A Markov Chain Based Pruning Method for Predictive Range Queries

Xiaofeng Xu, Li Xiong, Vaidy Sunderam, and Yonghui Xiao

Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322

{xxu37, lxiong, vss, yonghui.xiao@emory.edu}

ABSTRACT

Predictive range queries retrieve objects in a certain spatial region at a (future) prediction time. Processing predictive range queries on large moving object databases is expensive. Thus effective pruning is important, especially for long-term predictive queries since accurately predicting long-term future behaviors of moving objects is challenging and expensive. In this work, we propose a pruning method that effectively reduces the candidate set for predictive range queries based on (high-order) Markov chain models learned from historical trajectories. The key to our method is to devise compressed representations for sparse multi-dimensional matrices, and leverage efficient algorithms for matrix computations. Experimental evaluations show that our approach significantly outperforms other pruning methods in terms of efficiency and precision.

1. INTRODUCTION

With positioning techniques like GPS, huge amounts of location data are collected from continuous moving objects like vehicles and mobile device users. Predictive range queries can be performed on these location data to retrieve objects in a certain spatial region at the prediction time. Efficiently and accurately processing predictive range queries is important for many location-based services, such as real time ride sharing, location-based crowd sourcing, and navigation.

A typical strategy for answering predictive range queries is *pruning-verification*, where the pruning step quickly filters out objects that are likely not in the query result and the verification step verifies the remaining object according to predefined *prediction functions* [16, 20, 7, 19]. Objects surviving the pruning step are called *candidate objects* and form the *candidate set*. The verification step can be very expensive if pruning is not effective enough to reduce the size of candidate set.

We categorize prediction functions into motion functions (e.g. [16, 19]), trajectory patterns (e.g. [24, 15, 10, 11, 23, 25]) and descriptive models (e.g. [8, 3, 2, 26]). Motion functions can effectively and efficiently estimate near future (tens of seconds) locations of the moving objects, but cannot accurately estimate their long-term (tens of minutes) behaviors. Trajectory patterns or descriptive models can perform long-term predictions, but are usually very expensive. Therefore effective pruning is even more crucial for long-term predictive queries. Existing pruning methods, such as *query window enlargement* [7] that enlarges the query window based on object velocities, and *travel time grid* that records the average travel time between grid cells in the space domain [5], rely on simple predicates and have limited pruning capacities.

This paper focuses on the pruning step. We propose a (high-order) Markov chain based pruning method that efficiently and effectively reduces the size of candidate set for long-term predictive range queries. Specifically, we partition the predefined space domain with a uniform grid, where each grid cell is called a *state*. A *path* of a moving object is defined as a sequence of states on the grid. We assume that the paths follow an order-k Markov chain model, where the *transition matrix*, describing *transition probabilities* between states, is off-line learned from historical trajectories. Given a predictive range query, through matrix computations, we prune the paths that are not likely to transit into the query window at the prediction time and then use any state-of-the-art prediction functions for verification. Computation involving the Markov transition matrix can be very expensive in terms of both CPU and memory. Based on the observation that in real world scenarios, the Markov transition matrices are extremely sparse, especially those for high-order Markov chains, we propose a novel approach to compactly store the sparse transition matrices in main memory, and novel algorithms for performing arithmetic operations on these matrices, which significantly reduce the computation costs.

The contributions of this paper are summarized below.

- We propose an effective pruning algorithm, based on (high-order) Markov chain models, to reduce the candidate set for long-term predictive range queries.
- We propose a novel approach to compactly store sparse multi-dimensional matrices and support arithmetic operations involved in our pruning mechanism.
- Extensive experiments are performed to demonstrate that our method is effective and significantly outperforms existing pruning methods.

The remainder of this paper is organized as follows. In Sec-

SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA

DOI: http://dx.doi.org/10.1145/2996913.2996922

tion 2, we review related works. In Section 3, we present preliminary background and clarify the problem definition. Section 4 introduces the data structures storing the trajectories and sparse multi-dimensional matrices. In Section 5, we present the query processing pipeline using our Markov chain based pruning mechanism. Experimental studies are presented in Section 6. Finally in Section 7, we conclude this paper and propose future work.

2. RELATED WORK

In this section, we revisit some related works about indexing moving objects and pruning methods for processing predictive range queries.

2.1 Indexing moving objects

Data structures that augment R-tree [4] or R*-tree [1] are very popular for indexing moving objects (e.g. TPR-tree [16], TPR*-tree [19], STP-tree [20]). By indexing motion functions, R-tree based indexes are able to capture near future locations of the moving objects by monitoring the enlargement of *minimum bounding rectangles* (MBRs) [16].

Unfortunately, R-tree based indexing structures are usually expensive to maintain, thus are inefficient in applications with frequent data updates. Instead of the R-tree family, recent studies (e.g. [17, 18, 13, 21]) indicate that a simple uniform grid is better choice for indexing moving objects in main memory. Moreover, *space-filling* techniques can also be applied on a uniform grid to map the spatial coordinates into scalars that can be indexed by B$^+$-trees (e.g. the B^x-tree [7]).

2.2 Pruning methods

For processing predictive range queries, R-tree based indexes prune the nodes whose MBRs will not intersect with the range query window at the prediction time [16, 22], while grid based indexes apply the *query window enlargement* technique[7] to prune the grid cells that will not intersect with the enlarged query window at the prediction time. Note that the enlargements of MBRs and query windows are based on specific motion functions.

Unfortunately, enlarging MBRs or query windows cannot monitor positions of the objects in far future, as their motions might change over time. Hendawi et. al proposed *Panda* [5] a long-term predictive query processor that uses a pruning strategy based on the *travel time grid* (TTG) – a 2-dimensional array where each cell $TTG[i,j]$ stores average travel time between two grid cells C_i and C_j that are learned from historical trajectories. According to TTG, objects in the cells that are unreachable to the query window within the prediction time are pruned. However, the average travel time is far too simple a measure to capture transition patterns between these cells e.g. in real world highly variable traffic conditions.

3. PRELIMINARIES AND PROBLEM DEFINITION

We introduce some preliminary definitions and our problem settings in this section. Table 1 lists the notations used throughout this paper.

Trajectory and path. The predefined space domain is partitioned into uniform grid cells and each cell has an identifier, which is the sequence number of the grid cell in the underlying *space-filling curve* (see Section 4.2). We call the grid cells *states* and the set of all states, $\mathcal{S} = \{s_0, s_1, \ldots, s_{N-1}\}$, the *state space*, where $N = |\mathcal{S}|$ denoting the total number of states. In this paper, we alternatively denote s_i with its identifier i, $\forall 0 \leq i < N$, when there is no ambiguity. A *trajectory* of an object $O \in \mathcal{O}$ is defined as a sequence of timestamped locations $\mathcal{T} =< O_0, O_1, \ldots, O_t, \cdots >$, where $O_t = ((x,y),t)$ denotes that O locates at (x,y) at time t. As each location is associated with a state, we call a sequence of states of O, denoted as $\mathcal{I} =< i_0, i_1, \ldots, i_t >$, a *path* of O. We refer to the path of O in the h timestamps from $t-h+1$ to t as an *h-backward path* ending at t and denote it as $\mathcal{I}_h^t$, i.e. $\mathcal{I}_h^t =< i_{t-h+1}, i_{t-h+2}, \ldots, i_t >$. Particularly, we call an h-backward path ending at the current time t_c, $\mathcal{I}_h^{t_c}$, the *base h-backward path*.

