

Indexing, Retrieval, and Compression of Moving Objects in Networks: A String Processing Approach

KOIDE, Satoshi

Abstract

Nowadays, smartphones and automobiles equip location information devices, such as global positioning systems (GPS). Spatial data collected from those devices are used to provide useful information services, including maps, traffic information, and social networks, to name a few.

We focus on *trajectory data*, which is a series of positions sequentially collected from location devices. In particular, our interest is on *trajectories in (road) networks*. This particular form of trajectory is important, because human movements are essentially constrained in road networks. Especially, spatial trajectories obtained from automobiles play important roles in many applications, such as data-driven navigation systems and automotive development. In this thesis, we consider fundamental problems for data management of trajectories in networks; we develop a series of new techniques that are useful for trajectory processing.

In the first part of this thesis, we develop a novel indexing method that allows us to treat various types of spatio-temporal queries that involve routing in road networks such as 1) finding moving objects that have traveled along a given path during a given time interval, 2) extracting all paths traveled after a given spatio-temporal context, 3) enumerating all paths between two locations traveled during a certain time interval. Our finding is that these queries can be solved efficiently using a common data structure, which we refer to as *SNT-index*. A key feature of our approach is the use of *string indexing techniques*. We regard a trajectory as a string, because a trajectory constrained in a network can be regarded as a sequence of discrete symbols (i.e., road segment IDs). Concretely, trajectories are stored in *FM-index*, which is a fast and compact in-memory data structure for strings. We propose how to connect temporal information with spatial information stored in FM-index.

An issue of FM-index is its large memory footprint when we treat large trajectory datasets. To address this issue, we tackle the data compression problem for trajectories stored in an FM-index, which is the main topic of the second part of this thesis. We achieve a significant improvement compared with the existing techniques in string processing fields. This improvement comes from the fact that trajectories in road networks can be regarded as *walks in a large but sparse directed graph*. On basis of this idea, we develop *relative movement labeling*, which transforms trajectories into easy-to-

compress sequences. We also show that important features of FM-index (e.g., its fast query processing) are still available even if we apply such transformation. Moreover, we make detailed analysis from information theoretic perspective, and reveal that the road network sparsity plays an essential role in our method. Surprisingly, we show that query processing of our proposed method becomes faster than that of the uncompressed FM-index.

Acknowledgement

The author would like to express my sincere gratitude to my advisor Professor Yoshiharu Ishikawa for his continuous support of my Ph.D study and related research. Without his patience, encouragement, and immense knowledge, this thesis would not have been materialized. I also thank for offering me many opportunities that broaden my knowledge and network. The experiences during my Ph.D study will definitely benefit me in my future life. I would like to thank my thesis committee, Professor Shigeki Matsubara, Associate Professor Yousuke Watanabe, and Associate Professor Chuan Xiao for their insightful comments.

I want to express my gratitude to Associate Professor Chuan Xiao, as a co-author, for his continuous support and insightful comments during my Ph.D study. It has been a great opportunity and pleasure for me to have a technical discussion with him.

I also would like to thank to Toyota Central R&D Labs., Inc. and the colleagues, who understood and encouraged me during my Ph.D study. In particular, I thank my co-authors, Dr. Yukihiro Tadokoro and Dr. Takayoshi Yoshimura, who gave me helpful comments during the early phase of my Ph.D study. I also thank all co-authors of the papers that are not directly involved in the main topic of this thesis. I have greatly benefited from discussion with them regarding a broad research topics and fields.

Finally, I appreciate my wife, Kumiko, for her understanding and kind support during my Ph.D study. Also, I have to say thanks to my parents for their love and support; especially, without their financial support during the six years at Osaka University, I would not come here now. The last word goes to my baby boy, Kazuto. You have been the light of my life for the last two years of my Ph.D study.

Satoshi Koide
January 24, 2020

Contents

Abstract	iii
Acknowledgement	v
List of Figures	xii
List of Tables	xiii
Abbreviations	1
I. Introduction	3
1. Introduction	5
1.1. Background	5
1.2. Research Objective and Contribution	6
1.3. Thesis Organization	7
2. String Processing and Spatial Trajectories: Preliminaries	9
2.1. Trajectory as a String	9
2.2. Strings and Related Data Structures	11
2.2.1. Strings	11
2.2.2. Suffix Arrays	12
2.2.3. Burrows-Wheeler Transform	13
2.3. FM-index	16
2.3.1. Pattern Matching Query	16
2.3.2. Substring Extraction Query	17
2.3.3. Wavelet Trees	18
2.4. Summary	20

II. Trajectory Indexing	21
3. Research Issues and Problem Definition	23
3.1. Research Issues	23
3.2. Problem Definition and Path-based Queries	27
3.2.1. Data Model	28
3.2.2. SPQ: Strict Path Query	29
3.2.3. TEQ: Trajectory Extraction Query	31
3.2.4. TAPEQ: Time-period-based All Path Enumeration Query	32
3.2.5. Summary	33
4. Indexing and Querying Methods	35
4.1. SNT-index	35
4.1.1. Overview	35
4.1.2. Spatial FM-index	35
4.1.3. Temporal B ⁺ -trees	37
4.1.4. Index Construction and Implementation	39
4.1.5. Summary	39
4.2. Algorithm for SPQs	40
4.2.1. Proposed SPQ algorithm	40
4.2.2. Existing SPQ Algorithms	41
4.2.3. Summary	43
4.3. Algorithm for TEQs	43
4.3.1. Baseline Method: Adding TB-tree-like Pointers to NETTRA	43
4.3.2. Proposed TEQ Algorithm	44
4.3.3. Summary	45
4.4. Algorithm for TAPEQs	46
4.4.1. Baseline Method: PrefixSpan for TAPEQs	46
4.4.2. Proposed TAPEQ algorithm	46
4.4.3. Summary	49
4.5. Appending New Data to SNT-index	49
4.5.1. Partitioning the FM-index for Appending New Data	50
4.5.2. Spatial Partitioning of the FM-index	51
5. Experiments	53
5.1. Setup and Implementation Details	53
5.2. SPQ Results	54
5.3. TEQ Results	57
5.4. TAPEQ Results	59

5.5. Index Size and Index Construction Time	61
5.6. Effect of Buffer Caches	63
5.7. Summary	63
5.8. Discussion	64
6. Related Work	67
6.1. Trajectory Indexing	67
6.2. Related Queries	69
7. Summary	71
III. Trajectory Compression	73
8. Research Issues and Preliminaries	75
8.1. Research Issue	75
8.2. Preliminaries	77
8.2.1. Huffman-shaped Wavelet Tree	77
8.2.2. Compressed Variants of FM-index	80
9. Compressing FM-index for Trajectories	83
9.1. Relative Movement Labeling	83
9.2. Data Structure	86
9.3. Query Processing	87
9.3.1. PseudoRank	88
9.3.2. Suffix Range Query with CiNCT	89
9.3.3. Extracting a Substring with CiNCT	90
9.4. Theoretical Analysis	91
9.4.1. Optimality of RML	91
9.4.2. Space Complexity	92
9.4.3. Time Complexity	93
9.4.4. Comparison of RML with MEL	93
9.5. Proofs	94
9.5.1. Proof of Theorem 9.2	94
9.6. Proof of Theorem 9.3	97
10. Experiments	99
10.1. Experimental setup	99
10.2. Results	100
10.2.1. Comparison with Various FM-indexes	100

Contents

10.2.2. Comparison with Several Compression Methods	102
10.2.3. Effect of Labeling Strategy	103
10.2.4. Effect of ET-graph size/shape	104
10.2.5. Sub-path Extraction Time	105
10.2.6. Index Construction Time	105
11. Related Work	107
11.1. Trajectory Compression	107
11.2. FM-index	108
12. Summary	109
IV. Conclusion	111
13. Conclusion and Future Work	113
Bibliography	114
Publications by the Author	123

List of Figures

1.1. Thesis overview	7
2.1. Visualization of network-constrained trajectories	10
2.2. Schematic explanation of suffix array	13
2.3. Schematic explanation of the Burrows-Wheeler transform (BWT)	14
2.4. Schematic explanation of wavelet trees	20
3.1. Overview of SNT-index	24
3.2. Example trajectories	24
3.3. Typical data structure for NCT indexing	25
3.4. Strict path query	29
3.5. SPQ application scenario	30
3.6. Time-period-based all-path enumeration query (TAPEQ)	33
4.1. Data structure of SNT-index	36
4.2. SNT-index construction	38
4.3. Modified TB-tree	44
4.4. ST-join	48
4.5. Append-supported SNT-index	51
5.1. SPQ processing time for various $ P $	56
5.2. FM-search (the Singapore dataset)	56
5.3. Scalability (RGSxN, $ P = 50$, 100% temporal selectivity)	56
5.4. SPQ processing time for various $ I $ (high temporal selectivity; 25% – 100%)	57
5.5. SPQ processing time for various $ I $ (low temporal selectivity; 2% – 25%)	57
5.6. TEQ processing time for various L	58
5.7. TEQ processing time for various $ I $ (temporal selectivity 2% – 100%)	58
5.8. <i>FM-extract</i> (Singapore dataset)	59
5.9. TEQ scalability (RGSxN, $L = 10$, 100% temporal selectivity)	59
5.10. TEQ example ($L = 10$)	59
5.11. TAPEQ processing time for various ξ	60
5.12. TAPEQ processing time for various $ I $ (temporal selectivity 2% – 100%)	60
5.13. Effect of d_{uv} (Roma, $\xi = 2$, temporal selectivity 100%)	61

List of Figures

5.14. Scalability (RGSxN, $\xi = 50$, temporal selectivity 100%)	61
5.15. TAPEQ example	61
5.16. Effect of buffer caches for SPQs	64
5.17. Effect of buffer caches for TEQs	64
5.18. Effect of buffer caches for TAPEQs	64
8.1. Main idea of our trajectory compression	76
8.2. Schematic explanation of BWT	78
8.3. Schematic explanation of wavelet tree	79
8.4. Schematic explanation of compression boosting	80
9.1. ET-graph and labeled BWT	84
9.2. Comparison of Huffman trees	87
9.3. Schematic explanation of PseudoRank	89
9.4. Difference between RML and MEL	94
10.1. Comparison of data size and processing time	101
10.2. $ P $ vs. search time: (Singapore dataset)	102
10.3. Dependence on alphabet size σ	102
10.4. Dependence on out-degree	103
10.5. Comparison of labeling strategies	105
10.6. Extraction time	106
10.7. Index construction time (Singapore dataset)	106

List of Tables

3.1. Notation	27
5.1. Dataset statistics	53
5.2. Summary of algorithms and data structures used in the experiment . . .	54
5.3. Index size and index construction time (ICT)	62
10.1. Our proposed method and its competitors*	100
10.2. Statistics of each dataset	101
10.3. Compression ratio (larger is better)	103
10.4. Comparison of entropy (RML and MEL)	104

Abbreviations

BWT Burrows-Wheeler Transform

CB Compression Boosting

CiNCT Compressed-index for Network Constrained Trajectories

HWT Huffman-shaped Wavelet Tree

ISA Inverse Suffix Array

MSB Most Significant Bit

NCT Network-Constrained Trajectory

SA Suffix Array

SNT-index Suffix-array-based Network-constrained Trajectory index

SPQ Strict Path Query

TEQ Trajectory Extraction Query

TAPEQ Temporally-constrained All Path Enumeration Query

WT Wavelet Tree

Part I.
Introduction

1. Introduction

1.1. Background

In recent years, a massive amount of spatio-temporal trajectory data has become available from automobiles and smartphones. This data is used in many applications, such as traffic information systems, map generation, and location-based social networks [63]. Recent low-cost communication links allowed us to collect not only GPS data but also a massive amount of automotive sensor data, such as camera images. This rapid increase in the volume and variety of data collected has had a large impact on the development of autonomous vehicles, which is a hot topic in the automotive industry. Such advanced automotive systems are developed in a data-driven way, and the development involves frequent access to historical spatio-temporal data. Another application of such huge spatio-temporal data resources is data-driven navigation systems. Recently, for example, problems like time-dependent travel time distribution estimation [8] and time-dependent extraction of frequently-used routes between two locations [31] have been investigated on the basis of automotive trajectory data. While the recent increase in the volume of data available makes it possible to apply such data-driven methods, we also need to frequently access huge trajectory datasets when deploying such methods on a large scale. This motivates us to develop an efficient method for retrieving trajectories based on their routes.

As surveyed in [36, 42], numerous studies have been conducted on indexing and retrieving trajectories. Typically, most of these studies model trajectories as sequences of points in two or three dimensional Euclidean space, which are suitable to model raw GPS traces; these methods often support range queries, which find trajectories in a spatio-temporal rectangle. An important observation here is that trajectories generated by automobiles are usually constrained in a road network. By treating trajectories in a network-aware way, we can consider semantics of location information that is associated with road networks (e.g., type or speed-limit of road segments). For example, even if two coordinates are spatially close, they might have different semantics; one is traveling on a highway and another is not. Such a semantics is important for real applications such as navigation systems.

Basing the observation above, some studies have focused on indexing and retrieval

1. Introduction

of such *network-constrained trajectories* (NCTs) [9, 12, 25, 46, 48, 51, 58, 59]. Several types of queries have been considered for NCT representation. Range queries, which are also considered for trajectories in Euclid space, have been treated in early studies (e.g., [9, 12, 46]). Recent studies treat more complex but practical queries that are related to routing [25, 59, 60, 62]. However, there is plenty of room for research; e.g., improving efficiency of query processing, and defining new types of queries that are important in emerging applications. In order to clarify the novelty, we will make detailed comparison of our method with these related studies in each part of this thesis.

1.2. Research Objective and Contribution

In this thesis, we aim to develop a novel index structure and associated query processing algorithms that allow us to handle huge numbers of historical automotive trajectories. We focus on the fact that automobile trajectories are fundamentally constrained to a road network. Since road networks can be represented as directed graphs, each trajectory can be represented as a sequence of road segment IDs and timestamps, which we referred to as NCT in the previous subsection.

Our key finding throughout the present thesis is as follows.

- Data structures and algorithms developed in string processing field, also referred to as *stringology*, provide efficient methods for NCT processing.

We emphasize that this point is different from techniques used in the previous methods [9, 12, 25, 46, 48, 51, 58, 59], because those methods essentially rely on the traditional tree-based indexing and do not employ string algorithms.

Based on the finding, this thesis makes the following two main contributions for the research community.

- We propose a novel indexing method, **SNT-index**, that efficiently realizes several types of spatio-temporal queries for NCTs. We first provide a way to integrate the FM-index from string processing field with a method from spatial database field.
- In order to cope with the large memory footprint of FM-index used in the **SNT-index**, we propose a novel compression method of FM-index, referred to as **CiNCT**, that significantly improves not only memory footprint but also query processing speed. Our idea for trajectory compression is substantially different from those considered in database research field.

The contents that correspond to these two contributions are described in Part II and Part III, respectively. In each part, we mention detailed contributions for each method against the existing methods.

1.3. Thesis Organization

The present thesis is organized as follows.

- In the remainder of this Part I (Chapter 2), we introduce important concepts in stringology, including FM-index, commonly used in this thesis, as well as how we treat trajectories as strings.
- In Part II (Chapter 3–7), we propose the SNT-index, which integrates FM-index with spatial databases.
- In Part III (Chapter 8–12), we propose a novel FM-index compression method for trajectories. This reduces the memory footprint of the FM-index.
- At the end of the present thesis (part IV), we make concluding remarks, followed by the bibliography and the author’s publication list.

Readers who are only interested in the compression method (Part III) can skip Part II (and *vice versa*), because technical contents of these two parts are independent. Note that, as mentioned in the previous section, the motivations of these parts are strongly connected. The relationship among the four parts (Part I–IV) are depicted in Figure 1.1.

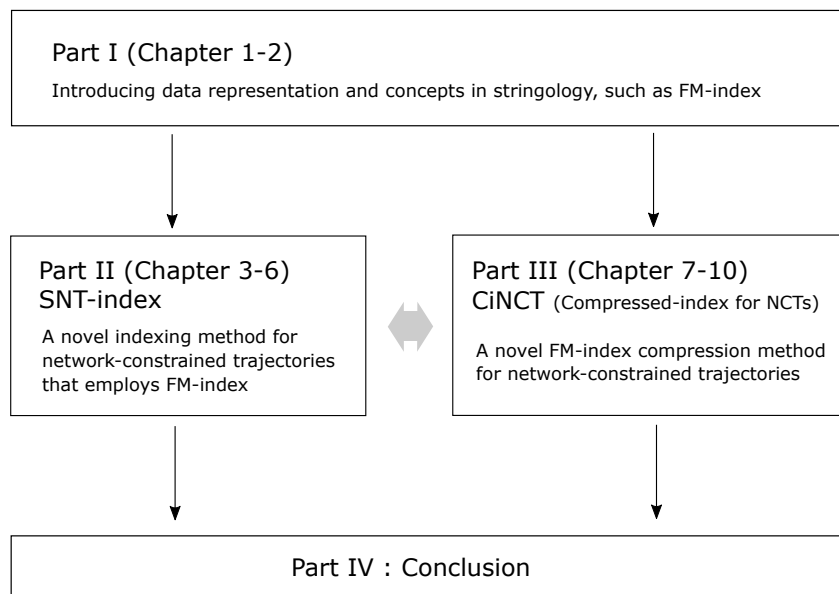


Figure 1.1.: Thesis overview

2. String Processing and Spatial Trajectories: Preliminaries

In this chapter, we describe string processing techniques, which are deeply integrated with our trajectory processing methods in the subsequent parts. In Section 2.1, we introduce the concept how to treat trajectory as a string. In Section 2.2, we describe several important concepts from the stringology, such as *suffix arrays*, *Burrows-Wheeler transform*, and *wavelet trees*. These are commonly used throughout this thesis. In Section 2.3, we provide a detailed description on a string index called FM-index, which is the most important concept in this thesis. Readers who are familiar with string processing can skip these Section 2.2 and Section 2.3.

2.1. Trajectory as a String

Throughout this thesis, we assume that road networks are modeled as directed graphs, denoted as $G = (V, E)$, where V is a set of vertices (or road intersections) and E is a set of road segments (or called *edges*). For an edge $e \in E$, we denote $head(e) \in V$ and $tail(e) \in V$ by its head and tail vertices, respectively. We assume the road network G is static, i.e., G does not vary during the data period. Although this assumption is valid for a short term, we will also show how to treat varying road networks in Chapter 4 (§ 4.5).

Trajectories are often obtained through GPS devices. Such trajectories are usually given as two dimensional time series, $x_1 \cdots x_m$ where $x_i \in \mathbb{R}^2$. Map-matching techniques are commonly used to convert such two dimensional time series into a sequence of *road segments*. Figure 2.1 visualizes map-matched trajectories in a road network. Throughout this thesis, we assume that trajectories are map-matched. Formally, we define trajectories as follows.

Definition 2.1 (Network-Constrained Trajectory; NCT) *A trajectory of length n is a sequence of road edges denoted as $e_1 \cdots e_n$, where $e_i \in E$. Importantly, e_{i-1} and e_i are assumed to be connected, i.e., $head(e_{i-1}) = tail(e_i)$. We refer to such a connected sequence of edges as a path. Also, unless otherwise noted, the term trajectory*

2. Preliminaries



Figure 2.1.: Visualization of trajectories in a road network (Roma, Italy). Colors represents frequency of occurrence of road segment in the dataset.

indicates such network-constrained trajectory (or NCT, shortly).

With this representation, a trajectory is regarded as a symbolic sequence, thus can be regarded as a string on E . Of course, trajectories consist of not only spatial information as mentioned above but also temporal information (e.g., timestamps). We introduce how to treat temporal information in Part II, and currently regard trajectories just as strings on E .

Remark 1 (Erroneous map-matching results) Simple map-matching algorithms (e.g., mapping a coordinate to the nearest road segment) often fail due to the observation noise included in GPS data. To improve the accuracy of the matching, an algorithm based on the *Hidden Markov model* (HMM) is often applied [41], where the observation is a sequence of GPS coordinates and the hidden states are road segments. However, the estimated sequence of road segments might still have error. We emphasize that all the methods presented in this thesis are independent from map-matching algorithms, and work even if the matching error is included; however, such erroneous sequences may not be found through the search algorithm (because a query sequence is in general a “correct” sequence). To fix the error, general strategies are (1) improving the data collection process (e.g., increasing sampling rate), and/or (2) using better map-matching algorithms. Although we use the most-widely used algorithm proposed in [41] throughout this thesis, we point out that there are many variants aiming at the improvement of accuracy and computational cost [30, 56].

Trajectory Strings As we introduce later, FM-index is a data structure that indexes one (large) string T . To use FM-indexes for document retrieval, a set of documents (strings) is usually concatenated into one large string. Similarly, we consider concatenation of spatial paths into a *trajectory string*, which plays important roles throughout this thesis, defined as follows.

Definition 2.2 (Trajectory String) Let $\mathbb{T} = \{p_i\}_{i=1}^N$ be a collection of trajectories in a road network $G = (V, E)$. A trajectory string T is defined as a string composed of the reversals of all paths p_i concatenated using the special character $\$$, i.e.,

$$T = p_0^r \$ p_1^r \$ \cdots \$ p_{D-1}^r \$ \#, \quad (2.1)$$

where the string p_i^r is the reverse of p_i . Here, $\#$ is another special character that marks the end of T . The alphabet set Σ is defined as $\Sigma := E \cup \{\$, \#\}$, where E is the set of road segments in the road network $G = (V, E)$.

Remark 2 As we describe later, pattern matching in FM-index proceeds backwardly. By storing reversed trajectories as in Definition 2.2, several queries can be processed efficiently.

Example 2.1 Suppose four trajectories: $p_1 = ABEF$, $p_2 = ABC$, $p_3 = BC$, and $p_4 = AD$, where A – F are road segment IDs. Then, the corresponding trajectory string is as follows.

$$T = FEBA\$CBA\$CB\$DA\$ \#. \quad (2.2)$$

Once we convert trajectories into a trajectory string, we expect that string processing techniques can be adopted. In the following sections, we introduce string algorithms that are fundamental but has not been employed in spatial trajectory processing thus far.

2.2. Strings and Related Data Structures

We describe important string concepts related to FM-index. We first introduce basic notation, followed by *suffix arrays*, *Buffows-Wheeler transform*, and *wavelet trees*. Then, we elaborate FM-index based on those concepts.

2.2.1. Strings

First, we introduce notation that is often used in stringology. Let Σ be an alphabet set and S be a string on Σ of length $|S|$.¹ To clarify the length n of S , we often use $S[0, n)$.

¹As defined in Definition 2.2, we define $\Sigma = E \cup \{\$, \#\}$ throughout this thesis.

2. Preliminaries

The size of Σ is denoted by σ . Σ^k and Σ^* denote a set of strings of length k and a set of strings of arbitrary length defined on an alphabet Σ , respectively. Strings (or arrays) have 0-based subscripts. The i th element of S and the substring from i to j are denoted by $S[i]$ and $S[i..j]$, respectively. We also use the half-open interval notation $S[i..j)$ to mean the substring that does not include $S[j]$. Further, we assume that the subscripts are defined in a circular manner, i.e., $S[i]$ is defined for any $i \in \mathbb{N}$ by $S[i \bmod |S|]$. An i th suffix of a string S is defined as a substring $S[i, |S|)$. The *concatenation* of two strings S_1 and S_2 is denoted by S_1S_2 . The *reversal* of S is denoted by S^r , and is defined by $S^r = S[|S| - 1]S[|S| - 2] \cdots S[0]$.

Lexicographic order We assume lexicographic order on $\Sigma = E \cup \{\$, \#\}$. This is important to define several concepts introduced in the subsequent sections. For our purposes, we may use any ordering, and thus we use the natural ordering defined by the road segment IDs, which are often given as integers. For the special characters ($\$$ and $\#$), we assume $\# < \$ < w$ ($\forall w \in E$).

Remark 3 (Integer-valued alphabet) Although Σ usually consists of non-integer symbols (like English letters), we map them onto integers $[0, \sigma)$ with keeping its lexicographical order. Hence, we can assume $\Sigma = [0, \sigma)$ without loss of generality. In addition, we assume that each symbol $w \in \Sigma$ appears at least once in T ; otherwise, we remove such a symbol w from Σ without loss of generality. These assumptions are useful when we define C-array in Section 2.2.3.


2.2.2. Suffix Arrays

Let T be a string of length n . Denote the i th *suffix* of T by $S_i := T[i..n)$ (see “Suffixes” in Figure 2.2). The suffixes $\{S_i \mid 0 \leq i < n\}$ are then sorted in lexicographical order (see “Sorted Suffixes” in Figure 2.2). The suffix array [33] SA of T is an integer array of length n whose j th element $SA[j]$ is equal to i iff S_i is the j th smallest suffix (where we use “smallest” in a lexicographical sense).

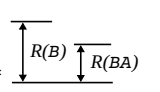
Example 2.2 In Figure 2.2, we have $SA[0] = 15$ because $S_{15} = \#$ is the smallest suffix. Besides, we have $SA[5] = 13$ because $S_{13} = A\#\#$ is the fifth smallest suffix, as we can see among the “sorted suffixes” in Figure 2.2.

We introduce an important fact regarding SA . In Figure 2.2, the substring **BA** appears as the prefix of two suffixes, S_2 and S_6 . In the list of sorted suffixes, they appear consecutively in the range $R(\mathbf{BA}) = [9, 11) = \{9, 10\}$ (R is formally defined below). We thus have $SA[9] = 6$ and $SA[10] = 2$, which tells us that **BA** appears in the original string


i	Suffixes S_i	Sorted Suffixes $S_{SA[j]}$	j	$SA[j]$	i	$ISA[i]$
0	FEBA\$CBA\$CB\$DA\$#	#	0	15	0	15
1	EBA\$CBA\$CB\$DA\$#	\$#	1	14	1	14
2	BA\$CBA\$CB\$DA\$#	\$CB\$DA\$#	2	8	2	10
3	A\$CBA\$CB\$DA\$#	\$CBA\$CB\$DA\$#	3	4	3	7
4	\$CBA\$CB\$DA\$#	\$DA\$#	4	11	4	3
5	CBA\$CB\$DA\$#	A\$#	5	13	5	12
6	BA\$CB\$DA\$#	A\$CB\$DA\$#	6	7	6	9
7	A\$CB\$DA\$#	A\$CBA\$CB\$DA\$#	7	3	7	6
8	\$CB\$DA\$#	B\$DA\$#	8	10	8	2
9	CB\$DA\$#	BA\$CB\$DA\$#	9	6	9	11
10	B\$DA\$#	BA\$CBA\$CB\$DA\$#	10	2	10	8
11	\$DA\$#	CB\$DA\$#	11	9	11	4
12	DA\$#	CBA\$CB\$DA\$#	12	5	12	13
13	A\$#	DA\$#	13	12	13	5
14	\$#	EBA\$CBA\$CB\$DA\$#	14	1	14	1
15	#	FEBA\$CBA\$CB\$DA\$#	15	0	15	0



Sort in the
lexicographical
order



$R(B)$ $R(BA)$



Inverse
Func.

Figure 2.2.: **Suffix array:** This example is based on the trajectory string given in Eq. (2.2). If suffix S_i is the j th smallest, we have $SA[j] = i$. The “ISA” at the rightmost is an inverse of SA, referred to as an *inverse suffix array*, which is introduced in Definition 2.5.

at positions 6 and 2. This observation leads us to the following important relationship between suffix arrays and pattern matching.

Lemma 2.1 *Let T be a string and Q be a query pattern of length m that appears at least once in T . Then, there exists a unique range $R(Q) := [sp, ep)$ such that $T[SA[j], SA[j] + m) = Q$ iff $sp \leq j < ep$.*

PROOF Considering all the suffixes of T sorted in the lexicographical order, there exists a unique range of the sorted suffixes such that the prefixes of the sorted suffixes in the range have a given pattern Q . Let the indexes of such a range be $R(Q) = [sp, ep)$. Remembering that $S_{SA[j]}$ is the j th smallest suffix, we have that $S_{SA[j]}[0, m) = Q$ and $sp \leq j < ep$ are equivalent. This leads to the conclusion because, for any i and m , we have $S_i[0, m) = T[i, i + m)$.

Definition 2.3 (Suffix Range) *Given a pattern $Q \in \Sigma^*$ and a string T , we refer to the unique range $R(Q)$ determined by Lemma 2.1 as the suffix range of Q .*

2.2.3. Burrows-Wheeler Transform

The *Burrows–Wheeler transform* (BWT) [5] is closely related to pattern matching and is used in FM-index. This is an invertible transform of a string, formally defined as follows.

2. Preliminaries

i	Rotations		j	Sorted Rotations	BWT
0	FEBA\$CBA\$CB\$DA\$#		0	#FEBA\$CBA\$CB\$DA\$	\$
1	EBA\$CBA\$CB\$DA\$#F		1	\$#FEBA\$CBA\$CB\$DA	A
2	BA\$CBA\$CB\$DA\$#FE		2	\$CB\$DA\$#FEBA\$CBA	A
3	A\$CBA\$CB\$DA\$#FEB		3	\$CBA\$CB\$DA\$#FEBA	A
4	\$CBA\$CB\$DA\$#FEBA		4	\$DA\$#FEBA\$CBA\$CB	B
5	CBA\$CB\$DA\$#FEBA\$		5	A\$#FEBA\$CBA\$CB\$D	D
6	BA\$CB\$DA\$#FEBA\$C		6	A\$CB\$DA\$#FEBA\$CB	B
7	A\$CB\$DA\$#FEBA\$CB	↻	7	A\$CBA\$CB\$DA\$#FEB	B
8	\$CB\$DA\$#FEBA\$CBA	Sort	8	B\$DA\$#FEBA\$CBA\$C	Last Column C
9	CB\$DA\$#FEBA\$CBA\$		9	BA\$CB\$DA\$#FEBA\$C	
10	B\$DA\$#FEBA\$CBA\$C		10	BA\$CBA\$CB\$DA\$#FE	E
11	\$DA\$#FEBA\$CBA\$CB		11	CB\$DA\$#FEBA\$CBA\$	\$
12	DA\$#FEBA\$CBA\$CB\$		12	CBA\$CB\$DA\$#FEBA\$	\$
13	A\$#FEBA\$CBA\$CB\$D		13	DA\$#FEBA\$CBA\$CB\$	\$
14	\$#FEBA\$CBA\$CB\$DA		14	EBA\$CBA\$CB\$DA\$#F	F
15	#FEBA\$CBA\$CB\$DA\$		15	FEBA\$CBA\$CB\$DA\$#	#

Figure 2.3.: **Burrows-Wheeler Transform:** BWT corresponds to the last column of the sorted rotations of T . This example is based on the trajectory string $T = \text{FEBA\$CBA\$CB\$DA\$#}$ in Eq. (2.2).

Definition 2.4 (BWT) *Given a string T of length n , BWT of T is also a string of length n , defined as follows.*

$$T_{\text{bwt}}[i] = T[SA[i] - 1] \quad (0 \leq i < n), \tag{2.3}$$

where SA is the suffix array of T . Note that, if $SA[i] = 0$, we have $T_{\text{bwt}}[i] = T[-1] = T[n - 1]$ because of the circular subscripts as mentioned in Section 2.2.1.

Example 2.3 *Besides the definition above, we provide another explanation that is equivalent to Definition 2.4. Given a string T , its BWT T_{bwt} also can be defined as the last column of all sorted rotations, as depicted in Figure 2.3. As a result, this example generates the BWT as follows.*

$$T_{\text{bwt}} = \$AAABDBBCCE$$$F#. \tag{2.4}$$

Note that the order of the sorted rotations is equivalent to that corresponding to Figure 2.2, because the smallest symbol $\#$ appears only once in T .

Next, we introduce an important property of BWT called *LF-mapping*, that makes BWT invertible. For the sake of this, we introduce *inverse suffix array*, *rank function*, and *C-array*, respectively defined as follows.

Definition 2.5 (Inverse suffix array; ISA) *Consider a string T of length n and its suffix array SA . Inverse suffix array, denoted by ISA , is an integer array of length n defined as the inverse function of SA , i.e., ISA satisfies $ISA[SA[i]] = i$ for $0 \leq i < n$. Note that the inverse function is well-defined because SA is a one-to-one map by definition.*

Definition 2.6 (Rank function) Given a string T of length n , a symbol $w \in \Sigma$, and a position $0 \leq i < n$, the rank function, denoted by $\text{rank}_w(T, i)$, returns a number of w 's occurrences within a prefix $T[0, i)$.

Definition 2.7 (C-array) Let T be a string on an alphabet Σ of size σ . An element $C[w]$ of the C-array of T , denoted by $C[0, \sigma]$, is defined as the number of elements in T that are (strictly) lexicographically smaller than $w \in \Sigma$. Formally, $C[w] := \#\{a \in \Sigma \mid a < w\}$.

The length of C-array is $\sigma + 1$. By definition, we always have $C[0] = 0$ and $C[\sigma] = n$, and C is an increasing function. Note that the C-arrays of T and T_{bwt} are equivalent because the numbers of symbol occurrences in T and T_{bwt} are the same by definition.

Example 2.4 We confirm the concepts above using the running example (note: $T_{\text{bwt}} = \$AAABDBBCCCE\$\$\$F\#$).

