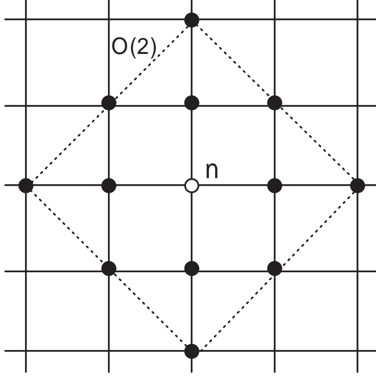



Figure 5.3: $O_i$ $(i = 2)$ in Uniform Grid. It comprises both $n$ and all solid dots.

Let $B_i$ denote the category that $i$ belongs to, and $ub, lb$ denote the upper and lower bound of the distance category. Then, $B_i.ub = c^{\lceil log_c iT^{-1} \rceil}T$, $B_i.lb = c^{\lfloor log_c iT^{-1} \rfloor}T$. According to the query processing algorithm, objects in $B_i$ are the objects and the only objects whose distances need to be compared with $i$. More specifically, for each object in $B_i$ whose actual distance to $n$ is $j$, we need to visit $j - B_i.ub$ number of nodes for their signatures, whose sizes are $|D| loglog_c SP \cdot T^{-1}$ (excluding the bits for backtracking links). As such,

$$cost(i) = |D| loglog_c SP \cdot T^{-1} \cdot \sum_{j=B_i.lb+1}^{B_i.ub} (j - B_i.lb)(O(j) - O(j-1)), \quad (2)$$

where $O(i)$ denote the number of objects within $i$ distance away from $n$. As can be observed from Equation 2, for any $i_1, i_2$, if $B_{i_1} = B_{i_2}$, then $cost(i_1) = cost(i_2)$. As such, we can rewrite Equation 1 into

$$Cost = \sum_{k=0}^{log_c SP \cdot T^{-1}} |D| c^k T(c-1) loglog_c SP \cdot T^{-1} \cdot \sum_{j=c^{k-1}T}^{c^k T} (j - c^{k-1}T)(O(j) - O(j-1)) \quad (3)$$

To solve Equation 3, we need to obtain $O(i)$. As the objects are uniformly distributed with probability $p$, the problem is reduced to the number of nodes within radius $i$, which is $2i^2 + i$ from Figure 5.3. Replacing $O(i)$ with $p(2i^2 + i)$, we rewrite Equation 3 as

$$Cost \approx c^{4 log_c SP \cdot T^{-1}} cpT^5 loglog_c SP \cdot T^{-1} = KcT loglog_c SP \cdot T^{-1}, \quad (4)$$

where $K$ is a constant. To minimize $Cost$, we get the partial derivative of $c$ and $T$, and let them be zero. As such, we obtain the optimal $c = e$ (the Euler number) and $T = \sqrt{\frac{SP}{e}}$. An interesting observation from the result is that, the optimal $c$ and $T$ are independent of $p$, the density of the dataset ($c$ is even a constant). Although the result is derived under the grid and uniform distribution assumptions, these optimal values can serve as guiding values for general road networks.

## 5.2 Signature Construction and Encoding

To construct the signature for a node $n$, the distance from $n$ to any object must be obtained. However, instead of building the shortest path spanning tree from $n$, which additionally computes the distance from $n$ to any node, we build the shortest path spanning tree for every object $o$ by the Dijkstra's algorithm, so that all the distances computed are necessary for the signatures.

The algorithm is initialized by allocating $(logM + logR) \cdot |D|$ bits for each node's signature, where $M$ is the number of categories and $R$ is the maximum degree of a node. When the spanning tree of $o$ extends to node $n$ by the Dijkstra's algorithm, $d(n, o)$ is computed and categorized to fill $s(n)[o]$ in the signature.

The original signature is quite large, since it uses fixed-length encoding on the category id. However, as is observed from Section 5.1, the number of objects in each category vary greatly: according to the grid and uniform distribution assumption, at each distance $i$, there are $(4i-1)p$ number of objects. With exponential partition, far more objects are in the latter categories which have larger distance ranges. As such, we devise a variable-length encoding scheme, called *reverse zero padding*, for the categories. The scheme is based on Huffman coding[2], where the last category is encoded as bit "1", and the second last category is encoded as "01", and in general category $B_i$ is encoded by padding a "0" on category $B_{i+1}$. The following theorem proves that, if $c > \frac{3}{2}$, this scheme is optimal in terms of the average code length.