Table 1: Notations

O	moving object	$\mathcal{O}$	set of all objects
N	number of states	Q	predictive range query
$\mathbf{q}$	query vector	R	query window
t_c	current time	t_q	prediction time
k	order of Markov chain	M_k	Markov transition matrix
s_i/i	state	$\mathcal{S}$	state space
ρ_k^t	state matrix	$\mathcal{I}_h^t$	h-backward path
ϕ	diagonal	ξ	offset of diagonal
$\mathcal{D}$	diagonal matrix	Ξ	offset array
$\mathbf{d}$	diagonal array	ι	shift of projection
$\mathcal{T}$	trajectory	$\mathcal{I}$	path/coordinate
ω_ϕ	valid range	H	number of phases

Predictive range query. Given a set of moving objects, we define predictive range queries as follows.

Definition 3.1 (Predictive range query). *Given a set of moving objects, $\mathcal{O}$, with their recent trajectories, $\mathcal{T}$, a spatial region R, and a prediction time t_q, the predictive range query, $Q(R, t_q)$, returns the set of objects in region R at time t_q.*

Generally speaking, a predictive range query is processed in two steps: pruning and verification [16, 20, 7, 19]. The pruning step aims to filter out non-qualifying objects that will not likely be in the query result, via a fast pruning mechanism. Candidate objects surviving the pruning step are fed to the verification step that computes their probabilities of satisfying the query predicate. We call the base h-backward paths of the candidate objects the *candidate paths*. Since the verification step dominates the overall execution time for query processing (see Section 6) it is important to have effective pruning mechanisms to reduce the number of candidate objects.

The Markov chain model. In most real world scenarios, future paths of the moving objects, such as vehicles in a city road network, are usually uncertain but follow certain patterns. We use time-homogeneous Markov chains [9] to capture such mobility patterns.

Definition 3.2 (Order-k Markov chain). *A stochastic process o_t, is called an order-k Markov chain if and only if $\forall i_t, i_{t-1}, \ldots, i_0 \in \mathcal{S}$*

$$\begin{aligned} &P(o_t = i_t | o_0 = i_0, \ldots, o_{t-k} = i_{t-k}, \ldots, o_{t-1} = i_{t-1}) \\ =&P(o_t = i_t | o_{t-k} = i_{t-k}, \ldots, o_{t-1} = i_{t-1}) \end{aligned} \quad (1)$$

We call Equation (1) the *local Markov property* of order-k Markov chains. The conditional probability is called *transition probability*, which indicates the probability for an object

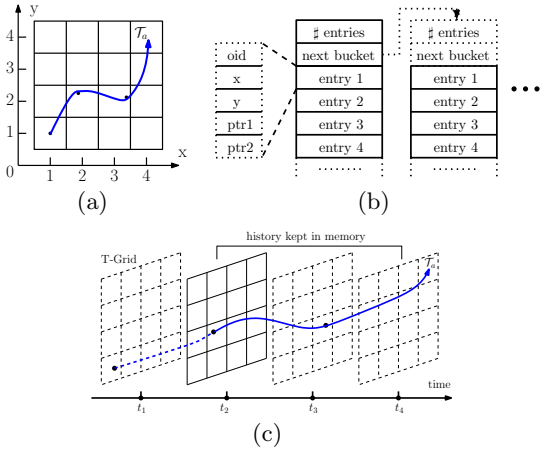


Figure 1: Structure of the T-Grid

moving to state i_t given its previous k states $i_{t-k}, \ldots, i_{t-1}$. The $\underbrace{N \times N \times, \ldots, \times N}_{\text{k+1 N's}}$ multi-dimensional matrix M_k is the transition matrix of the order-k Markov chain where

$$M_k[i_{t-k}, i_{t-k+1}, \ldots, i_t] = P(i_t | i_{t-k}, \ldots, i_{t-1}) \qquad (2)$$

In the rest of this paper, we denote $\underbrace{N \times N \times, \ldots, \times N}_{\text{d N's}}$ as $N^{(d)}$ for simplicity. Note that the coordinate $[i_{t-k}, i_{t-k+1}, \ldots, i_t]$ of an element in M_k inherently forms a path $\mathcal{I}_k^t$. The main goal of our proposed method is to efficiently reduce the candidate set for predictive range queries with the aid of (high-order) Markov chains.

Sparse matrix storage. Transition matrices, especially those for high-order Markov chains, are usually sparse in spatio-temporal settings. One storage format for sparse matrices [14, 6] is *dictionary of keys* (DOK), a dictionary that maps (row, column)-pairs to values for non-zero elements. DOK is ideal for incrementally constructing matrices but poor for arithmetic operations. The *compressed row storage* (CSR) format stores a sparse matrix using three 1-dimensional arrays (A, IA, JA): where A holds all the nonzero entries in *row-major* order, IA records the start and end indexes in A for each row, and JA contains the column index of each element of A. The CSR format is efficient for arithmetic operations such as inner-product, but inefficient for incremental construction. Therefore, one typical strategy is to use DOK format for construction and then convert the matrix to CSR format while performing arithmetic operations. For matrices that consist of a few diagonals, the diagonal format (DIA) stores the diagonals in a rectangular $N_{diag} \times N$ array, where N_{diag} is the number of diagonals and N is the number of columns. Note that these formats only support 2-dimensional matrices and are not applicable in high dimensional scenarios. We present new data structures for storing sparse high-dimensional transition matrices in next section.

4. DATA STRUCTURES

In this section, we introduce the data structures to store object trajectories and sparse Markov transition matrices.

4.1 The trajectory grid

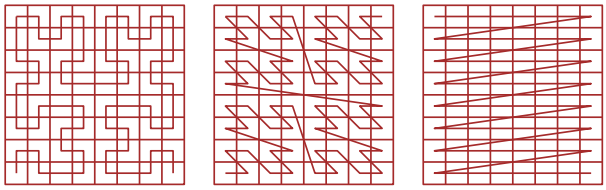


Figure 2: Space-filling curves

Table 2: Occupancy ratio of Markov transition matrices

Grid size	Order	Element	Hilbert	Morton	Raster
64 × 64	order-1	0.003	0.175	0.093	**0.008**
	order-2	6.39e-10	3.89e-4	2.563e-4	**2.55e-5**
	order-3	5.56e-24	1.89e-7	1.308e-7	**2.17e-8**
128 × 128	order-1	8.59e-4	0.133	0.074	**0.007**
	order-2	1.1e-11	9.04e-5	5.7e-5	**7.72e-6**
	order-3	3.8e-28	1.36e-8	9.13e-9	**1.33e-9**
256 × 256	order-1	2.23e-4	0.096	0.054	**6.15e-4**
	order-2	1.79e-13	2.31e-5	1.41e-5	**1.81e-6**
	order-3	2.06e-32	9.29e-10	6.52e-10	**8.72e-11**

Figure 1(a) shows part of a trajectory in the 2-dimensional uniform grid, where the dots represent sampled locations. We assume that all trajectories are sampled with equal intervals and synchronized timestamps. Interpolation is performed when this assumption does not hold. The *trajectory grid* (T-Grid) consists of a series of uniform grids aligned by timestamps, which we call *phases*. Each phase stores only the sampled locations with a certain timestamp. Figure 1(c) shows an example of the T-Grid and the corresponding trajectory in Figure 1(a). Note that only the most recent H phases are kept in main memory and $H \geq k$, where k is the order of the underlying Markov chain. The main purpose of introducing phases is to support fast object/trajectory retrievals by timestamps, which is useful in query processing (see Section 5).

