Proc. the 3rd IIAI International Conference on e-Services and Knowledge Management (IIAI ESKM'12),
Fukuoka, September 2012. (to appear)

Trajectory Pattern Matching Based on Bit-Parallelism for Large GPS Data

Hirohito Sasakawa
Graduate School of IST, Hokkaido University,
N14, W9, Sapporo 060-0814, Japan
Email: sasakawa@ist.hokudai.ac.jp

Hiroki Arimura
Graduate School of IST, Hokkaido University,
N14, W9, Sapporo 060-0814, Japan
Email: arim@ist.hokudai.ac.jp

***Abstract*—In this paper, we consider an approximate search problem for massive trajectory data. We first propose multi-resolution symbolic encoding for 2-dimnetional trajectory data based on a fixed-length stopper code. Then, we present an efficient bit-parallel string matching algorithm on encoded texts for the classes of multi-resolution trajectory patterns. Finally, we ran experiments on the real world trajectory data to evaluate the efficiency of the proposed algorithm. The results showed good performance enough for real applications.**

***Keywords*-GPS trajectory data, bit-parallel algorithm, multi-byte string matching**

I. Introduction

By the rapid progress of mobile devices and sensors, massive amount of trajectory data from GPS and mobile sensors have emerged for the last decade. Since they have different characteristics from traditional static databases, study on efficient trajectory search technology has attracted increasing attention for recent years [1], [2], [3], [4].

In this paper, we consider an approximate search problem for massive trajectory data. We first propose multi-resolution symbolic encoding for 2-dimnetional trajectory data based on a fixed-length tagged multi-letter coding. Then, we present an efficient bit-parallel string matching algorithm on encoded texts for the classes of multi-resolution trajectory patterns.

Finally, we ran experiments on the real world trajectory data to evaluate the efficiency of the proposed algorithm. In the experiments, our algorithm achieved throughput of 130 mega byte per second on a dataset of total size 45 mega bytes containing 536 trajectories and 11,219,955 points from Cabspotting project [5].

A. Related work

Research of trajectory search is mainly classified into two categories, trajectory search indexes and online trajctory search. There have been many researches on trajectory search indexes [6], which are based on existing spatial indexes such as R^*-trees [7], and are suitable to a computer with a few number of CPUs and large external storages. On the other hand, online trajectory search systems based on string matching technology are mainly main memory oriented and suitable to PC clusters equipped with very fast GPUs [8] as proven in event stream processing field. However, it seems that there has not been any existing research on the application of string matching to online trajectory search. Hence, we aim to develop base technologies for online trajectory search based on string matching.

B. Organization

The organization of this paper is as follows. Sec.II gives basic definitions on trajectory seach problems. Sec.III introduces our fixed-length stopper coding for 2-dimentional trajectories. Sec.IV presents an efficient bit-parallel string matching algorithm for multi-resolution trajectory patterns on encoded trajectories. Sec.V gives experimental results. Sec.VI gives conclusion and future problems.

II. Preliminaries

In this section, we give basic definitions on our trajectory search problem. For the definitions not found here, see textbooks, e.g., [9] and [10].

A. Basic definitions

We denote by $\mathbb{N}$ and $\mathbb{R}$ the sets of all non-negative integers and all real numbers, respectively. Let $\mathbb{R}_+ = \{\, x \in \mathbb{R} \mid x \geq 0 \,\}$. For $a, b \in \mathbb{R}$ $(a < b)$, we define the interval $[a, b] = \{\, x \in \mathbb{R} \mid a \leq x \leq b \,\}$. For $i, j \in \mathbb{N}$ $(i \leq j)$, we define $[i..j] = \{i, \ldots, j\}$.

For a set A, a *string over* A of length $n \geq 0$ is a consecutive sequence $s = s[1] \cdots s[n]$ of elements from A, where $|s| = n$ is the *length* of s. We denote by ε the *empty string* of length 0. We define by $s[i..j]$ the *substring* of s starting from i and ending at j. We denote by A^* the set of all possibly empty strings over A and let $A^+ = A \setminus \{\varepsilon\}$.

