Succinct Trit-array Trie for Scalable Trajectory Similarity Search

Shunsuke Kanda  
RIKEN AIP  
shunsuke.kanda@riken.jp

Keisuke Fujii  
Nagoya University  
fujii@i.nagoya-u.ac.jp

Koh Takeuchi  
Kyoto University  
takeuchi@i.kyoto-u.ac.jp

Yasuo Tabei  
RIKEN AIP  
yasuo.tabei@riken.jp

ABSTRACT
Massive datasets of spatial trajectories representing the mobility of a diversity of moving objects are ubiquitous in research and industry. Similarity search of a large collection of trajectories is indispensable for turning these datasets into knowledge. Current methods for similarity search of trajectories are inefficient in terms of search time and memory when applied to massive datasets. In this paper, we address this problem by presenting a scalable similarity search for Fréchet distance on trajectories, which we call trajectory-indexing succinct trit-array trie (tSTAT). tSTAT achieves time and memory efficiency by leveraging locality sensitive hashing (LSH) for Fréchet distance and a trie data structure. We also present two novel techniques of node reduction and a space-efficient representation for tries, which enable to dramatically enhance a memory efficiency of tries. We experimentally test tSTAT on its ability to retrieve similar trajectories for a query from large collections of trajectories and show that tSTAT performs superiorly with respect to search time and memory efficiency.

KEYWORDS
Trajectory data mining, scalable similarity search, Fréchet distance, succinct data structures

1 INTRODUCTION
With advances in location-acquisition technology and mobile computing, spatial trajectory data representing the mobility of a diversity of moving objects such as people, vehicles, and animals are ubiquitous in research and industry [47]. For example, a city monitoring system with the global positioning system (GPS) enables us to record a huge number of complex trajectories from vehicles [3]. A camera-based tracking system such as SportVU also enables us to collect a large number of fast and dynamic movements of sports players precisely [1]. There is, therefore, a strong and growing demand for developing new powerful methods to make the best use of huge collections of trajectories toward data mining applications such as predicting a taxi demand in a city in the near future and making an effective game strategy in sports.

Searching for similar trajectories in a large collection for a given query is an example of effective use of huge collections of trajectories, and it has a wide variety of applications from route recommendation [31, 39] to sports play retrieval and clustering [23, 37]. Several distance measures for trajectories (e.g., Fréchet distance, dynamic time warping, and longest common subsequence [41]) have been proposed thus far. Fréchet distance is the defacto standard measure [4] to evaluate the similarity of trajectories. Fréchet distance can be intuitively explained as the length of the shortest leash enabling a person to walk a dog such that they move along each of their own trajectories at their own speed without going back, and it has successfully been used in various applications including detection of commuting patterns [10], handwriting recognition [40], protein structure alignment [44], and other applications [12, 28, 48]. In addition, an algorithm competition focusing on similarity searches of trajectories using Fréchet distance was held in the ACM SIGSPATIAL Cup 2017 [43]. While these applications of Fréchet distance show its practicality, similarity search of trajectories using Fréchet distance is computationally demanding because computing the Fréchet distance between a pair of trajectories is quadratic to their length [9], which limits large-scale applications using Fréchet distance in practice.

Locality sensitive hashing (LSH) for Fréchet distance has been proposed for scalable similarity searches of trajectories using Fréchet distance. The first LSH method proposed by Indyk [24] approximates Fréchet distance via a product metric. Recently, more efficient LSH methods with tighter approximation bounds have been proposed [6, 13, 14], which map trajectories into non-negative integer vectors (called sketches) such that Fréchet distance is preserved as the Hamming distance among sketches. LSH for Fréchet distance is expected to enhance the scalabilities of similarity searches from a large collection of trajectories.

FRESH [13] is the state-of-the-art method applying LSH for Fréchet distance to similarity searches of trajectories in practice. FRESH uses an inverted index implemented by the hash table data structure, which stores values associated with each key. For a collection of sketches of fixed length $L$, FRESH builds $L$ hash tables whose key is an integer at each position of sketches and value is the set of sketch identifiers with the integer. Given a query sketch of the same length $L$, FRESH retrieves the hash table with the integer at each position of the query and computes a set of sketch identifiers with the integer at each position, resulting in $L$ sets of sketch identifiers in total for $L$ positions of the query. The Hamming distances between each sketch in the set and the query are computed by sequentially counting the number of appearances of each integer in these sets. However, this approach suffers from performance degradation caused by the sequential counting of integers if large sets of identifiers are computed. In addition, FRESH consumes a large amount of memory for indexing large collections of trajectories because of the memory inefficiency of hash tables, which limits large-scale applications in practice. Thus, an important open challenge is to develop a memory-efficient data structure for fast similarity searches of sketches for trajectories.
Trie [18] is an ordered labeled tree data structure for a set of strings and supports various string operations such as string search and prefix search with a wide variety of applications in string processing. Examples are string dictionaries [27], n-gram language models [35], and range query filtering [45]. A typical pointer-based representation of trie consumes a large amount of memory. Thus, recent researches have focused on space-efficient representations of trie (e.g., [27, 35, 45]). To date, trie has been applied only to the limited application domains listed above. However, as we will see, there remains great potential for a wide variety of applications related to trajectories.

Our Contribution. We present a novel method called trajectory-indexing succinct trie array trie (tSTAT) that efficiently performs a similarity search for trajectories with LSH for Fréchet distance. As in FRESH, multiple data structures according to the sketch length are built for efficient query retrievals. However, unlike FRESH, tSTAT enables faster similarity searches by effectively partitioning sketches into several blocks by the multi-index approach [22] and building the trie data structure for each block. tSTAT’s query retrieval is performed by an efficient trie traversal that bypasses the sequential counting of integers for computing Hamming distance. While preserving fast similarity searches, tSTAT successfully reduces the memory usage by presenting two novel techniques of node reduction and a space-efficient representation of trie by leveraging succinct data structures (i.e., compressed representations of data structures while supporting various data operations in the compressed format) [26, 34].