Sampled locations of the trajectories are stored in the corresponding grid cells. A grid cell is stored as a list of *buckets* where each bucket contains the number of data entries stored, the pointer that links to the next bucket when the number of entries in the current bucket exceeds a predefined *capacity*, and an array of data entries. Each data entry stores a single trajectory record, which consists of object ID (*oid*), spatial coordinate (x and y) and the pointer that links to the position of the next record in the same trajectory. Each such pointer contains two fields $ptr1$ and $ptr2$ that represent cell ID and in-bucket position of the next record (data entry), respectively. Figure 1(b) illustrates the storage structure of a grid cell. In this paper, the capacity is 1000 by default.

4.2 The MDIA format

To store Markov transition matrices that encode consecutive states of moving objects as multi-dimensional arrays, we map cells in 2-d space to a 1-d sequence via *space-filling* curves e.g. Hilbert, Morton, and Raster as shown in Figure 2. Markov transition matrices, especially those of high-order Markov chains, are usually extremely sparse in real-world spatio-temporal settings, due to geographic restrictions and velocity limitations. Table 2 shows occupancy ratios of Markov transition matrices derived from Beijing taxi data (see Section 6.1). In this table, column *Element* repre-

sents *element occupancy* while *Hilbert*, *Morton*, and *Raster* columns represent *diagonal occupancy* with the corresponding space-filling techniques. Element occupancy and diagonal occupancy denote the ratio of nonzero elements and (partially) occupied diagonals (see Definition 4.1), respectively.

As we can observe, the transition matrices become increasingly sparse when either the order of Markov chain or the grid size increases. Furthermore, the transition matrices have very low diagonal occupancy, which means that they can be fully and compactly represented by only a small portion of diagonals. The intuition is that moving objects transit to neighboring locations, so non-zero transition probabilities tend to be clustered around diagonals. Moreover, among the three space-filling techniques, Raster achieves lowest diagonal occupancy, and is thus selected as the default space-filling technique in this paper. Additionally, for high-order Markov transition matrices, element occupancies are much lower than diagonal occupancies, which means that the diagonals themselves might be sparse as well.

Based on the above observations, we propose a multi-dimensional diagonal (MDIA) representation for extremely sparse matrices. The MDIA format of a sparse matrix consists of two components: 1) the diagonal matrix, $\mathcal{D}$, which is an $N_{diag} \times N$ matrix (using its CSR format [14] when sparse), where N_{diag} denotes the number of (major) diagonals, and 2) *offsets* (with respect to the major diagonal starting from the origin) of the diagonals, which form an array of tuples denoted as Ξ. We define diagonals and offsets of a multi-dimensional matrix as follows.

DEFINITION 4.1 (DIAGONAL AND OFFSET). *Given an $N^{(m)}$ ($m \geq 2$) matrix M, the tuple $\phi = (\xi, \mathbf{d})$ is called a diagonal of M if and only if $\mathbf{d}$ is a 1-dimensional array with $\mathbf{d}[i] = M[i + \delta_0, i + \delta_1, \ldots, i + \delta_{m-2}, i]$, where $0 \leq i + \delta_j < N, \forall 0 \leq i < N, 0 \leq j < m$ and $\xi = (\delta_0, \ldots, \delta_{m-2})$ is called the offset of ϕ.*

According to this definition, the element $\mathbf{d}[i]$ is valid only when $0 \leq i + \delta_j < N$, $\forall 0 \leq j < m$, which is equivalent to $max(0, -\delta_{min}) \leq i < min(N, N - \delta_{max})$, where δ_{min} and δ_{max} denote the minimum and maximum values of δ_i, $0 \leq i \leq m-2$. We denote $\omega_\phi = [max(0, -\delta_{min}), min(N, N - \delta_{max}))$ as the *valid range* of ϕ. Given a matrix coordinate $\mathcal{I} =< i_0, i_1, \ldots, i_{m-1} >$, offset of the diagonal where it resides and the in-diagonal position are computed by

$$\xi = (i_0 - i_{m-1}, i_1 - i_{m-1}, \ldots, i_{m-2} - i_{m-1}), \ \kappa = i_{m-1} \quad (3)$$

Conversely, given the offset $\xi = (\delta_0, \ldots, \delta_{m-2})$ and the in-diagonal position $\kappa \in \omega_\phi$, the corresponding coordinate is

$$\mathcal{I} =< \kappa + \delta_0, \kappa + \delta_1, \ldots, \kappa + \delta_{m-2}, \kappa > \quad (4)$$

We also define, for any 1-dimensional matrix/array M, $\mathbf{d} = M$ as the only diagonal with offset $\xi = \emptyset$. Figure 3 shows examples of diagonals, $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$, and their offsets of a 2-dimensional matrix, where the solid part of $\mathbf{d}_1$ through $\mathbf{d}_3$ represent the valid ranges. The following shows an example of a high dimensional MDIA matrix.

EXAMPLE 4.1. *Let M be a 3-dimensional matrix where $M[0,:,:] = \begin{bmatrix} 1&0&0 \\ 1&0&0 \\ 0&0&0 \end{bmatrix}$, $M[1,:,:] = \begin{bmatrix} 0&1&0 \\ 0&1&0 \\ 0&1&0 \end{bmatrix}$, $M[2,:,:] = \begin{bmatrix} 0&0&0 \\ 0&0&1 \\ 0&0&1 \end{bmatrix}$. M has three diagonals $((0,0),[1,1,1])$, $((0,1),[1,1,0])$, and $((0,-1),[0,1,1])$. The MDIA format of M is $(\mathcal{D}, \Xi)$ where $\Xi = \begin{bmatrix} (0,0) \\ (0,1) \\ (0,-1) \end{bmatrix}$ and $\mathcal{D} = \begin{bmatrix} 1&1&1 \\ 1&1&0 \\ 0&1&1 \end{bmatrix}$. The CSR format of $\mathcal{D}$ is (A, IA, JA), where $A = [1,1,1,1,1,1,1]$, $IA = [0,3,5,7]$, and $JA = [0,1,2,0,1,1,2]$.*

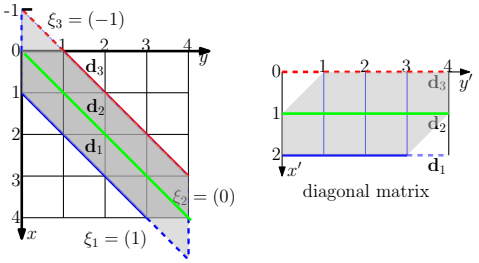


Figure 3: Diagonals

5. ALGORITHMS

In this section, we introduce our proposed pruning algorithm for predictive range queries using Markov chains.

5.1 Markov chain based pruning

Generate candidate paths. Given a predictive range query $Q(R, t_q)$, a binary vector $\mathbf{q}$ represents the spatial region R. Specifically, the i^{th} position in $\mathbf{q}$ is set when state s_i intersects with R. We call $\mathbf{q}$ the *query vector* of Q.