B. Trajectory search

We define the trajectory search problem as follows. Let $\mathcal{A} = [0, u]^2 \subseteq \mathbb{R}^2$ be a subspace of the Euclidean space, called an *area*. Without loss of generality, we simply assume $\mathcal{A} = [0, 1]^2$ with $u = 1$. A *discrete trajectory* in $\mathcal{A}$ is a sequence $s = (p_1, \ldots, p_n) \in \mathcal{A}^*$ of 2-dimensional points in $\mathcal{A}$, where $|s| = n$ is the *length* of s. The input of the problem is a collection $\mathcal{S} = \{s_1, \ldots, s_h\}$ $(h \geq 0)$ of discrete trajectories in $\mathcal{A}$, called a *trajectory database*. Let $\mathcal{G} \subseteq 2^{\mathcal{A}}$ be a class of regions, in $\mathcal{A}$, called *mesh cells* which will be defined later.

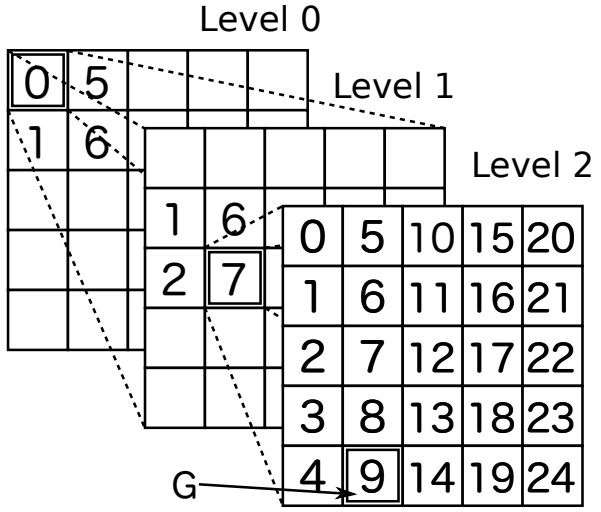

Figure 1. An example of a hierarchical mesh structure $(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3)$ of level $K = 3$ and resolution $R = 5$, and the number of mesh cells $D = 25$. The mesh cell G has the address $addr(G) = 0.7.9$.

Then, a *multi-resolution trajectory pattern* (*pattern*, for short) is a sequence $P = (G_1, \ldots, G_M) \in \mathcal{G}^*$, where $|P| = M$ is the *length* of P, and for every $i = 1, \ldots M$, G_i is a region in $\mathcal{G}$. A pattern $P = (G_1, \ldots, G_M)$ *matches* a trajectory $s = (p_1, \ldots, p_n)$ if there exists some position $1 \leq h \leq n$ such that $p_{h+i-1} \in G_i$ holds for every $i = 1, \ldots, M$. Then, we say that the *right occurrence* (*occurrence*, for short) of P in s is $pos = h + M - 1$. The *occurrence* of P in $\mathcal{S}$ is a pair (s, pos), where pos is an occurrence of P in $s \in \mathcal{S}$.

Now, we state our trajectory search problem as follows: the *approximate trajectory search* is the problem to preprocess given a trajectory database $\mathcal{S}$ so that trajectory search queries are evaluated efficiently. A *trajectory search query* is, given a multi-resolution trajectory pattern P, to find all the occurrences of P in $\mathcal{S}$.

Our goal is to devise an efficient search algorithm that answers the above trajectory search queries under appropriate preprocessing of an input trajectory database.

C. Model of computation

As the model of computation, we assume a *unit-cost RAM* with word length w [11]. For any bitmask length $m \geq 0$, A *bitmask* is a vector $X = b_m \cdots b_1 \in \{0,1\}^m$ of m bits. For bitmasks with $m \leq w$, we assume that the following boolean and arithmetic operations are executed in $O(1)$ time: *bitwise* AND "&", *bitwise* OR "|", *bitwise* NOT "$\sim$", *bitwise* XOR "$\oplus$", *left shift* "$\ll$", *right shift* "$\gg$" on $RAM()$, *integer addition* "+" and *integer subtraction* "$-$" on $RAM(+)$. The space complexity is measured in the number of words.