- $ISA[i]$ can be regarded as a function that returns the lexicographical rank of i th suffix S_i . In Figure 2.2, as the suffix $S_{15} = \#$ is the zeroth smallest suffix, we have $ISA[15] = 0$.
- Consider the rank function for $\text{rank}_w(T_{\text{bwt}}, i)$. If $i = 5$, we have $T_{\text{bwt}}[0, 5) = \$AAAB$; hence, we have $\text{rank}_A(T_{\text{bwt}}, 5) = 3$ because $w = A$ occurs three times.
- Finally, we see the C-array of T_{bwt} . For example, $C[A] = 5$ because the frequency of symbols $\$, \# < A$ is five in total. Similarly, $C[B] = 8$ because $\$, \#, A$ appear eight times.

Property of BWT Let us show an important observation regarding BWT. In the first column of Figure 2.3, we observe four “\$” from $i = 1$ to $i = 4$. At the last column, we also see four “\$”. Importantly, the order of these symbols are the same. For example, “\$” for $i = 1$ in the first column corresponds to that for $i = 0$ in the last column. Similarly, $i = 2$ at the first column is connected to $i = 11$; $i = 3$ goes to $i = 12$; and $i = 4$ goes to $i = 13$. This is because that the order of sorted rotations with the same first symbol is kept even if the first symbol is removed.

Now, we are ready to explain the LF-mapping. Roughly, the LF-mapping maps the last column of the sorted rotations in Figure 2.3 to the corresponding positions of the first column (note: “LF” stands for Last-to-First). For example, consider the last character \$ of the zeroth sorted rotation in Figure 2.3. This \$ appears at the first position in the first sorted rotation. Therefore, $i = 0$ is mapped to $i = 1$. Similarly, the last A for $i = 1$ is

2. Preliminaries

mapped to the first A for $i = 5$. By definition, this mapping is denoted by $ISA[SA[i] - 1]$ (rank of the suffix corresponding to the previous position of the i th smallest suffix). The following well-known proposition provides how to compute the mapping.

Proposition 2.1 (LF-mapping) *Suppose $T_{\text{bwt}}[i] = w$. We have*

$$ISA[SA[i] - 1] = C[w] + \text{rank}_w(T_{\text{bwt}}, i). \quad (2.5)$$

PROOF By definition, $\text{pos} = \text{rank}_w(T_{\text{bwt}}, i)$ represents the number of w in $T_{\text{bwt}}[0, i)$. This corresponds the rank of $w = T_{\text{bwt}}[i]$ among all occurrences of w . In the first column of the sorted rotations, w begins at position $C[w]$. Therefore, pos -th w appears at position $C[w] + \text{pos}$.

2.3. FM-index

FM-index [11] is a data structure based on BWT.² Concretely, given a string T to be indexed, we store its BWT in a data structure called *wavelet tree*. We first introduce two types of queries used in this thesis (i.e., pattern matching and extraction queries), and then introduce wavelet trees, which enable efficient query processing.

2.3.1. Pattern Matching Query

Throughout this thesis, we define *pattern matching query*, also referred to as *suffix range query*, in the following sense.

Definition 2.8 (Pattern matching query) *Given a string $T \in \Sigma^*$ and a query $Q \in \Sigma^*$, pattern matching query, or suffix range query, is to find the suffix range $R(Q)$, which was defined in Definition 2.3.*

Algorithm With BWT T_{bwt} , we can answer the pattern matching query as shown in Algorithm 1. This algorithm starts from the last symbol of a query Q (of length m) and sequentially updates the suffix range $[sp, ep)$. First, we initialize this range as $sp := C[Q[m - 1]]$ and $ep := C[Q[m - 1] + 1]$, which corresponds to the suffix range of $Q[m - 1]$.

Suppose we are at the i th step ($i > 1$). Then, $[sp, ep)$ corresponds to the suffix range of $Q[i + 1, m)$. Considering the next symbol $w = Q[i]$, we must have the following: substrings in T corresponding to $Q[i, m)$ appear at positions where the substring $Q[i + 1, m)$ appears *and* w is the previous symbol. Such positions are equivalent to those

²“FM” stands for the names of authors of the original paper by Ferragina and Manzini.

Algorithm 1: Pattern Matching Query: Finding the suffix range $R(Q) = [sp, ep)$ for a given query Q of length m based on T_{bwt}

Input: BWT string: T_{bwt} , Query string: Q , Integer array: C

- 1 $w \leftarrow Q[m - 1]$
- 2 $sp \leftarrow C[w]$
- 3 $ep \leftarrow C[w + 1]$
- 4 **for** $i \leftarrow 2$ **to** m **do**
- 5 $w \leftarrow Q[m - i]$
- 6 $sp \leftarrow C[w] + \text{rank}_w(T_{\text{bwt}}, sp)$
- 7 $ep \leftarrow C[w] + \text{rank}_w(T_{\text{bwt}}, ep)$
- 8 **if** $sp \geq ep$ **then return** `NotFound`
- 9 **return** $[sp, ep)$

satisfying $T_{\text{bwt}}[sp, ep) = w$ since $T_{\text{bwt}}[i]$ represents the previous symbol of the position $SA[i]$. Therefore, using LF-mapping, we can obtain the suffix range of $Q[i, m)$ as follows.

$$sp := C[w] + \text{rank}_w(T_{\text{bwt}}, sp),$$

$$ep := C[w] + \text{rank}_w(T_{\text{bwt}}, ep).$$

If w does not appear in $T_{\text{bwt}}[sp, ep)$, it implies the substring $Q[i, m)$ never appears in T . Hence we return `NotFound` at Line 8.

2.3.2. Substring Extraction Query

In this thesis, we also focus on another query, *substring extraction query*, that recovers a part of the original string T from the BWT string T_{bwt} . This is formulated as follows.

Definition 2.9 (Substring extraction query: Type I) *Given a position on the suffix array j and an extraction length L , the substring extraction query recovers a substring $T[SA[j] - L, SA[j])$ from the BWT T_{bwt} of T .*

In addition, we also consider a modified version of the query. Instead of the extraction length L , we specify a symbol $v \in \Sigma$ and continue extraction until v appears.

Definition 2.10 (Substring extraction query: Type II) *Given a position on the suffix array j and a symbol $v \in \Sigma$, we extract a substring $T[SA[j] - L', SA[j])$ from the BWT T_{bwt} of T , where $T[SA[j] - L'] = v$ and $T[SA[j] - k] \neq v$ for $1 \leq k < L'$.*

Example 2.5 *Suppose $j = 3$ in Figure 2.3 and we extract a substring of length $L = 4$. As $SA[3] = 4$, the query extracts $T[SA[3] - L, SA[3]) = T[0, 4) = FEBA$. In fact, this corresponds to the last four symbols at $j = 3$ rd row (underlined). In other words, a length*

2. Preliminaries

L suffix of the j th smallest sorted rotation is extracted by this query. Furthermore, if $j = 15$ and $L = |T| = 16$, the substring extraction query recovers the whole T . In this sense, BWT is invertible.

Algorithm Algorithm 2 and Algorithm 3 show how we can extract substrings from T_{bwt} . Starting from a given position j , these algorithms read $T_{\text{bwt}}[j]$ (using `access` method) and repeatedly apply the LF-mapping to update the current position j until meeting the stopping condition.

Algorithm 2: *FM-extract*(T_{bwt}, j, l): Extract the original substring $T[i - l, i)$ of length l , where $i = SA[j]$.

```

1  $S \leftarrow$  empty string
2 for  $k \in 1, \dots, l$  do
3    $w \leftarrow$  access( $T_{\text{bwt}}, j$ ) //  $T_{\text{bwt}}[j]$ 
4    $S \leftarrow wS$  // concatenate  $w$ 
5    $j \leftarrow C[w] + \text{rank}_w(T_{\text{bwt}}, j)$ 
6 return  $S$  as  $T[i - l, i)$ 

```

Algorithm 3: *FM-extract-until* (T_{bwt}, j, v): This variant decodes the original substring until v is found.

```

1  $S \leftarrow$  empty string
2 repeat
3    $w \leftarrow$  access( $T_{\text{bwt}}, j$ ) //  $T_{\text{bwt}}[j]$ 
4    $S \leftarrow wS$  // concatenate  $w$ 
5    $j \leftarrow C[w] + \text{rank}_w(T_{\text{bwt}}, j)$ 
6 until  $w \neq v$ 
7 return  $S$ 

```

2.3.3. Wavelet Trees

Algorithms 1–3 all require computing $\text{rank}_w(T_{\text{bwt}}, j)$. This indicates that fast calculation of rank_w enables the fast execution of those algorithms because all the operations except for $\text{rank}_w(T_{\text{bwt}}, i)$ are merely either substitutions or summations. However, naïve calculation of rank_w with cumulative counting incurs an unacceptable $O(|T_{\text{bwt}}|)$ time.

A *wavelet tree* [16] storing T_{bwt} enables fast calculation of $\text{rank}_w(T_{\text{bwt}}, i)$; its time complexity does not depend on the data size $|T_{\text{bwt}}|$. In the following, we explain wavelet tree using an example in Figure 2.4. See [38] for detailed description.

Data structure A wavelet tree storing $S = \$AAABDBBCCCE\$\$F\#$ is depicted in Figure 2.4. The bit representation of each symbol $w \in \Sigma$ is predefined, as illustrated on the right side (e.g., Huffman coding based on the frequency of each symbol in S). Each node v in the tree stores a bit vector B_v . In the root node v_0 , B_{v_0} stores the most significant bit (MSB) of each symbol in S . As the MSB of $\{\$, A, C\}$ is 1, those 1's are stored in B_{v_0} with keeping the original order. For the others, we store 0's. At the second level, the symbols are divided into two parts based on the bit value at the first level, while keeping the ordering. Each bit vector stores the second MSB. Repeating such partitioning recursively, we obtain the wavelet tree. In fact, B_v is stored in a *succinct dictionary* [20, 49], which is a bit vector that supports a bit-wise rank (i.e., $rank_0(B_v, j)$ and $rank_1(B_v, j)$) in $O(1)$ time using a lookup table.

Rank computation With wavelet trees, we can compute $rank_w(S, j)$ by traversing the tree along with the bit representation of w . We explain how to compute $rank_w(S, j)$ with the wavelet tree shown in Figure 2.4. Suppose $j = 5$ and $w = \$$, whose bit representation is '10'. At the root node, we compute the bit-wise rank of $\$$'s MSB ($=1$), that is, $rank_1(B_{v_0}, 5) = 4$. We then go to the right child, say v , because MSB 1 corresponds to the right child. We know four symbols come from the parent node owing to the rank value at the parent. As the second MSB of $\$$ is 0, we compute $rank_0(B_v, 4) = 1$. Here, we used $j = 4$, which is the rank result at the parent. Then, we go to the left child, which is a leaf node. Arriving at a leaf node, we return the current bit-wise rank value, 1, as $rank_{\$}(S, 5)$.

Remark 4 Importantly, the time complexity of rank computation does not depend on the length of string S because bit-wise rank is (assumed to be) computed in $O(1)$ time, as mentioned above. Therefore, the query processing algorithms of FM-index run very efficiently even for large data.

Remark 5 (Time and space complexities) In Part III, time complexity of rank computation and space complexity (i.e., how small wavelet tree is) are elaborated. In Part II, we employ uncompressed wavelet trees and does not go into details of the FM-index data structure itself. Here, we mention the time/space complexities of the uncompressed wavelet tree. Unlike HWTs, the uncompressed wavelet tree uses equi-length bit representation; In our example, all symbols have 3 bits because the alphabet size of Σ is $8 = 2^3$ ($\Sigma = \{\#, \$, A, B, C, D, E, F\}$). This means that the total size of bit vectors in the uncompressed wavelet tree is $|T| \lg |\Sigma|$ bits. The time complexity of rank computation is $O(\lg |\Sigma|)$ because the depth of wavelet tree is $\lg |\Sigma|$ and the complexity of bit-wise rank is $O(1)$. Hence, the time complexity of Algorithm 1 is $O(|P| \lg |\Sigma|)$,

2. Preliminaries

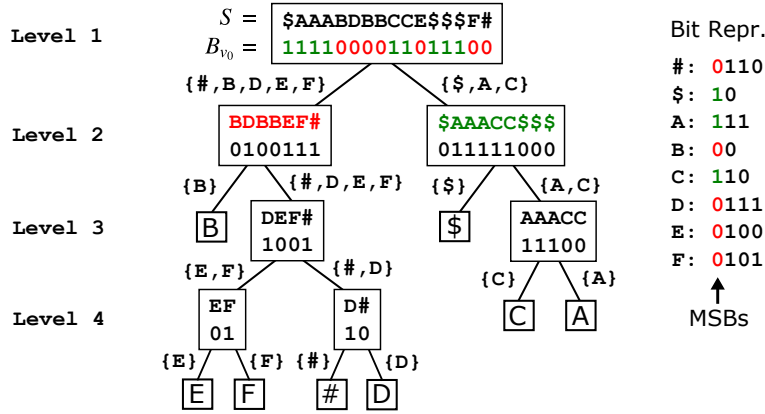


Figure 2.4.: Wavelet tree: a bit representation of each symbol in a string S is stored in a binary tree (this example is the HWT of the BWT of the trajectory string Eq. (2.4)). Note that only bit vectors are stored in each node.

because rank values are computed $|P|$ times in the algorithm. Similarly, Algorithm 2 runs in $O(L \lg |\Sigma|)$ time. In Part II, we refer to these complexity results.

2.4. Summary

In this chapter, we first described how to represent trajectory as a string. Then, we described FM-index and the related string data structures and algorithms. In particular we showed two kinds of queries, namely *pattern matching query* and *extraction queries*. These queries can be efficiently answered with FM-index, a data structure in which BWT string is stored in a wavelet tree. In Part II, we propose a data structure for spatio-temporal queries for trajectories based on FM-index introduced here. As FM-index is memory-intensive if used for large data, we need to reduce memory usage. In Part III, we propose a compression method for FM-index that significantly reduces the memory footprint.

Part II.
Trajectory Indexing

3. Research Issues and Problem Definition

3.1. Research Issues

As we discussed in Chapter 1, data management for large trajectory data is important in modern applications, such as data-driven navigation systems and automotive development.

In Part II, we develop an efficient index structure and associated query processing algorithms that allow us to handle huge numbers of historical trajectories in road networks. Figure 3.1 shows an overview of the proposed method. As we defined in Section 2.1, we focus on the fact that automobile trajectories are fundamentally constrained to a road network. Such network-constrained trajectories (NCTs) can be represented as a sequence of road segment IDs and timestamps. Figure 3.2 illustrates four example NCTs. In spatial database research, several papers have focused on methods for indexing NCTs [9, 12, 25, 46, 48, 51, 58, 59]. Unlike these existing methods, our method, referred to as SNT-index,¹ employs *suffix arrays*, which enable efficient retrieval of NCTs. In the remainder of this section, we motivate our research and describe some issues by reviewing the existing NCT-indexing methods in terms of query types they allow and data structures they use.

Several types of queries have been considered for NCTs. Early studies (e.g., Pfooser and Jensen [46], FNR-tree [12], and MON-tree [9]) mainly focused on *spatio-temporal range queries*, which find all trajectories that touch *one of* the road segments in a given spatial rectangle R during a given time interval I . This is one of the most widely-studied types of query in the spacial database research field. On the other hand, for applications like automotive development or navigation systems, data access under route rather than spatial rectangle constraints becomes important (detailed application scenarios are shown in Section 3.2.2). Such queries were first considered in Popa et al. [48], which find trajectories that touch one of road segments in a given path P (i.e., a sequence of road segments) during a given time interval I . *Strict path queries* (SPQs) [25] are

¹SNT-index stands for Suffix-array-based network-constrained trajectory index

3. Research Issues and Problem Definition

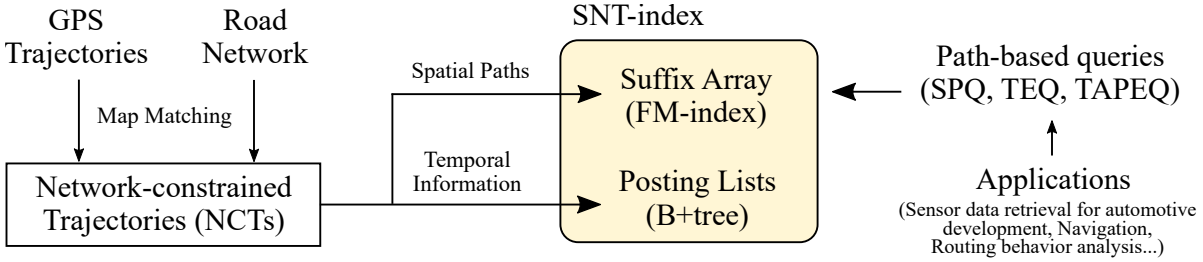


Figure 3.1.: Overview of SNT-index, suffix-array-based network-constrained trajectory index.

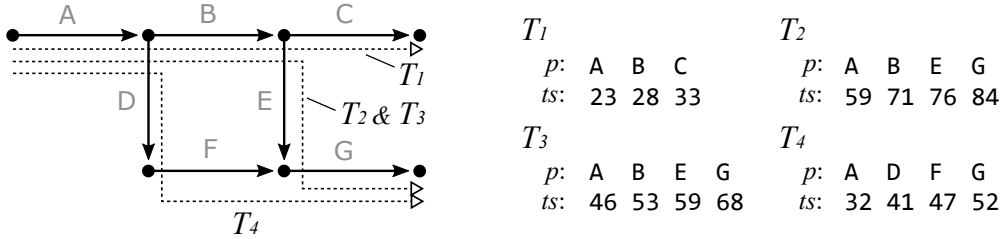


Figure 3.2.: Examples of NCTs (T_1, T_2, T_3 , and T_4). Here, p and ts show a sequence of traveled road segment IDs, and the corresponding timestamps, respectively.

another typical example of this type of query and are one of the types treated in this thesis. SPQs find all trajectories that have traveled along a given path P during a given time interval I . In this thesis, we mainly focus on such *path-based queries*, for which the precise shape of the path plays an important role (in the later sections, we introduce novel types of path-based queries).

The existing NCT-indexing methods all use a common data structure [9, 12, 25, 59], an example of which is shown in Figure 3.3. For each road segment $e \in E$, a table Φ_e is created, its records essentially consisting of (tid, ts) pairs, where tid is a trajectory ID and ts is the timestamp when trajectory tid traversed the road segment e . Then, for each Φ_e , a tree-based index is built using the ts values to support fast temporal filtering. In this example, ts is indexed using a B⁺-tree, as proposed by Vieira et al. [59].²

This data structure (Figure 3.3) is similar to an *inverted index* in document retrieval [34], which typically consists of postings lists Φ_e for each term e . Each postings list Φ_e stores the IDs of the documents that include term e . In the NCT-indexing methods discussed above, the trajectory ID (tid) and road segment ID (e) correspond to the document ID and term ID, respectively. We thus refer to such NCT-indexing methods as *inverted-index-like* methods in this thesis.

²In contrast, FNR-tree [12] stores both of entered and left timestamps and uses 1D R-tree.

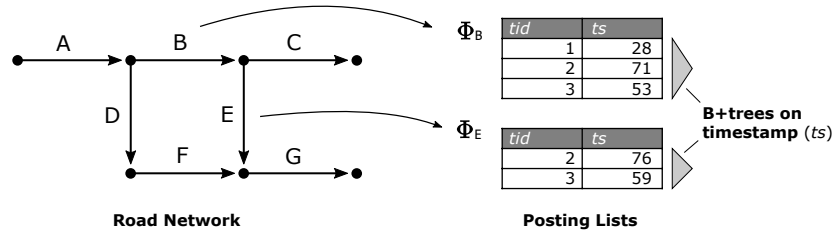


Figure 3.3.: Typical data structure used for NCT indexing (referred to as an *inverted-index-like method* in this thesis). Posting lists are defined for each road segment and store pairs consisting of a trajectory IDs (tid) and a timestamp (ts). A tree-based index is built based on the timestamps.

Research Issue and Main Idea For path-based queries, the inverted-index-like approach is slow due to the need for a large number of disk accesses. As an example, let us consider an SPQ for a given route P and time interval I . Since the SPQ aims to find the NCTs that visited *all of* the road segments e in P , the simplest approach based on an inverted-index-like method, would be as follows.

1. **(Select)** For each $e \in P$, find the trajectory IDs $S_e := \{tid\}$ that traveled along e during I by retrieving the postings list Φ_e (using B⁺-tree search for temporal selection).
2. **(Join)** Find the trajectory IDs that are common to all the S_e , that is, $\cap_{e \in P} S_e$. (To be exact, we also need to ensure that the ordering is valid.)

The number of B⁺-tree retrievals and join operations is $|P|$, which is problematic when $|P|$ is large. In particular, the first step is slow because the posting lists Φ_e must be stored on disk for huge trajectory datasets. Krogh et al. [25] proposed a method of reducing the number of these operations, but it still requires $O(|P|)$ operations to ensure there are no false positives (we review this method in Section 4.2.2). In data-driven automotive development, there are situations where we need to handle frequent SPQs, motivating us to develop a more efficient method for dealing with them (see the application scenarios in Section 3.2.2).

The proposed **SNT-index** addresses the problem discussed above. Technically, the **SNT-index** incorporates another important paradigm in document retrieval, the *suffix array* (SA) [33], which we introduced in Section 2.2 for efficient pattern matching. However, applying SAs for NCT-indexing is not straightforward because SAs cannot take account of temporal information. Our research question is thus summarized as follows:

- *How can we integrate the suffix arrays into NCT-indexing?*

3. Research Issues and Problem Definition

We answer this question by employing an *inverse suffix array*, and we show that inverse suffix arrays can successfully combine the conventional NCT-indexing methods (i.e., the inverted-index-like methods discussed above) used for spatial databases and the suffix arrays used in stringology. This technique allows us to achieve flexible and fast query processing, not only for the abovementioned SPQs but also for other path-based queries. In the subsequent sections, we formulate these queries and propose suitable query processing algorithms. In order to make the algorithms efficient, **SNT-index** employs an *FM-index* [11], an efficient suffix array implementation. As we described in Section 2.3, FM-indexes support not only pattern matching queries but also fast *substring extraction queries*. The use of substring extraction queries enables **SNT-index** to support several other types of queries besides SPQ. Experiments with real data show that the proposed algorithms are orders of magnitude faster than baseline algorithms that do not use suffix arrays.

That said, the price that must be paid for this significant improvement in query processing is that dynamic updating becomes impossible. We should emphasize, however, that this is not a big problem for the applications considered here, because our main motivation is the development of automotive systems, where retrieving historical data is more important. Nonetheless, we will also discuss strategies for mitigating this updating problem.

Contributions The technical contributions of this Part II can be summarized as follows.

- We propose **SNT-index**, which provides a concrete method of integrating suffix arrays with inverted-index-like approaches to NCT indexing. We incorporate two concepts from stringology into conventional NCT indexing: inverse suffix arrays and FM-index (a compact implementation for suffix arrays).
- We formulate various types of spatio-temporal path-based queries and propose efficient algorithms for processing these queries. These algorithms utilize properties of inverse suffix arrays and FM-indexes that have not been considered in NCT-indexing before.
- We provide a practical and efficient method of appending new data to the **SNT-index**, even though the FM-index is a static index.
- Experiments with real datasets show that the proposed algorithms can process target queries for more than one million trajectories in a few tens of milliseconds, which is a significant improvement over baseline algorithms that do not use suffix arrays.

Table 3.1.: Notation

Notation	Description
$G = (V, E)$	Road network (directed graph)
$\mathbb{T} = \{(p_{tid}, ts_{tid})\}_{tid=0}^{D-1}$	Target NCTs
Φ_e	Posting-list (a set of tuples) of a road segment $e \in E$
$U, W, X, Y, Z \subset \Phi_e$	Temporally-filtered posting lists
$x.\mathbf{attr}$	The value of attribute \mathbf{attr} of tuple $x \in X$
T, T_{bwt}	Trajectory string and its Burrows-Wheeler transform
$\mathcal{F}(T)$	FM-index of T
$\Sigma := E \cup \{\$, \#\}$	Alphabet set of T and T_{bwt}
$R(P) (= [sp, ep])$	Range of suffix for a pattern P
$S[i, j], S^r$	Substring (from i to j) and the reversed string of S
$C[w]$	The number of symbols in T_{bwt} lexicographically smaller than $w \in \Sigma$
$\mathit{rank}_w(T_{\text{bwt}}, k)$	Rank function. The number of $w \in \Sigma$ in $T_{\text{bwt}}[0..k)$

Outline of Part II The rest of Part II is organized as follows. In the rest of Chapter 3, we describe the problem definition. The several types of path-based queries considered in Part II are introduced. In Chapter 4, we describe the **SNT-index** data structure. In addition, algorithms for the target path-based queries are proposed. We also introduce a series of baseline algorithms and discuss the efficiency of the proposed algorithms. Furthermore, we discuss how to append new data to **SNT-index**. In Chapter 5, the results of experiments with real trajectory datasets are presented. Finally, we discuss related work and the conclusions of Part II in Chapter 6 and Chapter 7, respectively.

3.2. Problem Definition and Path-based Queries

In this section, we give formal definitions of our queries of interest — SPQs, TEQs and TAPEQs. We also discuss potential applications of these queries.

Throughout Part II, we basically use notation from string algorithms defined in Section 2.2. We also use notation from the relational database field. A relation (table) X is a set of tuples (also referred to as records). For each tuple $x \in X$, an attribute named \mathbf{attr} is accessed by $x.\mathbf{attr}$. For example, table $\Phi_{\mathbf{B}}$ in Figure 3.3 has three tuples, and we can access the timestamp of a given tuple $x \in \Phi_{\mathbf{B}}$ by $x.\mathbf{ts}$. Moreover, basic SQL is used to keep the notation simple. For convenience, we show the notation used throughout Part II in Table 3.1.

3.2.1. Data Model

Road networks are treated as directed graphs $G = (V, E)$ and each trajectory is represented as a path on G , as we introduced in Section 2.1. Similar to the existing methods, we additionally consider temporal information associated with the spatial path, as a sequence of timestamps.

Definition 3.1 *A network-constrained trajectory (NCT) of length m is a tuple (p, ts) where p is a path of length m on G , and ts is a sequence of timestamps of length m . Here, $ts[i]$ means the timestamp when the moving object left $p[i]$. A set of NCTs to be indexed is denoted by $\mathbb{T} := \{(p_{tid}, ts_{tid})\}_{tid=0}^{D-1}$, where tid is the trajectory identifier (ID) and D is the number of trajectories in \mathbb{T} .*

Remark 6 (Other possibilities of NCT representation) As mentioned above, we define $ts[i]$ as the timestamp when the moving object left the corresponding edge $p[i]$. We could also have considered $ts_{in}[i]$ to be the timestamp when it entered the edge $p[i]$ (like FNR-tree), but the explicit storage of $ts_{in}[i]$ leads to redundancy because $ts_{in}[i]$ is equal to $ts[i - 1]$. We thus consider only the timestamp when the object left the edge. This prevents us from searching based on $ts_{in}[i]$, but we do not consider this a problem for the following reason. As will be shown later, range searches that check whether a timestamp is within a given time interval I are important for our application. If the interval I is large, measured in hours or days, the two conditions $ts[i] \in I$ and $ts_{in}[i] \in I$ are almost equivalent because the difference $ts[i] - ts_{in}[i]$ is much smaller than $|I|$. We can therefore omit storing $ts_{in}[i]$ because I is measured in hours or days in our target application. We should, however, emphasize that it would be easy to store $ts_{in}[i]$ in SNT-index, using a similar way to the FNR-tree.

One might also wonder why we have adopted an *edge-based* representation, instead of the *vertex-based* representation that represents NCTs as sequences of vertexes. Although these two representations are essentially equivalent and convertible with each other, we chose an edge-based representation for the following reason: the number of NCTs in \mathbb{T} that follow a road segment $e \in E$ during a given I is typically much smaller than the number that pass the edge's tail node v because many other NCTs also pass v going in other directions. This implies that join operations (or other refinement steps) can be more expensive if we adopt a vertex-based representation.

Storage model We essentially store all data in a secondary memory, such as magnetic or solid-state disks. We also allow the data to be stored in the main memory, but our assumption is that we have sufficient secondary memory (e.g., tens of terabytes) while main memory is limited (e.g., tens of gigabytes).

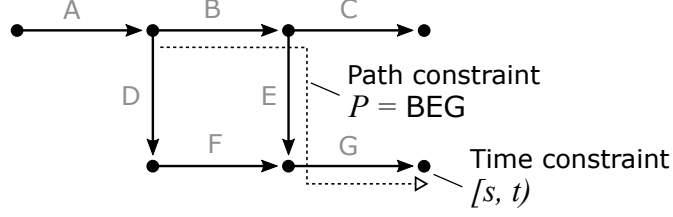


Figure 3.4.: Strict path query: $\text{SPQ}_{\text{simple}}(P, I)$ finds all trajectories that include P as a subtrajectory and where the timestamp at the end of P is within I .

3.2.2. SPQ: Strict Path Query

Given a path P and a time interval I , *strict path queries* (SPQs) aim to find all the trajectories in \mathbb{T} that traveled the subtrajectory P during the time interval I . It is formally defined as follows.

$$\text{SPQ}(P, I) = \{tid \mid \exists i \text{ s.t. } (p_{tid}[i..i+|P|] = P) \wedge (ts_{tid}[i] \in I) \wedge (ts_{tid}[i+|P|-1] \in I)\}. \quad (3.1)$$

The timestamps are constrained at the beginning and end of the given path, namely $P[0]$ and $P[|P|-1]$, respectively.³ In the present study, we also consider a simplified version of an SPQ (see Fig. 3.4):

$$\text{SPQ}_{\text{simple}}(P, I) = \{tid \mid \exists i \text{ s.t. } (p_{tid}[i..i+|P|] = P) \wedge (ts_{tid}[i+|P|-1] \in I)\}. \quad (3.2)$$

This is different from a standard SPQ in that the timestamp is only constrained at the end of P . Although our method supports both types of time constraints, $\text{SPQ}_{\text{simple}}$ can be processed more efficiently. We also note that $\text{SPQ}(P, I)$ and $\text{SPQ}_{\text{simple}}(P, I)$ return similar results when the time interval I is larger than the typical time needed to travel along P (see also the discussion in Remark 6).

To keep the explanation simple, we assume that a query P does not include loops, i.e., $P[i] \neq P[j]$ implies $i \neq j$. We would like to emphasize that this assumption is usually satisfied in practice.

SPQ Examples Here, we provide some examples of SPQs based on the example NCTs in Figure 3.2. Consider $P = AB$ and $I = [0, 70)$. The NCTs that have AB as a subtrajectory are T_1, T_2 , and T_3 . The T_1 timestamps for the road segments A and B

³This time constraint is slightly different from the one originally introduced in Krogh et al. [25], where the timestamp for the first edge $P[0]$ is constrained on the *head* node of $P[0]$. In contrast, our definition constrains the *tail* node of $P[0]$. Although we consider this modification is not to be critical practice, we also note that the original constraint can easily be applied by posterior refinement with the additional information ts_{in} (see the discussion in Section 3.2.1).

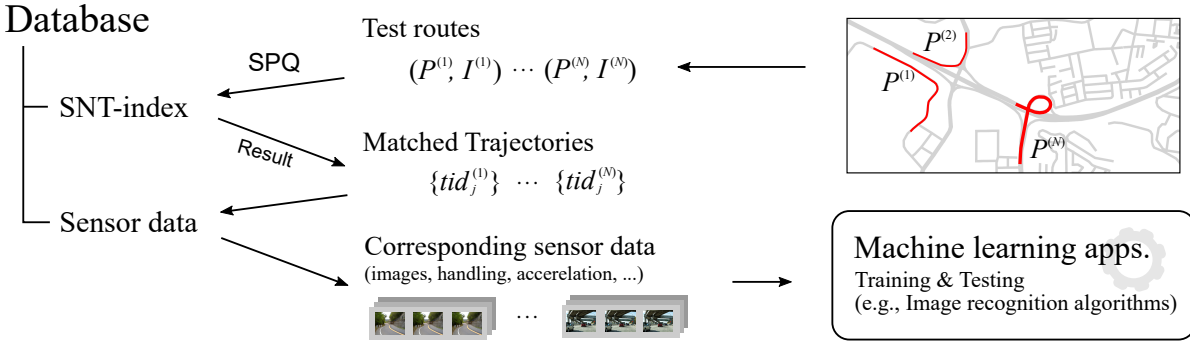


Figure 3.5.: SPQ application scenario

are 23 and 28, respectively. Hence, T_1 satisfies the time constraint I . Also, the T_3 timestamps for the road segments A and B are 46 and 53, respectively. Hence, T_3 also satisfies I . However, T_2 traveled along B at $ts = 71 \notin [0, 70)$, indicating that only T_1 and T_3 satisfy the constraint and $SPQ(P, I) = \{1, 3\}$. Similarly, if $P = ABE$ and $I = [0, 70)$, we obtain $SPQ(P, I) = \{3\}$ because T_1 does not follow $P = ABE$. Finally, if $P = ABE$ and $I = [0, 50)$, $SPQ(P, I) = \emptyset$ (the empty set) because the T_3 's timestamp at E is $59 \notin [0, 50)$.