We experimentally test tSTAT on its ability to retrieve similar trajectories for a query from large collections of real-world trajectories. The performance comparison with state-of-the-art similarity search over sketches in the research areas of trajectory data analysis and data mining. We briefly review the state-of-the-art methods, which are also summarized in Table 1.

3 LITERATURE REVIEW

Several efficient data structures have been proposed for similarity search over sketches in the research areas of trajectory data analysis and data mining. We briefly review the state-of-the-art methods, which are also summarized in Table 1.

3.1 Related Work on Trajectory Data Analysis

While several methods for exact similarity searches of trajectories have been proposed in trajectory data analysis [7, 11, 15], they are slow due to the large cost for computing Fréchet distance. Thus, recent researches have focused on approximate similarity searches using LSH for Fréchet distance.

The first data structure for approximate nearest neighbor searches on trajectories was proposed by Indyk [25]. While the search time is \(O(nO(1) \log n)\) for the number of trajectories \(n\) and the maximum length of trajectory \(m\), it consumes a large amount of memory in \(O(|A|\sqrt{m(n\sqrt{m}\log n)^2})\) words for the size of the domain \(|A|\) on which the trajectories are defined, which limits practical applications of the data structure. Later, improved data structures [6, 14] were presented, and they take \(O(n \log n + mn)\) memory words while performing a similarity search in \(O(m \log n)\) time.

FRESH [13] is a practical data structure for approximate similarity search of trajectories, and it uses \(L\) inverted indexes with hash tables. The search time of FRESH is \(O(L \cdot \max(1, n/(\sigma)) + C_{\text{hash}})\) assuming that the hash table has a uniform distribution, where \(C_{\text{hash}}\) is the verification time for candidates. The memory usage is \(O(Ln \log n + Ln \log \sigma)\) bits.
### 3.2 Related Work on Data Mining

In data mining, recent similarity search methods for sketches use the multi-index approach [22] as follows. The approach partitions sketches in S into several blocks of short sketches and builds multiple inverted indexes from the short sketches in each block. The similarity search consists of two steps: filtering and verification. The filtering step roughly obtains candidates of similar sketches assuming that sketches are uniformly distributed.

Several data structures with the multi-index approach were presented specifically for similarity searches on binary sketches (i.e., σ = 2) [20, 33, 36]. These methods are not directly applicable to similarity searches on integer sketches (i.e., σ > 2). HmSearch [46] is an efficient similarity search with the multi-index approach for integer sketches. It generates sketches similar to the ones in S and adds them to S, which enables fast similarity searches with hash tables. The search time is \( O(L \cdot \max(1, \frac{L}{\sigma L - 2L/K + 1})) + C_{hms} \) assuming that the sketches in S are uniformly distributed, where \( C_{hms} \) is the verification time for candidates. HmSearch needs a large memory consumption of \( O(L \cdot \max(1, \frac{L}{\sigma L - 2L/K + 1})) \) bits because of the amplification of the sketches in S. Although several similarity search methods applicable to integer sketches have been proposed (e.g., [29, 30, 32]), HmSearch performs the best [46].

Despite the importance of a scalable similarity search for massive trajectories, no previous work achieving fast search and memory efficiency in trajectory data analysis and data mining exists. As reviewed in this section, the current best possible methods are (i) to use L inverted indexes with hash tables or (ii) to use the multi-index approach with hash tables. However, those methods scale poorly due to the large computation cost of the search algorithm and memory inefficiency of the data structure.

### 4 TRAJECTORY-INDEXING SUCCINCT TRIT-ARRAY TRIE

tSTAT proposes to use an efficient approach called multi-index [22] for solving the similarity search problem of sketches on the Hamming space. Each sketch \( S_i \in S \) is partitioned into B disjoint blocks \( S^1_i, S^2_i, \ldots, S^B_i \) of fixed length L/B. For convenience, we assume that L is divisible by B. The j-th block of S for \( j = 1, 2, \ldots, B \) is denoted by \( S^j = \{ S^j_1, S^j_2, \ldots, S^j_n \} \). Figure 1 shows an example collection of six sketches \( S \), and S is partitioned into two blocks \( S^1 \) and \( S^2 \).

For the similarity search for query T, we first partition T into B blocks \( T^1, T^2, \ldots, T^B \) in the same manner. We then assign smaller Hamming distance thresholds \( K^1, K^2, \ldots, K^B \) to each block, which is detailed later. We obtain the candidate sets of sketch ids \( C^j = \{ i | \text{Ham}(S^j_i, T^j) \leq K^j \} \) for each block \( j = 1, 2, \ldots, B \) and compute the union of the candidate sets as \( C = C^1 \cup C^2 \cup \cdots \cup C^B \). Finally, we remove false positives by computing Ham(Si, T) for each i ∈ C, resulting in solution set I.

The threshold assignment requires \( K^j \) such that each solution in I is included in \( C^j \) for any \( j = 1, 2, \ldots, B \). Such assignment is satisfied by setting \( K^j \) such that \( K^1 + K^2 + \cdots + K^B = K - B + 1 \), which is ensured by the general pigeonhole principle [36].

For the similarity search with \( K = 3 \) in Figure 1, we assign \( K^1 = 1 \) and \( K^2 = 1 \) such that \( K^1 + K^2 = K - B + 1 = 2 \) based on the general pigeonhole principle. Then, \( C^1 = \{ 1, 4 \} \) and \( C^2 = \{ 5, 6 \} \) are obtained. Since \( I = \{ 1, 4, 5 \} \) is a subset of \( C = \{ 1, 4, 5, 6 \} \), we can obtain I by verifying each element in C.