Algorithm 1 A bit-parallel algorithm for multi-resolution trajectory search

```
1: procedure MultiResSHIFT-ANDmain(P, T)
   Input: a pattern P of size m, a text T of size n;
2:     M ← Preprocess(enc_tmc(P));
3:     Runtime(T, M);
4: end procedure

5: procedure Preprocess(enc_tmc(P))
6:     Compute and return a set M of bitmasks for
       NFA N corresponding to enc_tmc(P) by the method in
       Sec. IV-B;
7: end procedure

8: procedure Runtime(T: text, M: a set of bitmasks)
9:     ((CH[c])_{c∈Σ}, INIT, ACCEPT, SKIPPOS) ← M;
10:    Loop ← INIT;
11:    State ← 0^{m+1};
12:    State ← State | Loop;
13:    for i = 0 to n do
14:        State ← (State ≪ 1) & CH[T[i]];
15:        Loop ←
               Loop | ((State & SKIPPOS) ≪ 1);
16:        if T[i] ∈ Σ_1 then
17:            State ← State | Loop;
18:            Loop ← INIT;
19:            if State & ACCEPT ≠ 0 then
20:                Report an occurrence at i;
21:            end if
22:        end if
23:    end for
24: end procedure
```

Figure 2. Multi-Resolution SHIFT-AND algorithm

III. Encoding of trajectory data

For efficient trajectory search by string matching technology, we apply the idea of multi-byte stopper encoding of Moura *et al.* [12] and Tarhio *et al.* [13] to encode trajectory data. We propose the *multi-resolution tagged mesh coding* scheme for 2-dimentional trajectory data based on hierarchical mesh decomposition defined as follows. For the notions used in the followings, see textbooks, e.g. [14].

Granurality of our encoding is controled by two encoding parameters D and $K \geq 1$, called the *number of mesh cells* and the *level parameter*, respectively. D is often a squre number $D = R^2$ for a resolution parameter $R \geq 1$, but it is not necessarily true in general.

A. Hierarchical mesh structure

Suppose that we are to encode any 2-dim point $p = (x, y) \in \mathcal{A}$. We encode p as K-digits D-ary number $addr(p) = z_1 \cdots z_K$, in $[0..D-1]^K$, called the (D, K)-*address* of p, as follows.

Let $k = 0$, and $\mathcal{A}_0 = \mathcal{A}$ be the top-level plane. For every $k = 1 \ldots K-1$, we define the k-th digit $z_k \in [0..D-1]$ as follows. Suppose that $k > 0$ and $\mathcal{A}_{k-1}$ is defined. We devide $\mathcal{A}_{k-1}$ into $D = R^2$ equal-sized cells, called *mesh cells* of level k (k-mesh cells), and number them from 0 to $D-1$ systematically as in Fig. 1, e.g., visiting columns from left to right, and then cells from top to bottom. We define the *mesh* of level k (or k-*mesh*), denoted by $\mathcal{G}_k$, to be the collection of all such k-mesh cells with size $\sigma = 1/R$. Then, we let $z = z_k$ be the the assigned number (or the id) of the unique mesh cell G_z in this level k that contains the point p. After letting $\mathcal{A}_k$ be the mesh cell G_z, we repeat the above process while $k \leq K-1$. Eventually, $k = K$ holds, and then we obtain the K-digits D-ary number $addr(p) = z_1 \cdots z_K$ for the point p as desired, which we call a *complete* address.

We can also encode any mesh cell G in any level $k < K$ by the same method. In this case, it is encoded in the k-digits D-ary number $addr(G) = z_1 \cdots z_k$, with length properly smaller than K, which we call a *partial* address for the mesh cell G.

Finally, we obtain the collection $\mathcal{G} = \cup_{1 \leq k \leq K} \mathcal{G}_k$ of all mesh cells in any level. We call $(\mathcal{G}_1, \ldots, \mathcal{G}_K)$ the hierarchical mesh structure w.r.t. encoding parameters (D, K). If it is clear from context, we identify a point in an element of the bottom mesh G_K .