SPQ Application Scenarios Let us consider a machine learning application that aims to detect lane markings based on in-vehicle camera images (Figure 3.5). We have to test, as exhaustively as possible, whether the image recognition algorithm works, even in extreme conditions. To do this, we first prepare thousands of test routes involving extreme conditions, such as tight curves, rainy nights, and backlit conditions. These routes can be extracted based on historical weather conditions and roadmap geometries. For each given route $P^{(i)}$ and corresponding time interval $I^{(i)}$, we execute an SPQ and obtain the trajectory ID set $\{tid_j^{(i)}\}$. Based on the retrieved trajectory IDs, we can obtain the corresponding camera images that are required for the exhaustive image recognition algorithm test.⁴ In this application, quick retrieval of historical data is important because thousands of queries must be executed.

Another direct application of SPQs is for travel time distribution estimation [8]. Here, we need travel time histograms for several paths P_i during a given period. These can be calculated directly from the differences in the timestamps at the first and the last road segments in the SPQ results. Furthermore, SPQs can also be used as a component for

⁴Considering this application more precisely, we need not only camera images but also other sensor data (e.g., precise positions on the traveling road segment, speed profile, and so on), or ground truth data for the supervised learning (i.e., “true” lane information in the images annotated by hand in case of lane marking detection). Such additional information can be attached on the records in the postings-list (using relational tables). We do not discuss this issue in this thesis any more because our method is independent from the additional information, and it highly depends on the application.

processing more complicated queries related to navigation and route prediction, which are described in the following sections.

3.2.3. TEQ: Trajectory Extraction Query

Models for predicting future positions of moving objects are important, because they have a variety of potential applications in automotive systems, including anticipatory driver warning systems (e.g., of hazardous road conditions), information services (e.g., advertising), and automatic monitoring and control of the vehicle’s behavior (e.g., engine load anticipation). Krumm [27] studied probabilistic models based on k th-order Markov chain that aims to predict the next road segment e after traveling a path P of length k (i.e., the conditional probability $p(e|P)$ based on the frequency). We consider the following two natural extensions of this basic idea.

- (C1) Time-dependent models: for the objects that traveled along P during I , which road segments do they go next?
- (C2) L -step ahead models: predict the next L (≥ 2) road segments after P .

To obtain the frequency corresponding to a given (P, I, L) , precomputation is not realistic because there are too many possibility of (P, I, L) ; hence, we need to retrieve the data on demand. We can formulate this problem as the following *trajectory extraction query* (TEQ).

$$\text{TEQ}(P, I, L) = \{p_{tid}[i..i + \hat{L}] \mid tid \in \text{SPQ}(P, I) \wedge (\exists i \text{ s.t. } p_{tid}[i - |P|..i] = P)\}, \quad (3.3)$$

$$\text{where } \hat{L} := \min(L, |p_{tid}| - i). \quad (3.4)$$

The first condition selects the moving objects that traveled along P during I using an SPQ (corresponding to C1). The second condition then selects the subscripts i such that the substring $p_{tid}[i - |P|..i]$ matches P . For each tid and i , the subtrajectory of length L after P is extracted as $p_{tid}[i..i + \hat{L}]$ (corresponding to C2). When trajectory tid has less than L edges after P , we cannot extract L edges. Therefore, we stop extraction at the end of p_{tid} . This is why we use \hat{L} instead of L .

TEQ Example: Now, we give an example of a $\text{TEQ}(P, I, L)$ query using Figure 3.2. Let us consider $P = \text{AB}$, $I = [0, 70]$, and $L = 2$. First, we call $\text{SPQ}(P, I)$, finding that T_1 and T_3 match the condition (see the example in Section 3.2.2). The paths of T_1 and T_3 after $P = \text{AB}$ are C and EG , respectively. Hence, $\text{TEQ}(P, I, L) = \{\text{C}, \text{EG}\}$. Note that the length of the extracted trajectory for T_1 is one because T_1 does not have subtrajectory of length $L = 2$ after P .

3.2.4. TAPEQ: Time-period-based All Path Enumeration Query

Route planning is another important application of spatio-temporal trajectories. Navigation systems are often given two locations (i.e., the origin and destination) and asked to output routes between them. These routes are usually calculated by minimizing the total cost [10]. By using different link cost definitions (e.g., physical distance or average travel time), we can extract different routes.

An alternative approach to route planning is to use the frequencies of real trajectories, as we might expect that real trajectories reflect drivers' preferred routes. Existing methods [6, 31] aim to find *the most frequent route* (MFR) using specialized data structures. In contrast, we consider queries that *enumerate all routes* that occur in the database. This can be regarded as a generalization of MFR, because by definition the MFR is included among the enumerated routes. We call this a *time-period-based all-path enumeration query* (TAPEQ). TAPEQs could be useful for navigation (finding not just popular routes but also little-known shortcuts), and urban planning (e.g., analyzing changes in drivers' routing choices).

Formally, for a given road segment pair $u, v \in E$ and a time interval I , a TAPEQ enumerates all u - v paths that were traveled during I as shown in Fig. 3.6. Let $\Pi(u, v)$ be the set of all possible paths between u and v . We assume that each path $P \in \Pi(u, v)$ passes u and v only once, i.e., $P[j] \neq u$ and $P[j] \neq v$ ($1 \leq j \leq |P| - 2$). This assumption is reasonable because, in practical applications like route planning, we do not need paths with loops. In addition, let $\text{supp}(P, I)$ be the number of NCTs in \mathbb{T} that follow the path P during I . We take this as the size of SPQ, i.e., $\text{supp}(P, I) := |\text{SPQ}(P, I)|$. Note that $\text{supp}(P, I) > 0$ implies that at least one NCT in \mathbb{T} traveled along P during I . A TAPEQ is formally defined as follows:

$$\text{TAPEQ}(u, v, I, \xi) = \{P \in \Pi(u, v) \mid \text{supp}(P, I) \geq \xi\}, \quad (3.5)$$

where $\xi > 0$ specifies the minimum support of the result. Although \mathbb{T} consists of historical data, prior enumeration is impossible because there are infinitely many possible (u, v, I) configurations.

We should also emphasize that TAPEQs also consider NCTs whose *subpaths* are u - v paths. In other words, NCTs considered in TAPEQ are not only NCTs that match regular expression $u.*v$ but also those that match $.*u.*v.*$ (here, $.*$ matches any sequence).

TAPEQ Examples: Here, we give four examples of a $\text{TAPEQ}(u, v, I, \xi)$ based on the example in Figure 3.2.

- a) Let us consider $u = A$, $v = G$, $I = [0, 100)$, and $\xi = 1$. We can see that there

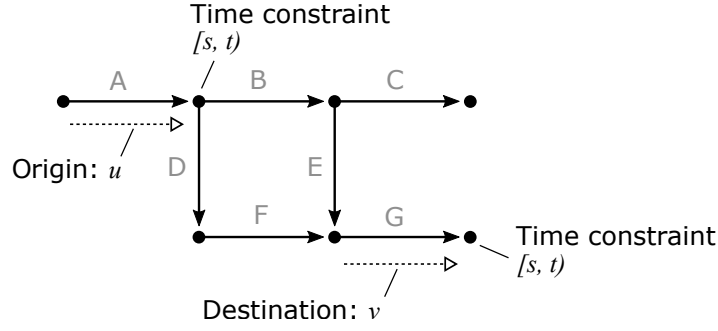


Figure 3.6.: Time-period-based all-path enumeration query: $\text{TAPEQ}(u, v, I)$ finds all u - v paths that appear at least once (or, optionally, more than ξ times) in the database as subtrajectories.

are two possible paths from A to G: $P_1 = ABEG$ and $P_2 = ADFG$. We then find $\text{SPQ}(P_1, I) = \{2, 3\}$ and $\text{SPQ}(P_2, I) = \{4\}$. Since $|\text{SPQ}(P_1, I)| = |\{2, 3\}| = 2 \geq \xi$ and $|\text{SPQ}(P_2, I)| = |\{4\}| = 1 \geq \xi$, we obtain $\text{TAPEQ}(A, G, [0, 100], 1) = \{P_1, P_2\}$.

- b) If $\xi = 2$ and the other constraints are as before, we obtain $\text{TAPEQ}(A, G, [0, 100], 2) = \{P_1\}$ because P_2 occurs only once.
- c) If $I = [0, 60)$, $\xi = 1$, and u and v are as before, we have $\text{TAPEQ}(A, G, [0, 60), 1) = \{P_2\}$ because $\text{SPQ}(P_1, [0, 60))$ is an empty set.
- d) Let us consider $u = A$, $v = E$, $I = [0, 100)$, and $\xi = 1$. In this case, there is only one path from A to E: $P_3 = ABE$. Since both T_2 and T_3 satisfy the time constraint, we obtain $\text{TAPEQ}(A, E, I, \xi) = \{P_3\}$.

3.2.5. Summary

In this chapter, we have defined the data model and the SPQs, TEQs, and TAPEQs considered in Part II, and have also described some applications of these queries. These queries all operate on paths in road networks, which are represented as sequences of road segment IDs. We will show that these queries can be processed using a common indexing structure, SNT-index, which is introduced in the next chapter.

4. Indexing and Querying Methods for Trajectories

In this chapter, we describe index structure and query processing algorithms of SNT-index. In Section 4.1, we provide a key statement that connects temporal information with spatial paths stored in FM-index (Lemma 2.1). Subsequently, we describe the proposed algorithms for SPQs, TEQs, and TAPEQs (Section 4.2–4.4).

4.1. SNT-index

4.1.1. Overview

In this section, we describe the proposed data structure, the SNT-index, which consists of the following two data structures (Figure 4.1):

1. an *FM-index* for spatial paths, and
2. posting lists (using B^+ -tree indexed by timestamps).

We need to develop a method of integrating the spatial information and temporal information because standard FM-index cannot consider temporal information. The key idea for SNT-index is to combine these two data structures using *inverse suffix array* (ISA). Unlike the existing inverted-index-like methods shown in Figure 3.3, the SNT-index postings-lists, shown in Figure 4.1, have an additional column (*isa*) to store the ISA values. In the remainder of this section, we first show a key statement regarding the ISA in Section 4.1.2. Then, we describe the postings-lists (B^+ -tree for timestamps) in Section 4.1.3. As we will see later, the *isa* column makes a connection with the FM-index and enables efficient query processing. The query processing algorithms for SPQs, TEQs, and TAPEQs are then described.

4.1.2. Spatial FM-index

In this section, we describe how SNT-index indexes the spatial information of NCTs and a key statement how we connect spatial information with temporal information.

4. Indexing and Querying Methods

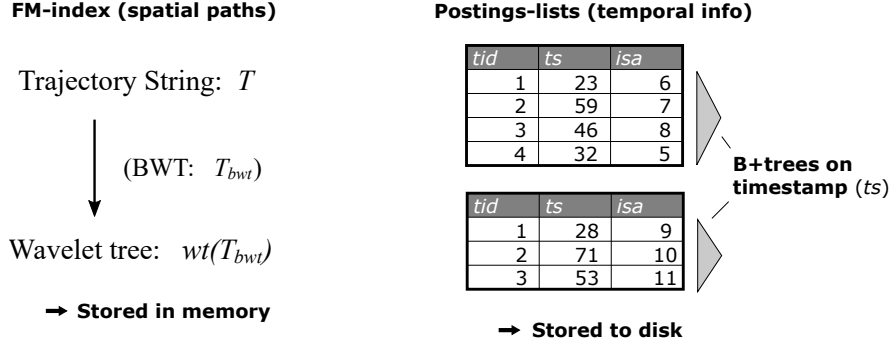


Figure 4.1.: An SNT-index consists of two data structures; an FM-index and B⁺-trees of posting lists. Unlike the existing inverted-index-like methods, our postings-lists additionally store an *inverse suffix array* (ISA).

Let $\mathbb{T} = \{p_{tid}, ts_{tid}\}_{tid=0}^{D-1}$ be a set of NCTs to be indexed. We first consider the *trajectory string*, denoted by T , consisting of all paths in \mathbb{T} (Definition 2.2):

$$T = p_0^r \$ p_1^r \$ \cdots \$ p_{D-1}^r \$ \#. \quad (4.1)$$

With the example trajectories in Figure 3.2, we obtain the following trajectory string, which is used as a running example below:

$$T = \text{CBA\$GEBA\$GEBA\$GFDA\$}\#. \quad (4.2)$$

Next, the trajectory string T is stored in an in-memory FM-index. We call this FM-index *the spatial FM-index* $\mathcal{F}(T)$. As described in Section 2.3, the spatial FM-indexes enable fast pattern matching and substring extraction, functions that play crucial roles in the proposed algorithms.

Key Property of ISA As defined in Definition 2.5, the inverse suffix array ISA is the inverse function of SA , that is, $ISA[SA[j]] = j$ ($0 \leq j < |T|$). Further, the suffix range $R(Q)$ was defined in Definition 2.3 to represent the range of sorted suffixes whose prefix is Q . We have the following proposition, which provide a bridge between the suffix range and ISA.

Proposition 4.1 *Consider a string T and its inverse suffix array ISA . Let $R(Q) := [sp, ep)$ be the suffix range of a given pattern Q of length m . Then, the following two statements are equivalent: 1) $sp \leq ISA[i] < ep$, and 2) $T[i, i + m) = Q$.*

PROOF In Lemma 2.1, let us replace the terms $SA[j]$ and j with i and $ISA[i]$, respectively. This leads to the statement.

This proposition implies that, if we know the suffix range $R(Q) = [sp, ep)$ for a given Q , we can check whether Q appears at position i just by checking the inequality $sp \leq ISA[i] < ep$. This property plays important roles in the proposed algorithms for SPQs, TEQs, and TAPEQs. *Importantly, as mentioned in Section 2.3, we can find the suffix range $R(Q)$ very efficiently using FM-index.* As shown in the next section, we describe a data structure that stores both the inverse suffix array and temporal information; combining such a data structure with the spatial FM-index $\mathcal{F}(T)$, we obtain efficient algorithms.

There are several FM-index variants that have different strengths in terms of query processing speed and compression performance. In this Part II, we employ the simplest one: we store T_{bwt} in a balanced wavelet tree with uncompressed bit vectors. With this implementation, time complexity to compute $rank_w(T_{\text{bwt}}, j)$ is $O(\log |\Sigma|)$.

4.1.3. Temporal B⁺-trees

For the temporal B⁺-trees, SNT-index employs an inverted-index-like approach. For each road segment $e \in E$, the postings-list Φ_e is formally defined as a table (relation) consisting of tuples of the form $(\text{tid}, \text{ts}, \text{isa})$. The postings-list Φ_e represents a set of NCTs that traveled along the road segment e . Figure 4.2(a) shows the postings-lists for the example trajectories in Figure 3.2. The postings-list Φ_A stores four records because four NCTs, T_1 , T_2 , T_3 , and T_4 traveled along the road segment A at $\text{ts} = 23, 59, 46$, and 32, respectively.

The “NCT-table” in Figure 4.2(b) explains how the isa column of each postings-lists is defined. Each row corresponds to an NCT element. For example, since T_1 (i.e., $\text{tid} = 1$) traveled along the road segment $\text{eid} = \text{C}$ at $\text{ts} = 33$ (see T_1 in Figure 3.2), these values appear in the first row of the NCT-table. The records are sorted in the same order as in the trajectory string (Eq. (4.2)), which means that the eid column of this table corresponds to the trajectory string T . There are also two additional columns, i and isa , defined as follows.

- Column i gives the position i in T for which data is given in that row.
- Column isa gives the i th element of the inverse suffix array of T (i.e., $ISA[i]$).

To build the posting list Φ_e , all records with $\text{eid} = e$ are selected and stored in Φ_e (see the arrows from (b) to (a) in Figure 4.2). Since Φ_e has to support range queries by timestamp, a B⁺-tree is built using the ts column as key. This allows us to quickly find records in Φ_e within a given time interval I , i.e.,

$$\text{RangeQuery}(e, I) = \{(\text{tid}, \text{ts}, \text{isa}) \in \Phi_e \mid \text{ts} \in I\}. \quad (4.3)$$

4. Indexing and Querying Methods

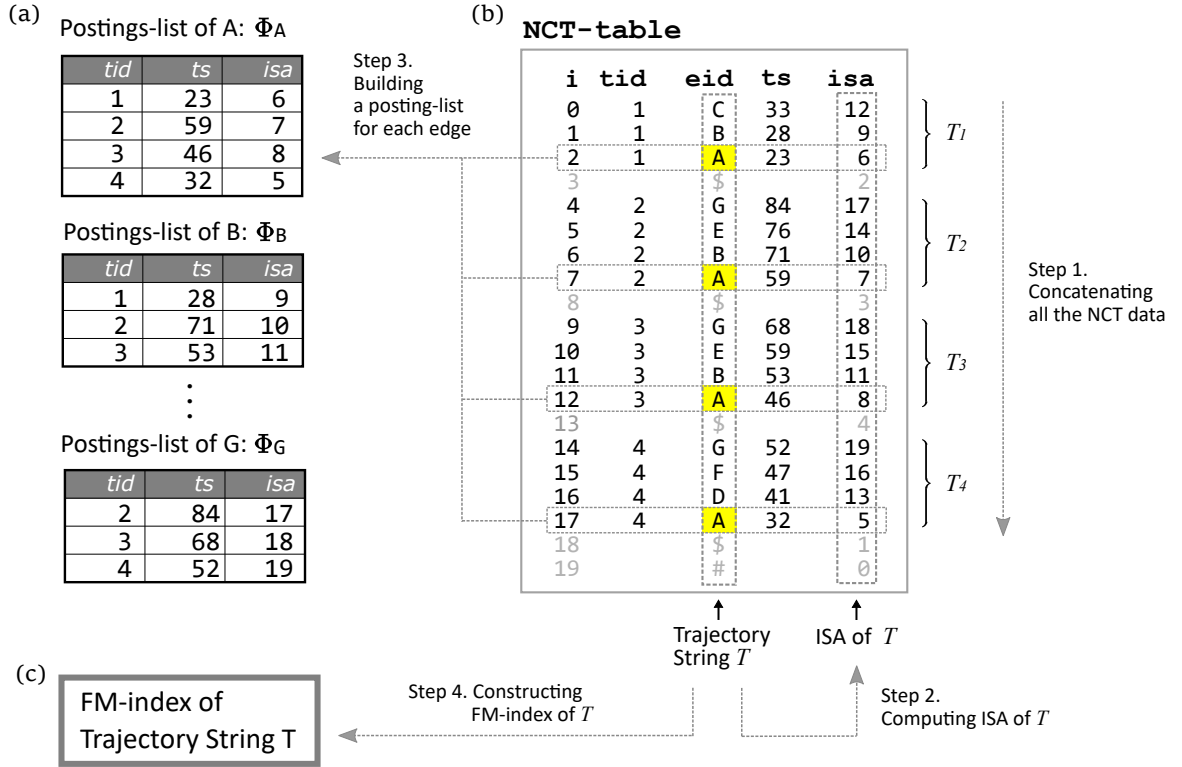


Figure 4.2.: Schematic representation of SNT-index construction. (a) Postings-lists $\{\Phi_e\}$ for each road segment $e \in E$. (b) This NCT-table explains how the postings-list is constructed; (Step 1) All the NCT elements in \mathbb{T} are concatenated; (Step 2) ISA of the trajectory string T is computed; (Step 3) The records in NCT-table are inserted in the corresponding posting list Φ_e . (c) FM-index of the trajectory string T is constructed (Step 4).

This can be seen as filtering the postings-list Φ_e based on the given time interval I . We refer to the result as a *temporally-filtered postings-list*. The I/O complexity of this query is $O(occ + \log |\Phi_e|)$, where occ is the number of records in the result set.

Relationship to the Existing NCT Indexing Methods Our approach is different from that of the existing methods in the following sense. Compared with the inverted-index-like methods for NCTs, SNT-index has an additional column, *isa*, that connects the FM-index with the postings-lists and plays an important role in query processing.

The data structure proposed by Krogh et al. [25] has a different additional column for efficient SPQ processing. Since this method is closely related to our method, we briefly describe it in Section 4.2.2 and conduct an experimental comparison in Chapter 5.

4.1.4. Index Construction and Implementation

We summarize the index construction scheme and discuss an actual implementation using an off-the-shelf RDBMS. To construct an SNT-index for a given set \mathbb{T} of NCTs, we first calculate the suffix array of the trajectory string T (Definition 2.2). We can calculate SA in linear time, $O(|T|)$, by using the induced sorting algorithm [43]. Then, the BWT T_{bwt} is calculated using SA , and the FM-index $\mathcal{F}(T)$ is obtained by storing T_{bwt} in an in-memory wavelet tree (the arrow (b) \rightarrow (c) in Figure 4.2). Since the FM-index is a compact data structure, we expect that it will fit in memory. If there is insufficient memory, however, we can divide the FM-index into sections and store them in a distributed manner (Section 5.8), or use a *compressed FM-index* (Part III).

To build the postings-lists, the inverse suffix array ISA is calculated from SA . Since ISA is just the inverse of SA , the calculation is straightforward (and thus runs in linear time, $O(|T|)$). We then store the records in the postings-lists $\{\Phi_e\}$, as shown in Figure 4.2 (b \rightarrow a). Finally, a B⁺-tree is constructed for each Φ_e .

We employ an off-the-shelf RDBMS to implement the postings-lists, storing all the lists in one table. This table is similar to `NCT-table` shown in Fig. 4.2 (b). To simulate $RangeQuery(e, I)$, we created a B⁺-tree index on (eid, ts) . This index was constructed as a *clustered index*, which speeds up disk access by storing the records physically in (eid, ts) -order. Using the `NCT-table`, $RangeQuery(e, I)$ could be implemented as the following SQL query (where $I = [I_{\text{begin}}, I_{\text{end}}]$).

```

RangeQuery(e, I) : SELECT tid, ts, isa FROM NCT-table
                    WHERE eid = e AND ts BETWEEN I_begin AND I_end.

```

4.1.5. Summary

In this section, we have described the data structure of SNT-index, which consists of two indexes, namely an FM-index and B⁺-trees. The spatial paths of the NCTs are indexed using an FM-index. Although this does not store temporal information, it does enable fast pattern matching and substring extraction, and it is compact and can thus fit in memory. The temporal information is indexed using B⁺-trees. This is similar to the existing inverted-index-like methods, but SNT-index employs an additional column for the *inverse suffix array* (ISA), which connects the B⁺-trees with the FM-index. In the following sections, we present algorithms for several different types of queries using SNT-index.

Algorithm 4: Proposed algorithm for $SPQ_{\text{simple}}(P, I)$ (proposed- SPQ_{simple})

```

1  $W \leftarrow \phi$ 
2  $[sp, ep] \leftarrow FM\text{-search}(P^r, T_{\text{bwt}})$ 
3  $Y \leftarrow RangeQuery(P[|P| - 1], I)$ 
4 for each record  $y \in Y$  do
5   if  $sp \leq y.\text{isa} < ep$  then
6      $W \leftarrow W \cup \{y\}$ 
7 return  $W.\text{tid}$ 

```

Algorithm 5: Proposed algorithm for $SPQ(P, I)$ (proposed- SPQ)

```

1  $W \leftarrow \phi$ 
2  $[sp, ep] \leftarrow FM\text{-search}(P^r, T_{\text{bwt}})$ 
3  $Y \leftarrow RangeQuery(P[|P| - 1], I)$ 
4 for each record  $y \in Y$  do
5   if  $sp \leq y.\text{isa} < ep$  then
6      $W \leftarrow W \cup \{y\}$ 
7  $X \leftarrow RangeQuery(P[0], I)$ 
8  $U \leftarrow ST\text{-join}(X, W)$ 
   // intersection of X and W
9 return  $U.\text{tid}$ 

```

4.2. Algorithm for Strict Path Queries

4.2.1. Proposed SPQ algorithm

Here, we propose algorithms for SPQ with SNT-index. We first present the algorithm for SPQ_{simple} , followed by the algorithm for SPQ.

The method of dealing with $SPQ_{\text{simple}}(P, I)$ is shown in Algorithm 4. *FM-search* at Line 2 conducts a spatial search for the route pattern P using the spatial FM-index (P^r is the reverse of P). Then, *RangeQuery* at Line 3 conducts a temporal search to find the NCTs that visited the last road segment $P[|P| - 1]$ during I . Only this *RangeQuery* requires disk access in this algorithm. Finally, Lines 4-6 integrate the spatial and temporal search results. The correctness of Algorithm 4 is guaranteed by the following proposition.

Proposition 4.2 *Algorithm 4 finds the correct result of $SPQ_{\text{simple}}(P, I)$.*

PROOF At Line 3, *RangeQuery* finds the set Y of NCTs that visited the last road segment $P[|P| - 1]$ during I , meaning that $SPQ_{\text{simple}}(P, I) \subset Y.\text{tid}$. Here, $Y.\text{tid}$ denotes the set of trajectory IDs in Y . Thus, all we need to show is that the remainder of the algorithm removes all the elements in Y that do not match P . Remembering that $y.\text{isa}$ means $ISA[i]$ where i is the corresponding position in the trajectory string T , $sp \leq y.\text{isa} < ep$ is equivalent to $T[i..i + |P^r|] = P^r$ due to Proposition 4.1. Since the trajectories are stored in reverse order in the trajectory string T , we can say that $y.\text{tid}$ matches P iff $sp \leq y.\text{isa} < ep$. Thus, the result $W.\text{tid}$ at Line 7 is equivalent to $SPQ_{\text{simple}}(P, I)$.

Next, we propose an algorithm to handle $SPQ(P, I)$ as shown in Algorithm 5. This is similar to Algorithm 4 but we have to consider the additional temporal constraint at

$P[0]$. By definition, we have a relation $\text{SPQ}(P, I) \subset \text{SPQ}_{\text{simple}}(P, I)$. Hence, our strategy is as follows: we first find $\text{SPQ}_{\text{simple}}$ (Lines 1–6), and we then filter out NCTs that do not travel along $P[0]$ during I (Lines 7–8). Lines 1–6 are the same as those in Algorithm 4. At Line 7, we find the posting list X of NCTs that visited the first road segment $P[0]$ during I . At Line 8, the result of $\text{SPQ}_{\text{simple}}$, denoted by W , is filtered with *ST-join* using X to remove the NCTs in W that did not visit $P[0]$ during I . This *ST-join* takes the intersection of two temporally-filtered postings-lists W and X w.r.t. the trajectory IDs, that is,

ST-join(X, W) :

$$\text{SELECT } W.* \text{ FROM } X \text{ JOIN } W \text{ ON } X.\text{tid} = W.\text{tid} \text{ and } X.\text{ts} < W.\text{ts}. \quad (4.4)$$

Here, we have an additional condition $X.\text{ts} < Y.\text{ts}$, which is needed to filter out NCTs that traveled along $P[0]$ *after* $P[|P| - 1]$. At Line 9, the set of trajectory IDs in the result of *ST-join* ($U.\text{tid}$) are returned as the $\text{SPQ}(P, I)$ result.

Efficiency In our proposed method, pattern matching for paths P of any length is replaced by a single scalar inequality, $sp \leq y.\text{isa} < ep$. With this technique, only one *RangeQuery* call is required in Algorithm 4 for $\text{SPQ}_{\text{simple}}$ (and two in Algorithm 5 for SPQ). This is much fewer calls than with the existing algorithms, which need to execute *RangeQuery* $O(|P|)$ times for pattern matching, as will be discussed in the next section. Instead, our method has to calculate $[sp, ep)$ for a given P using Algorithm 1. As mentioned in Remark 5, this algorithm runs in $O(|P| \log |\Sigma|)$ time; this is several orders of magnitude faster than *RangeQuery* in practice thanks to the in-memory processing. Our method is therefore expected to be much faster than the existing methods, especially when $|P|$ is large.

4.2.2. Existing SPQ Algorithms

Here, we briefly review the NETTRA data structure and three existing SPQ algorithms proposed by Krogh et al. [25]. These algorithms are compared with our proposed method in Section 5.

These algorithms are based on “hash” values $h(e)$ (integers) that are defined for each road segment $e \in E$. Two methods were considered for defining the hash values: 1) the physical length of e , and 2) the rounded logarithm of a random (and large) prime number. Given a path P , its hash value is defined as the sum of hash values of the corresponding road segments, i.e., $h(P) = \sum_{i=0}^{|P|-1} h(P[i])$. For the i th element of path p_{tid} , the hash

Algorithm 6: Approximation algorithm for SPQ(P, I) (DHash-join) [25]

```

1  $\Delta = h(P)$ 
2  $X \leftarrow \text{RangeQuery}(P[0], I)$ 
3  $Y \leftarrow \text{RangeQuery}(P[|P| - 1], I)$ 
4  $X \leftarrow \Delta\text{-join}(X, Y, \Delta)$ 
5 return  $X.tid$  // Traj-IDs in  $X$ 

```

Algorithm 7: Naïve algorithm for SPQ(P, I) (Full-join) [25]

```

1  $X \leftarrow \text{RangeQuery}(P[0], I)$ 
2 for  $i = 1..(|P| - 1)$  do
3    $Y \leftarrow \text{RangeQuery}(P[i], I)$ 
4    $X \leftarrow \Delta\text{-join}(X, Y, h(P[i]))$ 
5 return  $X.tid$  // Traj-IDs in  $X$ 

```

value is defined as $\text{hash}[i] := h(p_{tid}[0..i])$. Krogh et al. [25] proposed to store these hash values for each element in the posting lists. By definition, we have $\text{hash}[j] - \text{hash}[i] = h(P)$ if $P = p_{tid}[i..j]$, which allows us to check whether a subpath of p_{tid} matches a given pattern P just by looking up the two hash values. However, incorrect matches (false positives) can happen due to hash collisions, and thus we cannot guarantee $P = p_{tid}[i..j]$ even if the hash values are the same.

Based on these hash values, the three SPQ algorithms were proposed. First, an approximate SPQ algorithm using this hash approach is shown in Algorithm 6 (DHash-join). This requires only two *RangeQuery* calls, for the first and the last road segments of P . These two lists are then joined using the the following subroutine, which we refer to as *Δ -join*.

$\Delta\text{-join}(X, Y, \Delta)$:

```
SELECT Y.* FROM X JOIN Y ON X.tid = Y.tid AND  $\Delta = Y.\text{hash} - X.\text{hash}$ ,
```

where X and Y are the temporally-filtered postings-lists. This method might yield false positive results because more than two routes can have the same hash value.¹

Second, an exact algorithm for SPQ(P, I) was proposed using *Δ -join* (Algorithm 7), which we call **Full-join**. This algorithm takes the intersection of all the temporally-filtered posting lists corresponding to $e \in P$. A drawback of the **Full-join** algorithm is the large number of disk accesses required due to the need for multiple range queries ($|P|$ queries), which is inefficient when $|P|$ is large.

To improve efficiency, a third method (**Optimal-join**) was introduced. This can guarantee exact results and the number of joins required is less than with **Full-join**. This reduction is achieved by partitioning P into substrings (i.e., $P = P_1P_2 \cdots P_k$), each of which guarantees no hash collisions. This partition is calculated via a shortest path al-

¹We cannot avoid possibility of this hash collision in principle because there can be a combinatorial number of routes between two positions while the hash value is an integer with the finite number of bits (e.g., 32 bits or 64 bits).

gorithm in terms of the hash values. The **Optimal-join** only requires $k + 1$ range queries, at $P_i[0]$ ($1 \leq i \leq k$) and $P_k[|P_k| - 1]$. Although k is less than $|P|$ in general, it is usually proportional to $|P|$, implying that the number of *RangeQuery* calls is still $O(|P|)$. See Krogh et al. [25] for further details of these algorithms.

4.2.3. Summary

In this section, we have proposed SPQ algorithms based on SNT-index. The multiple ($O(|P|)$) B⁺-tree retrievals required by the existing algorithms (**Full-join** and **Optimal-join**) are replaced with two B⁺-tree retrievals and some in-memory operations (*FM-search*). Although the *FM-search* operations require $O(|P| \log |\Sigma|)$ time, this can be ignored in practice as we will see in the experiments. The other existing algorithm (**DHash-join**) also requires only two B⁺-tree retrievals, but its query results may contain false positives.

4.3. Algorithm for Trajectory Extraction Queries

4.3.1. Baseline Method: Adding TB-tree-like Pointers to NETTRA

Before describing the proposed algorithm for TEQs, we first present a baseline algorithm that does not use an SNT-index. For this, we extend the NETTRA data structure described in Section 4.2.2, because the existing inverted-index-like methods do not support TEQs. As defined in Eq. (3.3), TEQs aim to extract the paths of length L that were followed after traveling along P during I . A natural algorithm for $\text{TEQ}(P, I, L)$ can be described as follows.

1. Execute $\text{SPQ}(P, I)$ to obtain the NCTs that traveled along P during I .
2. For each trajectory in $\text{SPQ}(P, I)$, extract the path of length L that was followed after traveling P .