THEOREM 5.1. *Under exponential partition ($c > 3/2$) and uniform dataset distribution assumptions, reverse zero encoding has the minimal average code length.*

PROOF. From the scheme, reverse zero padding follows Huffman coding's paradigm and recursively merges the first two categories. Since Huffman coding has been proven to be optimal in terms of the average code length, we only need to prove that all merges satisfy the criterion in Huffman coding, that is, the two merged categories have the lowest access probabilities (i.e., the fewest objects). This is equivalent to proving that the number of objects in a category is larger than the sum of all categories prior to it. In other words, for any category $B_k$, $|B_k| > \sum_{j=0}^{k-1} |B_j|$, or equivalently, $O(B_k.ub) > 2O(B_k.lb)$. Replace $|B_k|$ with $c^k T$ and $O(i)$ with $p(2i^2 + i)$, the inequation is

$$2c^{2k-2}T^2(c^2 - 2) > c^{k-1}T(2 - c) \Rightarrow 2c^{k-1}T(c^2 - 2) > 2 - c \quad (5)$$

As $c > 1$, the left hand side of the inequation increases monotonously with $k$. Therefore, we only need to assure that the inequation holds for $k = 1$. And since $T \geq 1$, the inequation is reduced to $2(c^2 - 2) > 2 - c$. Solve this inequation and we get $c > \frac{3}{2}$. □

We now estimate the average code length of the reverse zero padding scheme. The total code length of all objects is,

$$\sum_{k=0}^{M-1} (O(B_k.ub) - O(B_k.lb))(M - k)$$

$$\approx \sum_{k=0}^{M-1} 2pc^{2k}T(M-k) \approx \frac{2pc^{2M}T^2}{c^2-1} \quad (6)$$

Therefore, the average code length is,

$$\frac{2pc^{2M}T^2}{(c^2-1)O(B_{M-1}.lb)} \approx \frac{c^2}{c^2-1} \quad (7)$$

. It can be observed that the average code length is very close to 1, especially when $c$ is large. As for the optimal case when $c = e$, the average code length is about 1.2.

## 5.3 Signature Compression

Another approach for reducing the size of the signature is called *compression*. It is motivated by the observation that in the signature of node $n$, many objects share the same backtracking link; furthermore, once the signature of a single object $u$ is determined, the signature of another object $v$ which shares the same link may be obtained by adding up the signatures of $s(n)[u]$ and $s(u)[v]$. This is especially true when $u$ is much closer to $n$ than $v$. Therefore, we can replace $s(n)[v]$ with a 1-bit flag to designate that $s(n)[v]$ should be computed by adding up $s(n)[u]$ and $s(u)[v]$. This is a typical method of "exchanging time for space", and we apply it to compress the signatures because:

- The node in a road network usually has few adjacent nodes, so many objects share the same backtracking link.
- The distance categories are exponentially partitioned, so the signatures of many remote objects can be represented by adding up two signatures.
- Queries normally focus on local areas only, so distant objects are not frequently accessed. As such, the decompression (i.e., retrieving s(n)[v] by adding up two signatures) also occurs infrequently and incurs little CPU overhead.

It is noteworthy that, as the signature index already stores the distance of any two objects and caches it in memory for approximate distance comparison (see Section 3.2.2), the compression and decompression require no additional memory storage. Nonetheless, there are two tasks remaining, namely, the selection of object $u$ and the definition of the "add-up" operation. For the former task, we choose to select the closest object (in terms of the distance categories), resolving ties by their positions in the sequence of $s(n)$. For the latter task, since categories are partitioned exponentially, the normal integer summation does not appropriately represent the actual summation of distances. Therefore, we define the summation of signatures as follows. If two signatures are not equal, the summation is defined as the larger of the two, because it is the dominant distance in the summation; if the two signatures are equal, the summation is defined as their signatures incremented by 1. This is based on the reasoning that, under the grid and uniform distribution assumptions, the average distance of an object in a category is larger than the medium of its upper and lower bound, because the number of objects at distance $i$ is proportional to $i$. Based on this reasoning, the summation of two objects in the same category is likely to exceed its upper bound. The summation operation is summarized as follows:

DEFINITION 5.1. *For any objects $u, v$ and node $n$,* $s(n)[u] + s(u)[v] =$

$$\begin{cases} max(s(n)[u], s(u)[v]) & \text{if } s(n)[u] \neq s(u)[v] \\ s(n)[u] + 1 & \text{if } s(n)[u] = s(u)[v] \end{cases}$$

Algorithm 7 shows the pseudo-code for the compression algorithm. It reads an incoming signature and finds out the closest object for every backtracking link. Then it sequentially reads the whole signature again and tests if the signature of any object can be compressed.

**Algorithm 7** Signature Compression Algorithm

**Input:** the signature of node $n$, $s(n)$
**Output:** the compressed signature $s'(n)$
**Procedure:**

```
1: read s(n) to find the closest object for each backtracking
   link
2: for each object v do
3:    u is the closest object such that s[u].link = s[v].link
4:    if s(n)[u] + s(u)[v] = s(n)[v] then
5:       flag s(n)[v] as compressed
```

## 5.4 Signature Update

Distance signature is efficient in update, that is, a change on the nodes or edges only causes a limited number of signatures to be updated because: (1) distance categories are exponentially partitioned, so a local change on the nodes or edges is not likely to make a distant object change its category, (2) the backtracking link only indexes the next node in the shortest path, which is also less likely to be remotely affected.

As node insertion/deletion can be reduced to edge(s) insertion/deletion, we consider edge update only. The main idea is to maintain the shortest path spanning trees of all objects (the intermediate results during signature construction). Besides these spanning trees, we also need a reverse index for each edge on the objects whose spanning trees comprise this edge. This index is used to identify the spanning trees that are affected by edge removal or edge weight increase.

### 5.4.1 Adding Edge/Decreasing Edge Weight

For the spanning tree from object $o$, let nodes $a, b$ denote the two nodes adjacent to this edge and $d(o, a) > d(o, b)$. Then $b$ is tested to see if $d(o, a) + w(a, b) > d(o, b)$ ($w(a, b)$ is the

weight of this edge). If it is true, $d(o,b)$ is updated and all nodes $i$ adjacent to $b$ with $d(o,i) > d(o,b)$ are tested if their distances should also be updated, i.e., if $d(o,b) + w(b,i) > d(o,i)$. The update is thus propagated until there are no more updates.

#### 5.4.2 *Removing Edge/Increasing Edge Weight*

First of all, the reverse index of this edge is checked to get the spanning trees that are affected. For any affected spanning tree from object $o$, let nodes $a, b$ denote the two nodes adjacent to this edge and $d(o,a) > d(o,b)$. Then $d(o,b)$ is recomputed by considering all of its adjacent nodes (including $a$). The update on $d(o,b)$ is then propagated to the subtree rooted from $b$ in the spanning tree until there are no more updates.

The aforementioned update algorithm is only on the spanning trees and reverse index. To update the signature of each node $n$, the updates on $n$ are aggregated and only the changes on distance category or backtracking link are updated in the signature.

## 6. PERFORMANCE EVALUATION

In this section, we present the experimental results on the signature index. We used two road networks in the simulation. The first one is synthetic for controlled experiments. It was created by generating 183,231 planar points and connecting neighboring points by edges with random weights between 1 and 10. The degrees of the nodes follow an exponential distribution with mean set to 4 (i.e., the degree for a two-road intersection). The second one is a real road network obtained from Digital Chart of the World (DCW). It contains 594,103 railroads or roads, and 430,274 junctions in US, Canada, and Mexico. Similar to [10], we employed the connectivity-clustered access method (CCAM) [12] to sort and store the nodes, their adjacency lists, and the signatures. The page size was set to 4K bytes. The testbed was implemented in C++ on a Win32 platform with 2.4 GHz Pentium 4 CPU and 512 MB RAM.