B. Multi-resolution tagged mesh code

In our encoding of K-digit D-ary numbers for points and meshes in $\mathcal{G}$, we use an alphabet $\Sigma = \Sigma_0 \uplus \Sigma_1$ of ℓ-bit letters satisfying the following conditions. Let Σ_0 and Σ_1 be mutually disjoint alphabets, that are large enough to represent any D-ary digit. Elements of Σ_0 and Σ_1 are called *cotinuers* and *stoppers*, respectively. Specifically, for every $i = 0, 1$, Σ_i is an alphabet of ℓ-bit tagged letters with highest bit set to i, where $\ell = \lceil \log_2 D \rceil + 1$. Since $2^{\ell-1} \geq D$, the alphabet Σ_i can encode any D-ary digit in $[0..D-1]$.

Then, we define the *multi-resolution tagged mesh code* as follows. To easily detect code boundaries in a text, we add tag bits at the highest bit-positions of each mesh letter. The tagged mesh codeword for the point p with (D, K)-address $addr(p) = z_1 \cdots z_K$ is defined by the sequence

$$enc_{\mathrm{tmc}}(p) = c_1 \cdots c_k \in \Sigma^k \tag{1}$$

where for every $1 \leq i \leq k$, c_i is a tagged letter in Σ. We call the codeword *complete* if the length satisfies $k = K$, and *partial* if $k < K$. Moreover, c_i is a continuer in Σ_0 if $i \neq K$, and c_i is a stopper in Σ_1 if $i = K$. Therefore, a complete tagged mesh codeword always terminates with a stopper, and used in both trajectories and patterns. On the other hand, a partial tagged mesh codeword consists of only continuers, and used only in patterns.

$$P_1 = (G_1, G_2, G_3)$$
$$enc_{\mathrm{tmc}}(P_1) = (0A)(0A)(1B)\#(0B)\%\#(0C)(0A)\%\#$$

Figure 3. Examples of a multi-resolution trajectory pattern P_1 and its encoded version $enc_{\mathrm{tmc}}(P_1)$, where G_1, G_2 and G_3 are regions, $0A$ and $1A$ indicates continuer and stopper letters encoding A, respectively, $\#$ denotes a code boundary, and $\%$ denotes a special symbol for skipping the rest of a codeword.

C. Encoding of trajectory data

We encode a trajectory $s = (p_1, \ldots, p_n)$ of length n by

$$enc_{\mathrm{tmc}}(s) = (enc_{\mathrm{tmc}}(p_1), \ldots, enc_{\mathrm{tmc}}(p_n)) \in \Sigma^*,$$

where each $enc_{\mathrm{tmc}}(p_i)$ is a complete mesh codeword. Finally, we encode an input trajectory database $\mathcal{S} = \{s_1, \ldots, s_m\}$ of size m by a collection

$$enc_{\mathrm{tmc}}(\mathcal{S}) = \{\, enc_{\mathrm{tmc}}(s) \mid s \in \mathcal{S} \,\} \subseteq \Sigma^*$$

of encoded strings on Σ. Since each point is encded in a codeword of length K letters, the total size of $enc_{\mathrm{tmc}}(\mathcal{S})$ is $O(KM)$, where $M = ||\mathcal{S}||$ is the original input size.

For example, in our experiments in Sec. V, we used the encoding parameters $R = 8$, $D = R^2 = 64$ and $K = 4$, which are sufficient to encode points in our trajectory data [5] with x- and y-coordinates in 12-bit integers. Then, define each tagged letter to have bit-width $\ell = 8$ with one bit for tag and one bit for padding. Thus, this encoding becomes a byte-based encoding.

IV. Algorithms

We use a fast string matching algorithm tailored to texts encoded in the multi-resolution tagged mesh coding, to implement fast trajectory search queries.

A. Outline of algorithms

In Fig.2, we present an efficient bit-parallel string matching algorithm MultiResShiftAnd, which is an extension of Extended SHIFT-AND algorithm, proposed by Navarro and Raffinot[15], to multi-resolution tagged mesh coding.