How, then, do we handle the second extraction step? In inverted-index-like methods like NETTRA, the records in the postings-lists do not include pointers to records corresponding to the next road segment. Hence, we cannot easily find the subsequent records starting from a given record. This problem can be solved by adding pointers to the next record as shown in Figure 4.3. Pointers from Φ_A to Φ_B and from Φ_B to Φ_C have been added to the records with $tid = 1$ because the spatial path of T_1 ($tid = 1$) in Figure 3.2 is ABC. This idea was proposed for TB-tree [47], an indexing method for free (i.e., non-constrained) trajectories.

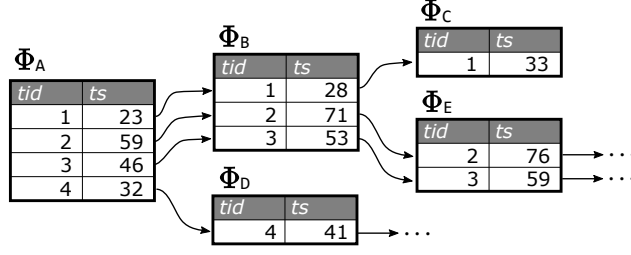


Figure 4.3.: Baseline method for TEQs (modified TB-tree). Successor record pointers are added to the postings-lists.

Using this structure, we can process TEQs as follows. For the first step, the **Optimal-join** algorithm can be used to process the SPQ. Then, for each record found in the first step, the corresponding paths of length L are extracted by tracing the pointers. This baseline method is relatively efficient compared with the case without pointers, however, we still need to access the disk every time we follow a pointer. Our algorithm, proposed in the next subsection, avoids need for the disk access by using the spatial FM-index.

4.3.2. Proposed TEQ Algorithm

Our SNT-index allows for a more efficient algorithm by using the spatial FM-index. First, we introduce a fundamental property that connects the temporal B⁺-trees in the SNT-index with *FM-extract* (Algorithm 2). For example, let us focus on the $i = 16$ th record of the NCT-table in Figure 4.2, that is, $(i, \text{tid}, \text{eid}, \text{ts}, \text{isa}) = (16, 4, \text{D}, 41, 13)$. By the *FM-extract* definition (Algorithm 2), we have $\text{FM-extract}(T_{\text{bwt}}, \text{isa} = 13, l = 2) = T[i - l, i] = T[14, 16] = \text{GF}$, which corresponds to the path of length 2 traveled by this NCT (T_4) after $\text{eid} = \text{D}$. More generally, we have the following property.

Proposition 4.3 *Consider any road segment $e \in E$ and any record $u := (\text{tid}, \text{ts}, \text{isa}) \in \Phi_e$. Then, $\text{FM-extract}(T_{\text{bwt}}, u.\text{isa}, L)$ gives the path of length L after the corresponding record u , i.e., the path of length L traveled by the corresponding NCT ($u.\text{tid}$) after the road segment e at time $u.\text{ts}$.*

PROOF As defined in Section 4.1.3, we have $u.\text{isa} = \text{ISA}[i]$, where i specifies the record in the NCT-table (Figure 4.2(b)). Hence, by definition, $\text{FM-extract}(T_{\text{bwt}}, u.\text{isa}, L) = \text{FM-extract}(T_{\text{bwt}}, \text{ISA}[i], L)$ returns $T[i - L, i]$. Since the trajectory string T stores the paths in reverse order, $T[i - L, i]$ gives the path of length L after the record u .

Algorithm 8 (**proposed-TEQ**) is our proposed TEQ algorithm. As in the baseline algorithm in the previous subsection, we first execute $\text{SPQ}(P, I)$ and obtain the temporally-filtered postings-list U (Line 2), in this case using **proposed-SPQ** (Algorithm 5). Although

the original **proposed-SPQ** returns $U.\text{tid}$, we can easily modify it to return U instead (i.e., U includes `isa` values). By definition, we have $U \subset \Phi_{\tilde{e}}$, where \tilde{e} is the last element of P . Therefore, the desired subtrajectories of length L can be obtained by executing $FM\text{-extract}(T_{\text{bwt}}, u.\text{isa}, L)$ using the ISA values of the tuples u in U (Proposition 4.3). Finally, Line 5 removes symbols in p after $\$$ (if there is $\$$ in p) because $FM\text{-extract}$ returns a subtrajectory of length L even if the corresponding NCT does not have a subtrajectory of length L after traveling P .

Algorithm 8: Proposed algorithm for $TEQ(P, I, L)$ (proposed-TEQ)

```

1  $Paths \leftarrow \emptyset$  // Empty set
2  $U \leftarrow \text{proposed-SPQ}(P, I)$  // Return U instead of U.tid
3 for  $u \in U$  do
4    $p \leftarrow FM\text{-extract}(T_{\text{bwt}}, u.\text{isa}, L)$  // Path extraction of length L
5    $p \leftarrow \text{RemoveAfter-}\$(p)$  // Remove symbols after $ (e.g. AB$CD->AB)
6    $Paths \leftarrow Paths \cup \{p\}$  // Add the extracted path p to the result
7 return  $Paths$ 

```

Efficiency The disk access cost for the baseline algorithm in Section 4.3.1 is $O(L)$, because we need to trace the pointer L times to extract a subtrajectory of length L . The **proposed-TEQ** algorithm replaces the pointer-tracing step with $FM\text{-extract}$. Although the time complexity of $FM\text{-extract}$ is $O(L \log |\Sigma|)$ as mentioned in Remark 5 (Chapter 2), in practice $FM\text{-extract}$ is much faster because it only needs memory access. In the proposed algorithm, disk access is only needed when we call **proposed-SPQ** (i.e., two *RangeQuery* calls for any given (P, I, L)). Hence, the proposed algorithm is expected to be much faster than the baseline. In fact, we will show experimentally that our algorithm is tens of times faster (Section 5.3).

4.3.3. Summary

In this section, we have proposed an algorithm for TEQs that utilizes the `isa` values in the posting lists in a different way from the **proposed-SPQ** algorithm. This idea, extracting subtrajectories based on the `isa` values in the posting lists, are also used in the proposed TAPEQ algorithm.

4.4. Processing Time-period-based All-Path Enumeration Queries

4.4.1. Baseline Method: PrefixSpan for TAPEQs

Remembering the definition of a TAPEQ, a naïve solution for TAPEQs is to check whether $\text{supp}(P, I) \geq \xi$ holds by executing $\text{SPQ}(P, I)$ for all $P \in \Pi(u, v)$. However, this would be impossible because there can be a massive number of possible paths between u and v . For example, let us consider a small 32×32 grid network and paths from top left to bottom right. Even if we consider only the shortest paths, there are $\binom{32}{64} > 2^{60}$ possible paths, which is an unrealistic number to process.

TAPEQs are closely related to *sequential pattern mining*, because they can be regarded as enumerating all symbol sequences in a database satisfying a given condition. In this section, we present a baseline approach based on the inverted-index-like method without suffix arrays and highlight its inefficiency. Again, we employ the NETTRA structure for this.

For sequential pattern mining, the PrefixSpan algorithm [45] is one of the most widely-used algorithms in practice. Although there are numerous variants of this algorithm, we employ one similar to *Traj-PrefixSpan* [37] here as a baseline for TAPEQ algorithm. Traj-PrefixSpan restricts the sequence P to be a substring of the sequences in the database (i.e., patterns with gaps are prohibited). Algorithm 9 shows our customized Traj-PrefixSpan algorithm, which can be executed using NETTRA. Unlike the original Traj-PrefixSpan, our modified version considers temporal constraints (Line 1 in Algorithm 10). In Algorithm 9, $\text{OutGoingEdges}(P.\text{last})$ returns a set of road segments that are directly accessible from the last segment of P . Further, the path length $\text{dist}(P)$ is restricted to θ (Line 1). Without this restriction, this algorithm would not terminate in a reasonable time (note that this restriction is not part of the original TAPEQ). To process $\text{TAPEQ}(u, v, I, \xi)$, we call $\text{PrefixSpan}(u, \mathbb{T}|_u)$, where $\mathbb{T}|_u = \text{RangeQuery}(u, I)$. Note that this algorithm requires one disk access (*RangeQuery*) per recursion in the *Projection* function, which means that the disk access cost is large.

4.4.2. Proposed TAPEQ algorithm

Unlike the previous PrefixSpan algorithm, our proposed TAPEQ algorithm does not require an explicit projection operation, and thus can avoid the large associated disk access cost. The proposed algorithm consists of the following two steps.

1. *Candidate generation*: Find all NCTs that traveled along both u and v during I .

Algorithm 9: PrefixSpan algorithm tailored for TAPEQ (baseline):
PrefixSpan($P, \mathbb{T}|_P, I$)

```

1 if  $|\mathbb{T}|_P| < \xi$  or  $dist(P) > \theta$  then
2   return // Terminate if  $\mathbb{T}|_P$  is
           too small or  $P$  is too long
3 if  $P.last = v$  then
4   yield  $P$  and return
5 for  $\forall w \in OutGoingEdges(P.last)$  do
6    $P' \leftarrow Pw$  // Concatenate  $P$  and  $w$ 
7    $\mathbb{T}|_{P'} \leftarrow Projection(w, \mathbb{T}|_P, I)$ 
8   PrefixSpan( $P', \mathbb{T}|_{P'}, I$ )

```

Algorithm 10: Obtain Pw -projected database:
Projection($w, \mathbb{T}|_P, I$)

```

1  $\mathbb{T}|_w \leftarrow RangeQuery(w, I)$ 
2 return
    $\Delta\text{-join}(\mathbb{T}|_P, \mathbb{T}|_w, h(w))$ 

```

2. *Path recovery:* For each NCT found in the previous step, recover the route traveled between u and v using the FM-index.

This method is expected to be more efficient than PrefixSpan because only two disk accesses are required (for u and v) in the first step. The complete algorithm is shown in Algorithm 11.

Algorithm 11: Proposed algorithm for TAPEQ(u, v, I, ξ)

```

1  $ITree \leftarrow$  Empty interval tree
2  $X \leftarrow RangeQuery(u, I)$ 
3  $Y \leftarrow RangeQuery(v, I)$ 
4  $Z \leftarrow RestrictToAdj(ST\text{-}join2(X, Y))$  // NCTs that visited  $u \rightarrow v$  during  $I$ 
5 for  $z \in Z$  do
6    $R \leftarrow ITree.find(z.isa_v)$  //  $R$  is a suffix range that contains  $isa_v$ 
7   if  $R$  is null then
8      $P \leftarrow FM\text{-}extract\text{-}until(v, z.isa_u, T_{bwt})$  // Recover the  $u$ - $v$  path
9      $R \leftarrow FM\text{-}search(P, T_{bwt})$  // Find suffix range for  $P$ 
10     $ITree.insert(R)$  // Register the suffix range  $R$  to  $ITree$ 
11     $A[R] \leftarrow (path = P, count = 0)$ 
12     $A[R].count \leftarrow A[R].count + 1$ 
13 return  $[(A[R].path, A[R].count) \text{ for } R \text{ in } A.keys \text{ if } A[R].count \geq \xi]$ 

```

Lines 2–4 correspond to the first step. Here, Z is the set of candidate NCTs that traveled along both u and v during I . In this algorithm, we need isa_u and isa_v , ISA values from the posting list of u and v , respectively. We thus modify *ST-join* to keep

4. Indexing and Querying Methods

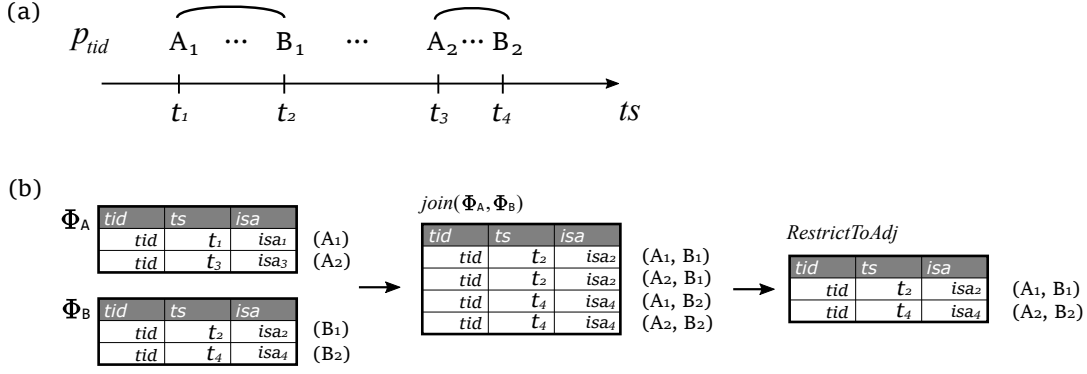


Figure 4.4.: An explanation of *RestrictToAdj*. (a) An example NCT with loops (road segments A and B are traveled twice). (b) Only the neighboring pairs (corresponding to the arcs in (a)) are selected.

them, as follows.

$$\begin{aligned}
 ST\text{-}join2(X, Y) : & \text{ SELECT } X.isa \text{ AS } isa_u, Y.isa \text{ AS } isa_v \\
 & \text{ FROM } X \text{ JOIN } Y \text{ ON } X.tid = Y.tid \text{ AND } X.ts < Y.ts \quad (4.5)
 \end{aligned}$$

Note that, at Line 4, *RestrictToAdj* (Figure 4.4) is applied in order to filter out pairs that are not temporally-adjacent. This is useful because we do not aim to extract paths $P \in \Pi(u, v)$ that travels along u (or v) more than once, as discussed in Section 3.2.4.

The remainder (Lines 5–12) corresponds to the second step, which we will describe from Line 8. For each NCT z in Z , the actual route between u and v is extracted as P (Line 8). Similar to Proposition 4.3, *FM-extract-until* (Algorithm 3) extracts P from the FM-index $\mathcal{F}(T)$, starting from $isa.u$, until v is found. Once a new P has been found, its suffix range R is calculated (Line 9) and is inserted into the interval tree *ITree* (Line 10). The newly found path P is registered to the association array A using R as a key (Line 11). Now, we return to Lines 6–7. If possible, we would like to avoid *FM-extract-until* operations, because these operations are relatively expensive. Fortunately, we can avoid this extraction step if P has already been found: if $z.isa_v$ is within one of the ranges that are already registered in *ITree*, the corresponding path has already been found because of Proposition 4.1. Therefore, we can avoid *FM-extract-until* and only need to increment the count of the corresponding u - v path (Line 12). Finally, at Line 13, the u - v paths that occur more than ξ times are returned as the result of the TAPEQ.

Efficiency Clearly, our algorithm requires only two disk accesses thanks to the use of an FM-index. In contrast, the baseline PrefixSpan algorithm needs at least L accesses, where L is the number of distinct road segments in the TAPEQ result (which can be

large if u and v are distant).

Although the path extraction phase (Lines 5–12) is an in-memory operation, it can be costly if Z is large. To improve the efficiency of this phase, we can filter out some of the elements in Z based on estimated frequency using the hash values discussed in Section 4.2.2. To implement this idea, we also calculate the difference between the hash values of u and v at Line 4 by selecting $\Delta := Y.\text{hash} - X.\text{hash}$ in Eq. (4.5). After Line 4, we count how many times each hash value appears in Z . If $\text{count}(\Delta) < \xi$ for the hash value Δ , all $z \in Z$ with this hash value must correspond to a path P that does not satisfy $\text{supp}(P, I) \geq \xi$.² We can therefore remove these candidates from Z , reducing the number of *FM-extract-until* operations required at Line 8. As will be demonstrated in the experiments, our algorithm can be hundreds times faster than PrefixSpan in practice.

4.4.3. Summary

In this section, we have discussed two TAPEQ algorithms. First, we presented a baseline algorithm based on the famous PrefixSpan algorithm, which is based on an existing inverted-index-like method and involves many B⁺-tree accesses. Then, we proposed a more efficient algorithm based on SNT-index, which uses the spatial FM-index for path enumeration and only requires two B⁺-tree accesses.

4.5. Appending New Data to SNT-index

SNT-index does not explicitly support dynamic updating for two reasons. First, the suffix array (FM-index) is essentially a static index. Second, even if the suffix array could be updated dynamically, data insertion would affect the ISA values of all (existing) leaves of the B⁺-trees because the insertion would destroy the suffix order.

As we have repeatedly emphasized, because our target application is the retrieval of historical data, real-time updating is not needed for our purposes. However, it may still be necessary to add new data at a certain time interval. There are two ways to do this. The first method is to reconstruct the index, including the new data. While this would allow us to apply the proposed method without changing the data structure at all, the update cost would increase over time. In the following, we show the second method.

²Although the same Δ can refer to different paths due to hash collisions, $\text{count}(\Delta)$ can give an upper bound on the true support $\text{supp}(P, I)$, which is enough to guarantee the filtering scheme discussed here.

4.5.1. Partitioning the FM-index for Appending New Data

The second method is to build an FM-index using only the newly added data, that is, to build a new FM-index \mathcal{F}_τ for each period τ (Figure 4.5). For a given set of new NCTs $\mathbb{T}^{(\tau)}$ at a period τ , we first generate its trajectory string $T^{(\tau)}$, and then compute the corresponding BWT string $T_{\text{bwt}}^{(\tau)}$. This new BWT string is separately stored in a new wavelet tree, which consists of a new FM-index \mathcal{F}_τ . For a given pattern P , we can conduct a pattern matching using Algorithm 12. Unlike Algorithm 1, we obtain a *set* of ranges each of which corresponds to a period. This algorithm requires multiple *FM-search* executions, but it can be calculated in parallel if the FM-indexes are stored in a distributed manner. Given J FM-indexes, the time complexity of *Multi-FM-search* is $O(J \cdot |P| \log |\Sigma|)$ (non-parallel case), and $O(|P| \log |\Sigma|)$ (parallel case). We consider that this multiple execution of *FM-search* is generally not a problem because the processing time of *FM-search* is two orders of magnitude faster than *RangeQuery* as we see in our experiments.

Algorithm 12: *Multi-FM-search*($P, \{T_{\text{bwt}}^{(\tau)}\}$): Find the suffix range for a pattern P of length m for given multiple BWT strings $\{T_{\text{bwt}}^{(\tau)}\}$

```

// The following loop can be parallelized if  $T_{\text{bwt}}^{(\tau)}$ 's are stored in a
// parallel manner.
1 for each  $T_{\text{bwt}}^{(\tau)}$  in  $\{T_{\text{bwt}}^{(\tau)}\}$  do
2    $[sp^{(\tau)}, ep^{(\tau)}] \leftarrow \text{FM-search}(T_{\text{bwt}}^{(\tau)}, P)$ 
3 return  $\{[sp^{(\tau)}, ep^{(\tau)}]\}$  // A set of suffix ranges over all periods are
   returned

```

In this case, we should store not just the ISA value but also an identifier for the period τ in the leaves of the B^+ -trees. Here, we refer to this new column as “period”, and thus the new postings-lists consist of tuples of the form $(\text{tid}, \text{ts}, \text{period}, \text{isa})$, as illustrated in Figure 4.5. With these modifications, we can process SPQs as follows. Let $[sp^{(\tau)}, ep^{(\tau)})$ be the result of the suffix range query for \mathcal{F}_τ for a given pattern P . In this case, the condition in Proposition 4.1 (and thus Line 5 in Algorithms 4 and 5) simply becomes

$$\bigvee_{\tau} (\text{period} = \tau \wedge sp^{(\tau)} \leq \text{isa} < ep^{(\tau)}), \quad (4.6)$$

where \bigvee_{τ} means logical OR over τ corresponding to a given interval I . Hence, this change imposes little additional cost on the query processing phase. In addition, we emphasize that, in this scheme, the cost for index construction per period does not increase over time because we do not need to update the data records we have already stored.

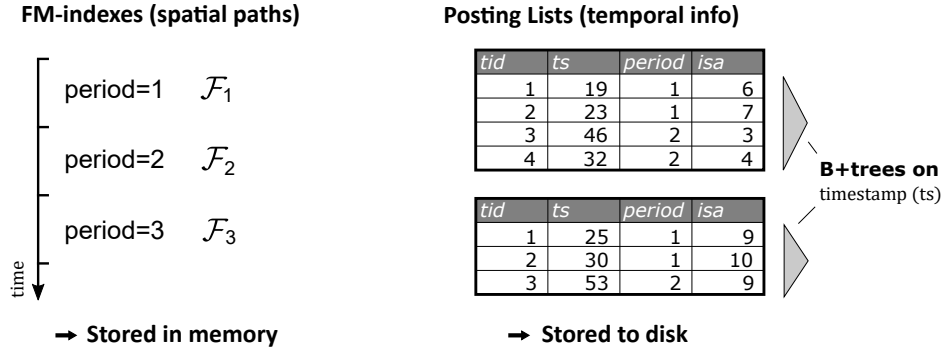


Figure 4.5.: Append-supported SNT-index structure. A new column “period” is used to specify one of the partitioned FM-indexes.

The proposed algorithms for TEQs and TAPEQs in the previous sections require extraction of a sub-path for a given ISA value in a postings-list record. With the modified structure above, this extraction is straightforward because the `period` ($= \tau$) value is now stored in B⁺-trees, which allows us to specify the FM-index \mathcal{F}_τ and execute *FM-extract* starting from the `isa` value stored together. More specifically, T_{bwt} is replaced with $T_{\text{bwt}}^{(\tau)}$ (Line 4 in Algorithm 8 and Line 8 in Algorithm 11). For TAPEQs, we also have to replace *FM-search* with *Multi-FM-search* in Line 9 of Algorithm 11. Since *Multi-FM-search* returns multiple ranges, the interval tree used in this algorithm also has to be parametrized by τ (i.e., $ITree^{(\tau)}$). Accordingly, $ITree.find(z.isa_v)$ becomes $ITree^{(z.\text{period})}.find(z.isa_v)$.

Such a semi-dynamic update scheme would be conceptually close to data warehousing. Spatial FM-indexes represent summary data in data warehousing, that is, they are intended to make heavy query processing more efficient, rather than focusing on processing transactions to maintain consistency.

4.5.2. Spatial Partitioning of the FM-index

Finally, we discuss a possibility of spatial partitioning of the FM-index, which would be useful when the spatial FM-index is too large to fit into the memory of a single server. Unlike the time-period partitioning discussed above, we would divide the data based on spatial proximity in this case. Specifically, given a set \mathbb{T} of NCTs, we divide \mathbb{T} into groups, say $\mathbb{T}_1, \dots, \mathbb{T}_R$, considering the spatial proximity (e.g., we can use a trajectory clustering method). When storing each \mathbb{T}_r into postings-lists, we also record the subscript r in a new `region` column. Furthermore, we build an FM-index \mathcal{F}_r for each \mathbb{T}_r . Similar to the above temporal partitioning case, we can support SPQs, TEQs, and TAPEQs using this new `region` column. In this spatial partitioning case, we have to develop an efficient partitioning/clustering scheme. Such a partitioning scheme should

4. *Indexing and Querying Methods*

be determined based on the application scenario.

5. Experiments

5.1. Setup and Implementation Details

Dataset We use three datasets for the evaluation. The first dataset is called **Singapore** and consists of real trajectories obtained from taxi cabs in Singapore [55]. The trajectories have already been matched to the map, but the NCTs in this dataset sometimes have gaps (i.e., there are transitions between road segments that are not physically connected). We therefore filled in these gaps with shortest paths and calculated the corresponding timestamps using linear interpolation based on driving distance. We also split NCTs into multiple pieces if they stayed on the same road segment more than 10 minutes. The resulting dataset contained more than 340K NCTs, each consisting of road segment sequences of length 290 (on average). Therefore, the total length of the trajectory string T for **Singapore** dataset was about 98 million.

The second dataset is called **Roma**, and consists of GPS coordinates from taxi cabs in Rome, Italy. We applied HMM map-matching [41] to obtain NCT representations. The resulting dataset contained more than 130,000 NCTs with an average length of 92. The total length of the trajectory string T was therefore 12 million.

The third dataset is the **RGSx5** dataset which was synthesized by random sampling based on 5-gram probability $p(e_t|e_{t-5}, e_{t-4}, e_{t-3}, e_{t-2}, e_{t-1})$ calculated using the **Singapore** dataset. The average travel times were used to generate timestamps. This dataset was five times larger than the **Singapore** dataset, containing about 1.7 million NCTs of average length 290. Hence, the total length of the trajectory string T was about 500 million. We also used the **RGSx1**, **RGSx2**, **RGSx3**, and **RGSx4** datasets, created similarly, to evaluate the scalability of the algorithms. The statistics of the dataset are summarized in Table 5.1.

Table 5.1.: Dataset statistics

Dataset	#traj	$ T $	$ \Sigma $	Storage	Source
Singapore	340K	98M	55,892	SSD	Song et al. [55]
Roma	130K	12M	56,653	SSD	Bracciale et al. [3]
RGSx5	1.7M	500M	55,892	HDD	Synthesized based on Singapore

5. Experiments

Implementation As discussed in Section 4.1.4, the posting lists were implemented as one table (`NCT-table`) using PostgreSQL (version 9.6.2) with default settings. The size of cache buffers were set to 3GB. The Singapore and Roma datasets were stored on the SSD, while the HDD was used for the largest RGSx5 dataset. All algorithms were implemented in C++ and compiled with g++ (version 4.8.4) with the `-O3` option. We used the `sdsl-lite`¹ library for the (in-memory) wavelet trees. The BWTs were calculated using `sais.hxx`² which implements a linear-time sorting algorithm [43]. All experiments are conducted on a workstation with the following specifications: Intel Core i7-K5930 3.5GHz CPU (64-bit, 12 cores), 32GB DDR4 RAM, Ubuntu Linux 14.04.

Table 5.2 summarizes the algorithms implemented in this experiment. All the algorithms used for comparison were introduced in previous sections, which are based on NETTRA [25], the state-of-the-art (disk-based) NCT-indexing method for path-based queries. Our NETTRA implementation is also based on PostgreSQL, which is the same as the original implementation in Krogh et al. [25]. NETTRA also uses a hub-labeling index as an in-memory auxiliary data structure. We implemented this by modifying publicly available library.³ For the hash function, we used the physical length of each road segment (in meter).

Table 5.2.: Summary of algorithms and data structures used in the experiment

Query	Algorithm	Data structure / Description
SPQ	Full-join	NETTRA (Section 4.2.2)
	Optimal-join	NETTRA / Additional hub-labeling index is needed (Section 4.2.2)
	DHash-join	NETTRA / Exact result is not guaranteed (Section 4.2.2)
	Proposed-SPQ	SNT-index (Section 4.2.1)
	Proposed-SPQ _{simple}	SNT-index / Slightly different query definition Eq. (3.2) (Section 4.2.1)
TEQ	Modified TB-tree	NETTRA + TB-tree-like pointers (Section 4.3.1)
	Proposed-TEQ	SNT-index (Section 4.3.2)
TAPEQ	Modified PrefixSpan (MPS)	PrefixSpan algorithm implemented on NETTRA (Section 4.4.1)
	Proposed-TAPEQ	SNT-index (Section 4.4.2)

5.2. SPQ Results

We evaluated the average query processing time over 1000 queries, randomly sampled from the dataset. We tested using queries of various lengths $|P| \in \{5, 10, \dots, 50\}$. In

¹<http://github.com/simongog/sdsl-lite/>

²<http://sites.google.com/site/yuta256/sais/>

³<http://github.com/savrus/hl>

the first experiment, we used the time interval $I = [t_{\min}, t_{\max}]$, where t_{\min} and t_{\max} are the minimum and the maximum timestamps appearing in the dataset, respectively (i.e., 100% temporal selectivity).

For comparison, we used the three algorithms discussed in Section 4.2.2: Full-join, Optimal-join, and DHash-join. Note that the DHash-join method does not guarantee exact results due to potential hash collisions. (This is the only algorithm in this study that can produce false positive results for SPQs.)

Effect of $|P|$: Figure 5.1 shows the average query processing time results. Here, “Proposed-SPQ_{simple}” and “Proposed-SPQ” are the proposed methods (Algorithms 4 and 5). While the processing times for Proposed-SPQ_{simple}, Proposed-SPQ and DHash-join were constant with respect to the length $|P|$, those for Full-join and Optimal-join increased as $|P|$ increased. Although the Optimal-join method was about four times faster than the Full-join for the Singapore dataset, the processing time still grew linearly, indicating that the average size of the *skips* realized by the substring partitioning in the Optimal-join was about four. The DHash-join method was as fast as the Proposed-SPQ because the both methods access the B⁺-trees twice per query. In this experiment, 0.1% of the results returned by the DHash-join were false positives while the other methods yielded no false positives for any dataset (these false positives can be reduced if we use a different hash function, see Section 5.8 for details). In addition, the Proposed-SPQ_{simple} method was twice as fast as that of the Proposed-SPQ because the Proposed-SPQ_{simple} only requires one B⁺-tree retrieval, while the Proposed-SPQ needs two.

As discussed previously, the proposed methods require additional operations using the FM-index to find the suffix range $R(P)$. Theoretically, this cost is $O(|P| \log \sigma)$, but the actual processing time of the proposed methods appeared to be constant for different lengths $|P|$. This is because the additional processing time required to find $R(P)$ is negligible compared with the cost of the B⁺-tree retrievals. The average processing time only to find $R(P)$ using the FM-index in Figure 5.2. Since the time needed for this operation is less than 0.1ms, it does not affect the total processing time. This is why the Proposed-SPQ method shows similar processing times compared to the DHash-join method. In fact, we observe that the DHash-join is slightly faster than Proposed-SPQ (e.g., see Figure 5.5). This difference is due to the additional *FM-search* execution in the Proposed-SPQ method. Therefore, the difference is less than 0.1ms, as observed in Figure 5.2.

Scalability: Figure 5.3 shows how the SPQ processing time scaled as the dataset size increased. For this evaluation, we used the RGSx1, RGSx2, \dots , and RGSx5 datasets. The proposed methods show the best scalability, indicating that the SNT-index works very well in practice for huge datasets. Despite the fact that the time complexity of

5. Experiments

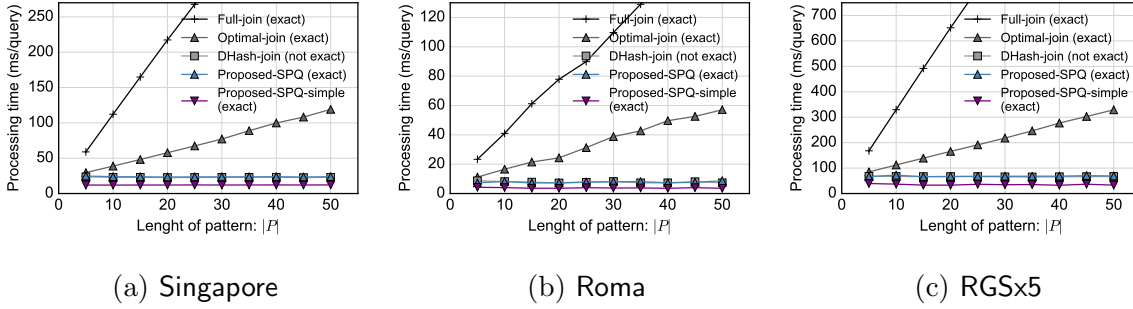


Figure 5.1.: SPQ processing time for various $|P|$

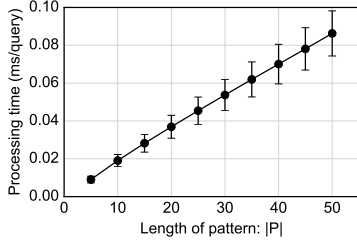


Figure 5.2.: FM-search (the Singapore dataset)

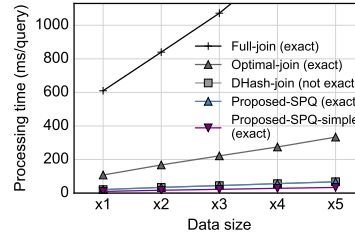
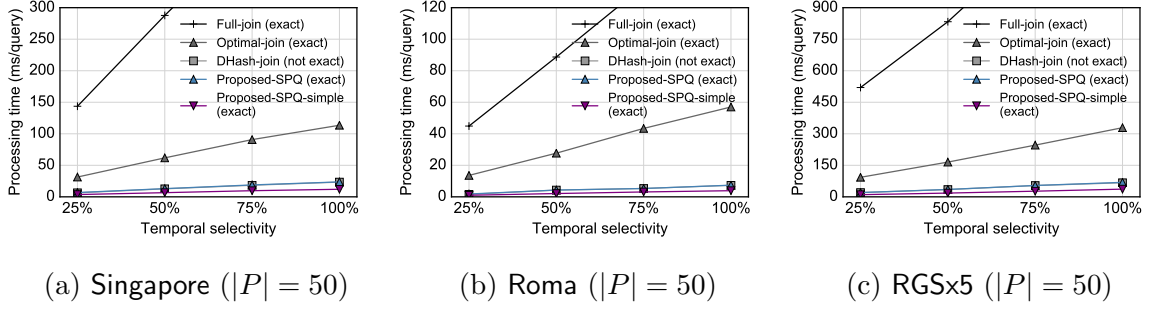
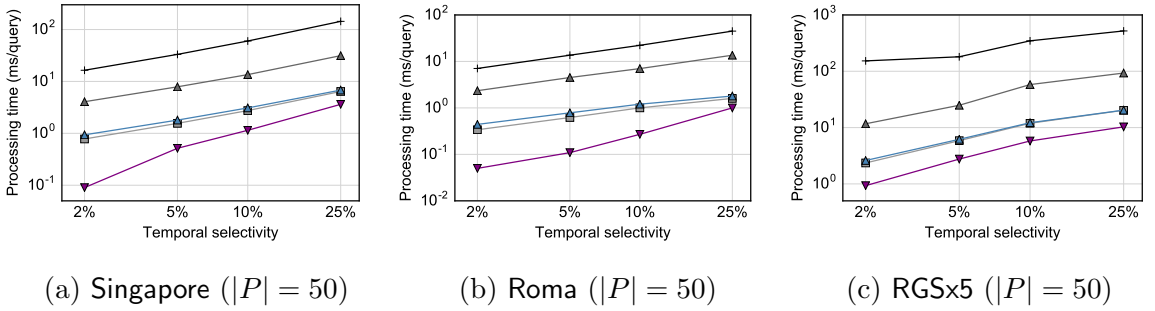


Figure 5.3.: Scalability (RGSxN, $|P| = 50$, 100% temporal selectivity)

B^+ -tree retrieval is $O(\log(\text{data size}))$, the processing time increased linearly. The reason for this is similar to the one in the temporal selectivity case: when data size doubles, occ also doubles on average, and thus the cost grows linearly with data size.