### 4.2 Trie Implementing Multi-Index

Trie \( X^j \) is an edge-labeled tree indexing sketches in \( S^j \) (see Figure 2). Each node in \( X^j \) is associated with the common prefix of a
subset of sketches in $S$, and the root (i.e., node without a parent) is not associated with any prefix. Each leaf (i.e., node without any children) is associated with input sketches of the same integers and has the list of their sketch ids. All outgoing edges of a node are labeled with distinct integers. For a node $u$ in $X^j$, the number of edges from the root to $u$ is the level of $u$. Figure 2 shows an example of $X^1$ and $X^2$ built from $S^1$ and $S^2$ in Figure 1, respectively.

Searching similar sketches for query sketch $T^j$ and threshold $K^j$ is performed by traversing trie $X^j$ in a depth-first manner. We start to traverse nodes from the root at level 0. At each level $ℓ$, we compare the $ℓ$-th integer of $T^j$ with labels associated with outgoing edges from nodes at level $ℓ$. For each node $u$ at level $ℓ$, we compute the Hamming distance $\text{dist}_u$ between the sequence of edge labels from the root to $u$ and the subsequence from the first position to the $ℓ$-th position in $T^j$. If $\text{dist}_u$ becomes more than $K^j$ at each reached node $u$, we can safely stop traversing down to all the descendants under node $u$. The candidate set $C^j$ can be obtained by accessing the list of sketch ids at each reached leaf $v$, which means $\text{dist}_v \leq K^j$.

In Figure 2, $\text{dist}_u$ for each node $u$ and query $T^j$ or $T^2$ is represented in each node. When $K^1 = 1$ and $K^2 = 1$, we stop traversing at each node including a red number. We obtain candidates $C^1 = \{1, 4\}$ and $C^2 = \{5, 6\}$, associated with the reached leaves.

The algorithm can prune unnecessary portions of the search space depending on $K^j$. The number of traversed nodes in $X^j$ is bounded by $O((L/B)^{K^j+2})$ when assuming the complete $σ$-ary trie [5]. When we assign $K^j$ in a round robin manner, it holds $K^j \leq K/B$; thus, the algorithm takes $O((L/B)^{K/B+2} + |C^j|)$ time for $X^j$.

When we represent trie $X^j$ by using pointers straightforwardly, the scalability becomes a critical issue since the memory usage is $O(N^j \log(σN^j))$ bits. In the remaining subsections, we present two techniques of node reduction and STAT data structure for compactly representing $X^j$.

### 4.3 Node Reduction

A crucial observation for candidate set $C$ is that it allows false positives for solution set $J$ for the Hamming distance problem. This observation enables us to make tries more memory efficient by removing their redundant nodes.

The weight of node $u$ is defined as the total number of sketch ids associated with leaves of the subtrie with root $u$ and satisfies an anti-monotonicity, i.e., the weight of node $u$ is no less than the one of $u$’s child. The weight of the root in trie $X^j$ is $n$.

Our node reduction is a depth-first traversal leveraging the anti-monotonicity. The algorithm starts from the root and visits each node. If the weight of node $u$ is no more than hyper-parameter $λ$, the algorithm eliminates the subtree with root $u$ from the tree. Then, $u$ becomes a new leaf of the tree and has the sketch ids of previous leaves associated with the subtree with root $u$.

Figure 3 shows an example of node reductions with $λ = 1$ and $λ = 2$ for trie $X^2$ in Figure 2. The trie with 16 nodes in Figure 2b is reduced to the tries with 11 nodes in Figure 3a and 6 nodes in Figure 3b by the node reduction algorithm with $λ = 1$ and $λ = 2$, respectively.

$λ$ is a trade-off parameter that can control the balance between the tree traversal time and the verification time for removing false positives. The larger $λ$ becomes, the smaller the number of nodes becomes and the larger the number of false positives becomes, resulting in a smaller tree traversal time and a larger verification time. This trade-off is investigated with various $λ$ in Section 5.

### 4.4 Succinct Rank and Select Data Structures

STAT is an integer array representation of trie leveraging succinct data structures [26].

Given an integer array $A$ of length $M$, the data structures support the following operations in compressed space:

- $\text{Rank}_c(A, i)$ returns the number of occurrences of integer $c$ between positions 0 and $i−1$ on $A$.
- $\text{Select}_c(A, i)$ returns the position in $A$ of the $i$-th occurrence of integer $c$; if $i$ exceeds the number of $c$’s in $A$, it always returns $M$.

For $A = (0, 1, 0, 2, 1, 1, 2, 0)$, $\text{Rank}_0(A, 3) = 2$ and $\text{Select}_3(A, 1) = 6$.

For a bit array $A$ consisting of integers from $[0, 1]$, $A$ can be stored in $M + o(M)$ bits of space while supporting the operations in $O(1)$ time [26, 42]. For a bit array $A$ consisting of integers from $[0, 1, 2]$, $A$ can be stored in $M \log_2 3 + o(M)$ bits of space while supporting the operations in $O(1)$ time [17, 34].

### 4.5 STAT Data Structure

STAT consists of three arrays $H$, $G$, and $V$ for compactly representing trie $X^j$ (see Figure 4). To define the arrays, we define the orders of internal nodes and leaves at each level, as follows.

We traverse $X^j$ in a breadth-first order. The $i$-th internal node at a level is the $i$-th encountered node for internal nodes at the level.
Similarly, the \( i \)-th leaf at a level is the \( i \)-th encountered node for leaves at the level. The node orders start at the zeroth. For example, level 2 of \( X^2 \) in Figure 3a consists of the zeroth internal node, the first internal node, and the zeroth leaf from top to bottom.