In bit-parallel string matching approach, in preprocessing phase, we first receive an input pattern P, transform P into a non-deterministic automaton (NFA) $N(P)$, and then, build a set of bit-masks that represents the NFA. In runtime phase, we simulate the string matching of the NFA based on the given set of bit-masks by using bit- and arithmetic operations on registers.

A key to the algorithm is synchronization of state transition of the underlying pattern matching NFA at code

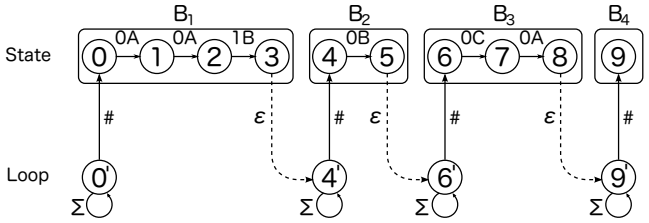

Figure 4. The NFA N_1 correspoding to the pattern P_1 in Fig. 3.

boundaries. For the purpose, we split the NFA into the main part for foward transitions and the sub-part for loop transitions. At every time, we detect the code boundary by stopper symbol, we move bits from/to the main-part states to/from the sub-part states to synchronize.

B. Preprocessing phase

1) Transformation of an input trajectory pattern: Suppose that we are given a multi-resolution trajectory pattern $P = (G_1, \ldots, G_M) \in \mathcal{G}^*$, where M is the length of the pattern counted in the number of regions. for every $1 \leq i \leq M$, G_i is a mesh cell of level k_i. Let $\#$ and $\% \notin \Sigma$ be special symbols called a code boundary letter and a skip letter, respectively.

Then, the encoded pattern is given by

$$\begin{aligned} enc_{\mathrm{tmc}}(P) &= B_1 \# B_2 \# \ldots \# B_M \# \\ &= \alpha_1 \alpha_2 \cdots \alpha_m \in (\Sigma \cup \{\#, \%\})^*, \end{aligned}$$

where m is the number of all symbols in the encoded pattern, and each α_i is either a base tagged letter in Σ, a code boundary letter $\#$, or a skip letter $\%$. For every $1 \leq i \leq M$, sub-sequence $B_i = enc_{\mathrm{tmc}}(G_i) \in \Sigma^{k_i}$ is the tagged mesh codeword for $G_i \in \Sigma$ consisting of k_i tagged letters in Σ, and $\# \notin \Sigma$ is a special symbol for a code boundary that is not actually encoded in a trajectory.

In the above codeword, the i-th component B_i has one of the following forms:

1) If G_i represents a point, i.e., $k_i = K$, then B_i is a complete codeword $c_1^i \cdots c_K^i$ for G_i.
2) Otherwise, G_i represents a mesh, i.e., $k_i < K$, then B_i has the form $\alpha\%$, where $\alpha = c_1^i \cdots c_k^i$ is a partial codeword for G_i, and $\%$ is a special symbol that indicates the rest of any codeword.

In Fig.3, we show an example of an input pattern.

2) Construction of a pattern NFA: Our algorithm constructs the correspoinding NFA from the above pattern, and a set of bit-masks for it, and then simulate the transition computation of the NFA on encoded texts by using bit-parallel method.

In Fig. 4, we show an example of NFA for trajectory pattern $P = (0A)(0A)(1B)\#(0B)\%\#(0C)(0A)\%\#$ with $m = 9$. In the figure, the first complete codeword $(0A)(0A)(1B)$ is transformed into a sub-automaton B_1 from state 0 to state 3. For the next code boundary $\#$, we create a new state $4'$ with self-loop, and add an ε-edge from the terminal state 3 of B_1 to $4'$ and a $\#$-edge from $4'$ to the initial state 4 of the next block B_2. For codewords B_2 and B_3, we repeat a similar construction.