Effect of $|I|$: Here, we investigate the effect of changing the length of the time interval I . For this experiment, we used $|P| = 50$ (fixed) and varied the length of I . Figure 5.4 and Figure 5.5 show the average processing times over 1,000 SPQs, randomly sampled in the same manner as in the previous experiment. The horizontal axis represents temporal selectivity, which was calculated as follows. First, we calculated the $ts_{2\%}$, $ts_{5\%}$, $ts_{10\%}$, $ts_{25\%}$, $ts_{50\%}$, $ts_{75\%}$ quantiles of ts in the database, and then we processed 1,000 random SPQs for $I_{x\%} := [ts_{\min}, ts_{x\%})$.

In Figure 5.4 and Figure 5.5, the proposed method shows the best performance among all the competitors. Again, this result is mostly explained by the number of B^+ -tree retrievals needed to process each SPQ. Unlike Figure 5.1, the processing times of the proposed methods (proposed-SPQ and proposed-SPQ_{simple}) slightly increase as $|I|$ increases, despite the fact that the number of *RangeQuery* calls is constant (two and one, respectively). This is because the number of candidates $occ := |Y|$ (see Section 4.2.1) increased for longer time intervals. Because the actual number of data blocks fetched from storage is $O(occ) \simeq O(|I|)$, the processing time increased slightly as $|I|$ increased.

Figure 5.4.: SPQ processing time for various $|I|$ (high temporal selectivity; 25% – 100%)Figure 5.5.: SPQ processing time for various $|I|$ (low temporal selectivity; 2% – 25%).
The legend is the same as Figure 5.4.

5.3. TEQ Results

To evaluate the processing time required for $\text{TEQ}(P, I, L)$, we used randomly-generated paths P of length 5. For comparison, we implemented the *modified TB-tree* described in Section 4.3.1.

Figure 5.6 shows the processing time for various extraction lengths L (100% temporal selectivity case). The proposed method was more than ten times faster than the modified TB-tree (50 times faster for $L = 20$ for the **Singapore** dataset), and its processing time slightly increased as L increased. This is due to the substring extraction used in the proposed algorithm (Line 4 in Algorithm 8), which takes $O(L \log \sigma)$ time. The actual processing time for substring extraction from the spatial FM-index $\mathcal{F}(T)$ is shown in Figure 5.8 for the **Singapore** dataset. While this was as fast as *FM-search* (Figure 5.2), substring extraction is executed $|U|$ times per query, leading to a total time of $O(|U|L \log \sigma)$. Here, $|U|$ is the number of NCTs that followed P during I (see Algorithm 8). This differs from the SPQ case, which requires only one *FM-search*. For the **Singapore** dataset, $|U|$ was about 2,000 on average, and thus the processing time increased slightly as L increased, even though substring extraction is an in-memory operation. However, we should emphasize that the processing time of the proposed method increased much

5. Experiments

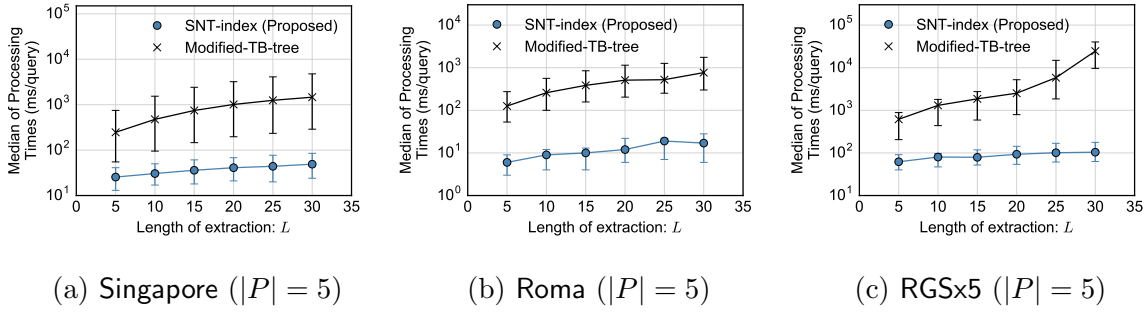


Figure 5.6.: TEQ processing time for various L . Error bars show 25%-75% quantiles.

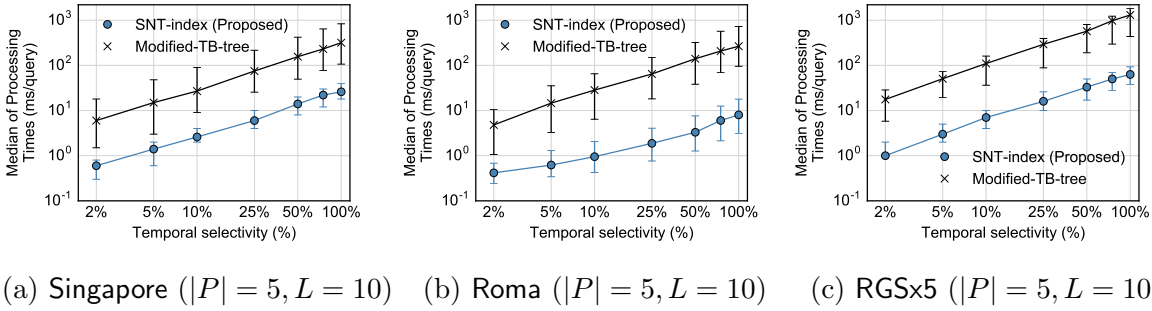
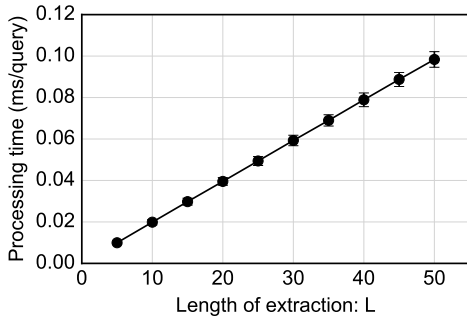
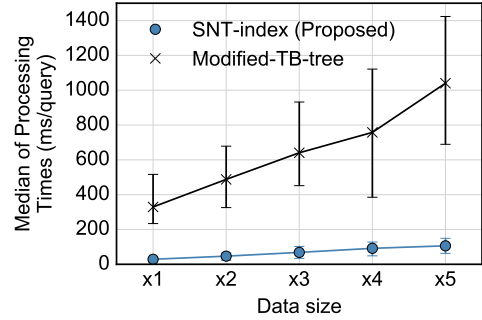


Figure 5.7.: TEQ processing time for various $|I|$ (temporal selectivity 2% – 100%). Error bars show 25%-75% quantiles.

more slowly than that of the modified TB-tree with respect to L , as shown in Figure 5.6. In fact, for the **Singapore** dataset, the proposed method took $0.9L + 21.3$ ms per query, as opposed to $49.5L$ ms for the modified TB-tree.

Figure 5.7 shows the TEQ processing time for various temporal selectivity (2% – 100%). The results show that the processing times of both methods are proportional to the temporal selectivity. This is because the number of NCTs to be extracted, $|U|$, is proportional to the temporal selectivity $|I|$.

Next, Figure 5.9 shows the change in processing time as the data size increases. The datasets used for this evaluation were the same as those used in Figure 5.3 (i.e., RGSx1, RGSx2, ..., and RGSx5). The proposed method showed better scalability than the modified TB-tree method. Finally, Figure 5.10 visualizes an example TEQ result ($|P| = 5, L = 10$) for the **Singapore** dataset. The red road segments correspond to the query pattern P , and the other road segments are shaded based on the frequency which they appeared in $\text{TEQ}(P, I, L)$. These results give a probabilistic prediction of the object's likely movements after P during I .

Figure 5.8.: *FM-extract* (Singapore dataset)Figure 5.9.: TEQ scalability (RGSxN, $L = 10$, 100% temporal selectivity)Figure 5.10.: TEQ example ($L = 10$)

5.4. TAPEQ Results

Here, we evaluate the average TAPEQ processing time for two algorithms: modified PrefixSpan (the baseline algorithm described in 4.4.1) and the proposed algorithm (Section 4.4.2).

For this evaluation, we generated 100 random queries as follows: 1) for u and v , we randomly drew from the 100 most frequent road segments; 2) for I , we used $I = [t_{\min}, t_{\max}]$, as in the SPQ experiments, and 3) for ξ , we evaluated $\xi \in \{1, 2, 5, 10\}$. The modified PrefixSpan (MPS) algorithm has an additional parameter, θ , that restricts the length of the extracted paths to $\theta \cdot d_{uv}$, where d_{uv} is the shortest-path distance between u and v . We evaluated values for θ , namely $\theta \in \{1.5, 2.0, 3.0\}$, and refer to algorithms as **modified-PrefixSpan(1.5)**, **modified-PrefixSpan(2.0)**, and **modified-PrefixSpan(3.0)**, respectively. For the proposed algorithm, we used the modified version, enhanced with the hash values, for evaluation (see Section 4.4.2).

Figure 5.11 shows a comparison of the TAPEQ processing times for each dataset. The proposed method is 100–1000 times faster than the competing methods. Similar to the SPQ case, this improvement is due to the small number of B⁺-tree retrievals: the

5. Experiments

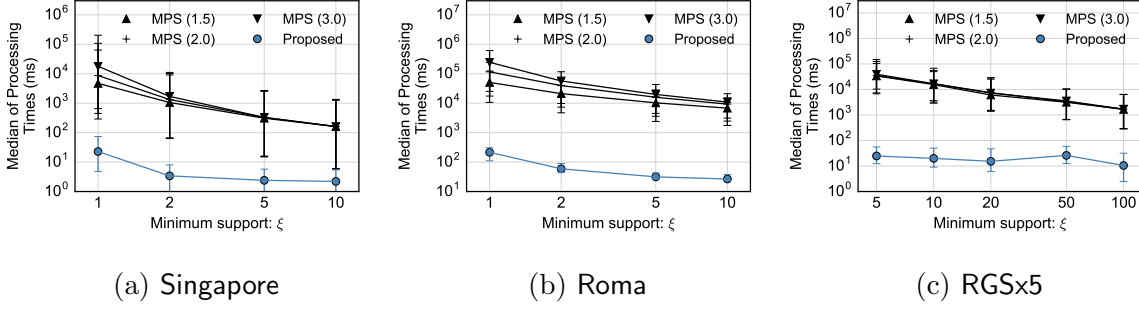


Figure 5.11.: TAPEQ processing time for various ξ with 25%-75% quantiles. MPS (θ) is an abbreviation for Modified-PrefixSpan (θ).

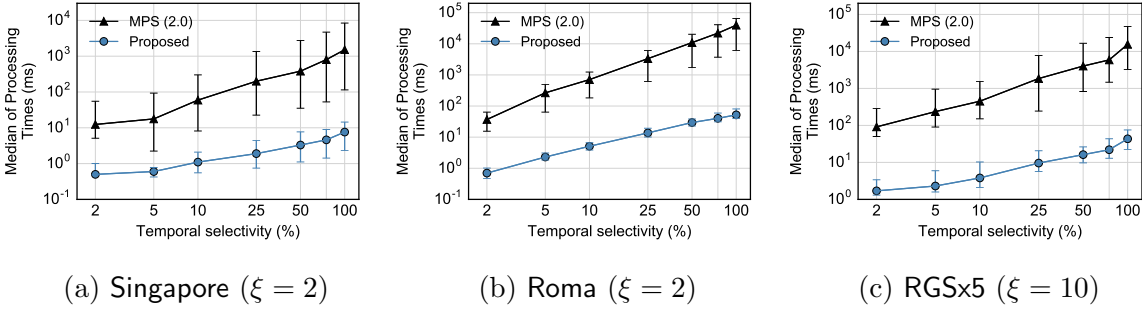


Figure 5.12.: TAPEQ processing time for various $|I|$ (temporal selectivity 2% – 100%). Error bars show 25%-75% quantiles.

proposed algorithm only requires two, while the `modified-PrefixSpan(θ)` requires many more B^+ -tree retrievals.

Figure 5.12 shows the TAPEQ processing time for various temporal selectivity (2% – 100%). Again, the proposed method is faster than `modified-PrefixSpan(2.0)`. We observe that the difference between the proposed method and the `modified-PrefixSpan(2.0)` gets smaller for lower temporal selectivity. This is because, for low temporal selectivity, the condition $|T|_P < \xi$ in Algorithm 9 (`modified-PrefixSpan`) prunes the search branches in early stage and this reduces the number of `RangeQuery` calls.

Figure 5.13 shows how the TAPEQ processing time increased with the shortest-path distance d_{uv} between the origin u and destination v . Each data point corresponds to one query. The processing times of both methods increased as the distance d_{uv} became longer, because there tend to be more paths between u and v if they are distant. However, the proposed method demonstrated even greater improvement for more distant d_{uv} . For example, when d_{uv} was 1000 meters, the proposed method was 1000 times faster. Figure 5.14 illustrates the scalability of TAPEQ, again showing that our method gave better results than the competitors for every data size.

Figure 5.15 shows an example TAPEQ result, obtained with $\xi = 3$ for the Singapore

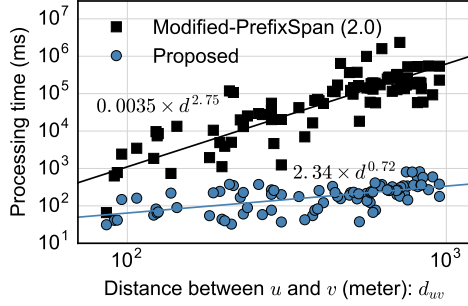


Figure 5.13.: Effect of d_{uv} (Roma, $\xi = 2$, temporal selectivity 100%)

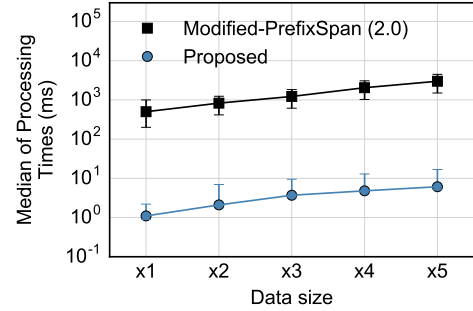


Figure 5.14.: Scalability (RGSxN, $\xi = 50$, temporal selectivity 100%)

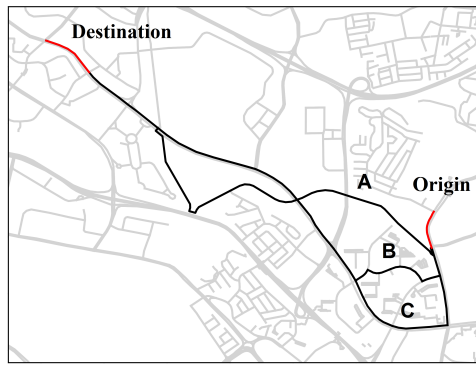


Figure 5.15.: TAPEQ example

dataset. This shows that there were three frequent routes (A, B, and C) between the origin and the destination, with frequencies $N_A = 4$, $N_B = 29$, and $N_C = 3$. We can therefore guess that route B was the standard route and that the others were detour. If we instead use $\xi = 1$ for this $u-v$ pair, many noisy routes are found, indicating that appropriate ξ values need to be chosen to obtain suitable route recommendations. However, because the appropriateness depends on the application scenario, we will not discuss this issue further here.

5.5. Index Size and Index Construction Time

As we have repeatedly emphasized, SNT-index consists of B^+ -trees stored on disk, while the spatial FM-index is held in memory. The disk storage⁴ and memory storage require-

⁴The disk storage measurement includes not only the table size but also the B^+ -tree index size. The table and index sizes are calculated using PostgreSQL built-in functions, `pg_relation_size` and `pg_indexes_size`. Therefore, the size includes all meta data and system columns.

5. Experiments

ments for each dataset are shown in Table 5.3. This shows that the memory (FM-index) required by an SNT-index is not very large, about 4% of the amount stored on disk. The memory footprint of our FM-index implementation is about $|T| \log_2 \sigma$ bits in total. Therefore, we need $\log_2 \sigma \simeq 16$ bits (2 bytes) per symbol for all datasets. Note that the memory usage shown here is for an uncompressed FM-index; to reduce the memory footprint, we could also have compressed the FM-index using the techniques described in Part III. Table 5.3 also shows the NETTRA index size (for both the original data structure and the TB-tree version). The disk storage for NETTRA+TB-tree is larger because the pointers introduced by TB-tree increase the size of the B⁺-tree index. The memory required by NETTRA represents the size of the Hub-labeling index, which is required by the Optimal-join algorithm to avoid dynamic programming for the shortest path computation. Note that this size depends only on the road network. Finally, the disk space required for the SNT-index is slightly larger than that for original NETTRA, due to the additional `isa` column.⁵

The ICT column in Table 5.3 shows the time needed to construct the spatial FM-index. (Note that no ICT values are given for the other methods, because these do not use an FM-index.) This shows that the FM-index construction time is relatively short, even for the largest RGSx5 dataset, because we can calculate the BWT in $O(|T|)$ using linear-time sorting [43]. This is typically much shorter than the time needed for B⁺-tree construction on disks: B⁺-tree construction took more than 1 hour for the RGSx5 dataset, for both the SNT-index and NETTRA. Hence, FM-index construction is not a bottleneck during the index construction phase.

Table 5.3.: Index size and index construction time (ICT)

Dataset	Method	Disk (GB)	Memory (GB)	ICT (sec)
Singapore	SNT-index	5.96	0.209	113
	NETTRA (original)	5.18	0.023	—
	NETTRA (+TB-tree)	8.00	0.023	—
Roma	SNT-index	0.73	0.033	14
	NETTRA (original)	0.63	0.038	—
	NETTRA (+TB-tree)	0.98	0.038	—
RGSx5	SNT-index	30.4	1.066	580
	NETTRA (original)	26.4	0.023	—
	NETTRA (+TB-tree)	40.8	0.023	—

⁵In our implementation, we retained the `hash` column in the SNT-index because it is useful for TAPEQs. Without this column, the disk space required by the SNT-index and the original NETTRA would have been the same.

5.6. Effect of Buffer Caches

In the previous experiments, we used 3GB buffer caches for PostgreSQL. In addition, the SNT-index uses an in-memory FM-index. As we have shown in Table 5.3, the memory footprint of the FM-index was about 1 GB for the largest RGSx5 dataset. One might wonder whether the comparative methods become more efficient if more cache buffers are used, which can reduce the I/O cost.

In this section, we consider three configurations that have different buffer cache sizes (3 GB, 6 GB, and 9 GB) and investigate how the efficiency of each method scales. Generally, the buffer cache effect depends on the access pattern. In this experiment, we consider randomly generated queries (SPQs, TEQs, and TAPEQs) as in the previous experiments. Figures 5.16, 5.17, and 5.18 show the results for SPQs, TEQs, and TAPEQs, respectively. Here, the processing times are normalized by the corresponding 3 GB buffer cache case. These results indicate that the buffer cache size does not have significant impact on the processing time, which would imply that 3 GB buffer caches are sufficient. Exceptionally, for RGSx5 dataset, the results for TEQs and TAPEQs (100% temporal selectivity) show that larger buffer cache size can reduce the processing time by up to 40%. This would be because

- RGSx5 is the largest dataset (more than 25 GB disk storage), which is larger than the buffer cache size, and
- TEQs and TAPEQs sequentially access the same posting-lists in one query execution, which can increase the cache hit ratio.

Although these results can change if we use different configurations (e.g., non-random query pattern, and different cache algorithm), we would like to emphasize that the following consequences do not change.

- The SNT-index enables efficient processing (two orders of magnitude faster for some cases) by adding in-memory data structure of moderate size, and
- this is more efficient than appending buffer caches of the same size.

5.7. Summary

In this section, we have evaluated the processing times for SPQs, TEQs, and TAPEQs. The proposed algorithms showed overwhelmingly (10–1000 times) better performance compared with baseline algorithms that did not use the spatial FM-index. The improvements were all due to the reduction in the number of B⁺-tree accesses (*RangeQuery*).

5. Experiments

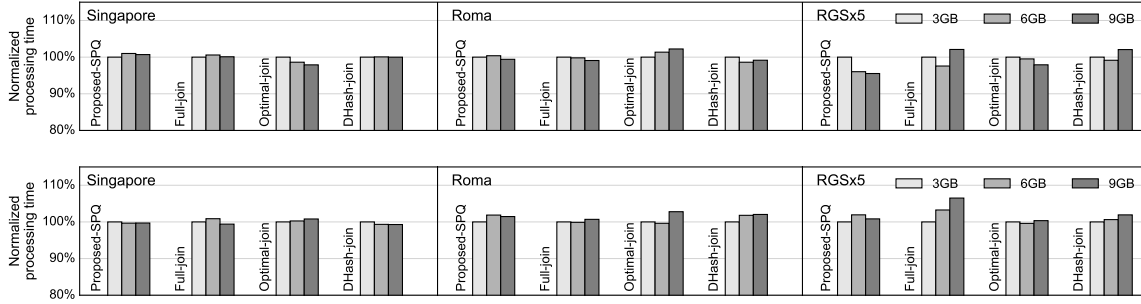


Figure 5.16.: Effect of buffer caches for SPQs (top: 100% temporal selectivity, bottom: 10% temporal selectivity). The processing times are normalized by the corresponding 3 GB case.

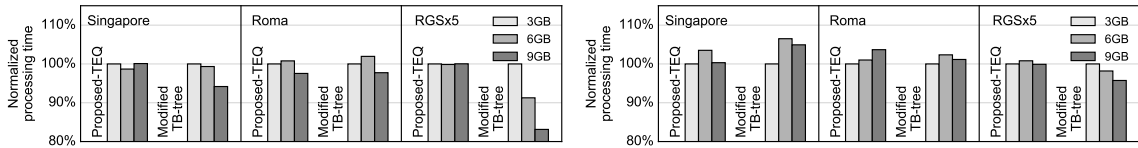


Figure 5.17.: Effect of buffer caches for TEQs (left: 100% temporal selectivity, right: 10% temporal selectivity). The processing times are normalized by the corresponding 3 GB case.

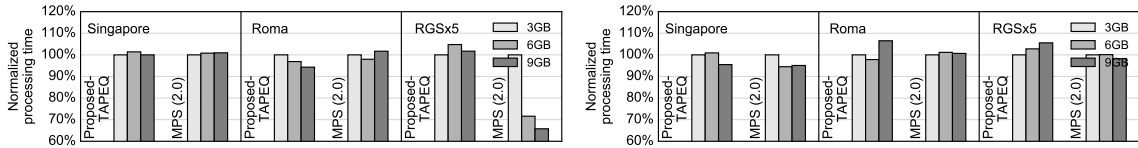


Figure 5.18.: Effect of buffer caches for TAPEQs (left: 100% temporal selectivity, right: 10% temporal selectivity). The processing times are normalized by the corresponding 3 GB case.

Moreover, the proposed algorithms showed better scalability. The index size evaluation also showed that the memory required for the FM-index was relatively small (less than 3% of the amount stored on disk), which easily fitted into memory, even for the largest dataset. These results demonstrate the practical advantages of the proposed method compared with existing methods that do not use string algorithms.

5.8. Discussion

SNT-index and NETTRA In the previous section, we have compared SNT-index and NETTRA (and its modified version). For TEQs and TAPEQs, the proposed SNT-index was faster than the baseline algorithms. For SPQs, the Proposed-SPQ method

was faster than the Full-join and Optimal-join methods and showed similar performance to the DHash-join. In our experiment, we used network distance for the hash function. In Krogh et al. [25], another hash function based on randomly chosen prime values was also proposed (here, we call it log-prime-hash). Generally, this log-prime-hash makes the Optimal-join slower, but the number of false positives in the DHash-join is significantly reduced. Therefore, NETTRA with the log-prime-hash can be a good solution in some applications. However, we cannot guarantee no false positive theoretically because the number of routes between two locations is a combinatorial number that cannot be represented as a 64-bit hash integer. On the other hand, in return for the relatively larger memory footprint, the SNT-index guarantees that there is no false positive. For automotive development (Section 3.2.2), we cannot admit false positives because such false positive data would behave as noisy training data that may lead to a bad recognition model. In summary, the SNT-index and the NETTRA have several trade-offs among processing speed, memory usage, supported queries, updating support, and guarantee of no false positive. Users have to choose them based on their application requirements. We consider that the memory usage is the main drawback of the proposed method. We discuss how to mitigate this in detail at the end of this section.

Possible Alternative Data Structures and Queries Classical NCT-indexing methods, like FNR-tree and MON-tree, manage temporal information as time intervals and use R-tree to manage them. We could also adopt such a data structure to implement the posting lists in SNT-index, instead of the forest of B⁺-trees used in our implementation. We would like to emphasize that SNT-index provides a general concept how to utilize suffix arrays for NCT-indexing, rather than a particular data structure. In addition, we did not consider using a spatial index (2D R-tree) for the road network $G = (V, E)$ in this study for simplicity. For SNT-index, it would be straightforward to use such a 2D R-tree to index the road network, and this would allow us to handle window queries (finding moving objects within a given spatial rectangle during a time interval I) in the same way as the classical NCT-indexing methods. Furthermore, to enhance the window query performance, we can combine an advanced NCT-indexing method, such as PARINET [48], instead of the simple posting lists used in the SNT-index. Specifically, we can implement this idea by storing an ISA value to each records of PARINET. Note that window queries on SNT-index with this 2D R-tree would show similar performance compared to the corresponding classical methods, because the FM-index cannot be used to process those queries.

Choice of FM-index The practical performance (i.e., size and search speed) of the FM-index depends on the choice of wavelet tree. In this Part II, we did not need to

5. Experiments

compress the FM-index because the uncompressed FM-index we used was moderate in size (approximately 1 GB, even for the largest RGSx5 dataset). If we had used larger dataset, we may need compression techniques. In Part III, we propose a compression method for FM-index storing a trajectory string.

6. Related Work

6.1. Trajectory Indexing

Many methods have been proposed for indexing spatial trajectories, and these can be divided into two categories: indexing for non-constrained trajectories and indexing for network-constrained trajectories (NCTs). A comprehensive list of these methods can be found in [36, 42].

For non-constrained trajectories, there have been a large number of studies, including TB-tree [47], which we treated in Section 4.3.1. Methods for non-constrained trajectories are typically based on R-trees [17]. Since NCTs can also be regarded as non-constrained trajectories, we can use those indexing methods to index NCTs as well, but the space efficiency would be low because the geographic coordinates of NCTs tend to concentrate around road segments. This is why dedicated indexing methods for NCTs have been considered.

Several NCT indexing methods also have been studied. FNR-tree [12] is one of the earliest, and consists of a 2D R-tree to index line segments and a forest of 1D R-trees for each segment to index time intervals. MON-tree [9] improves on FNR-tree in terms of data model. In the present thesis, we refer to these data structure as *inverted-index-like methods* because the forest of R-trees for temporal information is similar to an inverted index in document retrieval [34] (i.e., a path can be regarded as a document). Another important NCT-indexing method is T-PARINET [51], which can also be regarded as an inverted-index-like methods. This improves disk access locality by grouping neighboring road segments via graph partitioning. This idea could also be applied to **SNT-index**, although we have not implemented it explicitly in this thesis. An in-memory indexing method called SPNET [26] also partitions road graphs based on spatial proximity to enhance query processing, although it is not an inverted-index-like method. TRIFL [58] proposed a cost model and a self-tuning algorithm when trajectories are stored in flash storage. Although these methods handle NCTs, they mainly focus on range queries, which find moving objects that intersect a given spatial region during a given time interval. The most closely-related topic to **SNT-index** was studied in Krogh et al. [25], which proposed the NETTRA indexing structure and **SPQs**. An advantage of NETTRA over **SNT-index** is its support for real-time updating. The inability of **SNT-index** to

6. Related Work

handle real-time updates can be considered as the price of query processing efficiency. The choice of indexing method therefore depends on the application scenario.

The hash-based method (Section 4.2.2) proposed in NETTRA can be regarded as a special case of a *rolling hash* [23] in pattern matching. The suffix array [33] is also one of the most important tools in pattern matching. These relationships imply that string algorithms and NCT-indexing are strongly connected. We therefore expect that NCT-indexing will be further improved by advanced string algorithms in the future.

An important aspect of trajectory indexing is *trajectory preservation*, which enables accessing the data sequentially along a trajectory. This property was first considered in TB-tree [47] for non-constrained trajectories. The FM-index and ISA fields employed in SNT-index also provide sequential access to the next road segment, and can be regarded as an in-memory emulation of TB-tree pointers. In this sense, SNT-index has the trajectory preservation property while classical NCT-indexing methods, such as FNR-tree, MON-tree, and NETTRA, do not.

As pointed out in Section 5.8, our proposed method is conceptually related to *trajectory data warehousing*, which stores several aggregate measures of moving objects. Orland et al. [44] discussed data warehousing for non-constrained trajectories and proposed an efficient method for counting trajectories in a given spatial region during a given time interval.

In spatio-temporal database research, there are a number of data structures that support queries over past, present, and futures. For non-constrained trajectories, for example, RUM-tree [53] provides an efficient indexing for current position of moving objects, and TPR-tree/TPR*-tree [50, 57] index future positions of moving objects. Our SNT-index is essentially for historical data and cannot deal with present and future data. In the following, we discuss present data, and then discuss future data. In our opinion, data structures for past and present data should be built separately, otherwise we cannot support several path-based queries for huge past data efficiently. Based on this idea, we decided to omit an ability for frequent updating but to obtain an ability to process more complicated queries (SPQs, TEQs, and TAPEQs). For queries over *future*, Jeung et al. [21] and Hendawi et al. [19] treated the predictive queries for NCTs. We consider that the future position prediction has to be made based on large historical data. SNT-index can be useful when constructing probabilistic prediction models in an on-demand manner. It is an interesting future research question whether such various predictive queries can be supported by extending the proposed SNT-index.

Recently, in-memory indexing methods for NCTs have been proposed. Brisaboa et al. [4] proposed a data structure for counting queries, such as counting NCTs with origin $u \in E$ and destination $v \in E$ under a temporal constraint. Although the spatial component is similar to our method, the temporal information is stored in an in-memory data

structure called *wavelet matrix* [7]. As shown in their experiment, lossless compression of temporal information is difficult, that is, it consumes much more space than the spatial information. This is one of the reasons why we have not employed an in-memory data structure for temporal information. Some researchers have instead proposed lossy compression for temporal information [26, 55]. To the best of our knowledge, however, no lossless compression method exists that can achieve high compression ratio while still supporting useful queries. In this thesis, we did not consider compressing the temporal information because we assumed a sufficient amount of disk space. This is a reasonable assumption because disk prices have been dropping in recent years.

6.2. Related Queries

Spatio-temporal range queries have been widely studied for trajectory retrieval in classical settings. Although this type of query is still one of the most important queries for now, it is not adequate for the recent data-centric development in automotive technology and systems mentioned in Section 3.2.2. In Part II, we have considered SPQs, TEQs, and TAPEQs, which involve the spatial paths of the NCTs, but now we briefly describe related queries and methods.

Vieira et al. [59] conducted important related research, which treats cell phone trajectories, also regarded as symbolic trajectories, using an inverted-index-like structure. They considered a type of query called a *flexible pattern query* (FPQ), which is conceptually an SPQ superset. Their FPQ algorithm involves multiple *RangeQuery* executions and join operations, similar to Algorithm 7. SNT-index can also handle FPQs because our proposed data structure is equivalent to Vieira’s structure, apart from the FM-index and the ISA column in the postings-lists.

The concept of *combined queries* [47] has been proposed for non-constrained trajectories. This is a query of the following form: for all objects in the spatial range R during I , extract their trajectories during hour after they left R . It can be answered by combining a spatio-temporal range query with trajectory extraction (using TB-tree). This is similar to a TEQ, with a spatial range query instead of an SPQ. Therefore, SNT-index can process this combined query by replacing the SPQ in a TEQ with a spatio-temporal range query.