DEFINITION 5.1 (STATE MATRIX). *Given a predictive range query $Q(R, t_q)$ with query vector $\mathbf{q}$, at any time t and $t_c \leq t \leq t_q$, a state matrix ρ_k^t of an order-k Markov chain is an $N^{(k)}$ matrix, where $\rho_k^t[i_{t-k+1}, \ldots, i_t] = P(\mathbf{q}|\mathcal{I}_k^t)$ denoting the probability that the k-backward path $\mathcal{I}_k^t =< i_{t-k+1}, \ldots, i_t >$ moves into R at time t_q.*

In order to reduce the candidate set, we need to compute the probability for each object, given its base k-backward path, moving into R at time t_q. Such probabilities are stored in the state matrix $\rho_k^{t_c}$, which is called the *base state matrix*. Based on the local Markov property and Bayes rules, each element in ρ_k^t, $t_c \leq t < t_q$, can be calculated by

$$\begin{aligned} P(\mathbf{q}|\mathcal{I}_k^t) &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) P(\mathcal{I}_k^{t+1}|\mathcal{I}_k^t) \\ &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) \frac{P(\mathcal{I}_{k+1}^{t+1})}{P(\mathcal{I}_k^t)} \\ &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) P(i_{t+1}|\mathcal{I}_k^t) \end{aligned} \quad (5)$$

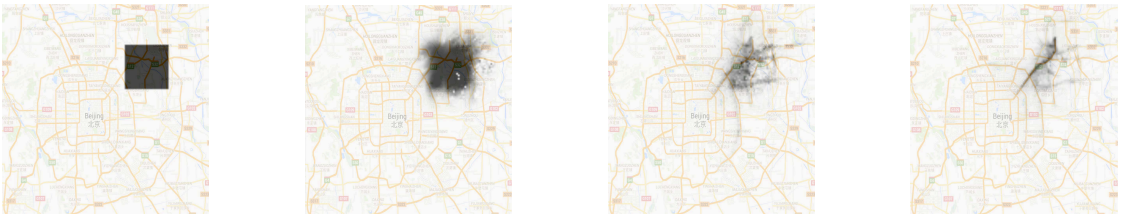
Note that $P(i_{t+1}|\mathcal{I}_k^t) \equiv P(i_{t+1}|i_{t-k+1}, i_{t-k+2}, \ldots, i_t)$ equals to $M_k[i_{t-k+1}, i_{t-k+1}, \ldots, i_{t+1}]$, where M_k is the order-k Markov transition matrix. Thus, we can rewrite Equation (5) as

$$\rho_k^t = \begin{cases} M_k \odot \mathbf{q} & t = t_q - 1 \\ M_k \odot \rho_k^{t+1}, & t_c \leq t < t_q - 1 \end{cases} \quad (6)$$

where $(\odot)$ represents the *multiplication* operation:

DEFINITION 5.2 (MULTIPLICATION). *Let M be an $N^{(m)}$ matrix, and L an $N^{(l)}$ matrix, where $m \geq 2$ and $1 \leq l < k$. $K = M \odot L$ is computed by*

$$K[i_0, \ldots, i_{m-2}] = \sum_{0 \leq j < N} M[i_0, \ldots, i_{m-2}, j] L[i_{m-l}, \ldots, i_{m-2}, j] \quad (7)$$



(a) Query window (b) Order-1 (c) Order-2 (d) Order-3

Figure 4: Base state matrices with different order of Markov chains

Algorithm 1: Generate candidate path trie

```
   input  : M_k (Markov transition matrix)
            Q (predictive range query)
            ε (pruning sensitivity)
   output: T (candidate path trie)
   /* compute base state matrix                                  */
1  q ← query vector of Q;
2  ρ_k^{t_q-1} ← M_k ⊙ q;                          /* Equation (6) */
3  for t ← t_q − 2 to t_c do
4      ρ_k^t ← M_k ⊙ ρ^{t+1};                      /* Equation (6) */
5  T ← empty transition matrix;
   /* iterate nonzero elements of ρ_k^{t_c}                      */
6  foreach diagonal φ(ξ, d) in ρ_k^{t_c} do
7      foreach nonezero element e in d do
8          κ ← position of e within d;
9          I ← coordinate of (ξ, κ);                /* Equation (4) */
10         if M[I]) > ε then
11             T.insert(I);
12 return T;
```

Note that when $m = 2$, multiplication degenerates to the classic inner-product operation between a 2-dimensional matrix and a 1-dimensional array. The base state matrix $\rho_k^{t_c}$ is computed by iteratively performing Equation (6).

Figure 4 visualizes base state matrices computed with order-k ($1 \leq k \leq 3$) Markov transition matrices learned from the Beijing dataset (see Section 6.1). Grayscales of the masks in Figure 4(b) through 4(d) reflect values of the corresponding elements in the base state matrices (probabilities for the corresponding paths to transit into R at time t_q). We can see that masks for high-order Markov chains largely follow the underlying road network, which also implies the transition patterns of vehicles. Intuitively, we prune paths with lower probabilities (lighter grayscales) to obtain the candidate paths.

Candidate paths are stored in the *Candidate Path Trie* (CPT), which is a Trie structure containing paths/coordinates corresponding to nonzero elements in the base state matrix with values greater than a threshold ϵ, the *pruning sensitivity*. CPT stores base k-backward paths that are promising to transit into R at time t_q according to the base state matrix and the transition matrix. Algorithm 1 summarizes the computation of CPT. Given a query Q, an order-k Markov transition matrix M_k, and the pruning sensitivity ϵ, we first compute the base state matrix $\rho_k^{t_c}$ using Equation (6), followed by generation of coordinates for each nonzero element in $\rho_k^{t_c}$ according to Equation (4), and insertion into T if its value is greater than ϵ. Note that Algorithm 1 is generic and applies to both ordinary and CSR diagonal matrices. An example of computing the puring predicate with an order-1 Markov transition matrix is shown below.

Algorithm 2: Process predictive range query

```
   input  : TG (T-Grid)
            Q (predictive range query)
            M_k (transition matrix)
   output: R (query result)
1  P ← compute candidate path trie ;          /* Algorithm 1 */
2  C ← generate candidate set via TG and P;
3  foreach object o in C do
4      I ← from TG get o's base k-backward path;
5      if VERIFY(o, Q) is true then
6          R.add(o);
7  return R;
```

EXAMPLE 5.1. *Given three historical paths,* $< 0, 1, 1 >$, $< 2, 1, 1 >$, *and* $< 2, 0, 2 >$, *the order-1 Markov transition matrix learned from these paths is* $M_1 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix}$. *Given a query* Q *with query vector* $\mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ *and prediction time* $t_q = 2$ *(note that current time* $t_c = 0$*). The state matrices are computed by*

$$\rho_1^1 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} \tag{8}$$

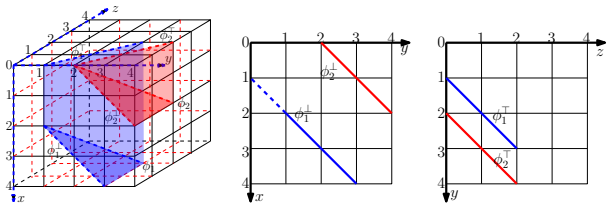
$$\rho_1^0 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.25 \end{bmatrix} \tag{9}$$

If $\epsilon = 0.2$, *the candidate path trie is* $\boxed{root} \longrightarrow \boxed{2}$, *which means only objects in state 2 at the current time are considered in the verification step.*

Process predictive range query. Algorithm 2 describes the query processing using CPT generated from Algorithm 1. The candidate set $\mathcal{C}$ is generated using the T-Grid TG and the CPT P. Details about generating candidate set is skipped here due to space limitations. An object $o \in \mathcal{C}$ is added to the result set if it passes the verification step, where $VERIFY$ can be any prediction function mentioned in Section 1, which predicts if a single object will enter the target region R at the prediction time. We skip details about the verification step here since this paper focuses on the pruning step.