3) Construction of bit-masks: To simulate the pattern matching NFA N, we construct a set of bit-masks INIT, ACCEPT and $\mathsf{SKIPPOS} \in \{0,1\}^{m+1}$ and an array of bit-masks $(\mathsf{CH}[c])_{c \in \Sigma} \in (\{0,1\}^{m+1})^{|\Sigma|}$ defined as follows.

- $\mathsf{INIT} = 0^m 1 \in \{0,1\}^{m+1}$ is the L-bit mask that sets 1 at the bit position 0 for state 1.
- $\mathsf{ACCEPT} = 10^m \in \{0,1\}^{m+1}$ is the L-bit mask that sets 1 at the bit position for the final state m.
- $\mathsf{SKIPPOS} \in \{0,1\}^{m+1}$ is the L-bit mask that indicates all bit positions of the start state x of an ε-edge outgoing from x to a loop state y.
- For every $c \in \Sigma$, $\mathsf{CH}[c] \in \{0,1\}^{m+1}$ is the L-bit mask for indicating all bit position of letter edges labeled with c. That is, $\mathsf{CH}[c][i] = 1$ if and only if state i has an incoming edge labeled with $c \in \Sigma$. For letter $\#$ this is not defined at all.

4) Runtime phase: In Fig. 2, we present an algorithm RUNTIME that simulates the NFA N for a pattern using bit-parallelism on encoded texts in multi-resolution tagged coding. The algorithm uses two working bit-masks S and $Loop \in \{0,1\}^{m+1}$ in L bits for simulating theforward states and the loop states.

At initialization, we initialize the working bit-masks $State$ and $Loop$ for forward and loop states at Line 10 - 12:

$$Loop \leftarrow \mathsf{INIT} \tag{2}$$

$$State \leftarrow 0^{m+1} \tag{3}$$

$$State \leftarrow State \mid Loop \tag{4}$$

Given an input base letter $c \in \Sigma$ at postion $i = 1, \ldots, n$, we make a letter transition on $N(Q)$ by the letter c using SHIFT-AND method by the following code. at Line 14:

$$State \leftarrow ((State \ll 1)) \,\&\, \mathsf{CH}[c] \tag{5}$$

We also make the ε-transitions from a skip state in $State$ to a loop state in $Loop$ on $N(Q)$ at Line 15 by the following code:

$$Loop \leftarrow Loop \mid ((State \,\&\, \mathsf{SKIPPOS}) \ll 1) \tag{6}$$

The above code also make the self-loop transition from a loop state to itself by copying the value of $Loop$.

Next, we check if we are at a code boundary. This is done by testing if c is a stopper or a continuer at Line 16. There are two cases below. If c is a continuer in Σ_0, then we do nothing and skip the rest of the for-loop. Otherwise, c is a stopper in Σ_1. Then, we simulate the virtual letter transition

with a special code boundary letter # on $N(Q)$ at Line 17 by the following code:

$$State \leftarrow State \mid Loop \tag{7}$$

At the same time, we have to explicitly make the self-loop transition from a the initial loop state $0'$ to itself by the following code:

$$Loop \leftarrow \mathsf{INIT} \tag{8}$$

since there is no previous state of the initial loop state $0'$ in our bit-parallel representation.

Finally, at Line 19, we test the acceptance of the NFA $N(Q)$ by the following code:

$$\text{if } (S \ \& \ \mathsf{ACCEPT} \neq 0) \text{ then report an occurrence}$$

and if so, report the matching of the query pattern Q at the present position i.

C. Analysis

We give theoritical analysis on the computational complexity of the proposed algorithm for multi-resolution trajectory query patterns. Let $w = O(\log n)$ be the bit-length of a register in an underlying computer.