Krumm [27] studied a driver turn prediction model based on a high-order Markov chain. TEQ generalizes this concept, as pointed out in Section 3.2.3. Other related probabilistic methods include, for example, *Predestination* [28], *SubSyn* [61], and *PRO-CAB* [64]. By observing partial initial trajectories, these aim to predict their destinations based on probabilistic models. TEQs would be useful as a preprocessing step for these methods, because it can find the paths that follow a given initial trajectory.

6. Related Work

Time-period-based most-frequent path (TPMFP) [31] queries are conceptually similar to TAPEQs. Although the detailed definition is different, the motivation is the same: find routes between two locations based on frequencies during a given time interval. We can say that TAPEQ includes TPMFP, because TAPEQ finds *all* routes, including the ones with low frequency. Furthermore, we also note that our method is more versatile because the proposed SNT-index can process not only TAPEQs but also other types of queries (i.e., SPQs, TEQs, as well as spatio-temporal range queries mentioned in Section 5.8). In contrast, TPMFP needs a specialized data structure.

In addition, probabilistic models for predicting driver’s routes between two locations have been also studied [54, 64]. Like these methods, TAPEQ results can also be regarded as probabilistic predictions (see Figure 5.15). TAPEQ prediction accuracy would be improved by using such a probabilistic smoothing method for the retrieved routes.

7. Summary

In this Part II, we have proposed **SNT-index**, a novel indexing method for historical trajectories on a road network (NCTs) that is able to process path-based queries efficiently. In Chapter 3, we discussed the fact that the existing NCT-indexing methods need multiple joins to process path-based queries, which is inefficient when the related paths are long. Then, we described the proposed method in Chapter 4. In Section 4.1, we presented a novel indexing method based on two data structures, B⁺-trees and a suffix array (FM-index), showing that these two structures can be connected with an inverse suffix array. Using the proposed **SNT-index**, we have also proposed algorithms for processing several types of path-based queries, including **SPQs**, **TEQs**, and **TAPEQs** (Sections 4.2–4.4). We believe that these algorithms demonstrate a general strategy on how to utilize FM-indexes and inverse suffix arrays for retrieval of NCTs, which could be useful in designing algorithms for queries not treated here. Comprehensive experiments have also been performed to evaluate the efficiency and effectiveness of the proposed method (Chapter 5), showing great improvements in processing time by factors of a hundred or more in some cases.

In this way, we have successfully introduced string indexing and algorithms (i.e., suffix range and substring extraction queries) to NCT-indexing. Although stringology is an important research field, based on well-established theory, it has not been studied in the context of spatial databases. For trajectory processing, we have to take account of not only spatial movement but also other attributes, such as temporal information, which are not usually considered in string processing. There may even be other, still-unknown connections between NCT-indexing and string algorithms that have not been investigated here. It would be interesting to find such unknown links that can improve the processing of practically-important spatio-temporal queries.

We have focused on efficient query processing for static data, rather than real-time update. This is mainly because our target application is accessing historical data, which is required in the development of automotive systems based on data-driven methodologies like machine learning. Nonetheless, for the case where index updates are required, we have also proposed a semi-dynamic update scheme based on temporal partitioning in Section 4.5. However, some applications may require more frequent updates, and developing a suitable data structure based on string algorithms is an interesting future

7. *Summary*

research direction.

Part III.

Trajectory Compression

8. Research Issues and Preliminaries

In Part II (Chapters 3–7), we presented that the SNT-index can process several path-based queries for trajectories in road networks. FM-index storing spatial paths, referred to as *trajectory string*, is key to SNT-index. As FM-index is an in-memory index, it would not fit in memory if we have a huge number of trajectories. In this Part III, we study compression methods for spatial paths stored in FM-index.

8.1. Research Issue

Thus far, several compression methods for trajectories and those for strings have been studied independently. We first review compression methods for spatial paths. Lossless compression methods based on *shortest-path encoding* have been studied in [18, 24], and [26]. To compress spatial paths, these methods remove partial shortest paths included within each spatial path because such shortest paths can be recovered from the road network itself. However, this approach cannot be applied to FM-index straightforwardly. Another drawback is that it cannot guarantee the information-theoretic upper bound of the compressed data size. A recent lossless path compressor introduced in [18] called *minimum entropy labeling* (MEL) guarantees a theoretic bound and also achieves practically higher compressibility than the shortest-path encoding methods.

In stringology, many compression methods for strings have been studied, including several variants of FM-index. In Part II, we employed a simple implementation of FM-index; we stored the BWT T_{bwt} in a balanced-shaped wavelet tree with uncompressed bit vectors. This version of FM-index is, however, not very efficient for NCTs due to the following reason. As we described in Section 2.3.3 (and Figure 2.4), wavelet trees store the bit representation of each symbol $w \in \Sigma$. Accordingly, if the alphabet size $\sigma := |\Sigma|$ is large, the resulting wavelet tree has many nodes and the tree becomes deep. This makes the wavelet tree larger (and also makes the pattern matching query slower).

In general, alphabet size of trajectory strings is huge because the alphabet consists of all edges in a certain city, which is potentially huge. Therefore, the FM-index for trajectory strings is memory-intensive and suffers from slow query processing. Note that genomic sequences, one of the most important and widely-used application fields of FM-index, include only four characters (i.e., A, C, G, and T), and they are compactly

stored and efficiently processed using the simple FM-index (i.e., space consumption is only 2 bits/symbol).

Research Issue and Main Idea As mentioned above, we study a compression method for trajectory strings. A noteworthy feature of our approach is that our method utilizes a structure of road networks, while the most existing methods in stringology are designed for general strings. Concretely, our main idea relies on the following fact:

- *NCTs can only move along physically connected road segments.*

This feature is quite different from general sequences, as illustrated in Figure 8.1. In Figure 8.1(a), we show four example NCTs in a small network with six road segments (A–F). The corresponding graph shown in Fig. 8.1(b) represents symbol transitions for these four NCTs. Here, each vertex corresponds to a symbol (i.e., a road segment), and directed edges exist between two vertices if the corresponding two symbols can appear successively. For example, in Figure 8.1(b), vertex A is connected with vertexes B and D because we can only move to road segment B or D from A. For NCTs, this *empirical transition graph* (ET-graph) becomes a sparse graph, reflecting the physical topology of road networks. This sparsity cannot be obtained for general sequences, which leads to a denser ET-graph, as illustrated in Fig. 8.1(c).

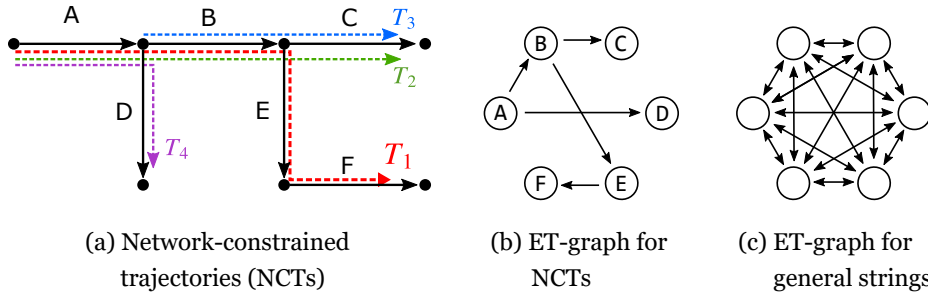


Figure 8.1.: (a) Network-constrained trajectories (NCTs), and both (b) sparse and (c) dense symbol transition graphs (ET-graphs).

Contribution Our proposed method, *Compressed-index for NCTs (CiNCT)*, significantly improves the compression and pattern matching operations when applied to sequences with a sparse ET-graph. To realize a high compression ratio with keeping its ability of fast pattern matching, we propose a series of novel techniques. We summarize our contributions as follows.

- We propose *relative movement labeling* (RML), which converts sequences on a sparse graph to low-entropy sequences. We theoretically prove its optimality and

show that RML provides a more compact representation of NCTs than that of the MEL method [18].

- We incorporate RML into FM-index by introducing a new concept called *Pseudo-Rank*, which leads to significant improvements in both size and query processing speed (i.e., the speed of pattern matching and sub-path extraction) as compared to existing FM-index variants. We also explain theoretically why this occurs.
- Using several real NCT datasets, we show that our method outperforms the state-of-the-art methods in stringology that do not consider graph sparsity.

Outline of Part III The remainder of Part III is organized as follows. In the rest of Chapter 8, we describe necessary concepts regarding compression, especially on Huffman-shaped wavelet trees and *compression boosting*, as well as some concepts from information theory. In Chapter 9, we describe our proposed method. Also, we provide information theoretic analysis of the proposed method and reveal the reason that our proposed FM-index is compact and fast. In Chapter 10, we demonstrate that our method outperforms existing methods using real datasets. In Chapter 11 and Chapter 12, we describe the related work and summary of Part III, respectively.

8.2. Preliminaries

In this section, we introduce necessary concepts for compression in addition to those introduced in Chapter 2. Specifically, in Section 8.2.1, we introduce Huffman-shaped wavelet tree (HWT) and the related complexities. In Section 8.2.2, further compression technique called *compression boosting* is introduced with the related compressed bit vector called *RRR*. With the compression boosting, FM-index can be compressed to the *k*th order empirical entropy (Eq. (8.4)). At the end of this section, we discuss the issues of these techniques.

8.2.1. Huffman-shaped Wavelet Tree

As a running example, we consider the following trajectory string consisting of four NCTs in Figure 8.1:

$$T = \underbrace{\text{FEBA}}_{T_1} \$ \underbrace{\text{CBA}}_{T_2} \$ \underbrace{\text{CB}}_{T_3} \$ \underbrace{\text{DA}}_{T_4} \$ \# . \quad (8.1)$$

8. Research Issues and Preliminaries

i	Rotations		j	Sorted Rotations	BWT
0	FEBA\$CBA\$CB\$DA\$#		0	#FEBA\$CBA\$CB\$DA\$	\$
1	EBA\$CBA\$CB\$DA\$#F		1	\$#FEBA\$CBA\$CB\$DA\$	A
2	BA\$CBA\$CB\$DA\$#FE		2	\$CB\$DA\$#FEBA\$CBA\$	A
3	A\$CBA\$CB\$DA\$#FEB		3	\$CBA\$CB\$DA\$#FEBA\$	A
4	\$CBA\$CB\$DA\$#FEBA		4	\$DA\$#FEBA\$CBA\$CB\$	B
5	CBA\$CB\$DA\$#FEBA\$		5	A\$#FEBA\$CBA\$CB\$D	D
6	BA\$CB\$DA\$#FEBA\$C		6	A\$CB\$DA\$#FEBA\$CB\$	B
7	A\$CB\$DA\$#FEBA\$CB\$		7	A\$CBA\$CB\$DA\$#FEB	B
8	\$CB\$DA\$#FEBA\$CBA\$		8	B\$DA\$#FEBA\$CBA\$C	Last C
9	CB\$DA\$#FEBA\$CBA\$	Sort	9	BA\$CB\$DA\$#FEBA\$C	Column C
10	B\$DA\$#FEBA\$CBA\$C		10	BA\$CBA\$CB\$DA\$#FE	E
11	\$DA\$#FEBA\$CBA\$CB\$		11	CB\$DA\$#FEBA\$CBA\$	\$
12	DA\$#FEBA\$CBA\$CB\$		12	CBA\$CB\$DA\$#FEBA\$	\$
13	A\$#FEBA\$CBA\$CB\$D		13	DA\$#FEBA\$CBA\$CB\$	\$
14	\$#FEBA\$CBA\$CB\$DA\$		14	EBA\$CBA\$CB\$DA\$#F	F
15	#FEBA\$CBA\$CB\$DA\$		15	FEBA\$CBA\$CB\$DA\$#	#

Figure 8.2.: The BWT of T is defined to be the last column of the sorted rotations of T . This example is based on the trajectory string T in Eq. (8.1).

For convenience, we illustrates the corresponding BWT in Figure 8.2, which is obtained as follows.

$$T_{\text{bwt}} = \$AAABDBBCCE$$$F#. \tag{8.2}$$

For each symbol $w \in \Sigma$, we can assume several coding schemes. The Huffman coding assigns a shorter code to a more frequent symbol, while the standard coding assigns equi-length codes. For the example string T , the codes are assigned as follows.

w	Frequency	Huffman	Standard
#	1	0110	000
\$	4	10	001
A	3	111	010
B	3	00	011
C	2	110	100
D	1	0111	101
E	1	0100	110
F	1	0101	111
Avg. Code Len.	—	2.8125	3.0

With the Huffman coding, we obtain a wavelet tree storing T as illustrated in Figure 8.2. As we see in the table above, the average code length of the Huffman coding (2.8125) is shorter than that of the standard coding (3.0). It is known that the average code length of the Huffman coding for a string S is at most $(1 + H_0(S))$ bits; here, $H_0(S)$

is the 0 th order empirical entropy [35]:

$$H_0(S) = \sum_{w \in \Sigma} \frac{n_w}{n} \lg \frac{n}{n_w}, \quad (8.3)$$

where n_w is the number of occurrences of w in S .

Wavelet trees with Huffman coding is referred to as *Huffman-shaped wavelet trees* (HWTs; Figure 8.3). If we use the standard coding, we obtain the *balanced-shaped wavelet trees*, which has been employed throughout Part II.

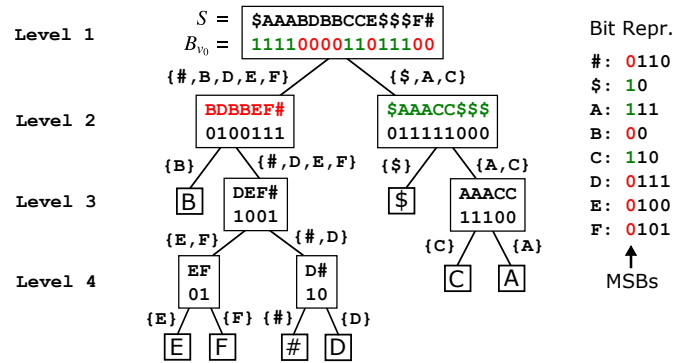


Figure 8.3.: Wavelet tree for the string $S = \$AAABDBBCCCE$$$F\#$.

As described in Section 2.3, each node v in a wavelet tree stores a bit vector B_v . The total length of bit vectors B_v over nodes in a wavelet tree can be represented as $\sum_v |B_v|$. For HWTs, the total length is $|S|(1 + H_0(S))$, which implies the total number of bits stored in HWT becomes small if S becomes a string with small $H_0(S)$. As will be described, our method uses this fact by converting T_{bwt} to a string with smaller entropy.

To calculate $\text{rank}_w(S, j)$, the wavelet tree calculates the bit-wise rank value at each node v_0, v_1, \dots, v_k between the root and the leaf corresponding to the bit representation $w = b_0b_1 \dots b_k$ (see Section 2.3). This indicates that bit-wise rank operations required to obtain $\text{rank}_w(S, j)$ is equal to k (i.e., the length of the bit representation of w). This fact leads to the following result [38].

Theorem 8.1 (Rank on HWT) *If $\text{rank}_w(S, j)$ is executed on uniformly random w over $S[0, n)$, it runs in $O(1 + H_0(S))$ time on average.*

This implies that a string with small entropy $H_0(S)$ achieves not only a small size but also a fast rank computation, which plays an important role in our theoretical analysis.

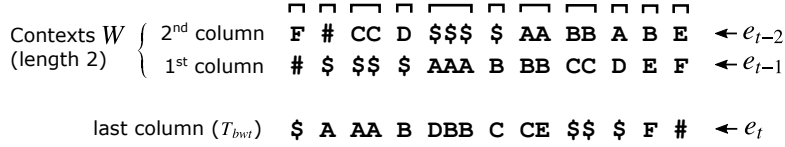


Figure 8.4.: Compression boosting of FM-index: T_{bwt} is divided into *contexts* and each partition is compressed separately.

8.2.2. Compressed Variants of FM-index

We consider further compression of FM-index. Let us imagine a sub-path of length 3 in a real NCT dataset: $e_{t-2} e_{t-1} e_t$. It is unlikely that two right turns occur in a row because most vehicles go toward their destinations. Considering such high-order correlations among symbols, we can boost the compression. Let us go back to Figure 8.2. A substring $W = \text{“BA”}$ appears as prefix in the range $R(\text{BA}) = [9, 11)$. The other prefixes $W \in \Sigma^2$ have their corresponding ranges.

Let us divide T_{bwt} based on such prefixes W (called *contexts* of length two) as in Figure 8.4. These context blocks represent the next segment e_t given the context $W = e_{t-1} e_{t-2}$. We have a chance of compression because the frequency of symbols in each context is biased as discussed above.

Compression Boosting (CB) The above idea can be generalized to any length of context. Let us divide T_{bwt} into l blocks of context $W \in \Sigma^k$ of length k : $T_{bwt} = L_1 L_2 \cdots L_l$ ($l \leq \sigma^k$). Storing each L_j in a 0-th order entropy compressor (i.e., in $|L_j| H_0(L_j)$ bits; this is approximately achieved by HWTs), we can compress T_{bwt} in $n H_k(T)$ bits. Here, H_k is k -th order empirical entropy [35]:

$$H_k(T) := \sum_{W \in \Sigma^k} \frac{n_W}{n} H_0(T_W), \tag{8.4}$$

where T_W is the concatenation of all symbols in T that precede the occurrences of the context W . To support a fast rank operation on those divided blocks, we need to precompute and store the rank results at each location of l blocks for all $w \in \Sigma$.

Taking larger k seems to be desirable because we have $H_k(T) \geq H_{k+1}(T)$ for all $k \geq 0$ [35]. However, partitioning into many blocks leads to the following problems in practice:

- P1) Blocks of variable length lead to inefficient random access to T_{bwt} .
- P2) Index size increases because of the overhead of block-wise storage (e.g., pointers in Huffman trees or overhead spaces due to bit vectors).
- P3) We have to save $l\sigma$ integers for the rank results. This is unrealistic for huge σ even

if $k = 1$ ($l = \sigma$).

Practical Implementation of Compression Boosting There are some CB variants that avoid the above problems. Fixed-block boosting [22] adopts blocks of a fixed size. Although this solves P1 (and P2 partially), problem P3 remains for huge σ . *Implicit compression boosting* (ICB) [32] avoids such explicit block partitioning by using compressed bit vectors called *RRR* [49] in the wavelet tree of T_{bwt} . This solves P1 and P3. In the following, we consider two types of implicit compression boosters, namely ICB-Huff and ICB-WM. The former is ICB with an HWT, while the latter is ICB with a *wavelet matrix* [7], which is an efficient alternative to a wavelet tree. As discussed in our theoretical analysis, ICBs still suffer from large overheads when applied to a string with large alphabet, such as a trajectory string.

9. Compressing FM-index for Trajectories

For NCTs, the alphabet size σ can be millions because it is the number of road segments in a road network. As discussed in the previous chapter, this makes the compression of trajectory strings inefficient, because the redundant bits in wavelet trees increase as σ increases. To avoid this, we propose to convert trajectory strings into strings on a small alphabet via *relative movement labeling* (RML), which relies on the sparsity of road networks. We give an overview of how to construct the proposed data structure (CiNCT) in the following.

1. Convert a set of NCTs into a trajectory string T .
2. Calculate the BWT of T and obtain T_{bwt} .
3. Construct an *ET-graph* G_T and a *relative movement labeling* (RML) function ϕ based on T (Section 9.1)
4. Label T_{bwt} based on the RML function ϕ and obtain the *labeled BWT* $\phi(T_{bwt})$ (Section 9.2).
5. Store $\phi(T_{bwt})$ in an HWT with RRR and obtain the proposed index structure (Section 9.2).

As steps 1 and 2 are straightforward, we describe the details of steps 3–5 in the subsequent sections. We emphasize that the NCTs are labeled *after* the BWT construction (step 4), otherwise we would be unable to implement the pattern matching query. Due to this labeling step, we need to develop an algorithm that differs from Algorithm 1. Such an algorithm is described in Section 9.3. The theoretical consequences of CiNCT are described in Section 9.4.

9.1. Relative Movement Labeling

The RML converts trajectory strings into strings with small alphabet based on the following fact: *NCTs can only move between physically connected road segments*. First,

9. Compressing FM-index for Trajectories

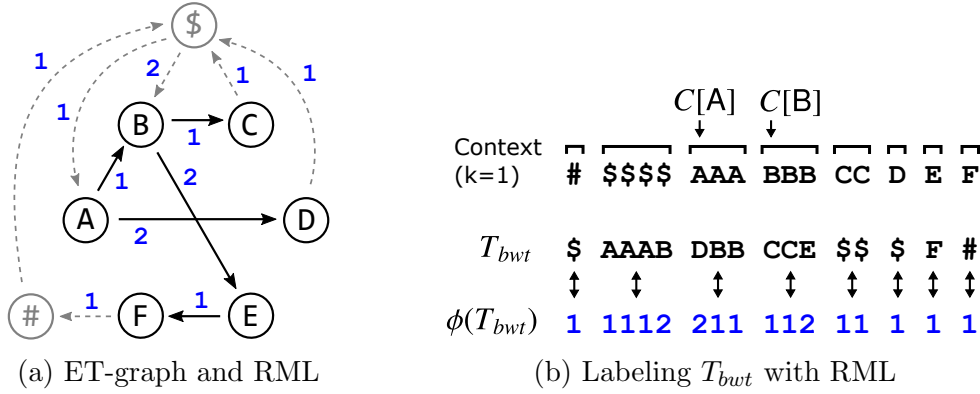


Figure 9.1.: (a) ET-graph of the example trajectory string $T = \text{FEBA\$CBA\$CB\$DA\$#}$: each node represents a road segment, and an edge exists if the corresponding transition occurs in T . The integer on each edge is the corresponding label. (b) Relative movement labeling: $\phi(T_{bwt})$ produces a string with a lower entropy than that of T_{bwt} .

we describe its idea based on the example in Figure 8.1 (a). If a vehicle is traveling along a road segment $w' = A$, the next segment w has to be B or D. Hence, we label them 1 and 2, respectively. Generally, if there are k connected road segments from a certain segment, we can label them with $1, \dots, k$. The sequences converted with this *relative movement labeling* (RML) are expected to have small alphabet because k is smaller than the maximum out-degree of the road network. To define RML formally, let us define an *empirical transition graph* (ET-graph).

Definition 9.1 (ET-graph) Let T be a string defined on an alphabet Σ . An ET-graph G_T of T is a directed graph defined as follows: 1) the vertex set is Σ ; 2) a directed edge $(w', w) \in \Sigma \times \Sigma$ exists iff T contains a substring ww' at least once. The edge set is denoted by E_T . In the following, we call a substring of length two a bigram.

In other words, an edge exists if and only if a direct transition between w' and w exists in T . An ET-graph constructed from a trajectory string of real data is expected to be a sparse graph because it has a similar topology to the original road network.

Remark 7 (Sparse Graph) *Out-degree* of a vertex v of a directed graph is the number of edges $e \in E$ that have v as their tail node, i.e., $\text{tail}(e) = v$. The *average out-degree* is an average over all vertices. In general, the maximum out-degree of a vertex can be $|V| - 1$. We use the term ‘*sparse graph*’ as a graph whose average out-degree is much smaller than $|V| - 1$. Similarly, the term ‘*maximum out-degree*’ is used as the maximum value of out-degrees over all vertices.

Example 9.1 Figure 9.1 (a) illustrates the ET-graph of the trajectory string

$$T = FEBA\$CBA\$CB\$DA\$#.$$

As a bigram $ww' = BA$ appears in T , we have an edge from $w' = A$ to $w = B$. Another bigram $ww' = BD$ never appears in T ; hence there is no edge from $w' = D$ to $w = B$. For convenience, we add an edge from the first symbol $w' = F$ to the last symbol $w = \#$. Note that ET-graphs include the special symbols $\$$ and $\#$.

Remark 8 Rigorously, the ET-graph of a trajectory string would have a large out-degree at the vertex corresponding to the special symbol ‘ $\$$ ’, because any symbol $w \in E$ is possible before ‘ $\$$ ’; in other words, trajectory can start from any road segments. As we see later, this is usually not a problem because the ratio of ‘ $\$$ ’ in T is relatively low.

Definition of RML RML is defined as an integer assigned on each edge of the ET-graph (see Figure 9.1 (a)). For example, the transition $A \rightarrow B$ is labeled 1. We denote such labeling as $\phi(B|A) = 1$. The transition $A \rightarrow D$ must have a different label, otherwise we cannot distinguish them.

In general, for transition $w' \rightarrow w$, we denote such a labeling function by $\phi(w|w')$. To make the labeling *distinct* given the previous symbol w' , the RML function ϕ must satisfy the following requirement:

- Given a symbol $w' \in \Sigma$, the RML function $\phi(\cdot|w')$ must be a one-to-one map.

Now, we discuss how to construct the RML function ϕ that satisfies the requirement above. Let us consider the *out-vertex set* of w' , defined as $N_{out}(w') = \{w|(w', w) \in E_T\}$, that determines the set of vertices directly accessible from w' . Based on the ET-graph and out-vertex set, we define $\phi(\cdot|w')$ as follows. Given w' , assign a different small integer $c_{ww'}$ to each $w \in N_{out}(w')$ and define $\phi(w|w') := c_{ww'}$, where $c_{ww'}$ is an integer between 1 and $k = |N_{out}(w')|$. It is clear that $\phi(\cdot|w')$ is a one-to-one map. If $w \notin N_{out}(w')$, we cannot define $\phi(w|w')$; however, this is not a problem because $w \notin N_{out}(w')$ indicates that the string ww' is not found in T , which tells us the result of pattern matching is null. This point is important for our search algorithm.

Finding Optimal RML (Step 3) The RML function ϕ described above does not define a unique labeling function because we have not yet specified a concrete way to assign the small integer $c_{ww'}$ (i.e., there are degrees of freedom for permutation). Here, we propose a strategy based on a bigram count $n_{ww'}$ (i.e., the frequency of the bigram ww' in T). The elements in $N_{out}(w')$ are sorted in descending order of bigrams $n_{ww'}$. The vertex w

with the largest bigram count is given the smallest label, 1. The second-most frequent vertex is labeled 2, the third-most frequent vertex is labeled 3, and so on.

Example 9.2 *The labels in Figure 9.1 (a) are actually determined in this way. Consider the same trajectory string $T = \underline{FEBA}\$CBA\$CB\$DA\$#\$. Bigrams BA and DA appear twice and once, respectively; hence we have $n_{BA} > n_{DA}$. Therefore, the edge from A to B has the smallest label 1 (i.e., $\phi(B|A) = 1$); the edge from A to D has the second smallest label, 2 (i.e., $\phi(D|A) = 2$).*

In the next section, we show how to convert T_{bwt} using this bigram-based RML function ϕ . One might wonder whether there exists a better labeling strategy; however, we prove the optimality of the labeling that leads to strong conclusions in Section 9.4.1: *our RML achieves the smallest size and the fastest search among all possible (and valid) labeling functions.*

9.2. Data Structure

In this section, we describe how to convert the plain BWT T_{bwt} into the labeled BWT $\phi(T_{\text{bwt}})$; then, we explain the concrete data structure. These respectively correspond to the steps 4 and 5 mentioned at the beginning of this chapter.

Labeling BWT (Step 4) Based on the RML function ϕ obtained in the previous section, the BWT T_{bwt} is converted to $\phi(T_{\text{bwt}})$ as follows. For each $w' \in \Sigma$, we consider the suffix range of w' , i.e., $[C[w'], C[w' + 1]]$, where C is the C-array of T . Then, each symbol $w = T[i]$ in this range (i.e., $\forall w \in T[C[w'], C[w' + 1]]$) is converted into $\phi(w|w')$. Note that $\phi(w|w')$ is guaranteed to be defined for this pair (w, w') because this ww' is a bigram that appears in T by definition of BWT.

Applying the conversion above for each suffix range of $w' \in \Sigma$, we obtain *the labeled BWT* $\phi(T_{\text{bwt}})$. This labeled BWT is expected to be a low entropy string because each symbol $T[i]$ is converted to a small integer. As noted in Remark 8, symbols within $[C[\$], C[\$ + 1])$ would not be small integers; however, the impact of these symbols on the entropy is expected to be small because the ratio of such symbols in T_{bwt} is low.

Example 9.3 *Let us focus on the third block of T_{bwt} , DBB, in Figure 9.1 (b). This block corresponds to the context of A, which indicates that the previous symbol of these DBB is A. Hence, DBB is labeled as 211 because $\phi(B|A) = 1$ and $\phi(D|A) = 2$ in Figure 9.1 (a). All the other blocks also can be labeled in the same manner. This labeling strategy generates a low-entropy sequence $\phi(T_{\text{bwt}})$ as shown in Figure 9.1 (b), because the distribution of the resulting symbols is biased toward smaller integers (i.e., 1 is the largest fraction). For this example, we have $H_0(T_{\text{bwt}}) \simeq 2.8$ and $H_0(\phi(T_{\text{bwt}})) \simeq 0.7$ (unit: bits/symbol).*

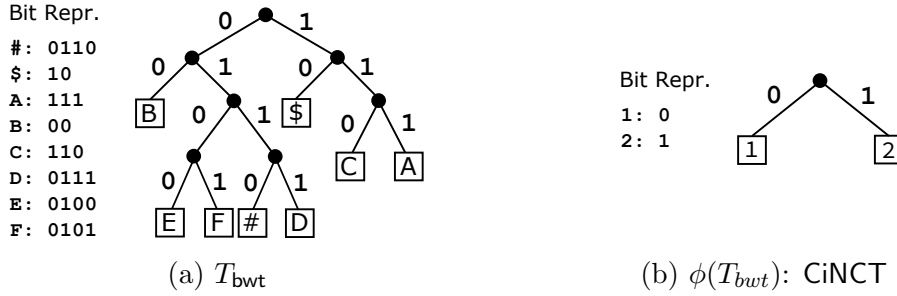


Figure 9.2.: Comparison of the Huffman trees of T_{bwt} and $\phi(T_{bwt})$: each leaf corresponds to a symbol. The tree generated by CiNCT (b) is much simpler than that from the standard Huffman tree (a).

Storing to a Compressed Wavelet Tree (Step 5) Finally, we store the labeled BWT $\phi(T_{bwt})$ to an HWT. For bit vectors in an HWT, we adopt a practical version of the compressed bit vector called RRR [39]. Figure 9.2 depicts the comparison of Huffman trees of T_{bwt} and $\phi(T_{bwt})$ for the example in Figure 9.1 (b). The Huffman tree of $\phi(T_{bwt})$ is obviously simpler than that of T_{bwt} . Because these tree shapes are the same as those of HWTs, this simplification explains intuitively why CiNCT is small and fast. We provide detailed theoretical analysis in Section 9.4.

An RRR bit vector has one parameter b , that controls the size of the internal blocks. For larger b , we obtain better compression but slower search (*rank* calculation) in general, and *vice versa*. This b is the only parameter in CiNCT. However, in Chapter 10, we show that this parameter has only a small influence on the index size and the search time; we also show that our method consistently better than baseline methods whatever parameter values for b are chosen.

Storing ET-graph We use an adjacency list to represent the ET-graph G_T . The value $\phi(w|w')$ is assigned to the edge $(w', w) \in E_T$. Thus $\phi(w|w')$ is obtained in $O(\delta)$ time by a linear search over $N_{out}(w')$. We also assign $C[w]$ to each vertex w in G_T . *Correction terms* $Z_{w'w}$, introduced in Section 9.3.1, are also attached to each edge. Since G_T is sparse, the space needed to store G_T is negligible when $|T|$ gets large. Note that ET-graphs can be implemented using *succinct graphs* (e.g., [40]); however, we do not use them because their impact on the data size was small in our preliminary experiments.

9.3. Query Processing

In the previous section, we introduced the labeled BWT, which is a string by converting the original BWT string. This requires us to develop a new algorithm for the pattern matching and substring extraction queries. In the following, we describe another key

concept of our method, *PseudoRank*, then show algorithms for the pattern matching and substring extraction queries.

9.3.1. PseudoRank

Fast computation of $rank_w(T_{bwt}, j)$ is needed for the pattern matching and substring extraction queries (Algorithm 1 and Algorithm 2 in Section 2.3). The original FM-index stores T_{bwt} in a wavelet tree to calculate ranks quickly. In our case, however, we do not have the original T_{bwt} but only have the labeled $\phi(T_{bwt})$. Can we obtain the rank values for the original BWT by using only the labeled BWT? Seemingly, this is difficult because different symbols are mapped to the same label (e.g., both A and C are converted to 1 as illustrated in Figure 9.1 (b)).

The key idea is to simulate the rank operation over T_{bwt} . To begin with, we explain the idea with Figure 9.3 and the following example.