5.2 Multiplying MDIA matrices

The major computational burden of constructing CPT resides in processing multiplication operations (lines 2-4 in Algorithm 1), which cannot be computed explicitly using Equation (7), since the matrices are stored in MDIA format. The traditional inner-product operation for 2-dimensional matrices is performed with row or column-major order, i.e. sequentially computes each row or column of the result matrix. For example, while computing $K = M \odot L$, where M and L both are $N \times N$ ordinary matrices, the j^{th} column



(a) (b) (c)

Figure 5: Projections

of K is calculated by performing inner-products between the j^{th} column of L and each row of M. However, this approach does not apply to MDIA matrices, since the elements are arranged in diagonals instead of rows or columns. In this paper, we propose an algorithm that efficiently performs the multiplication operation between MDIA matrices by using a diagonal-major approach that computes diagonals of the result matrix in a predefined order.

Projection table. We first introduce the *projection table*, which links diagonals with their *projections* defined as follows.

DEFINITION 5.3 (PROJECTION). *Let $\phi = (\xi, \mathbf{d})$ be a diagonal of an $N^{(m)}$ matrix, where $m \geq 2$ and $\xi = (\delta_0, \delta_1, \ldots, \delta_{m-2})$. $\phi^{\perp} = (\xi^{\perp}, \mathbf{d}^{\perp})$ is an H-projection of ϕ if*

$$\xi^{\perp} = \begin{cases} \emptyset & m = 2 \\ (\delta_0^{\perp}, \delta_1^{\perp}, \ldots, \delta_{m-3}^{\perp}) & m > 2 \end{cases} \tag{10}$$

where $\delta_i^{\perp} = \delta_i - \delta_{m-2}, \forall 0 \leq i < m-1$ and

$$\mathbf{d}^{\perp}[i + \iota] = \mathbf{d}[i], \forall 0 \leq i, i + \iota < N \tag{11}$$

where

$$\iota = \begin{cases} \delta_{m-2} & m = 2 \\ \delta_{m-2} - \delta_{m-3} & m > 2 \end{cases} \tag{12}$$

is the shift of the projection. $\phi^{\top} = (\xi^{\top}, \mathbf{d}^{\top})$ is a T-projection of ϕ if

$$\xi^{\top} = \begin{cases} \emptyset & m = 2 \\ (\delta_0^{\top}, \delta_1^{\top}, \ldots, \delta_{m-3}^{\top}) & m > 2 \end{cases} \tag{13}$$

where $\delta_i^{\top} = \delta_{i+1}$, $0 \leq i < m-1$ and

$$\mathbf{d}^{\top}[i] = \mathbf{d}[i], \forall 0 \leq i < N. \tag{14}$$

($\perp$) and ($\top$) are called the H and T-projectors, respectively. Intuitively, the H and T-projections project a diagonal of an m-dimensional matrix onto the first and last $m-1$ dimensions, respectively. The projections themselves are partial diagonals of a $(m-1)$-dimensional matrix. Note that the array elements of ϕ are not vertically projected onto its H-projection $\phi^{\perp}$, but are aligned according to a shift computed by Equation (12). Figure 5(a) shows an example of projecting two diagonals ϕ_1 (blue) and ϕ_2 (red). The darker and lighter areas denote the sweeping regions of projecting the diagonals onto the xy and yz sub-spaces, respectively. Figure 5(b) and 5(c) show the H and T-projections in the corresponding sub-spaces, respectively. The dashed line in Figure 5(b) illustrates shifting the H-projection of ϕ_1.

The projection table associated with an MDIA matrix $M(\mathcal{D}, \Xi)$ is obtained by performing H and T-projections on every single diagonal of M and collecting **distinct** offsets

Algorithm 3: Generate projection table

```
   input : M (an N^(k+1) MDIA matrix)
   output: PT (projection table)
1  PT, Ξ^⊥, Ξ^⊤ ← empty arrays;
2  foreach diagonal φ in M do
       /* compute projections of φ, φ^⊥ = (ξ^⊥, d^⊥),
          φ^⊤ = (ξ^⊤, d^⊤)                                  */
3      φ^⊥, φ^⊤ ← H and T-projections of φ;
4      if ξ^⊥ not in Ξ^⊥ then
5          Ξ^⊥.append(ξ^⊥);
6      if ξ^⊤ not in Ξ^⊤ then
7          Ξ^⊤.append(ξ^⊤);
8      α, β, γ ← index of ξ, ξ^⊥, ξ^⊤ in Ξ, Ξ^⊥, Ξ^⊤;
9      PT.append(α, β, γ);
   /* create hash index                                    */
10 create index IDX_β on PT(β);
11 return PT;
```

of the projections, denoted as $\Xi^{\perp}$ and $\Xi^{\top}$. The projection table is a 2-dimensional array that stores one tuple (α, β, γ) in each line, where α, β, and γ represent the index of ξ, $\xi^{\perp}$, and $\xi^{\top}$, in Ξ, $\Xi^{\perp}$, and $\Xi^{\top}$, respectively. The following shows an example of computing projections and generating the projection table.

EXAMPLE 5.2. *Let $M(\mathcal{D}, \Xi)$ be an MDIA matrix, where*

$$\mathcal{D} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ and } \Xi = \begin{bmatrix} (0,0) \\ (1,1) \\ (0,-1) \end{bmatrix}$$

The H-projections are

$$\mathcal{D}^{\perp} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \text{ and } \Xi^{\perp} = \begin{bmatrix} (0) \\ (1) \end{bmatrix}$$

while the T-projections are

$$\mathcal{D}^{\top} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ and } \Xi^{\top} = \begin{bmatrix} (0) \\ (1) \\ (-1) \end{bmatrix}$$

Finally, the projection table is

$$PT = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

The main purpose of introducing projection table is to instantly locate the related diagonals during the multiplication operations. For instance, when we compute $K = M \odot L$, the projection table can help us find the corresponding diagonals of M and L that contain all the data that is needed to compute a certain diagonal of K. Thus, we create a hash index, IDX_{β} on the second column of the projection table to support constant time retrievals by β. i.e. for each diagonal $\phi^{\perp}$ of K, we can find the diagonals of M that project onto $\phi^{\perp}$ and their T-projections in constant time. Algorithm 3 summarizes main steps of generating the projection table. Next we present the approach of multiplying Markov transition matrices and state matrices with the help of projection tables. Multiplication between transition matrices and query vectors can be performed similarly, and is therefore skipped.

Dense diagonal matrix. Algorithm 4 summarizes the approach to perform multiplication when the diagonal matrix $\mathcal{D}$ is dense and stored as a 2-dimensional array. In Algorithm 4, we first allocate the memory for storing the MDIA format of K (lines 1-3). Then we compute the array $\mathbf{d}^{\perp} = \mathcal{D}[\beta]$ (lines 5-9) and offset $\xi^{\perp} = M.\Xi[\alpha]^{\perp}$ (line 10),

Algorithm 4: Multiplication

```
   input : M (an N^(k+1) MDIA matrix)
           L (an N^(k) MDIA matrix)
           PT (projection table)
   output: K (L ⊙ M)
1  N_diag ← number of distinct β in PT;
2  Ξ ← array of N_diag tuples;
3  D ← N_diag × N array of 0's;
4  for β ← 0 to N_diag − 1 do
       /* get PT lines associated with β                    */
5      foreach (α, β, γ) in PT.IDX_β[β] do
6          for i ← 0 to N − 1 do
7              j ← i + ι;                          /* Equation (12) */
8              if 0 ≤ j < N then
9                  D[β][j] ← D[β][j] + M.D[α][i] × L.D[γ][i];
10     Ξ[β] ← M.Ξ[α]^⊥ ;                           /* Equation (10) */
11 K ← MDIA(D, Ξ);
12 return K;
```

projected from diagonals of M, for each diagonal of K. Finally we create the MDIA format of K from $\mathcal{D}$ and Ξ and return the result (lines 11, 12).