Since there is no existing work on a general-purpose index on road networks, we compare the signature index and the associated query processing algorithms with two closest competitors. The first is *full indexing*, which stores the exact distances of all objects for each node. The second is the Network Voronoi Diagram (NVD) used in the Voronoi-based Network Nearest Neighbor ($NV^3$) algorithm [8], which is known to be an efficient kNN algorithm for road networks. Since NVD does not support range query [8], we design a reasonable algorithm for it as follows: the network Voronoi polygon (NVP) of query node $n$ is obtained and the corresponding object is checked for its distance to $n$, which is already stored in NVD. If it is a result, the search is then expanded to all the adjacent $NVP$s until the distance exceeds the threshold. Regarding the performance metrics for query processing, we measured the CPU time and the number of disk *page accesses*.

### 6.1 Index Construction and Maintenance

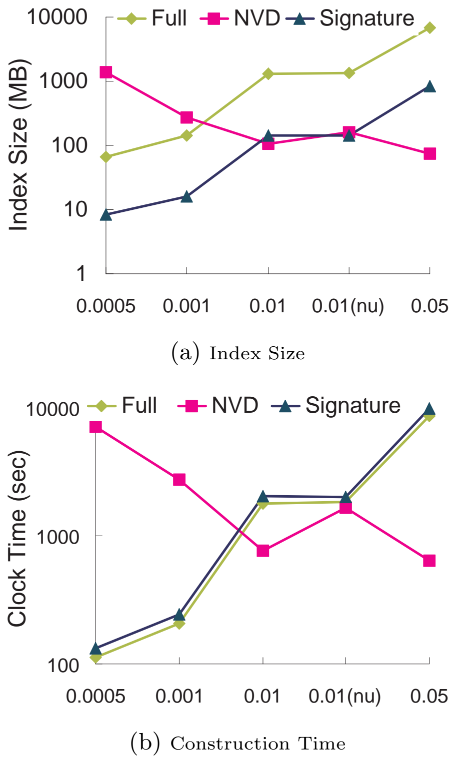
For each road network, we created four uniformly distributed datasets with density $p$ (the ratio of the number of the objects to the number of the nodes) set to 0.0005, 0.001, 0.01, and 0.05, respectively, and one non-uniform dataset that is composed of 100 clusters and $p = 0.01$ (denoted by $0.01(nu)$). For the full indexing, the shortest path trees of all objects were built and their distances to node $n$ were stored together in dedicated pages. For signature index, we set $c = e$ and $T = 10$, and the raw signatures were first obtained when the shortest path trees were built, followed by the encoding and compression processes. For NVD indexing, we built the NVP R-tree, NVD's, border-to-border ($Bor - Bor$) distances, and object-to-border ($OPC$) distances. Figure 6.4 shows the index sizes and the clock time for index construction on the synthetic road network with various datasets[2]. In Figure 6.5(a), the size of signature index is almost the smallest except for $p = 0.05$. In particular, the signature index is about $1/6 \sim 1/7$ the size of the full index. Given that 4 bytes (an integer) are used for each object in the full index and 3 bits are used for the backtracking link in the signature index, we can infer that the signature index only uses a little more than 1 bit on each distance category. We also observe that the sizes of both full index and signature index are proportional to the object density $p$. As for the NVD index, the index size increases as $p$ decreases, because $Bor - Bor$ and $OPC$ distances, which increase significantly as the Network Voronoi Polygon (NVP) expands, account for most of the NVD index storage. As an extreme example, the size of NVD index becomes forbiddingly high for sparse datasets ($p < 0.001$). Another observation is that while the full and signature indexes are not affected by the dataset distribution, NVD index is sensitive to it. This is because non-uniform datasets create more large NVPs which dominate the index size. In Figure 6.5(b), the construction time leads to similar observations, except that the signature index costs a bit more time than the full index, as it needs to encode the categories and compress the signatures besides the construction of shortest path trees. Nonetheless, it is still less costly than the $NVD$ index for most of the datasets. As such, we can conclude that the signature index is efficient for medium or sparse datasets and robust with various distributions.