Example 9.4 *Let us consider the suffix range of A, i.e., $R(A) = [C[A], C[B])$, and $j \in R(A)$. Remembering the substring $T_{bwt}[C[A], C[B]) = DBB$ is labeled as **211** by using the one-to-one map $\phi(\cdot|A)$ as described in Figure 9.1 (b), the following two counts are equivalent for $\forall j \in R(A)$:*

- *the number of occurrences of D within the range $R' := [C[A], j)$ in T_{bwt} (the shaded region in Figure 9.3), and*
- *the number of occurrences of 2 within R' in $\phi(T_{bwt})$.*

This equivalence holds in general. Let us consider a context w' . For all j such that $C[w'] \leq j \leq C[w'+1]$, let us consider a range $R' := [C[w'], j)$ (the shaded region in Figure 9.3). For a symbol $w \in N_{out}(w')$, the number of occurrences w within R' in T_{bwt} and that of the label $\eta := \phi(w|w')$ within R' in $\phi(T_{bwt})$ are the same because of the one-to-one requirement for $\phi(\cdot|w')$. This leads to the following balancing equation:

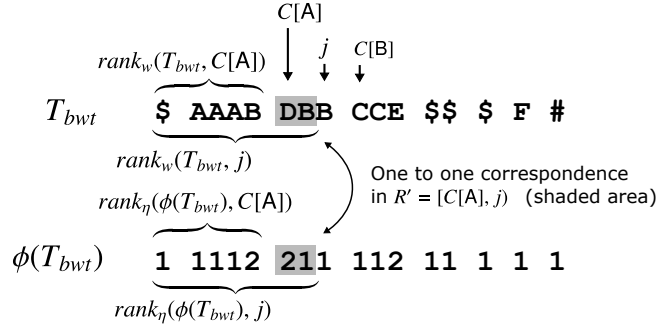
$$rank_w(T_{bwt}, j) - rank_w(T_{bwt}, C[w']) = rank_\eta(\phi(T_{bwt}), j) - rank_\eta(\phi(T_{bwt}), C[w']). \quad (9.1)$$

(The LHS is the number of occurrences of w in R' and the RHS is that of the label η of w in R' .) Rearranging this equation, we have the following theorem, which allows us to simulate the rank operation.

Theorem 9.1 (Pseudo-rank) *Suppose $w \in N_{out}(w')$ and $C[w'] \leq j \leq C[w'+1]$. Let $\eta := \phi(w|w')$, then we have*

$$rank_w(T_{bwt}, j) = rank_\eta(\phi(T_{bwt}), j) - Z_{w'w}, \quad (9.2)$$

$$\text{where } Z_{w'w} := rank_\eta(\phi(T_{bwt}), C[w']) - rank_w(T_{bwt}, C[w']). \quad (9.3)$$

Figure 9.3.: Basis of the balancing equation (Eq. (9.1)) for *PseudoRank*

We emphasize that the correction term $Z_{w'w}$ does not depend on j , implying that the number of correction terms needed is equal to $|E_T|$. Importantly, this property allows us to precompute and store the correction terms (as noted in Section 9.2, they are attached to each edge $(w', w) \in E_T$).

Theorem 9.1 provides Algorithm 13, which computes the rank values using only $\phi(T_{\text{bwt}})$. We also emphasize that *PseudoRank* does not allow us to compute rank values for all pairs of (w, j) . However, this limitation is not a problem for our search algorithm, as shown in the next subsection.

Algorithm 13: Emulating $\text{rank}_w(T_{\text{bwt}}, j)$ by using only $\phi(T_{\text{bwt}})$ (*PseudoRank*($\phi(T_{\text{bwt}}), j, w, w', Z_{w'w}$))

Input: Labeled BWT string $\phi(T_{\text{bwt}})$,
 Location of rank j ,
 Correction term $Z_{w'w}$,
 Target symbol w ,
 Previous symbol w'

Output: The value of $\text{rank}_w(T_{\text{bwt}}, j)$

```

1 if  $w \in N_{\text{out}}(w')$  and  $C[w'] \leq j \leq C[w' + 1]$  then
2    $\eta \leftarrow \phi(w|w')$  // RML
3   return  $\text{rank}_\eta(\phi(T_{\text{bwt}}), j) - Z_{w'w}$ 
4 return NotFound

```

9.3.2. Suffix Range Query with CiNCT

With the *PseudoRank*, we can simulate $\text{rank}_w(T_{\text{bwt}}, j)$ using only the wavelet tree of $\phi(T_{\text{bwt}})$ and the correction term $Z_{w'w}$ (Eq. (9.3)). Replacing the rank operations in Algorithm 1 with *PseudoRank*, we obtain our search algorithm (Algorithm 14), whose correctness is shown below.

Algorithm 14: Finding the suffix range $[sp, ep]$ for a given query P of length m based on $\phi(T_{bwt})$ (*LabeledSearchFM*)

Input: Labeled BWT string of length n : $\phi(T_{bwt})$,
 Query of length m : $P[0, m)$,
 Correction terms: $\{Z_{w'w}\}$

Output: Range of T_{bwt} that matches to P

```

1  $w \leftarrow P[m - 1]$ ;  $sp \leftarrow C[w]$ ;  $ep \leftarrow C[w + 1]$ 
2 for  $i \leftarrow 2$  to  $m$  do
3    $w' \leftarrow w$  // Save the previous symbol
4    $w \leftarrow P[m - i]$ 
5   if  $w \notin N_{out}(w')$  then
6     return NotFound
7    $sp \leftarrow C[w] + PseudoRank(\phi(T_{bwt}), sp, w, w', Z_{w'w})$ 
8    $ep \leftarrow C[w] + PseudoRank(\phi(T_{bwt}), ep, w, w', Z_{w'w})$ 
9   if  $sp \geq ep$  then
10    return NotFound
11 return  $[sp, ep)$ 

```

Correctness of the Algorithm To guarantee that Algorithm 14 is equivalent to Algorithm 1, we have to check the following two conditions on *PseudoRank* (Theorem 9.1) are satisfied immediately before Line 7: (c1) $w \in N_{out}(w')$; (c2) $C[w'] \leq sp \leq C[w' + 1]$ and $C[w'] \leq ep \leq C[w' + 1]$.

As noted previously, no substring ww' appears in T if $w \notin N_{out}(w')$; hence, **NotFound** is returned if $w \notin N_{out}(w')$ at Line 6. Therefore, (c1) $w \in N_{out}(w')$ holds immediately before Line 7. For (c2), before Line 7, sp satisfies

$$sp = C[w'] + rank_{w'}(T_{bwt}, sp'), \quad (9.4)$$

where sp' is the previous value. By the *rank* definition, $0 \leq rank_{w'}(T_{bwt}, j) \leq C[w' + 1] - C[w']$ ($0 \leq \forall j < |T|$) holds, where $C[w' + 1] - C[w']$ means the number of occurrences of w' in T . Combining this inequality with Eq. (9.4), we obtain $C[w'] \leq sp \leq C[w' + 1]$. We can prove the condition for ep in a similar manner.

9.3.3. Extracting a Substring with CiNCT

By replacing the rank computation in the substring extraction queries (Algorithm 2 with *PseudoRank*), we obtain Algorithm 15. Line 1 performs a binary search to find the last character $T[i] = w'$ such that $C[w'] \leq j < C[w' + 1]$. At Lines 4–6, we first access the j -th character of $\phi(T_{bwt})$ (i.e., the labeled $T_{bwt}[j]$), then decodes the $T_{bwt}[j] = T[i - k - 1] = w$ using the ET-graph. At Line 7, we move to the next position using *PseudoRank* version

Algorithm 15: Extracting a sub-path $T[i-l, i)$ for given $j = ISA[i]$ and $l > 0$

Input: Labeled BWT: $\phi(T_{\text{bwt}})$, Position on T_{bwt} : j ,
Extraction length: l , Correction terms: $\{Z_{w'w}\}$
Output: A substring $S := T[i-l, i)$

```

1  $w' \leftarrow \text{BinarySearch}(j, \{C[w']\})$ 
2  $S \leftarrow$  empty string
3 for  $k \leftarrow 1$  to  $l$  do
4    $\eta \leftarrow \text{access}(\phi(T_{\text{bwt}}), j)$  //  $\phi(T_{\text{bwt}})[j]$ 
5    $w \leftarrow \text{decode}(\eta|w')$ 
6    $S \leftarrow wS$ 
7    $j \leftarrow C[w] + \text{PseudoRank}(\phi(T_{\text{bwt}}), j, w, w', Z_{w'w})$ 
8    $w' \leftarrow w$  // Save previous symbol
9 return  $S$ 

```

of LF-mapping.

9.4. Theoretical Analysis

Here, we explain theoretically why CiNCT is compact and fast. We first show the optimality of RML, that is, the labeled BWT $\phi(T_{\text{bwt}})$ achieves the smallest entropy. Then, we explain that such a small entropy contributes high compressibility and fast query processing. We also show that RML is better than other labeling method called MEL, recently proposed in [18].

9.4.1. Optimality of RML

The 0th order entropy H_0 given in Eq. (8.3) plays important roles in our analysis. First, we show the labeling strategy based on bigram counts $n_{ww'}$ proposed in Section 9.1 achieves the minimum value of H_0 among all possible labelings (we provide the proof at the end of this section).

Theorem 9.2 (Optimality) *Let ϕ^* be the RML based on the bigram ordering strategy and ϕ be any possible RML that satisfies the one-to-one mapping requirement in Section 9.1. Then, we have*

$$H_0(\phi^*(T_{\text{bwt}})) \leq H_0(\phi(T_{\text{bwt}})). \quad (9.5)$$

As a special case of this theorem, we obtain an *unlabeled case* result, i.e., $H_0(\phi^*(T_{\text{bwt}})) \leq H_0(T_{\text{bwt}})$, by putting as $\phi = id$ (identity labeling). Importantly, we see that the entropy

of the labeled BWT is much smaller than that of the original BWT, i.e.,

$$H_0(\phi^*(T_{\text{bwt}})) \ll H_0(T_{\text{bwt}}) \quad (9.6)$$

holds for real NCT datasets in our experiments (Table 10.2).

9.4.2. Space Complexity

Evaluating Space Overheads The data structure of CiNCT consists of two parts: the labeled BWT $\phi(T_{\text{bwt}})$ and the ET-graph G_T . As noted in Section 9.2, the size of G_T is negligible when $|T|$ is large. Here, we compare the sizes of T_{bwt} and $\phi(T_{\text{bwt}})$ stored in HWTs with RRR. Note that these corresponds ICB-Huff and CiNCT, respectively. The main advantage of CiNCT comes from the lower space overhead due to RRR, as explained below. For a given bit vector B , it is known that the practical RRR with the parameter b (see Section 9.2) uses at most

$$|B|H_0(B) + |B| \cdot h(b) \quad (9.7)$$

bits where $h(b) = \frac{\lg(b+1)}{b}$ [39]. We call the second term the *RRR-overhead*. For $b = 63$, we have an overhead of $h(b) = (\lg 64)/63 \simeq 0.095$ bits per bit.

For a given string S , it is known that the average code length with Huffman coding is at most $(1 + H_0(S))$ bits [38]. Hence, the total length of bit vectors in the HWT is $\sum_v |B_v| \simeq |S|(1 + H_0(S))$. Summing the RRR-overheads over all internal nodes v in the HWT, we obtain total bits of the overhead:

$$\sum_v |B_v| \cdot h(b) \simeq |S|(1 + H_0(S)) \cdot h(b). \quad (9.8)$$

The right-hand side implies that the RRR-overhead of a sequence S is small if its entropy $H_0(S)$ is small. Therefore, Eq. (9.6) indicates that the space overhead for CiNCT is much smaller than that for ICB-Huff.

High-order Compression Here, we analyze the remaining first (and dominant) term in Eq. (9.7). Summing this term over all internal nodes v in the HWT, we find that the total bits needed for this term achieves the k -th order entropy Eq. (8.4) for all $k > 0$, as shown in the following Theorem 9.3. This theorem implies that our method guarantees high compressibility in an information theoretic sense. Note that this kind of entropic bound has not been guaranteed by the existing shortest-path based NCT compressors.

Theorem 9.3 *For all $k > 0$, the total bits required to store $\phi(T_{\text{bwt}})$ in an HWT with*

RRR, apart from the overhead Eq. (9.8), are $|T|H_k(T) + O(l\sigma b)$, where $l \leq \sigma^k$ is the number of distinct contexts $W \in \Sigma^k$ in T .

PROOF See Section 9.5.

9.4.3. Time Complexity

To evaluate whether Algorithm 14 is faster than Algorithm 1, we focus on the time complexity of the rank operation. As stated in Theorem 8.1, $rank_w(S, j)$ runs in $O(1 + H_0(S))$ time.¹ Hence, the relationship $H_0(\phi(T_{bwt})) \ll H_0(T_{bwt})$ (Eq. (9.6)) again explains why CiNCT is faster than ICB-Huff. Of course, Algorithm 14 incurs an additional cost in calculating $\phi(w|w')$, but this is not serious for a sparse G_T .

Moreover, we have the following theorem implying that the search time does not depend on the road network size σ but depends only on the maximum out-degree δ of the road network (which is usually less than four).

Theorem 9.4 (σ -independence) *Let $P \in E^*$ be any query path ($\$$ is not included). Algorithm 14 runs in $O(|P| \cdot \delta b)$ time.*

PROOF For any $w, w' \in E$, we have $\eta := \phi(w|w') \leq \delta + 2$. By the construction of RML, η is at least the $\delta + 2$ -th most frequent symbol in $\phi(T_{bwt})$. Thus η is at most located at the $\delta + 2$ level of the Huffman tree. Hence, $rank_\eta(\phi(T_{bwt}), j)$ in Eq. (9.2) runs in $O(\delta b)$ time (remember the bit-wise rank operation in practical RRR [39] requires $O(b)$ time). Since *PseudoRank* is calculated at most $2|P| - 2$ times in Algorithm 14, this leads to the conclusion.

Other FM-indexes do not satisfy this property. Note that this time complexity also does not depend on the data size $|T|$.

9.4.4. Comparison of RML with MEL

Minimum entropy labeling (MEL) is a labeling scheme for NCTs proposed in [18], which works as a preprocessor for general compressors, such as the Huffman coding or the LZ coding (i.e., pattern matching was not considered). Similar to RML, MEL converts a sequence of road edges to a low entropy sequence of small integers as follows: $w_1 w_2 \cdots w_n \rightarrow \psi(w_1) \psi(w_2) \cdots \psi(w_n)$ where $\psi : E \rightarrow \mathbb{N}$ is the MEL function. Different labels are assigned to road segments that shares a head node v (Fig. 9.4(b)). In contrast,

¹To be exact, this complexity is proportional to b because practical RRR [39] runs the bit-wise rank operation in $O(b)$ time.

9. Compressing FM-index for Trajectories

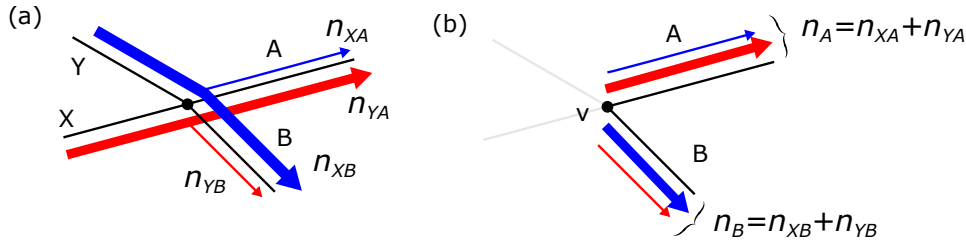


Figure 9.4.: Comparing two NCT labeling methods: (a) RML; and (b) MEL [18]

our RML conversion is as follows: $w_1 w_2 \cdots w_n \rightarrow \phi(w_1|\$)\phi(w_2|w_1) \cdots \phi(w_n|w_{n-1})$. Unlike RML, the MEL function ψ does not consider the previous symbol. Specifically, As in Figure 9.4(b), MEL labels based on the *unigram* frequencies, n_A and n_B . Conversely, our RML, shown in Fig. 9.4(a), is based on the *bigram* frequencies, n_{XA} , n_{XB} , n_{YA} , and n_{YB} .

Given these differences, the advantage of RML can be intuitively explained as follows. Real trajectories tend to go straight rather than turn left or right, as shown in Figure 9.4 (a). Because RML considers the previous road segment, it can take account the direction of the movement, whereas such information is lost in MEL. This implies that RML can capture a higher-order correlation compared to MEL. Although MEL also has the optimality of entropy, it cannot be better than RML. The experimental comparison is shown in Section 10.2.3. Mathematically, we have the following theorem.

Theorem 9.5 *For any trajectory string T , RML achieves a smaller 0th order empirical entropy than MEL does.*

PROOF Considering the size of the feasible labeling space, we find that our labeling space $\{\phi(w|w')\}$ is a superset of that of MEL, $\{\psi(w)\}$. In other words, MEL can be emulated by an RML $\bar{\phi}$ that might not be the optimal ϕ^* . Therefore, the optimality of RML (Theorem 9.2) leads to the conclusion.

9.5. Proofs

9.5.1. Proof of Theorem 9.2

To begin with, let us introduce some mathematical notations. Let us denote a set of integers as $[\sigma] := \{1, \dots, \sigma\}$. Consider σ discrete probability distributions p_1, \dots, p_σ on $[\sigma]$ defined by

$$p_{w'}(w) = \frac{n_{ww'}}{n_{\cdot w'}}, \quad (9.9)$$

where $n_{ww'}$ is the number of bigrams ww' in T and $n_{\cdot w'} = \sum_w n_{ww'}$. First, we define a permutation of a distribution.

Definition 9.2 *Let p be a discrete distribution on $[\sigma]$. A permuted distribution p^π is a distribution where $p^\pi(k) = p(\pi(k))$. Here π is a permutation on $[\sigma]$.*

In addition, we introduce the concept of a *decreasing distribution*:

Definition 9.3 *A discrete distribution p is decreasing iff $p(w) \geq p(w+1)$ for $\forall w \in [\sigma]$. Let \mathcal{F} be a set of decreasing distributions and \mathcal{F}^c be a set of non-decreasing distributions.*

Note that we can always find a permutation π that makes any distribution p decreasing, that is, $\exists \pi$ such that $p^\pi \in \mathcal{F}$.

Let us relate the above definitions to our problem. Since any possible RML corresponds to an assignment of distinct integers $c_{ww'} \in [\sigma]$, we can regard it as an array of permutations $\Pi = [\pi_1, \dots, \pi_\sigma]$. We denote such a labeling function as ϕ^Π . Our strategy, sorting by bigram $n_{ww'}$, corresponds to an array of permutations Π such that each π_i makes the distribution p_i decreasing. Note that, if $w \notin N_{out}(w')$, we can treat such cases as $p_{w'}(w) = 0$.

Our problem is to find a labeling function ϕ^Π that achieves the minimum $H_0(\phi^\Pi(T_{bwt}))$. Consider a mixture distribution

$$p^\Pi = \sum_{i \in [\sigma]} \alpha_i p_i^{\pi_i} \quad (9.10)$$

where $\alpha_i = n_{\cdot i} / \sum_j n_{\cdot j}$. Since the entropy of a discrete distribution is defined as

$$H(p) = - \sum_{k \in [\sigma]} p(k) \lg p(k), \quad (9.11)$$

the following equality holds:

$$H(p^\Pi) = H_0(\phi^\Pi(T_{bwt})). \quad (9.12)$$

Therefore, we can reformulate our optimization problem as follows.

$$\Pi^* = \operatorname{argmin}_{\Pi} H(p^\Pi). \quad (9.13)$$

Consider an optimal Π^* and any permutation π . Permutating elements in Π^* by π also yields another optimal solution by definition: $H(p^{\Pi^*}) = H(p^{\pi \circ \Pi^*})$ where $\pi \circ \Pi^* = \{\pi \circ \pi_1, \dots, \pi \circ \pi_n\}$. Here $g \circ f$ indicates a composite function. We can therefore assume p^{Π^*} is a decreasing distribution without loss of generality.

9. Compressing FM-index for Trajectories

We now prove the following theorem that directly leads to Theorem 9.2.

Theorem 9.6 *The optimal solution Π^* consists of permutations such that each $\pi_i \in \Pi^*$ makes the distribution p_i decreasing: $p_i^{\pi_i} \in \mathcal{F}$ for $\forall i \in [\sigma]$.*

We first consider the following Lemma which implies that a more concentrated distribution has smaller entropy.

Lemma 9.1 *If $a > b \geq 0$ and $\varepsilon > 0$, we have*

$$-a \lg a - (b + \varepsilon) \lg(b + \varepsilon) + (a + \varepsilon) \lg(a + \varepsilon) + b \lg b > 0. \quad (9.14)$$

PROOF Since $g(x) = (x + \varepsilon) \lg(x + \varepsilon) - x \lg x$ is a strictly increasing function, we have $g(a) - g(b) > 0$, which is equivalent to Eq. 9.14.

Now we are ready to prove Theorem 9.6.

PROOF (PROOF OF THEOREM 9.6) We prove optimality by contradiction. Let Π^+ be a set of permutations that minimizes H . As discussed above, we can assume $p^{\Pi^+} \in \mathcal{F}$ without loss of generality. Let us assume that there exists at least one $\pi_i \in \Pi^+$ such that $p_i^{\pi_i} \in \mathcal{F}^c$. Let us define $q := p_i^{\pi_i}$.

Since $q \in \mathcal{F}^c$, there exists $k \in [\sigma]$ such that $q(k) < q(k + 1)$. Based on Eq. 9.10, p^{Π^+} can be decomposed as

$$p^{\Pi^+} = (1 - \alpha_i) \hat{p} + \alpha_i q \quad (9.15)$$

where $\hat{p} := \frac{1}{1 - \alpha_i} \sum_{j \neq i} \alpha_j p_j^{\pi_j}$. If $\hat{p}(k) \leq \hat{p}(k + 1)$, we have $p^{\Pi^+}(k) < p^{\Pi^+}(k + 1)$, which contradicts the assumption $p^{\Pi^+} \in \mathcal{F}$. Therefore, we have $\hat{p}(k) > \hat{p}(k + 1)$.

Consider a permutation s_k that swaps only k and $k + 1$, and the corresponding permuted distribution q^{s_k} . Let us define $\alpha := \alpha_i$ and $\beta := 1 - \alpha_i$. We can calculate the difference of entropy functions between the optimal solution $p^{\Pi^+} = \beta \hat{p} + \alpha q$ and the swapped distribution $\beta \hat{p} + \alpha q^{s_k}$:

$$\begin{aligned} H(p^{\Pi^+}) - H(\beta \hat{p} + \alpha q^{s_k}) &= -\{\beta \hat{p}(k) + \alpha q(k)\} \lg\{\beta \hat{p}(k) + \alpha q(k)\} \\ &\quad -\{\beta \hat{p}(k + 1) + \alpha q(k + 1)\} \lg\{\beta \hat{p}(k + 1) + \alpha q(k + 1)\} \\ &\quad +\{\beta \hat{p}(k) + \alpha q(k + 1)\} \lg\{\beta \hat{p}(k) + \alpha q(k + 1)\} \\ &\quad +\{\beta \hat{p}(k + 1) + \alpha q(k)\} \lg\{\beta \hat{p}(k + 1) + \alpha q(k)\}. \end{aligned} \quad (9.16)$$

Using the notation $a = \beta\hat{p}(k) + \alpha q(k)$, $b = \beta\hat{p}(k+1) + \alpha q(k)$, and $\varepsilon = \alpha q(k+1) - \alpha q(k)$, we have $a > b \geq 0$ and $\varepsilon > 0$. Now Eq. 9.16 can be rewritten as

$$H(p^{\Pi^+}) - H(\beta\hat{p} + \alpha q^{s_k}) = -a \lg a - (b + \varepsilon) \lg(b + \varepsilon) + (a + \varepsilon) \lg(a + \varepsilon) + b \lg b > 0$$

where the last inequality holds from Lemma 9.1. This inequality indicates that $H(p^{\Pi^+}) > H(\beta\hat{p} + \alpha q^{s_k})$. Therefore,

$$[\pi_1, \dots, \pi_{i-1}, s_k \circ \pi_i, \pi_{i+1}, \dots, \pi_\sigma]$$

is better than Π^+ . However, this contradicts the optimality of Π^+ .

9.6. Proof of Theorem 9.3

We can prove Theorem 9.3 in a similar way to [32], which proves the theorem for ICB with a balanced wavelet tree.

PROOF To begin with, we introduce some facts about RRR [39]. Let us consider a bit vector B of length n . The RRR divides B into small blocks of length b : $B = B^{(1)}B^{(2)} \dots B^{(n/b)}$. Each $B^{(j)}$ is represented by its *class* c_j and *offset* o_j . Here the class is the number of 1's in $B^{(j)}$, and the offset is an index to distinguish the positions of 1's in $B^{(j)}$. In fact, the total space needed for the classes becomes the second term of Eq. (9.7) (see [39]). Since this term is already considered in Eq.(9.8), what we have to evaluate is the offsets. Each offset requires $\lg \binom{b}{c_j}$ bits because there are $\binom{b}{c_j}$ possible layouts of 1's for the class c_j .

Let us consider the partition of contexts of length k : $\phi(T_{bwt}) = L_1 L_2 \dots L_l$ ($l \leq \sigma^k$). Since a bit vector in a node v of a wavelet tree keeps the ordering, the bit vector B_v can be divided into l blocks: $B_v = B_1^v B_2^v \dots B_l^v$. Here each B_i^v corresponds to L_i .

Now, we can consider small blocks of RRR which is fully included in B_i^v . Let us denote such blocks as $B^{(1)}B^{(2)} \dots B^{(t)}$. The offsets for these blocks require

$$\sum_{j=1}^t \lg \binom{b}{c_j} \leq |B_i^v| H_0(B_i^v) \quad (9.17)$$

bits. There are at most two blocks at the boundary of B_i^v not considered above. Their offset requires $O(b)$ bits; therefore, the offsets for B_i^v need $|B_i^v| H_0(B_i^v) + O(b)$ bits in total.

Let us consider the space needed for L_i . Since there are at most $\sigma - 1$ inner nodes in

9. Compressing FM-index for Trajectories

a wavelet tree, summing the required spaces over v , we obtain

$$\sum_v |B_i^v| H_0(B_i^v) + O(\sigma b) = |L_i| H_0(L_i) + O(\sigma b) \quad (9.18)$$

bits; the RHS can be obtained by the recursive calculation technique discussed in [1].

Summing the above equation over i , we have

$$\sum_{i=1}^l |L_i| H_0(L_i) + O(l\sigma b). \quad (9.19)$$

Although L_i is a labeled string, the elements have a one-to-one correspondence with the non-encoded string because of the definition of the RML. Hence, $H_0(L_i)$ is equal to $H_0(T_W)$, where W is the corresponding context and T_W is defined in Eq.(8.4).

10. Experiments

10.1. Experimental setup

Implementation All methods were implemented in C++ and compiled with g++ (version 4.8.4) with the `-O3` option. We used the `sdsl-lite` library (version 2.0.1) for (in-memory) wavelet trees (<http://github.com/simongog/sdsl-lite/>). The BWT was calculated using `sais.hxx` (<http://sites.google.com/site/yuta256/sais/>). Experiments were conducted on a workstation with the following specifications: Intel Core i7-K5930 3.5GHz CPU (64-bit, 12 cores, L1 64kB×12, L2 256kB×12, L3 15MB), DDR4 32GB RAM, Ubuntu Linux 14.04.

Competitors Table 10.1 lists the competitors used in the experiment. We used five FM-index variants: uncompressed (UFMI, FM-GMR) and compressed (ICB-WM, ICB-Huff, FM-AP-HYB). The block-size parameter b had to be specified for CiNCT, ICB-Huff, and ICB-WM. Unless otherwise noted, we use $b = 63$. FM-GMR [15] and FM-AP-HYB [2] are FM-index variants that are tailored for huge σ and that support $O(\log \log \sigma)$ rank operation (faster than the $O(\log \sigma)$ of UFMI); they are available in the `sdsl-lite` library. These were the fastest (FM-GMR) and the smallest (FM-AP-HYB) methods for huge σ in a recent benchmark [14].

There are many possibilities for compressing NCTs by combining simple techniques such as run-length encoding. However, we do not consider such techniques in this study because pattern matching is not supported in sublinear time. In our prior evaluation, the Boyer-Moore method (linear time search) was at least four orders of magnitude slower than CiNCT. In this study, we thus only consider RePair [29], a standard benchmark in stringology which showed the best compression ratio in the initial evaluation, and PRESS [55], a shortest-path-based NCT compressor, and MEL [18], state-of-the-art labeling-based NCT compressor.

Measurement The search time was averaged over 500 suffix range queries of length 20 randomly sampled from data unless otherwise noted. For evaluation of the data size of the proposed method, the ET-graph was included.

Table 10.1.: Our proposed method and its competitors*

Method	Data	Description	C? [†]	Q? [‡]
CiNCT	$\phi(T_{bwt})$	HWT with RRR	✓	✓
UFMI	T_{bwt}	WM [◊] [7] with uncompressed bitmap [20]		✓
ICB-WM	T_{bwt}	WM with RRR [7]	✓	✓
ICB-Huff	T_{bwt}	HWT with RRR [32]	✓	✓
FM-GMR	T_{bwt}	FM-index for huge σ with $O(\log \log \sigma)$ rank [15]		✓
FM-AP-HYB	T_{bwt}	FM-index for huge σ with $O(\log \log \sigma)$ rank [2]	✓	✓
PRESS	T	The state-of-the-art trajectory compressor [55]	✓	
MEL	T	Min. entropy labeling [18]	✓	
Re-Pair ^{††}	T	A string compressor [29]	✓	

* For the first four methods, the type of WT is described / [◊] WM: wavelet matrix

[†] Uncompressed or compressed / [‡] Supports suffix range query or not

^{††} We used an implementation at <https://www.dcc.uchile.cl/~gnavarro/software/>

Datasets The datasets used in this study are as follows:

- **Singapore:** NCTs of taxi cabs used in [55]. This dataset contains many transitions without physical connection.
- **Singapore-2:** Preprocessed Singapore dataset such that transitions between two road segments without a physical connection are interpolated with the shortest path.
- **Roma:** GPS trajectories of taxi cabs in Rome. NCT representations were obtained by HMM map-matching [41] (<http://crawdad.org/roma/taxi/>).
- **MO-gen:** NCTs generated by the moving object generator (<http://iapg.jade-hs.de/personen/brinkhoff/generator/>).
- **Chess:** All chess game records (Blitz, 2006–2015, 1.87M games, <http://www.ficsgames.org>). First 10 moves are converted into hash values of Forsyth-Edwards notation.

Although Chess is not a vehicular dataset, it is included to show the possibility that CiNCT is applicable to targets other than NCTs. Table 10.2 lists the statistics of the datasets.

10.2. Results

10.2.1. Comparison with Various FM-indexes

Evaluation results for data size and processing time of suffix range queries are shown in Figure. 10.1. We observe that CiNCT requires *less than 2 bits per symbol* to store NCTs,

Table 10.2.: Statistics of each dataset

Dataset	$ T $	$\lg \sigma$	$H_0(T)$	$H_0(\phi)^\dagger$	$H_1(T)$	\bar{d}^\ddagger
Singapore	53M	15.5	13.8	1.8	1.5	26.8
Singapore-2	75M	15.5	14.0	1.3	1.1	4.0
Roma	12M	15.5	13.0	0.9	0.7	2.4
MO-Gen	193M	17.4	13.0	2.8	2.5	8.8
Chess	20M	18.8	10.3	2.0	1.4	1.6

$^\dagger H_0(\phi)$ means $H_0(\phi(T_{bwt}))$

$^\ddagger \bar{d}$ is the average out-degree of the ET-graph G_T .

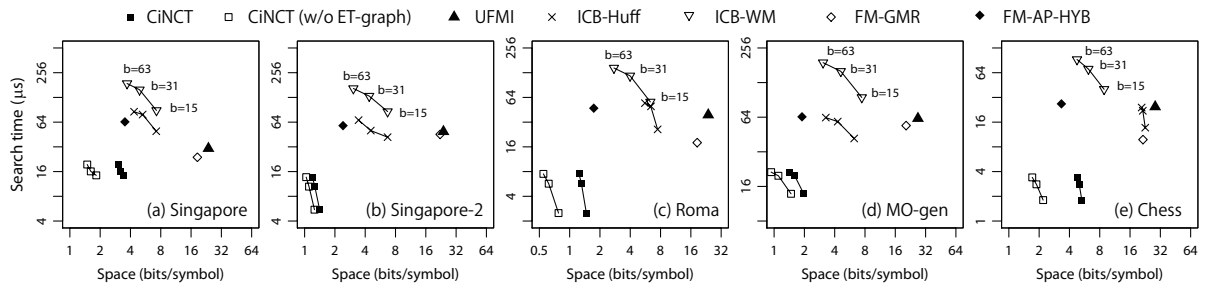


Figure 10.1.: Data size/search time (suffix range query): the proposed method shows the best performance. **CiNCT (w/o ET-graph)** is used to show the data size without the ET-graph (i.e., wavelet tree only). The block size used in the RRR bit vectors is parameterized as $b \in \{15, 31, 63\}$. Results for the other methods in Table 10.1 were omitted because their linear-time search was too slow.

and pattern matching of length 20 is processed in *a few tens of microseconds*. We also observe that *CiNCT outperforms the competitors in terms of both data size and query processing time*. We explain these results in detail below.