Sparse diagonal matrix. As we mentioned in Section 4.2, the diagonal matrix $\mathcal{D}$ of a Markov transition matrix might be sparse, in which scenario, it is stored in CSR format. We propose another algorithm, summarized in Algorithm 5, to process multiplication between CSR diagonal matrices, i.e. $\mathcal{D}$ is represented by three 1-dimensional arrays A, IA, and JA (see Section 3). Specifically, Algorithm 5 employs a *two-pass* approach. In the first pass (lines 3-19), it computes IA, which stores the number of nonzero elements in each diagonal of K, and in the second pass (lines 20-43) it fills A and JA with the values and in-diagonal positions of the nonzero elements of K. Finally, we sort the element indexes within each diagonal in ascending order using the *sort_index*() method (line 44). The sorting is an important procedure that keeps the MDIA format valid during the multiplication operation. Implementation details of *sort_index*() are explained in [14, 12] and omitted here.

Complexity analysis. Given an $N^{(k)}$ matrix, if no sparse storage format is applied, both space and time complexity for processing the multiplication operation is $O(N^k)$. For example, consider a 100×100 uniform grid, thus $N = 10,000$. M_2 is a $10,000 \times 10,000 \times 10,000$ matrix which contains 10^{12} elements. If each element is a 4-byte float number, M_2 will require 3 terabytes memory. Processing a single multiplication operation will take hundreds of minutes on a 1GHz CPU. If M is in the MDIA format with ordinary diagonal matrix, the space and time complexities reduce to $O(N_{diag} \cdot N)$, where $N_{diag} \ll N^{k-1}$ denotes the number of diagonals in M. The diagonal matrix is stored in CSR format if it is sparse, i.e. the number of nonzero elements $NNZ \ll N \times N_{diag}$, in which scenario, the space and time complexities further reduce to $O(NNZ+N_{diag})$, which is the theoretical optimum since any exact (in contrast to probabilistic) algorithm must iterate through every single diagonal and nonzero element to perform the multiplication operations.

Algorithm 5: Multiplication (CSR)

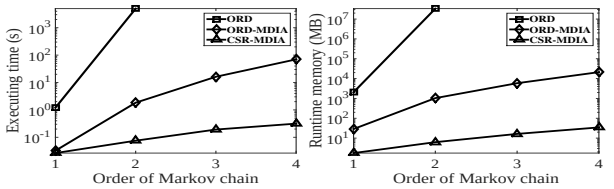
```
   input : M (an N^(k+1) MDIA matrix)
           L (an N^(k) MDIA matrix)
           PT (projection table)
   output: K (L ⊙ M)
1  N_diag ← number of distinct β in PT;
2  Ξ ← array of N_diag tuples;
   /* pass one:  compute IA                                */
3  IA ← array of N_diag + 1 0's;
4  mask ← array of N -1's;
5  nnz ← 0;
6  for β ← 0 to N_diag − 1 do
7      foreach (α, β, γ) in PT.IDX_β[β] do
8          ii ← M.IA[α], jj ← L.IA[γ];
9          while ii < M.IA[α + 1] and jj < L.IA[γ + 1] do
10             x ← M.JA[ii], y ← L.JA[jj];
11             if x = y then
12                 z ← x + ι;
13                 if mask[z] ≠ α then
14                     mask[z] ← α, nnz ← nnz + 1;
15                 ii ← ii + 1, jj ← jj + 1;
16             else if x > y then
17                 jj ← jj + 1;
18             else ii ← ii + 1;
19     IA[α + 1] ← nnz;
   /* pass two:  compute A and JA                          */
20 A ← array of nnz 0's, JA ← array of nnz 0's;
21 next ← array of N 0's, sums ← array of N -1's;
22 nnz ← 0;
23 for β ← 0 to N_diag − 1 do
24     head ← -2, len ← 0;
25     foreach (α, β, γ) in PT.IDX_β[β] do
26         ii ← M.IA[α], jj ← L.IA[γ];
27         while ii < M.IA[α + 1] and jj < L.IA[γ + 1] do
28             x ← M.JA[ii], y ← L.JA[jj];
29             if x = y then
30                 z ← x + ι;
31                 sums[z] ← sums[z] + M.A[ii] × L.A[jj]; if
                   next[z] = −1 then
32                     next[z] ← head, head ← z;
33                     len ← len + 1;
34                 ii ← ii + 1, jj ← jj + 1;
35             else if x > y then
36                 jj ← jj + 1;
37             else ii ← ii + 1;
38     for j ← 0 to len do
39         A[nnz] ← sums[head], JA[nnz] ← head;
40         nnz ← nnz + 1;
41         tmp ← head, head ← next[head];
42         next[tmp] ← -1, sums[tmp] ← 0;
43     Ξ[β] ← M.Ξ[α]^⊥
44 D ← csr_matrix(A, IA, JA);
45 sort_index(D);          /* sort diagonals in ascending order */
46 return MDIA(D, Ξ);
```

Table 3: Experimental settings

Space domain (m×m)	**50,000×50,000**
Number of trajectories	**50,000**
Grid size	32 × 32, 64 × 64, **128 × 128**, 256 × 256
Order of Markov chain	1, **2**, 3, 4
Sampling interval (s)	30, **60**, 90, 120
Pruning sensitivity	10^{-6}, 10^{-5}, . . . , 0,1, **0.2**, . . . , 0.5
Window length (km)	1, **2**, 3, . . . , 10
Query predict length (min)	10, **20**, 30, . . . , 60
Dataset	**Beijing**, Shenzhen

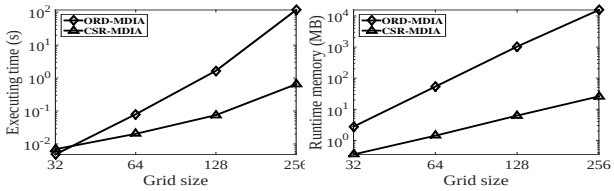
6. EXPERIMENTAL STUDY

In this section, we evaluated our proposed method and compare it with other existing methods. The experimental goal is to demonstrate that our method is effective and efficient in reducing the candidate set for predictive range queries. All algorithms are implemented with Python and C. The experimental machine is equipped with 2.6 GHz Intel Core i7 CPU and 16GB RAM and runs OSX 10.11.



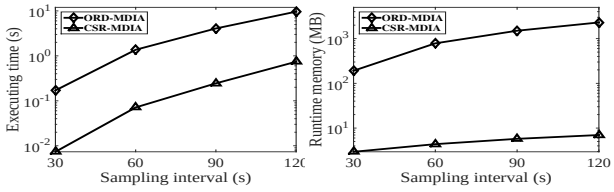
(a) Time cost (b) Space cost

Figure 6: Varying order of Markov chain



(a) Time cost (b) Space cost

Figure 7: Varying grid size



(a) Time cost (b) Space cost

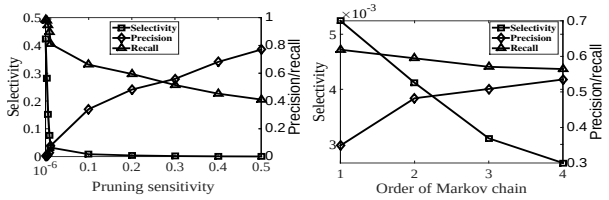
Figure 8: Varying sampling interval

6.1 Experimental setup

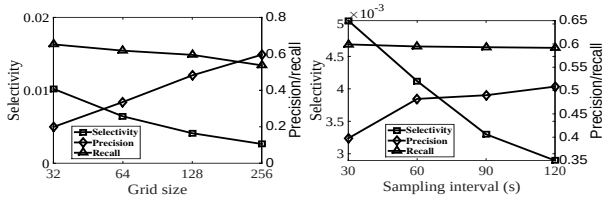
The experimental settings are shown in Table 3 where the default settings are boldfaced.