\( H_t \) is a trit (i.e., ternary digit) array and represents children of internal nodes at level \( \ell \). The children of an internal node are represented in a trit array of length \( \sigma \), and \( H_t \) is constructed by concatenating such trit arrays in the order of internal nodes. That is, the subarray of \( H_t \) between positions \( i \sigma \) and \((i+1)\sigma - 1 \) corresponds to the \( i \)-th internal node at level \( \ell \). The data structure is as follows:

- \( H_t[i\sigma + c] = 0 \) if the \( i \)-th internal node at level \( \ell \) does not have a child indicated with edge label \( c \) in the trie.
- \( H_t[i\sigma + c] = 1 \) if the \( i \)-th internal node at level \( \ell \) has a child indicated with edge label \( c \) as an internal node, and
- \( H_t[i\sigma + c] = 2 \) if the \( i \)-th internal node at level \( \ell \) has a child indicated with edge label \( c \) as a leaf.

\( H_t \) is implemented by the Rank data structure for trits. Let us consider the zeroth internal node at level 1 in Figure 4, \( H_t[0] = 1 \) and \( H_t[1] = 1 \) because it has two children as internal nodes indicated with edge labels 0 and 1; \( H_t[2] = 0 \) because it does not have a child indicated with edge label 2; and \( H_t[3] = 2 \) because it has a child as a leaf indicated with edge label 3.

\( G_t \) is a bit array representing the numbers of sketch ids associated with leaves at level \( \ell \) in unary encoding. That is, for \( g \) sketch ids associated with a leaf, the number \( g \) is encoded into \((g - 1)0s\) following 1. \( G_t \) is constructed by concatenating the unary codes from the zeroth leaf at level \( \ell \) and is implemented by the Select data structure for bits. In Figure 4, \( G_t = (1, 0, 1, 1) \) because the zeroth, first, and second leaves at level 4 have two, one, and one sketch ids, respectively.

\( V_t \) is an array of sketch ids associated with leaves at level \( \ell \) and is constructed by the sketch ids from the zeroth leaf at level \( \ell \). In Figure 4, \( V_t = (5, 6, 2, 1) \) because the zeroth, first, and second leaves at level 4 have sketch ids (5, 6), (2), and (1), respectively.

We present operations for computing children for a given internal node and sketch ids associated with a given leaf. Those operations are necessary for computing Hamming distances for a query on trie \( X^j \). Given the \( i \)-th internal node at level \( \ell \) and integer \( c \), there is not a child indicated with edge label \( c \) if \( H_t[i\sigma + c] = 0 \); if \( H_t[i\sigma + c] = 1 \), the child is the \( i' \)-th internal node at level \( \ell + 1 \), where \( i' = \operatorname{Rank}(H_t, i\sigma + c) \); if \( H_t[i\sigma + c] = 2 \), the child is the \( i' \)-th leaf node at level \( \ell + 1 \), where \( i' = \operatorname{Rank}_2(H_t, i\sigma + c) \). Given the \( i \)-th leaf node at level \( \ell \), the associated sketch ids are the elements between \( s \) and \( e \) in \( V_t \), where \( s = \operatorname{Select}_1(G_t, i) \) and \( e = \operatorname{Select}_1(G_t, i + 1) - 1 \).

In Figure 4, for the zeroth internal node at level 1, the child indicated with edge label 1 is the first internal node at level 2 because \( H_t[0] = 1 \) and \( \text{Rank}_1(H_t, 1) = 1 \); the child indicated with edge label 3 is the zeroth leaf at level 2 because \( H_t[2] = 0 \) and \( \text{Rank}_1(H_t, 3) = 3 \). For the zeroth leaf at level 4, the associated sketch ids are elements \( V_t[0] = 5 \) and \( V_t[1] = 6 \) because \( \operatorname{Select}_1(G_t, 1) = 0 \) and \( \operatorname{Select}_1(G_t, 2) - 1 = 1 \).

**Analysis for Memory Efficiency.** Let \( N_{\text{in}}^{j} \) denote the number of internal nodes in trie \( X^j \). The total length of \( H_t \) is \( \sigma N_{\text{in}}^{j} \), and the total memory usage of \( H_t \) is \( \sigma N_{\text{in}}^{j} \log_2 3 + o(\sigma N_{\text{in}}^{j}) = O(\sigma N_{\text{in}}^{j}) \) bits. The total length of \( G_t \) is \( n \). The total memory usage of \( G_t \) and \( V_t \) is \( n + o(n) + n \log_2 n = O(n \log n) \) bits.

The information-theoretic lower bound (ITLB) is the defacto standard criteria for investigating the memory efficiency of data structures and is defined as the minimum memory usage for representing tries. We analyze the memory efficiency of STAT by comparing the memory usage of STAT with that of ITLB for tries.

We ignore \( G_t \) and \( V_t \) in the comparison because the arrays are independent from trie data structures. The memory usage of STAT for a trie with \( N_{\text{in}}^{j} \) internal nodes is \( \sigma N_{\text{in}}^{j} \log_2 3 + o(\sigma N_{\text{in}}^{j}) \) bits. ITLB for a trie with \( N_{\text{in}}^{j} \) nodes is \( N_{\text{in}}^{j}(\sigma \log_2 \sigma - (\sigma - 1) \log_2(\sigma - 1)) \) bits [8]. Roughly, STAT becomes smaller than ITLB in that

\[
\frac{\sigma \log_2 3}{\sigma \log_2 \sigma - (\sigma - 1) \log_2(\sigma - 1)} < \frac{N_{\text{in}}^{j}}{N_{\text{in}}^{j}}.
\]

For example, when \( \sigma = 256 \), STAT becomes smaller in \( 43 < \frac{N_{\text{in}}^{j}}{N_{\text{in}}^{j}} \). The comparison shows that STAT is efficient for representing tries with large \( N_{\text{in}}^{j}/N_{\text{in}}^{j} \).