Data Size Compared with ICB-Huff and ICB-WM, CiNCT reduces the data size by up to 78% and 57%, respectively. As explained in Section 9.4, the space overhead decreases if $H_0(S)$ decreases. From Table 10.2 we can confirm that $H_0(\phi(T_{bwt})) \ll H_0(T_{bwt})$ holds for all datasets (note that $H_0(T) = H_0(T_{bwt})$). This explains why CiNCT shows this significant improvement. CiNCT even shows better compression than the smallest variant FM-AP-HYB, which was designed for huge σ . The improvement in **Singapore-2** is larger than that of **Singapore**. As “gapped” transitions are interpolated in **Singapore-2**, the ET-graph gets sparser ($\bar{d} = 26.8 \rightarrow 4$ in Table 10.2). This reduces the overhead regarding the ET-graph (this is confirmed through the difference of CiNCT and CiNCT (w/o ET-graph); w/o stands for without).

10. Experiments

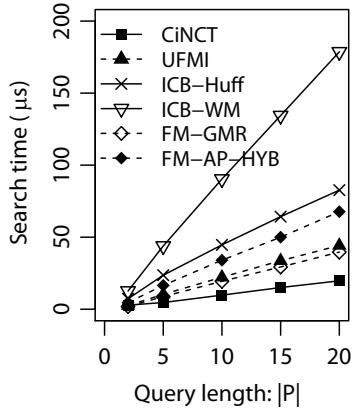


Figure 10.2.: $|P|$ vs. search time: (Singapore dataset)

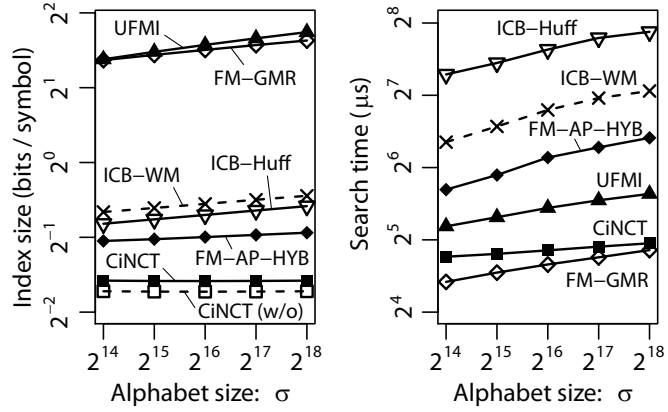


Figure 10.3.: CiNCT shows the best σ -dependence (Left: index size, right: search time / RandWalk dataset)

Processing Time of Pattern Matching Queries CiNCT is always much faster than ICB-Huff and ICB-WM; the speedups are up to 7 and 25 times, respectively. Surprisingly, CiNCT is even faster than those of the uncompressed indexes (UFMI and FM-GMR). Again, this speedup can be explained by the shallowness of the HWT of CiNCT. This decreases the number of bit-wise *rank* operations in the HWT (Section 9.4.3).

Effect of Block Size b As mentioned in Section 9.2, when b becomes larger, the results show better compression but slower search. However, as shown in Figure 10.1, *the sensitivity to the block size parameter b is very small for CiNCT*. This indicates that the proposed method is nearly *parameter-free*.

Effect of $|P|$ Figure 10.2 shows the processing time of suffix range queries against the query length $|P|$. For every method, the processing time grows linearly, because the numbers of iterations in Algorithms 1 and 14 are $O(|P|)$. We see that *CiNCT shows the slowest growth among all methods*.

10.2.2. Comparison with Several Compression Methods

Table 10.3 compares the compression ratio, defined as the uncompressed size (binary file of 32-bit integers) divided by the compressed size. We observe that CiNCT shows better compression than the existing methods. In particular, *our method is better than MEL, which showed the best compressibility in recent benchmark [18]*. This is explained as follows. First, as shown in Theorem 9.5 (and Table 10.4 in the next section), RML

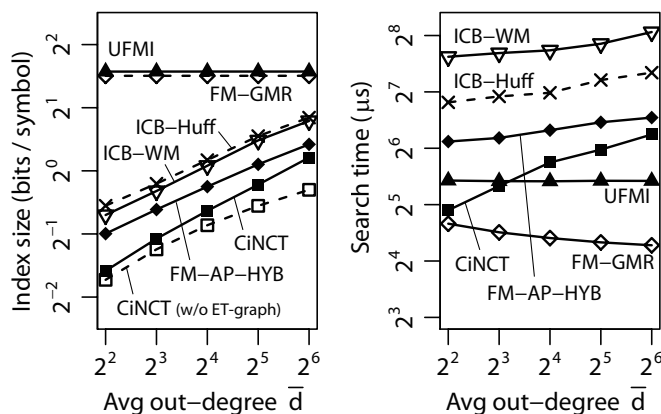


Figure 10.4.: Dependence on out-degree (Left: index size, right: search time / RandWalk dataset)

Table 10.3.: Compression ratio (larger is better)

	Singapore	Singapore-2	Roma	Mo-Gen	Chess
CiNCT	10.5	27.0	25.2	25.6	10.3
MEL [†]	N/A	15.8	21.2	N/A	N/A
Re-Pair	8.4	11.4	20.6	20.6	11.0
ICB-WM	8.8	9.4	11.3	12.0	10.5
bzip2	5.3	5.6	13.6	5.3	7.1
PRESS [‡]	4.6	N/A	N/A	N/A	N/A
zip	2.5	2.5	5.0	2.6	3.9

[†] Huffman coding was used after labeling, as in [18]. We evaluated only for ungapped datasets because MEL assumes no gap (see Singapore-2 explanation in Section 10.1).

[‡] Only the result for the Singapore dataset [55] is shown because no available implementation was found.

achieves smaller 0th order entropy than MEL (indicating a smaller average code length). Second, CiNCT is a higher order compressor (Theorem 9.3) whereas MEL is not. Note that the road network storage is not included in MEL evaluations whereas it is considered for CiNCT (as ET-graph).

10.2.3. Effect of Labeling Strategy

Comparison with MEL According to our analysis in Section 9.4.4, RML achieves lower entropy than MEL does. In Table 10.4, we show a comparison of the entropy achieved by RML and MEL for two “ungapped” NCT datasets, i.e., Singapore2 and Roma. We observe that *our RML obtained approximately 30% smaller entropy than that of MEL.*

Table 10.4.: Comparison of entropy (RML and MEL)

Dataset	RML (Proposed)	MEL [18]
Singapore2	1.26	1.93
Roma	0.76	0.99

Optimality In Section 9.1, we proposed a labeling strategy that assigns small integers $c_{ww'}$ sorted by the bigram counts $n_{ww'}$. The data size and search time under this strategy are expected to be better than those of any other possible labeling strategy, because we showed the optimality of our strategy (Theorem 9.2). Here, we compare our strategy with the *random sorting strategy*, which assigns randomly shuffled small integers $c_{ww'} \in \{1, \dots, |N_{out}(w')|\}$. Figure 10.5 shows the comparison for the five datasets ($b \in \{15, 31, 63\}$). We observe that *the index size and the search time of the bigram sorting strategy are always better than those of random sorting strategy*. Compared to the random strategy, it reduces the data size by up to 32%, and the search time by up to 57%. These results indicate the importance of the bigram sorting strategy.

10.2.4. Effect of ET-graph size/shape

Effect of Alphabet Size σ In Theorem 9.4, we showed that the search time of CiNCT does not depend on the size σ of the road map. Here, we investigate the effect of σ using synthetic RandWalk dataset: random walks on a directed random graph. The average out-degree \bar{d} of the graphs is fixed at four, and $|T|$ is set to 800σ . In Figure 10.3, *CiNCT shows good scalability against σ , whereas the index sizes and the search times of the existing methods both increase*. The search time of CiNCT is almost constant, as predicted by Theorem 9.4. The other methods do not show this property. For example, both the index size and the search time of UFMI at $\sigma = 2^{18}$ are 30% larger compared to the $\sigma = 2^{14}$ case.

Effect of Sparsity Here, we investigate the effect of the average out-degree \bar{d} . Figure 10.4 shows the results for the RandWalk dataset used in Section 10.2.4. For comparison, we fixed $\sigma = 2^{16}$ and $|T| = 100M$, and changed \bar{d} between 2^2 and 2^7 . We observe that *the sparsity of the ET-graph is the key factor for CiNCT*. Although the compression performance of CiNCT is the best, the data size grows quickly. This is due to two factors: the increase of ET-graph size and the increase of the depth of HWT. However, this result shows that our method works for larger \bar{d} than in the road network case, $\bar{d} \simeq 2^2$. This result opens the door to applications to datasets not mentioned in this thesis (e.g., symbol-valued time series).

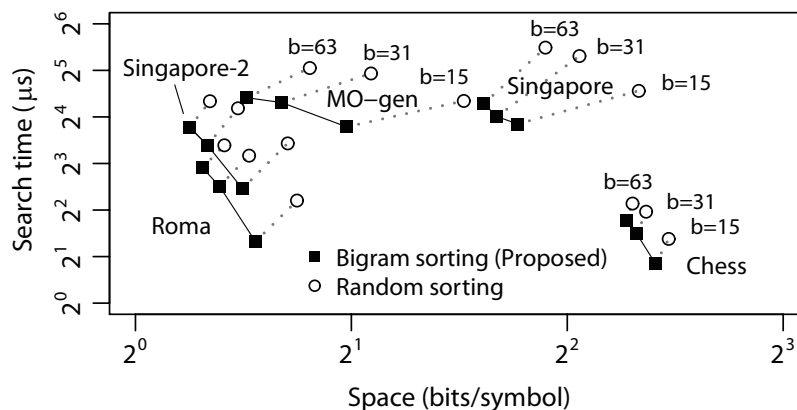


Figure 10.5.: Comparison of labeling strategies

10.2.5. Sub-path Extraction Time

Here, we evaluate *extract* queries described in Section 9.3.3. We evaluated the extraction time for obtaining the entire T , that is, $l = |T|$ and $j = 0$. Figure 10.6 compares the extraction times for the four datasets. We observe that *CiNCT shows the fastest extraction among the competitors* (twice as fast as UFMI). Again, this can be explained by the fast *rank* calculation in *CiNCT* (*PseudoRank*), as discussed above. Note that we omitted the results for FM-AP-HYB because random access to T_{bwt} was not supported in the *sdsl-lite* library.

10.2.6. Index Construction Time

Figure 10.7 compares the index construction times of FM-indexes. *The construction time of CiNCT is comparable to that of ICB-Huff, and shorter than those of the other methods.* ET-graph-build in Figure 10.7 includes all operations that are not needed for the other methods. Here, we can see the overhead for the construction of the ET-graph is not a serious problem. Note that all additional operations, including the construction of G_T from T , obtaining RML function ϕ , labeling T_{bwt} , and calculation of $Z_{w'w}$, can be executed in linear time $O(|T|)$, which implies the scalability of construction.

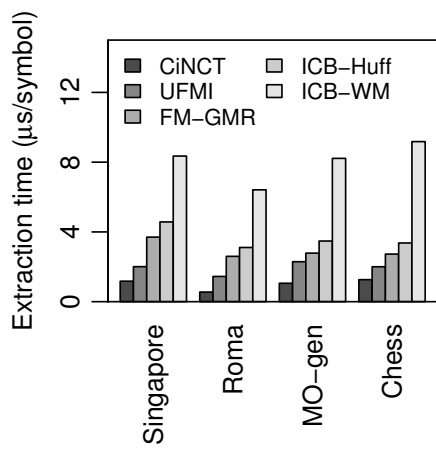


Figure 10.6.: Extraction time

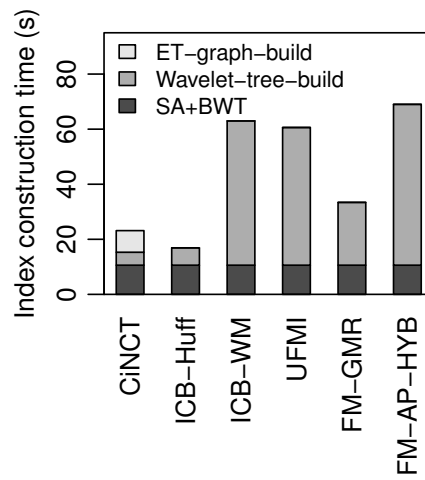


Figure 10.7.: Index construction time (Singapore dataset)

11. Related Work

11.1. Trajectory Compression

As noted in Section 8.1, shortest-path compression has been used to compress spatial paths in several papers [18, 24, 26, 55]. Although an NCT dataset is expected to have a small k -th order entropy (Eq. (8.4)), none of such shortest-path-based compressors have provided an information-theoretic evaluation of the compressed size. As an NCT compressor, our method first focuses on high-order entropy and gives an information-theoretic bound (Theorem 9.3). One of the methods proposed in [18], MEL, is a different type of spatial path compressor that achieves better compression than shortest-path-based methods. As shown in Section 9.4.4, the RML theoretically achieves a smaller entropy than MEL does. In [52], graph partitioning was used to reduce the size of spatial paths.

Imagine the following extreme case: trajectories always travel the shortest path between their origin and destination. In this case, shortest-path compressors can achieve high compression ratio (i.e., we need to record only the origin and destination), which might be better than any entropic compressor; however, as shown in our experiment (and also shown in [18]), entropy-based compression methods usually achieve better compression than shortest-path-based compressors. This fact implies that the assumption of shortest-path compressors (i.e., trajectories prefer the shortest path) is not very effective for real data.

To compress timestamps in NCTs, lossy compression methods are used in [18, 26] and [52], whereas lossless compression was used in [4]. Timestamp compression was not considered in this thesis because we assumed that timestamps are stored in disk as SNT-index does. However, existing trajectory compression methods (e.g., [4, 26]) usually store spatial/temporal information independently. This implies that our compression technique can be combined with temporal compression techniques proposed in those methods.

11.2. FM-index

FM-index, a compressed representation of suffix arrays [33], was proposed by Ferragina and Manzini [11]. We have already described FM-index and the related topics in Chapter 2, as well as in Section 8.2. Although there are a number of FM-index variants (e.g., [2, 15, 22, 32]), these are essentially designed for general strings. In Chapter 10, we compared our method also with FM-indexes designed for a large alphabet [2, 15]. For large σ , these methods can process suffix range queries in $O(|P| \log \log \sigma)$ time, which is much faster than typical $O(|P| \log \sigma)$ time. Importantly, we employed the domain-specific knowledge of the target data (i.e., sparse transition in road networks) to enhance the compression and query processing. This point is the largest difference from the FM-index family designed for general strings.

Recently, the *Wheeler graph* was defined to provide a unified view on BWT-related methods [13]. Similar to the ET-graph, this is also an edge-labeled graph. The differences are; 1) In the Wheeler graph, all edges *entering* a given node must have the *same* label, while it is not necessary in the ET-graph. 2) In the ET-graph, the edges *leaving* a given node must have *different* labels. It is not necessary in the Wheeler graph.

12. Summary

In Part III, we proposed CiNCT, a novel compressed data structure for NCTs. We incorporated the sparsity of road networks into the FM-index by using our proposed RML and *PseudoRank* techniques. The resulting data structure supports pattern matching (i.e., suffix range queries) and sub-path extraction from an arbitrary position while still achieving high compressibility.

The data size of our method is much smaller than the existing methods that are not tailored to trajectories. This means that, if we use CiNCT in SNT-index, we can significantly reduce the memory footprint. Our results also indicated that processing times of pattern matching queries and substring extraction queries are also much faster than the existing methods. As discussed in the experiment section in Part II (Chapter 5), the ratio of processing time of FM-index-related queries is not significant in SNT-index. Hence, the combination CiNCT + SNT-index will not improve the query processing time very much compared with the vanilla SNT-index (i.e., SNT-index with the standard FM-index). However, the proposed compression technique can be independently applied to other existing in-memory index methods, such as [4, 26]. We expect that such combinations will improve the query processing for applications beyond those considered in this thesis.

Part IV.
Conclusion

13. Conclusion and Future Work

In this thesis, we studied indexing and querying schemes for trajectories in road networks. We focused on data structures and algorithms in string processing and adopted them to spatio-temporal domain. In particular, we employed FM-index, which is a compact in-memory data structure for document retrieval.

In Part II, we presented how to integrate spatial information stored in FM-index with temporal information. To this end, we proposed to employ *inverse suffix arrays*. Based on the proposed indexing methods, we showed several path-based queries can be answered efficiently.

In Part III, we proposed a new compression method for FM-index storing trajectories. Our finding was based on the fact that possible edge-edge transitions are restricted due to the physical structure of road network. We proposed *relative movement labeling* to incorporate this idea into FM-index. Our theoretical and experimental analysis showed that our method outperforms existing FM-index and trajectory compression methods.

As a conclusion, we discuss possible future directions. First, supporting more types of queries is an important future work. An important extension is *similarity search*, which essentially allows the erroneous data in the database. We mainly treated *exact pattern matching* in this thesis; however, trajectory data (i.e., sequences of road segments) would include error due to map-matching algorithms. Similarity search would extend the utility of the trajectory database.

Another interesting direction is development of dynamic *SNT-index*, which allows us not only to append but also to delete trajectories. To this end, we need to seek for the possibility of data string data structures and algorithms other than FM-indexes. We believe that there are unknown links between string algorithms and trajectory processing.

Finally, we believe that application fields of the methods developed in this thesis is not limited to trajectory processing. As an example, we presented the results for Chess dataset in Chapter 10. The idea of linking the sequence data and other types of data (i.e., corresponding to road segments and timestamps in the trajectory case), or the idea of relative movement labeling for compression, would be widely applicable to other types

13. Conclusion and Future Work

of data. Seeking for new applications of the proposed methods is also an important and interesting research direction.

Bibliography

- [1] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. ISSN 0304-3975. doi: 10.1016/j.tcs.2013.10.019.
- [2] Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet Partitioning for Compressed Rank/Select and Applications. In *Proceedings of 21st International Symposium on Algorithms and Computation*, ISAAC’10, pages 315–326, 2010. doi: 10.1007/978-3-642-17514-5_27.
- [3] Lorenzo Bracciale, Marco Bonola, Pierpaolo Loreti, Giuseppe Bianchi, Raul Amici, and Antonello Rabuffi. CRAWDAD dataset roma/taxi (v. 2014-07-17). Downloaded from <https://crawdad.org/roma/taxi/20140717>, July 2014.
- [4] Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, and M. Andrea Rodríguez. Compact Trip Representation over Networks. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval*, SPIRE’16, pages 240–253, 2016. doi: 10.1007/978-3-319-46049-9_23.
- [5] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. In *Technical Report 124*. Digital Equipment Corporation, 1994.
- [6] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. Discovering Popular Routes from Trajectories. In *Proceedings of the 27th International Conference on Data Engineering*, ICDE ’11, pages 900–911, 2011. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767890.
- [7] Francisco Claude and Gonzalo Navarro. The wavelet matrix. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval*, SPIRE’12, pages 167–179, 2012. doi: 10.1007/978-3-642-34109-0_18.
- [8] Jian Dai, Bin Yang, and Christian S Jensen. Path Cost Distribution Estimation Using Trajectory Data. *Proc. VLDB Endow.*, 10(3):85–96, November 2016. ISSN 21508097. doi: 10.14778/3021924.3021926.

- [9] Victor Teixeira de Almeida and Ralf H. Güting. Indexing the Trajectories of Moving Objects in Networks. *Geoinformatica*, 9(1):33–60, 2005. ISSN 1384-6175. doi: 10.1007/s10707-004-5621-7.
- [10] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, pages 117–139. Springer, 2009. ISBN 978-3-642-02093-3. doi: 10.1007/978-3-642-02094-0_7.
- [11] Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS’00, pages 390–398, 2000. ISBN 0-7695-0850-2. doi: 10.1109/SFCS.2000.892127.
- [12] Elias Frentzos. Indexing Objects Moving on Fixed Networks. In *Proceedings of the 8th International Symposium on Spatial and Temporal Databases*, SSTD’03, pages 289–305, 2003. doi: 10.1007/978-3-540-45072-6_17.
- [13] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler Graphs: A Framework for BWT-based Data Structures. *Theoretical Computer Science*, 698:67–78, 2017. ISSN 03043975. doi: 10.1016/j.tcs.2017.06.016.
- [14] Simon Gog, Alistair Moffat, and Matthias Petri. CSA++: Fast Pattern Search for Large Alphabets. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments*, ALENEX’17, pages 73–82, 2017. doi: 10.1137/1.9781611974768.6.
- [15] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithm*, SODA’06, pages 368–373, 2006. ISBN 0-89871-605-5. doi: 10.1145/1109557.1109599.
- [16] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order Entropy-compressed Text Indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, SODA’03, pages 841–850, 2003. ISBN 0-89871-538-5.
- [17] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD ’84, pages 47–57, 1984. ISBN 0-89791-128-8. doi: 10.1145/602259.602266.
- [18] Yunheng Han, Weiwei Sun, and Baihua Zheng. COMPRESS: A Comprehensive Framework of Trajectory Compression in Road Networks. *ACM Trans. Database Syst.*, 42(2):11:1–11:49, May 2017. ISSN 0362-5915. doi: 10.1145/3015457.

- [19] Abdeltawab M. Hendawi, Jie Bao, Mohamed F. Mokbel, and Mohamed Ali. Predictive Tree: An Efficient Index for Predictive Queries on Road Networks. In *Proceedings of the 31st International Conference on Data Engineering, ICDE '15*, pages 1215–1226, 2015. ISBN 9781479979639. doi: 10.1109/ICDE.2015.7113369.
- [20] Guy Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, SFCS '89*, pages 549–554, 1989. ISBN 0-8186-1982-1. doi: 10.1109/SFCS.1989.63533. URL <https://doi.org/10.1109/SFCS.1989.63533>.
- [21] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, and Christian S. Jensen. Path Prediction and Predictive Range Querying in Road Network Databases. *VLDB Journal*, 19(4):585–602, 2010. ISSN 10668888. doi: 10.1007/s00778-010-0181-y.
- [22] Juha Kärkkäinen and Simon J. Puglisi. Fixed Block Compression Boosting in FM-Indexes. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval, SPIRE'11*, pages 174–184, 2011. ISBN 978-3-642-24583-1. doi: 10.1007/978-3-642-24583-1_18.
- [23] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987. ISSN 0018-8646. doi: 10.1147/rd.312.0249.
- [24] G. Kellaris, N. Pelekis, and Y. Theodoridis. Map-matched Trajectory Compression. *J. Syst. Softw.*, 86(6):1566–1579, 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2013.01.071.
- [25] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. Path-based Queries on Trajectory Data. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS'14*, pages 341–350, 2014. ISBN 978-1-4503-3131-9. doi: 10.1145/2666310.2666413.
- [26] Benjamin Krogh, Christian S. Jensen, and Kristian Torp. Efficient In-memory Indexing of Network-constrained Trajectories. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '16*, 2016. ISBN 978-1-4503-4589-7. doi: 10.1145/2996913.2996972.
- [27] John Krumm. A Markov Model for Driver Turn Prediction. In *Society of Automotive Engineers (SAE) 2008 World Congress*, 2008.

- [28] John Krumm and Eric Horvitz. Predestination: Inferring Destinations from Partial Trajectories. In *Proceedings of the 8th International Conference on Ubiquitous Computing*, UbiComp'06, pages 243–260, 2006. ISBN 978-3-540-39634-5. doi: 10.1007/11853565_15. URL http://link.springer.com/chapter/10.1007/11853565_{_}15.
- [29] N. Jesper. Larsson and Alistair Moffat. Offline Dictionary-based Compression. In *Proceedings of Data Compression Conference*, DCC'99, pages 296–305, 1999. doi: 10.1109/DCC.1999.755679.
- [30] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. Map-matching for Low-sampling-rate GPS Trajectories. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 352–361, 2009. ISBN 978-1-60558-649-6. doi: 10.1145/1653771.1653820.
- [31] Wuman Luo, Haoyu Tan, Lei Chen, and Lionel M. Ni. Finding Time Period-based Most Frequent Path in Big Trajectory Data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 713–724, 2013. doi: 10.1145/2463676.2465287.
- [32] Veli Mäkinen and Gonzalo Navarro. Implicit Compression Boosting with Applications to Self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval*, SPIRE'07, pages 229–241, 2007. doi: 10.1007/978-3-540-75530-2_21.
- [33] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990. doi: 10.1137/0222058.
- [34] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 9780521865715. doi: 10.1017/CBO9780511809071.
- [35] Giovanni Manzini. An Analysis of the Burrows-Wheeler Transform. *J. ACM*, 48(3):407–430, 2001. doi: 10.1145/382780.382782.
- [36] Mohamed F. Mokbel, Thanaa M Ghanem, and Walid G Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.
- [37] Mikołaj Morzy. Mining Frequent Trajectories of Moving Objects for Location Prediction. In *Proceedings of the 5th International Conference on Machine Learning*

- and Data Mining in Pattern Recognition*, MLDM '07, pages 667–680, 2007. ISBN 978-3-540-73498-7. doi: 10.1007/978-3-540-73499-4_50.
- [38] Gonzalo Navarro. Wavelet Trees for All. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, CPM'12, pages 2–26, 2012. doi: 10.1007/978-3-642-31265-6_2.
- [39] Gonzalo Navarro and Eliana Provedel. Fast, small, simple rank / select on bitmaps. In *Proc. SEA '12*, pages 295–306, 2012. doi: 10.1007/978-3-642-30850-5_26.
- [40] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016. ISBN 1107152380.
- [41] Paul Newson and John Krumm. Hidden Markov Map Matching Through Noise and Sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 336–343, 2009. ISBN 978-1-60558-649-6. doi: 10.1145/1653771.1653818.
- [42] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-Temporal Access Methods : Part 2 (2003 – 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, 2010.
- [43] Ge Nong, Sen Zhang, and Wai Hong Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. Comput.*, 60(10):1471–1484, 2011. doi: 10.1109/TC.2010.188.
- [44] Salvatore Orland, Renzo Orsini, Alessandra Raffaetà, Alessandro Roncato, and Claudio Silvestri. Trajectory Data Warehouses: Design and Implementation Issues. *Journal of computing science and engineering*, 1(2):211–232, 2007.
- [45] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 215–224, 2001. ISBN 0-7695-1001-9. doi: 10.1109/ICDE.2001.914830.
- [46] Dieter Pfoser and Christian S. Jensen. Indexing of network constrained moving objects. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, GIS '03, pages 25–32, 2003. ISBN 1581137303. doi: 10.1145/956676.956680.
- [47] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proceedings of the 26th*

Bibliography

- International Conference on Very Large Data Bases*, VLDB '00, pages 395–406, 2000. ISBN 1-55860-715-3.
- [48] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. PARINET : A Tunable Access Method for In-Network Trajectories. In *Proceedings of the 26th International Conference on Data Engineering*, ICDE'10, pages 177–188, 2010. ISBN 9781424454440. doi: 10.1109/ICDE.2010.5447885.
- [49] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees and Multisets. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, pages 233–242, 2002. ISBN 0-89871-513-X.
- [50] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2000. ISSN 01635808. doi: 10.1145/342009.335427.
- [51] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. Indexing In-network Trajectory Flows. *The VLDB Journal*, 20(5):643–669, October 2011. ISSN 1066-8888. doi: 10.1007/s00778-011-0236-8.
- [52] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, and Ahmed Kharrat. Spatio-Temporal Compression of Trajectories in Road Networks. *GeoInformatica*, 19(1): 117–145, 2014. ISSN 13846175. doi: 10.1007/s10707-014-0208-4.
- [53] Yasin N. Silva, Xiaopeng Xiong, and Walid G. Aref. The RUM-tree: Supporting frequent updates in R-trees using memos. *The VLDB Journal*, 18(3):719–738, 2009. ISSN 10668888. doi: 10.1007/s00778-008-0120-3.
- [54] Reid Simmons, Brett Browning, Yilu Zhang Yilu Zhang, and Varsha Sadekar. Learning to Predict Driver Route and Destination Intent. In *Proceedings of Intelligent Transportation Systems Conference*, ITSC'06, pages 127–132, 2006. ISBN 1-4244-0093-7. doi: 10.1109/ITSC.2006.1706730.
- [55] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. PRESS: A Novel Framework of Trajectory Compression in Road Networks. *Proc. VLDB Endow.*, 7(9): 661–672, 2014.
- [56] S. Taguchi, S. Koide, and T. Yoshimura. Online Map Matching with Route Prediction. *IEEE Transactions on Intelligent Transportation Systems*, 20(1):338–347, Jan 2019. ISSN 1558-0016. doi: 10.1109/TITS.2018.2812147.

- [57] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-Tree : An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proceedings of the 29th International Conference on Very Large Data Bases, VLDB'03*, pages 790–801, 2003. ISBN 0127224424.
- [58] Dai Hai Ton That, Iulian Sandu Popa, and Karine Zeitouni. TRIFL: A Generic Trajectory Index for Flash Storage. *ACM Trans. Spatial Algorithms Syst.*, 1(2): 6:1–6:44, 2015. ISSN 2374-0353. doi: 10.1145/2786758.
- [59] Marcos R. Vieira, Enrique Frías-Martínez, Petko Bakalov, Vanessa Frías-Martínez, and Vassilis J. Tsotras. Querying Spatio-temporal Patterns in Mobile Phone-Call Databases. In *Proceedings of the 11th International Conference on Mobile Data Management, MDM'10*, pages 239–248, 2010. doi: 10.1109/MDM.2010.24.
- [60] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. Torch: A Search Engine for Trajectory Data. In *SIGIR*, pages 535–544. ACM, 2018. ISBN 978-1-4503-5657-2. doi: 10.1145/3209978.3209989.
- [61] Andy Yuan Xue, Rui Zhang, Yu Zheng, Xing Xie, Jin Huang, and Zhenghua Xu. Destination Prediction by Sub-trajectory Synthesis and Privacy Protection Against Such Prediction. In *Proceedings of the 29th International Conference on Data Engineering, ICDE'13*, pages 254–265, 2013. ISBN 978-1-4673-4909-3. doi: 10.1109/ICDE.2013.6544830.
- [62] H. Yuan and G. Li. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *Proceedings of the 35th International Conference on Data Engineering, ICDE'19*, pages 1262–1273, 2019. doi: 10.1109/ICDE.2019.00115.
- [63] Yu Zheng and Xiaofang Zhou. *Computing with Spatial Trajectories*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 1461416280, 9781461416289.
- [64] Brian D. Ziebart, Andrew L. Maas, Anind K. Dey, and J. Andrew Bagnell. Navigate Like a Cabbie: Probabilistic Reasoning from Observed Context-aware Behavior. In *Proceedings of the 10th International Conference on Ubiquitous Computing, UbiComp'08*, 2008. ISBN 9781605581361. doi: 10.1145/1409635.1409678.

List of Publications

Journal Papers

- Satoshi Koide, Yukihiro Tadokoro, Takayoshi Yoshimura, Chuan Xiao, Yoshiharu Ishikawa, Enhanced Indexing and Querying of Trajectories in Road Networks via String Algorithms, ACM Transactions on Spatial Algorithms and Systems (TSAS), 2018, DOI: 10.1145/3200200
- 小出 智士, 肖 川, 石川 佳治, 道路ネットワーク上の軌跡データに対する圧縮索引 (Compressed Indexing for Trajectories constrained in Road Networks), 電子情報通信学会論文誌 D, 2020 (in Japanese)

International Conferences (Refereed)

- Satoshi Koide, Yukihiro Tadokoro, Takayoshi Yoshimura, SNT-index: Spatio-temporal index for vehicular trajectories on a road network based on substring matching, Proceedings of the First ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics (UrbanGIS), 2015, DOI: 10.1145/2835022.2835023
- Satoshi Koide, Yukihiro Tadokoro, Chuan Xiao, Yoshiharu Ishikawa, CiNCT: Compression and Retrieval for Massive Vehicular Trajectories via Relative Movement Labeling, IEEE International Conference on Data Engineering (ICDE), 2018, DOI: 10.1109/ICDE.2018.00102

Conferences (Non-refereed)

Presentations at non-refereed domestic/international workshops (only presentations being closely related are shown.)

- 小出 智士, 田所 幸浩, 吉村 貴克, 文字列索引によるネットワーク制約下の車両軌跡の索引化, データ工学と情報マネジメントに関するフォーラム (DEIM'16), 2016

Bibliography

- Satoshi Koide., An Application of Full-text Search to Spatio-temporal Trajectory Mining, Workshop on Compression, Text, and Algorithms, held conjunction with SPIRE'16, 2016
- 小出 智士, 吉村 貴克, 肖 川, 石川 佳治, ネットワーク上の軌跡データに対する時間制約付き二点間経路の列挙, データ工学と情報マネジメントに関するフォーラム (DEIM'18), 2018 (最優秀論文賞)
- 小出 智士, 肖 川, 石川 佳治, 道路ネットワークのスパース性に着目した車両軌跡の圧縮索引, データ工学と情報マネジメントに関するフォーラム (DEIM'19), 2019 (優秀論文賞)

Other Publications

The following publications are refereed papers by the author that are not closely related to the present thesis (only papers as the first author are shown).

- Satoshi Koide, Daisuke Furihata, Nonlinear and linear conservative finite difference schemes for regularized long wave equation, Japan journal of industrial and applied mathematics 26 (1), 15, 2009, DOI: 10.1007/BF03167544
- Satoshi Koide, Keisuke Kawano, Takuro Kutsuna, Neural Edit Operations for Biological Sequences, Proceedings of Neural Information Processing Systems, 2018