Datasets. We use two real world GPS tracking datasets: Beijing and Shenzhen. The Beijing dataset contains 15 million records collected from 10,358 taxis between Feb. 2, 2008 to Feb. 8, 2008 while the Shenzhen dataset contains 0.79 billion records collected from 26,572 taxis between March 20, 2014 to March 29, 2014. Each record includes the taxi ID, the location (longitude and latitude) and the timestamp. We select a 50,000 × 50,000 square meter area in the corresponding road networks as the space domain. We sample vehicle trajectories from the GPS records and perform interpolations to make all trajectories have equal sampling intervals and synchronized timestamps. From each dataset, we collect 50,000 one day long trajectories and divide them into two parts. The first part contains 40,000 trajectories that are used for training the models. The remaining 10,000 trajectories are prepared for testing. We sample 50,000 two hour long trajectories from the testing data and insert them into the T-Grids. Predictive range queries are randomly generated with the parameters specified in each experiment.

Evaluation metrics. We consider three metrics to evaluate the effect of our pruning methods: selectivity, precision and recall. Given the set of moving objects $\mathcal{O}$ with their trajectories and a predictive range query Q, selectivity equals $\frac{|\mathcal{O}'|}{|\mathcal{O}|}$, where $\mathcal{O}'$ is the candidate set and $|\cdot|$ denotes the cardinality of a set. Selectivity can be used to evaluate the pruning effect of a pruning method. Precision and recall are



(a) Pruning sensitivity (b) Order of Markov chain



(c) Grid size (d) Sampling interval

Figure 9: Pruning effects with different parameters

conventional measures of accuracy. Let $\mathcal{R}$ denote the result set of query Q. Precision equals $\frac{|\mathcal{O}' \cap \mathcal{R}|}{|\mathcal{O}'|}$ while recall equals $\frac{|\mathcal{O}' \cap \mathcal{R}|}{|\mathcal{R}|}$. In addition to these three metrics, we also evaluate the time and space costs of performing the pruning and query processing.

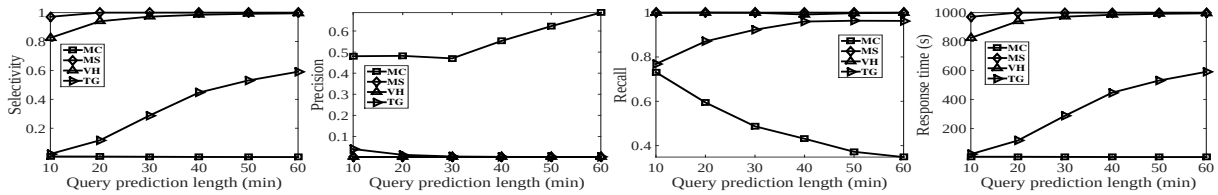
Competitors. We compared three alternative pruning methods with our proposed method. They are (1) Maximum speed (MS). This is a naive method that uses the maximum possible speed of all objects to enlarge the query window. (2) Velocity histogram (VH). This method is used in [7], which enlarges query windows by the maximum speed values of the objects in every grid cell that are stored in a 2-dimensional histogram. (3) Travel time grid (TG). This method is proposed in [5]. It tracks the average travel time between two grid cells, based on which the objects are not reachable to the query window within the prediction time are pruned.

6.2 Evaluation of our methods

We first evaluate the performance of our proposed method. We evaluate time and space costs of computing the CPTs and the pruning effect of Markov chains. We compare the baseline ordinary matrices (ORD) with two variants of the proposed MDIA format, MDIA with ordinary diagonal matrices (ORD-MDIA) and MDIA with CSR diagonal matrices (CSR-MDIA).

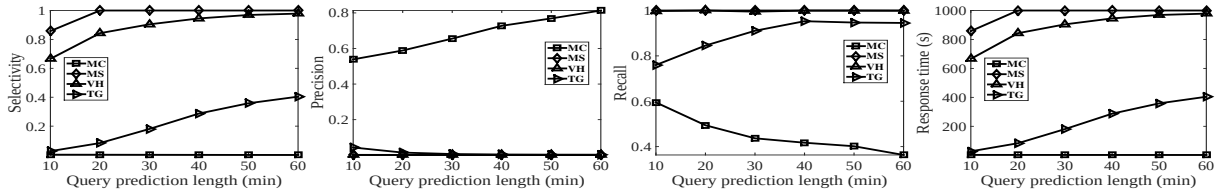
Time and space costs. Figure 6 shows time and space costs with varying order of Markov chains. We find that both execution time and memory rapidly grow as the order increases. Specifically, the size of Markov transition matrices grow exponentially, which makes ORD and ORD-MDIA infeasible with high-order Markov chains. However, CSR-MDIA is not as significantly affected by the order of Markov chains as other approaches. This is because the time and space complexity of CSR-MDIA is determined by the number of nonzero elements and that of occupied diagonals, which are far smaller than the size of the transition matrices.

Figure 7 shows the effect of varying grid sizes. We omit ORD here since it is not viable with order-2 (or greater) Markov chains. As before, both time and space requirements grow as grid size increases, since larger grids result in more states and larger transition matrices. Moreover, CSR-MDIA



(a) Selectivity (b) Precision (c) Recall (d) Response time

Figure 10: Varying query prediction length on Beijing dataset



(a) Selectivity (b) Precision (c) Recall (d) Response time

Figure 11: Varying query prediction length on Shenzhen dataset

significantly outperforms ORD-MDIA as grid size increases. Generally speaking, the order of Markov chain has more impact on the transition matrix than grid size, since growth in order increases the dimensions of the matrix instead of increasing the scope of each dimension.

Figure 8 shows the impact of varying the sampling interval. We find that larger sampling intervals lead to higher computation costs, since the transition matrices get denser as the sampling interval increases. It is noteworthy that CSR-MDIA can accomplish the computation for CPTs within one second and ∼10 megabytes of memory in all experimental settings.

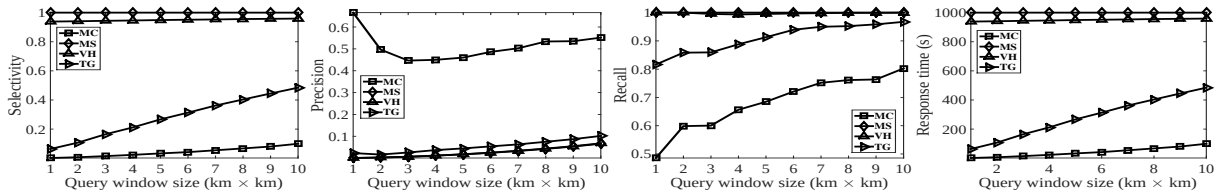
Pruning effect. In this experiment, we evaluate the pruning capability of Markov chains by comparing selectivity, precision and recall. The experimental results are summarized in Figure 9. Figure 9(a) shows the impact of pruning sensitivity (threshold), by varying its value from 10^{-6} to 0.5. We find that selectivity significantly reduces when pruning sensitivity increases. Moreover, precision climbs while recall drops as pruning sensitivity increases. The reason is that the higher the pruning sensitivity, the more objects are likely to be filtered out, resulting in smaller candidate set thus lower selectivity. Figure 9(b) shows the results with different orders of Markov chain. We find that both selectivity and recall significantly reduce while precision rises as order increases. Therefore, high-order Markov chains are more powerful in pruning while sacrificing prediction rate. Similarly, as shown in Figure 9(c), larger grid sizes (more finegrained grid cells) will result in fewer candidate paths, and thus result in lower selectivity and recall but higher precision. Similar trends with different sampling intervals are shown in Figure 9(d).