Lemma 1: The algorithm of Fig. 2 finds all occurrences of a given encoded query pattern P of state parameter m in $T = O(n\lceil \frac{m}{w} \rceil)$ time using $P = O(|\Sigma|m)$ preprocessing time and $S = O(|\Sigma|\lceil \frac{m}{w} \rceil)$ words of space, where $n = ||\mathcal{S}||$ is the total length of all encoded trajectories in $\mathcal{S}$. ◇

Proof: The preprocessing time and the space are obvious. We consider the running time as follows. For each input letter $c = T[i]$ at position $i = 1, \ldots, n$, the algorithm perform a constant number of bit-operations to update two bit-masks $State$ and $Loop$. Since we can simulate every operation on a bit-mask of length m by constant time operations on a set of $\lceil \frac{m}{w} \rceil$ registers of width w, the algorithm takes $O(\lceil \frac{m}{w} \rceil)$ time per input letter. Hence, the total running time is $O(n\lceil \frac{m}{w} \rceil)$. ■

V. Experiments

Finally, we ran computational experiments on real 2-dim trajectory data to evaluate the efficiency and the usefulness of the proposed algorithms.

A. Data

In the experiment, we use *Cabspotting* taxi GPS trajectory data[5]. This dataset consists of GPS traces of approximately 500 taxi over 30 days in San Francisco Bay Area, USA.

- The number of unique points is $11,219,936$.
- The dataset contains $11,219,955$ points with total size 379.0 (MB) consisting of 536 sequences.
- Each sequneces contains trajectories of a taxi in 30 days consisting of $20,932$ points in average.

In encoding, we used the following parameters.

- The resolution $R = 8$, the number of meshes $D = 64$, and the maximum level $K = 3$.
- The bit length of tagged letter $\ell = 8$ (bit).
- The sizes of the encoded and base alphabets are $|\Delta| = D^3 = 262,144$ and $|\Sigma| = 256$, respectively.

The total size of the encoded dataset is approximately 40 MB consisting of 536 sequences.

B. Method

We implemented our pattern matching algorithm in C++ language and compiled by g++ of GNU, version 4.3. We ran experiments on Intel Core i7 2.8GHz with 4GB of RAM running on Mac OSX 10.7.3.

We use the following patterns in the experiments.

- P_1 = (00AB) (00CD) (00EF) (10GH)# (00AB) (00CD) (00EF) (10IJ)# (00AB) (00CD) (00EF) (10GH)
- P_4 = (00AB) (00CD) (00KL) (10MN)# (00AB) (00CD) (00KL) (10MN)# (00AB) (00CD) (00KL) (10MN)# (00AB) (00CD) (00KL) (10MN)# (00AB) (00CD) (00EF) (10MN)# (00AB) (00CD) (00KL) (10MN)# (00AB) (00CD) (00EF) (10GH)# (00AB) (00CD) (00EF) (10GH).
- P_3 = (00AB)(00CD)%# (00AB)(00CD)(00EF)(10IJ).
- P_4 = (00AB)%# (00AB)%# (00AB)%# (00AB)%# (00AB)%# (00AB)%# (00AB) (00CD) (00EF) (10GH)# (00AB) (00CD) (00EF) (10GH).

P_1 and P_2 consist only of complete codewords, while P_3 and P_4 contain complete and partial codewords.

1) Exp 1: We collected the statistics of all patterns, and compare their number of occurrence in the text. We used the same patterns, P_1, P_2, P_3, and P_4 as in the Exp 1. Input text was 40 MB. The result is given by Table I.

Table I
STATISTICS OF PATTERNS IN EXPERIMENTS

pattern	complete	partial	#NFA states	#words	#occ
P_1	v		12	1	227
P_2	v		16	1	1
P_3	v	v	11	1	7,235
P_4	v	v	27	1	6,178

In the table, the features in the first row, the columns labeled with pattern, complete, partial, #NFA states, #words, #occ indicate the name of pattern, the existences of complete codes and partial codes in a pattern, reps., the number of states of the NFA, the number of words for a bit-mask, the number of occurrences of the patterns, respectively.

From column #words, we saw that all patterns fit into a computer word, whose bit-width was $w = 32$. From column #occ, the number of occurrences vary from pattern to pattern. For instance, P_3 has 7,235 occurrences in the database, while P_2 has the unique occurrence.