\( N_{\text{in}}^{j}/N_{\text{in}}^{j} \) is related to the average number of children for each internal node, i.e., the load factor of \( H_t \). Since nodes at a deeper level in a trie have fewer children, STAT is more effective for representing shallow tries. Thus, applying node reduction to a trie for eliminating trie nodes at deep levels enhances the space efficiency of STAT, as demonstrated in Section 5.

### 4.6 Complexity Analysis

The space complexity of tSTAT is derived as follows. Let \( N_{\text{in}} = \sum_{j=1}^{B} N_{\text{in}}^{j} \). B STAs are represented in \( O(\sigma N_{\text{in}} + Bn \log n) \) bits of space, where the left term is for \( H_t \) and the right term is for \( G_t \) and \( V_t \). In addition, we need to store collection \( \mathcal{S} \) in \( O(Ln \log \sigma) \) bits of space for verifying \( C \).

All the methods in Table 1 require \( O(Ln \log \sigma) \) bits of space to store \( \mathcal{S} \). In addition to the space, tSTAT uses \( O(\sigma N_{\text{in}} + Bn \log n) \) bits of space although FRESH and HmSearch use \( O(Ln \log n) \) bits of space. The factor of \( O(\log n) \) is obviously large for a massive collection with large \( n \). tSTAT can relax the large factor to \( B/L \). Although tSTAT needs \( O(\sigma N_{\text{in}}) \) bits of space, \( N_{\text{in}} \) can be reduced by the node reduction approach.

The time complexity of tSTAT is derived by simply assuming \( \lambda = 0 \). The algorithm consists of traversing nodes and verifying candidates of \( C \). The traversal time is \( O(B(L/B)^{K/B+2}) \) as presented in Section 4.2. The verification time \( \text{C}_{\text{stat}} \) contains the times of removing duplicated candidates in \( C^1, C^2, \ldots, C^B \) and verifying candidates in \( C \).
Table 2: Statistics of datasets. Min, Max, Mean, and Median regarding trajectory length are presented.

| Dataset | Number | Min | Max | Mean | Median |
|---------|--------|-----|-----|------|--------|
| Taxi    | 1,704,769 | 1,3881 | 48.9 | 41 |
| NBA     | 3,288,264 | 898 | 85.3 | 73 |
| OSM     | 19,113,525 | 2,000 | 13.3 | 7 |

5 EXPERIMENTS

We evaluated the performance of tSTAT through real-world trajectory datasets. We used three datasets with $d = 2$, as shown in Table 2. Taxi is 1.7 million trajectories of 442 taxis driving in the city of Porto for one year [3]. The GPS locations (i.e., latitude and longitude) were recorded every 15 seconds with mobile data terminals installed in the taxis. Each trajectory represents a taxi trip taken by a passenger. NBA is 3.3 million trajectories of 636 games in the 2015/16 NBA seasons [1]. Ten player locations were captured every 40 milliseconds by SportVU player tracking system. Each trajectory is segmented by stationary points (i.e., small moving distances). OSM is 19 million trajectories of various moving objects including cars, bikes and humans traced by GPS in Western United States, and it is created from OpenStreetMap project [2]. OSM is the largest dataset of 19 million trajectories among three datasets and enables us to evaluate the performance of similarity search methods on a huge dataset.

For each dataset, we sampled 1,000 trajectories as queries and excluded them from the collection. We selected Fréchet distance thresholds $R$ such that 1, 10, and 100 solutions are found on average per query, respectively, which resulted in $R = 567, 2720, 7263$ for Taxi, $R = 0.15, 0.26, 0.45$ for NBA, and $R = 899, 2506, 7824$ for OSM. LSH for Fréchet distance has two parameters of hash functions $\delta$ and $k$ [13]. Following the original paper [13], those parameters were set to $\delta = 8dR$ and $k = 1$.

We conducted all experiments on one core of quad-core Intel Xeon CPU E5-2680 v2 clocked at 2.8 GHz in a machine with 256 GB of RAM, running the 64-bit version of CentOS 6.10 based on Linux 2.6. We implemented all data structures in C++17 and compiled the source code with g++ (version 7.3.0) in optimization mode -O3. We implemented succinct Rank and Select data structures on bits using the succinct data structure library [19]. As far as we know, there is no any available implementation on trits; therefore, we developed a practical implementation of the data structures for trits. For a fast computation of Hamming distance on integers, we used an efficient computation algorithm exploiting a bit-parallelism technique [46]. See Appendix A for the implementation details. The source code used in our experiments is available at https://github.com/kam persanda/frechet_simsearch

5.1 Performance of LSH

We analyzed the performance of LSH for Fréchet distance while varying $L$ and $\sigma$. All the combinations of $L = 32, 64, 128$ and $\sigma = 2^1, 2^2, 2^3$ were tested. Setting $\sigma$ to $2^1, 2^2$ and $2^3$ means that the resulting sketches are binary, byte, and four-byte strings, respectively. For each combination of $L$ and $\sigma$, we computed recalls and precisions by varying Hamming distance threshold $K$ from 0 to $L/4$ with step width $L/32$.