6.3 Comparison with other methods

In this set of experiments, we compare the performance of our proposed Markov chain based method (MC) with three existing methods, maximum speed (MS), velocity histogram (VH), and travel time grid (TG). We conduct experiments on both Beijing and Shenzhen datasets. Besides pruning effects, we also evaluate the overall query response time. We use the prediction approach introduced in [26] (a hidden Markov model based approach) as the $VERIFY()$ function in Algorithm 3.

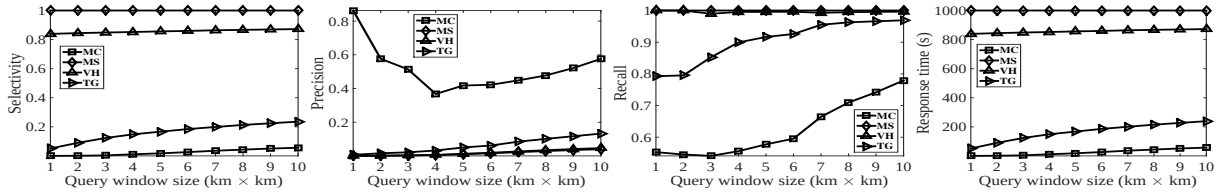
Query prediction length. We first study the impact of query prediction length and vary its value from 10 to 60 minutes. Figure 10 and 11 show the results on Beijing and Shenzhen datasets, respectively. We find that selectivity and response time grow as prediction length increases. This is because uncertainty of the object trajectories increases for longer term prediction, thus we include a larger portion of candidate objects. Moreover, the response time is nearly linear with selectivity, which implies that performing the $PREDICT()$ function with candidate objects dominates the execution time for answering predictive range queries. Precision drops as prediction time increases, since longer-term predictions are more challenging. Recall of our method drops as prediction time increases while those for other methods do not change much. This is because the main goal of our method is to accurately and efficiently reduce the candidate set with a trade-off of prediction rate, while other methods have less effects on pruning. However, this trade-off can be adjusted by the pruning sensitivity. Similar trends are displayed for Beijing and Shenzhen datasets. We find our method works a little better on Shenzhen data. This is probably because vehicle speeds in Shenzhen city are higher than those in Beijing city, thus the motions can be better captured by the transition probabilities. In summary, our method significantly outperforms other methods in terms of pruning capability.

Query window size. We also evaluate performance of the methods with different query window sizes. Figure 12 and 13 show the results on Beijing and Shenzhen datasets, respectively. Generally speaking, the pruning procedure becomes more effective as the query window increases. Similarly, we find that the response time is nearly linear to selectivity and our method works better on the Shenzhen dataset in different settings of query window sizes. Again, compared with other methods, our method enjoys significantly better performance in terms of selectivity, precision, and response time but scarifies recall.



(a) Selectivity (b) Precision (c) Recall (d) Response time

Figure 12: Varying query window size on Beijing dataset



(a) Selectivity (b) Precision (c) Recall (d) Response time

Figure 13: Varying query window size on Shenzhen dataset

7. CONCLUSIONS AND FUTURE WORKS

In this paper, we studied the usefulness of Markov chains as a pruning mechanism for long-term predictive range queries. In order to resolve the explosion of time and space costs arising with high-order Markov chains, we propose the MDIA format to store the Markov transition matrices. We also propose efficient algorithms for arithmetic operations involved in our pruning procedure with MDIA matrices. Extensive experiments illustrate that our proposed method enjoys excellent pruning performance and significantly outperforms other methods in terms of efficiency and precision.

To end this paper, we also propose some future works. First, we can extend the pruning mechanism to other kinds of spatio-temporal queries, e.g. k nearest neighbor query. We will also seek efficient formats for representing the n-step Markov transition matrices according to their sparse patterns. Moreover, the convergence property of n-step transition probabilities is another interesting problem in spatio-temporal settings. Pre-computing popular queries according to the workload will further improve performance of the system.

Acknowledgment

This research is supported by AFOSR DDDAS grant FA9550-15-1-0950 and NSF TWC grant IIS-1618932.

8. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[2] T. Emrich, H. Kriegel, N. Mamoulis, M. Renz, and A. Züfle. Querying uncertain spatio-temporal data. In *ICDE*, 2012.

[3] J. Froehlich and J. Krumm. Route prediction from trip observations. Technical report, SAE Technical Paper, 2008.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[5] A. M. Hendawi and M. F. Mokbel. Predictive spatio-temporal queries: a comprehensive survey and future directions. In *SIGSPATIAL*, 2012.

[6] E.-J. Im and K. A. Yelick. *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[7] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *VLDB*, 2004.

[8] J. Krumm. Real time destination prediction based on efficient routes. Technical report, SAE Technical Paper, 2006.

[9] S. P. Meyn and R. L. Tweedie. *Markov chains and stochastic stability*. Springer Science & Business Media, 2012.

[10] A. Monreale, F. Pinelli, R. Trasarti, and F. Giannotti. Wherenext: a location predictor on trajectory pattern mining. In *SIGKDD*, 2009.

[11] M. Morzy. Mining frequent trajectories of moving objects for location prediction. In *MLDM*, 2007.

[12] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[13] S. Ray, R. Blanco, and A. K. Goel. Supporting location-based services in a main-memory database. In *MDM*, 2014.

[14] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[15] A. Sadilek and J. Krumm. Far out: Predicting long-term human mobility. In *AAAI*, 2012.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, 2000.

[17] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys. Trees or grids?: indexing moving objects in main memory. In *SIGSPATIAL*, 2009.

[18] D. Sidlauskas, S. Saltenis, and C. S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *SIGMOD*, 2012.

[19] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD*, 2004.

[20] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[21] X. Xu, L. Xiong, and V. S. Sunderam. D-grid: An in-memory dual space grid index for moving object databases. In *MDM*, pages 252–261, 2016.

[22] X. Xu, L. Xiong, V. S. Sunderam, J. Liu, and J. Luo. Speed partitioning for indexing moving objects. In *SSTD*, 2015.

[23] A. Y. Xue, R. Zhang, Y. Zheng, X. Xie, J. Huang, and Z. Xu. Destination prediction by sub-trajectory synthesis and privacy protection against such prediction. In *ICDE*, 2013.

[24] G. Yavas, D. Katsaros, Ö. Ulusoy, and Y. Manolopoulos. A data mining approach for location prediction in mobile environments. *Data Knowl. Eng.*, 54(2), 2005.

[25] J. J. Ying, W. Lee, T. Weng, and V. S. Tseng. Semantic trajectory mining for location prediction. In *SIGSPATIAL*, 2011.

[26] J. Zhou, A. K. H. Tung, W. Wu, and W. S. Ng. A "semi-lazy" approach to probabilistic path prediction. In *SIGKDD*, 2013.