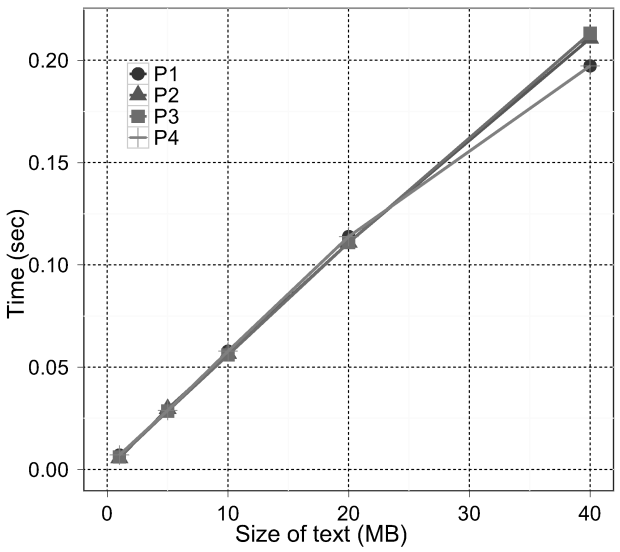

Figure 5. Running time against the input text size.

C. Results

1) Exp 2: In Fig. 5, we measure the running time of our matching algorithm by varing the values of parameters such as the pattern length and the number of skips. In this figure, we see the running time increases as the input size increases. Also, we observed that the running time does't chage by varing the pattern length and the number of skips.

VI. Conclusion

In this paper, we considered trajectory search from massive trajectory data. We proposed symbolic encoding called multi-resolution tagged mesh code for trajectory data, and presented an efficient bit-parallel pattern matching algorithm on encoded trajectories. Finally, we ran experiments on the real world trajectory data to evaluate the efficiency of the proposed algorithm, which showed good performance enough for real applications.

References

[1] P. Revesz, *Moving Objects Databases, Introduction to Databases: From Biological to Spatio-Temporal, Chapter 7, Texts in Computer Science*. Springer-Verlag, 2010, vol. 111.

[2] E. Frentzos, K. Gratsias, and Y. Theodoridis, "Index-based most similar trajectory search." in *Proc. the 23rd Int'l Conf. on Data Engineering (ICDE'07)*. IEEE, 2007, pp. 816–825.

[3] P. Bakalov, M. Hadjieleftheriou, E. Keogh, and V. J. Tsotras, "Efficient trajectory joins using symbolic representations," in *Proceedings of the 6th Int'l Conf. on Mobile Data Management (MDM'05)*. ACM, 2005, pp. 86–93.

[4] J. Han, Z. Li, and L. A. Tang, "Mining moving object, trajectory and traffic data," in *Proc. 15th Int'l Conf. on Database Systems for Advanced Applications (DASFAA'10)*, ser. LNCS, vol. 5982. Springer, 2010, pp. 485–486.

[5] M. Piorkowski, N. Sarafijanovoc-Djukic, and M. Grossglauser, "A Parsimonious Model of Mobile Partitioned Networks with Clustering," in *The 1st International Conference on Communication Systems and Networks (COMSNETS)*, January 2009.

[6] Y. Zheng and X. Zhou, Eds., *Computing with Spatial Trajectories*. Springer, 2011.

[7] D. P. Mehta and S. Sahni, *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[8] A. Margara and G. Cugola, "High performance content-based matching using gpus," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, ser. DEBS '11. New York, NY, USA: ACM, 2011, pp. 183–194.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[10] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, 2000.

[11] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[12] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," *ACM Trans. Inf. Syst.*, vol. 18, no. 2, pp. 113–139, Apr. 2000.

[13] J. Rautio, J. Tanninen, and J. Tarhio, "String matching with stopper encoding and code splitting," in *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM '02)*. London, UK, UK: Springer-Verlag, 2002, pp. 42–52.

[14] A. Moffat and A. Turpin, *Compression and Coding Algorithms*. Kluwer, 2002.

[15] G. Navarro and M. Raffinot, "Fast and simple character classes and bounded gaps pattern matching, with application to protein searching," in *Proceedings of the fifth annual international conference on Computational biology*, ser. RECOMB '01. New York, NY, USA: ACM, 2001, pp. 231–240.