Figure 5 shows recall-precision curves on Taxi and NBA. Each recall value (or precision value) was averaged per query. Overall, the larger Hamming distance thresholds were used, the larger the recall values became and the smaller the precision values became on each dataset. Under each fixed $L$, the recall and precision values for $\sigma = 2^8$ were better than those for $\sigma = 2^1$ on each dataset while the recall and precision values for $\sigma = 2^8$ were almost the same as those for $\sigma = 2^3$, which showed $\sigma = 2^8$ was reasonable for achieving high recall and precision values of similarity searches using LSH. Under each fixed $\sigma$, $L = 64$ achieved reasonable recall and precision values for similarity searches using LSH. In addition, $L = 64$ enables us to efficiently implement the fast algorithm for computing Hamming distance [46] which is applicable to not only tSTAT but also HmSearch and linear search. Thus, the following experiments were performed using $\sigma = 2^8$ and $L = 64$.

5.2 Efficiency of Node Reduction and STAT Data Structure

We evaluated the efficiency of node reduction in Section 4.3 and STAT data structure in Section 4.5 in terms of the memory efficiency and search time of tSTAT for solving the Hamming distance problem. We measured (a) the number of internal nodes $N_{in}$, (b) the memory usage, (c) the number of candidates $|C|$, and (d) the average search time per query by testing parameters $\lambda = 0, 2, 8, 32, 128, 512$. Setting $\lambda = 0$ results in no node being eliminated by the node reduction, i.e., the original trie is represented by the STAT data structure. All the combinations of block numbers $B = 8, 16$ and Hamming distance thresholds $K = 2, 4, 6, \ldots, 12, 14$ were tested. Figure 6 shows the experimental results on NBA for $R = 0.45$. The results with other parameter settings are presented in Appendix B.2.
Those results showed that setting $\lambda = 8$ for each method. The result showed tSTAT was the fastest for most results and demonstrated the efficiency of our search algorithm on tries with the multi-index approach. FRESH was slow because the implementation takes $O(n)$ time for $c_{\text{fresh}}$. For over 90% recalls, tSTAT was at most $60\times$, $34\times$, and $1600\times$ faster than FRESH on Taxi, NBA, and OSM, respectively. For over 90% recalls, tSTAT was at most $3.4\times$, $12\times$, and $2.3\times$ faster than HmSearch on Taxi, NBA, and OSM, respectively.

We also evaluated the scalability of the methods by varying the collection size $n$. We randomly sampled 20% to 100% of the trajectories from each dataset. Figure 7d shows the fastest search times with more than 90% recall for each method on NBA. tSTAT was much faster than LS and FRESH for a wide range of collection sizes $n$ and Fréchet distance thresholds $R$. While HmSearch was also fast for small $R$, the time efficiency was decreased for large $R$. This is because, as described in Section 3.2, HmSearch stores and retrieves many sketches additionally generated, resulting in amplifying the number of candidate sketches that need to be verified (i.e., the cost $c_{\text{total}}$ in Table 1) for large $R$.

FIGURES 6a–6b show the number of internal nodes $N_{\text{in}}$ and the memory usage of tSTAT, respectively. To observe the memory efficiency of tSTAT, Figure 6b also shows the ITLB estimated from the number of nodes $N = \sum_{j=1}^{K} N_j$. As $\lambda$ grew, the number of internal nodes $N_{\text{in}}$ and the memory usage were dramatically reduced. As shown in Section 4.5, since the value $N/N_{\text{in}}$ increased, the memory usage of tSTAT approached the ITLB. The reduction of the number of internal nodes and the memory usage converged at around $\lambda = 32$.

FIGURES 6c and 6d show the number of candidates $|C|$ and the search time, respectively. Since the search time was affected by the number of candidates, both figures showed a similar tendency. For $B = 8$ and $K \geq 8$, as $\lambda$ grew, the number of candidates and search time also increased. However, the effect of $\lambda$ for the number of candidates and search time also converged at around $\lambda = 32$. Those results showed that setting $\lambda = 8$ or 32 was beneficial for improving the search and memory efficiencies of tSTAT. For $\lambda = 8$, $R = 0.45$, and $B = 8$, tSTAT achieved an average search time of 5 milliseconds and a memory usage of 0.36 GiB. The next subsection shows the efficiency of tSTAT with $\lambda = 8$ and tSTAT can achieve a memory efficiency and fast search time.

5.3 Efficiency of tSTAT

We compared tSTAT with HmSearch, FRESH, and linear search (LS) in the computation of $I^\prime$. We fixed $L = 64$ for sketches. The parameters of LSH for Fréchet distance were set to $\sigma = 2^3$, $\delta = 8dR$, $k = 1$, and $K = 2, 4, 6, \ldots, 12, 14$, which were used in tSTAT, LS, and HmSearch. LS is a strawman baseline that computes the Hamming distance between a sketch in $S$ and a query one-by-one in $O(Ln)$ time and $O(Ln \log \sigma)$ bits of space. Following the original paper [13], FRESH was tested with the following parameter settings: $\sigma = 2^{32}$, $\delta = 4dR$, $k = 1, 2, 4$, and $K = 31, 35, 39, \ldots, 59, 63$. For a fair comparison between different methods, we measured the time performance and the memory usage for varying recall for solution set $I'$. The implementations of HmSearch and FRESH used in these experiments are downloadable from https://github.com/kampersand/hamsearch and https://github.com/Ceccar/FRESH, respectively. Figure 7 shows experimental results. A part of the experimental results are presented in Appendix B.1.

FIGURES 7a–7c show the search time for varying the recall up to 100% for each method. The result showed tSTAT was the fastest for most results and demonstrated the efficiency of our search algorithm on tries with the multi-index approach. FRESH was slow because the implementation takes $O(n)$ time for $c_{\text{fresh}}$. For over 90% recalls, tSTAT was at most $60\times$, $34\times$, and $1600\times$ faster than FRESH on Taxi, NBA, and OSM, respectively. For over 90% recalls, tSTAT was at most $3.4\times$, $12\times$, and $2.3\times$ faster than HmSearch on Taxi, NBA, and OSM, respectively.