To examine the effectiveness of the encoding and compression algorithms for signature index, we also measured the intermediate results before and after encoding. Table 1 shows the results for all 5 datasets. It is observed that, in all these datasets, the proportion of size reduced by the encoding algorithm is almost a constant 0.74, which is equivalent to reducing a category id from 3 bits to 1.4 bits. This result is consistent with our analysis in Section 5.1 that optimal code length is as short as 1.2 bits. Meanwhile, the effectiveness of compression algorithm increases as density $p$ increases, because more objects in distant categories can now be represented by closer objects and hence compressed. From the table, the average reduction ratio is 80%, which means that 70% of the objects are compressed, i.e., their category ids are replaced by the 1-bit compressed flag.

### 6.2 Query Search Result

We created workloads of range queries and type 3 kNN queries to compare the performance of the three indexes. For each workload, we randomly created $500 \sim 1000$ queries

[2]Since the results on the real road network show a similar trend as in the synthetic network, they are omitted for the interest of space.




(a) Index Size

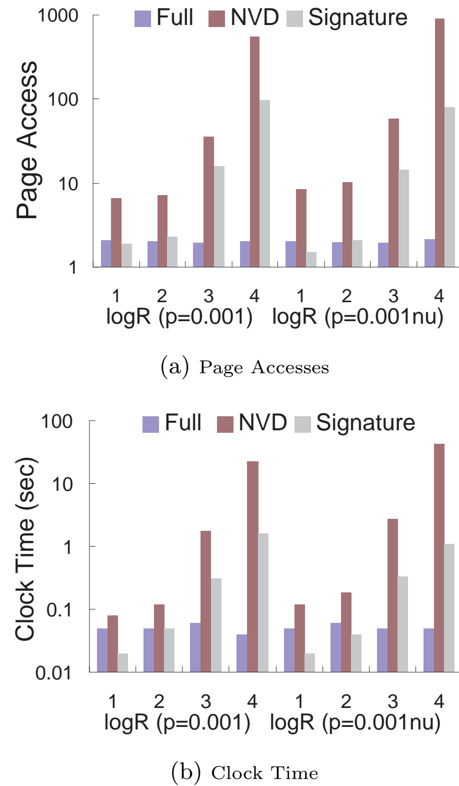


(b) Construction Time

Figure 6.4: Comparison on Index Construction Cost

|  | 0.005 | 0.001 | 0.01 | 0.01($nu$) | 0.05 |
|---|---|---|---|---|---|
| **Raw** | 12.6 | 26.5 | 247 | 252 | 1259 |
| **Encoded** | 9.3 | 19.6 | 176 | 188 | 965 |
| **Ratio** | **74%** | **74%** | **71%** | **74%** | **76%** |
| **Compressed** | 8.4 | 15.8 | 141 | 141 | 728 |
| **Ratio** | **90%** | **80%** | **80%** | **75%** | **75%** |

Table 1: Encoding and Compression on Signatures




(a) Page Accesses



(b) Clock Time

Figure 6.5: Comparison on Range Search

(depending on the query processing time) and measured the average performance.

The first set of experiments was based on the range query workload. We set the range threshold $R$ to 10, 100, 1000, and 10000, and plotted the number of page accesses and clock time. Since the five datasets show similar trend, Figure 6.5 only depicts the results for 0.01 and 0.01($nu$). From the figure, we can observe that: (1) as expected, the full index always achieves the best performance except for $R = 10$; (2) both NVD and signature index are as efficient as the full index for $R = 10, 100$, in particular, the signature index outperforms full index for $R = 10$, because within short distances, the signatures of few nodes need to be accessed; (3) NVD has a sharp increase when $R$ increases from 100 to 1000 because this is the distance range when the NVP of the query node is no longer sufficient to answer the query, and the phenomenon is more prominent in the non-uniform dataset; (4) the signature index has a similar trend as NVD, but instead of increasing linearly as $R$ increases, the performance of signature index (especially the clock time) is sublinear to $R$ and is still satisfactory (about 1 second) even when $R = 10000$, thanks to the CPU-efficient guided backtracking.

The second set of experiments was based on the kNN query workload, where $k$ ranges from 1 to 50. We measured the page accesses and clock time and plotted the results for $p = 0.01$ dataset in Figure 6.6. We observe that: (1) similar to the range query, the full index always achieves the