FIGURES 7e–7g show the memory usage for varying the recall up to 100% for each method. Various memories consumed by HmSearch were observed for different recalls because different data structures in HmSearch are built according to Hamming distance threshold $K$. Those results demonstrated a high memory efficiency of tSTAT. In fact, while tSTAT consumed only $2.4–2.7$ GiB of memory on OSM, HmSearch and FRESH consumed $27–65$ GiB and $50–65$ GiB of memories, respectively.

FIGURE 7h shows the construction time for varying the recall up to 100% for each method on OSM. We measured the execution time for producing sketches from input trajectories and building the index from the sketches. tSTAT took $5.0–6.1$ minutes in total while LS took $3.3$ minutes. On the other hand, the time needed for only building STATs from sketches was only $1.7–2.8$ minutes, which demonstrated fast algorithms for constructing STATs. HmSearch and FRESH took $32–57$ minutes and $18–51$ minutes, respectively, which were much larger than the construction times of tSTAT.

Considering the time and memory efficiency of tSTAT, the result demonstrated the feasibility of finding similar trajectories from...
We presented tSTAT, a novel method for fast and memory-efficient trajectory similarity search under Fréchet distance. Experimental results on real-world large datasets demonstrated that tSTAT was faster and more memory-efficient than state-of-the-art similarity search methods.

6 CONCLUSION

Figure 7: Performance evaluation of tSTAT, LS, HmSearch, and FRESH. Each of charts (a)–(c) shows average search times per query in milliseconds (ms) for varying recalls. Chart (d) shows average search times per query in ms for varying the number of trajectories, and it demonstrates scalabilities of similarity search methods. Each of charts (e)–(g) shows memory usages in GiB for varying recalls. Chart (h) shows construction times in minutes for varying recalls. The charts except (d) are plotted in the logarithmic scale.

Figure 8 shows an example of querying NBA trajectories using tSTAT with \( R = 0.45 \). Similar movements of NBA players for a short movement of Rajon Rondo in the match between Sacramento Kings and Oklahoma City Thunder on December 6, 2015 were retrieved from a large collection of 3.3 million trajectories in the NBA dataset. tSTAT successfully found similar movements of NBA players such as a movement of Tim Duncan in the match between San Antonio Spurs and Toronto Raptors on December 9, 2015. The result demonstrated an effectiveness of tSTAT for discovering similar movements of NBA payers, which would be beneficial to analyze various movements in sports data analyses [23, 37, 38].

large collections for a query, which would be beneficial to analyze trajectories that are similar to a query of interest.
In this paper, we presented tSTAT for similarity searches for discrete Fréchet distance on trajectories, but in principle it is also applicable to other distance measures such as continuous Fréchet distance and dynamic time warping distance on trajectories, as presented in [13, 14]. One important task of future work is to develop similarity searches for other distance measures by extending the idea behind tSTAT. This would be beneficial for users analyzing massive trajectory datasets in terms of similarity searches and trajectory mining.

ACKNOWLEDGMENTS

We would like to thank Yoichi Sasaki who taught us succinct data structures on trits at the StringBeginners workshop.

REFERENCES

[1] NBA Advanced Stats. https://stats.nba.com.
[2] OpenStreetMap Data Extracts. http://download.geofabrik.de/index.html.
[3] Taxi Service Route (TST) Prediction Challenge 2015. http://www.geolink.pt/ecmlpkoi2015-challenge/index.html.
[4] Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. Int. J. Comput. Geom. Appl., 5(01n02):75–91, 1995.
[5] Abdullah N Arslan and Ömer Ehcioğlu. Dictionary look-up within small trees of higher degree. Algorithmica, 31(2):320–345, 2001.
[6] Maria Astefanoaei, Paul Cesaretti, Panagiota Katsikouli, Mayank Goswami, and Rik Sarkar. Multi-resolution sketches and locality sensitive hashing for fast trajectory processing. In SIGSPATIAL, pages 279–288, 2018.
[7] Julian Baldus and Karl Bringmann. A fast implementation of near neighbors queries for Fréchet distance (GIS Cup). In SIGSPATIAL, page 99, 2017.
[8] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. Algorithmica, 43(4):275–292, 2005.
[9] Karl Bringmann. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. In FOCS, pages 722–731, 1994.
[10] Abdullah N Arslan and Ömer Ehcioğlu. Dictionary look-up within small trees of higher degree. Algorithmica, 31(2):320–345, 2001.
[11] Maximilian Konzack, Thomas McKetterick, Tim Ophelders, Maike Buchin, Luca Giuggioli, Jed Long, T’salyn Nelson, Michel A Westenberg, and Kevin Buchin. Visual analytics of delays and interaction in movement data. Int. J. Geogr. Inf. Sci., 31(2):320–345, 2017.
[12] Chen Li, Jiasheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In ICDE, pages 257–266, 2008.
[13] Alex X Liu, Ke Shen, and Eric Torn. Large scale hamming distance query processing. In ICDE, pages 553–564, 2011.
[14] Wunan Luo, Hayou Tan, Lei Chen, and Lionel M. Ni. Finding time period-based most frequent path in big trajectory data. In SSKOM, pages 713–724, 2013.
[15] Gourmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In WWW, pages 141–150, 2007.
[16] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast exact search in hamming space with multi-index hashing. IEEE T. Pattern Anal., 36(6):1107–1119, 2014.
[17] Mihai Patrascu. Succincter. In FOCS, pages 305–313, 2008.
[18] Guy Jacobson. Space-efficient static trees and graphs. In FOCS, pages 549–589, 1989.
[19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practical implementation of rank and select queries. In SIGMOD, pages 623–634, 2015.
[20] Kevin Toohey and Matt Duckham. Trajectory similarity measures. SIGSPATIAL Special, 10(1):43–50, 2015.
[21] Sebastian Vigna. Breadth implementation of rank/select queries. In WEA, pages 154–168, 2008.
[22] Martin Werner and Dev Oliver. ACM SIGSPATIAL GIS Cup 2017: Range queries under Fréchet distance. SIGSPATIAL Special, 10(1):24–27, 2018.
[23] Shuo Shang, Ruogu Ding, Kai Zheng, Christian S Jensen, Panos Kalnis, and Xiaofang Zhou. Personalized trajectory matching in spatial networks. The VLDB Journal, 23(3):449–468, jun 2014.
[24] E Siragavhandera, K Karthik, and Chiranjib Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In ICDAR, volume 1, pages 461–465, 2007.
A IMPLEMENTATION DETAILS

A.1 Succinct Data Structures on Tries

To handle trits on byte-aligned memory, we pack five trits into one tryte and store the tryte in one byte, as in [17]. For example, five trits 1, 2, 2, 0 and 1 are packed into a tryte as $1 \cdot 3^0 + 2 \cdot 3^1 + 2 \cdot 3^2 + 0 \cdot 3^3 + 1 \cdot 3^4 = 106$. Given $M$ trits, we construct a tryte vector $A$ of length $\lceil M/5 \rceil$ in which each element is represented as a byte. Then, the $i$-th trit can be extracted by getting the tryte $c \leftarrow A[\lfloor i/5 \rfloor]$ and computing $[c/3^i \bmod 3] \bmod 3$. The memory usage of $A$ is $8\lceil M/5 \rceil = \lceil 6.6M \rceil$ bits and is close to the optimal $\lceil M \log_2 3 \rceil = \lceil 1.58M \rceil$ bits [34].

We implement Rank data structures on $A$ in a similar manner for bit vectors (e.g., [21, 42]). We first conceptually partition $A$ into subarrays of 65,550 trits, calling these subarrays large blocks. Then, we partition each large block into subarrays of 50 trits, calling these subarrays small blocks. For each large block $j$, we store $LB_c[j] = \text{Rank}_c(A, 65550j)$. For each small block $k$, we store $SB_c[k] = \text{Rank}_c(A, 50k) - LB_c[\lfloor 50k/65550 \rfloor]$.

By using the arrays $LB_c$ and $SB_c$, $\text{Rank}_c(A,i)$ is computed as follows. We first compute $\text{Rank}_c(A,50\lfloor i/50 \rfloor)$ as $LB_c[\lfloor i/65550 \rfloor] + SB_c[\lfloor i/50 \rfloor]$, i.e., the Rank value at the beginning of small block $\lfloor i/50 \rfloor$, and then scan the remaining trits within the small block. We implement $LB_c$ and $SB_c$ as 64-bit and 16-bit arrays, respectively. $LB_c$ and $SB_c$ uses $64\lceil M/65550 \rceil + 16\lceil M/50 \rceil = 0.32M$ bits of space.

A.2 Fast Hamming Distance Computation

We consider to compute the Hamming distance between integer sketches $S$ and $T$ of length $L$. The computation time with a naive comparison approach is $O(L)$, assuming that two integers can be compared in $O(1)$ time (i.e., an integer can be represented within one machine word).

Zhang et al. [46] proposed a fast approach by exploiting a vertical format and bit-parallelism offered by CPUs. This approach encodes $S$ into a $\hat{S}$ in a vertical format, i.e., the $i$-th significant $L$ bits of each character of $S$ are stored to $\hat{S}[i]$ of consecutive $L$ bits. Given sketches $\hat{S}$ and $\hat{T}$ in the vertical format, we can compute $\text{Ham}(S, T)$ as follows. Initially, we prepare a bitmap bits of $L$ bits in which all the bits are set to zero. For each $i = 0, 1, \ldots, \lfloor \log_2 \sigma \rfloor - 1$, we iteratively perform $\text{bits} \leftarrow \text{bits} \lor (\hat{S}[i] \oplus \hat{T}[i])$, where $\lor$ and $\oplus$ denote bitwise-OR and -XOR operations, respectively. For the resulting bits, Popcnt(bites) corresponds to $\text{Ham}(S, T)$, where Popcnt() counts the number of 1s and belongs to the instruction sets of modern CPUs. The operations $\lor$ and Popcnt() can be performed in $O(1)$ time per machine word. Let $w$ be the machine word size in bits. We can compute $\text{Ham}(S, T)$ in $O(\lfloor \log_2 \sigma \rfloor \cdot \lceil L/w \rceil)$ time. In practice, setting $L$ to $w$ is efficient because $\hat{S}[i]$ of $L$ bits can be represented within one machine word, i.e., $L = 64$ in our experimental environment.

B EXPERIMENTAL RESULTS

This section shows the other experimental results not presented in Section 5.

B.1 Efficiency of tSTAT

Figure 9 shows the search time for showing the scalability on Taxi and OSM. Figure 10 shows the construction time on Taxi and NBA.

B.2 Efficiency of Node Reduction and STAT Data Structure

Figures 11–13 show the other experimental results not presented in Section 5.2 for showing an efficiency of node reductions. The results for $R = 2720$ on Taxi, $R = 0.26$ on NBA, and $R = 2506$ on OSM are omitted for the space limitation. In addition, the results on the numbers of candidates for varying Hamming distance thresholds $K$ are omitted for the same reason.
Figures 11, 12, and 13: Results of node reduction on Taxi, NBA, and OSM, respectively.