สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

อัลกอริทึมการเปรียบคู่โดยใช้รายการผกผัน

MATCHING ALGORITHMS USING INVERTED-LISTS

T137186

เชาวลิต ขันคำ

CHOUVALIT KHANCOME

วพ
ส741ค
2555

เลขหมู่.....
เลขทะเบียน.....137186
วัน,เดือน,ปี.....2 ส.ค. 2558

b.....1246112X
i.....

วิทยานิพนธ์นี้สำหรับการศึกษาตามหลักสูตรปริญญาปรัชญาดุษฎีบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2555

KMITL – 2012 – SC – D – 002 - 040

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

MATCHING ALGORITHMS USING INVERTED-LISTS

CHOUVALIT KHANCOME

A THESIS SUBMITTED IN FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2012

KMITL – 2012 – SC – D – 002 - 040

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

COPYRIGHT 2012

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

หัวข้อวิทยานิพนธ์	อัลกอริทึมการเปรียบคู่โดยใช้รายการผกผัน
นักศึกษา	นายเชาวลิต ขันคำ
รหัสประจำตัว	49062905
ปริญญา	ปรัชญาดุษฎีบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2555
อาจารย์ที่ปรึกษาวิทยานิพนธ์	รศ.ดร.วีระ บุญจริง

บทคัดย่อ

วิทยานิพนธ์นี้มีวัตถุประสงค์เพื่อพัฒนางานใหม่ของสองหลักการที่สำคัญทางด้านวิทยาการคอมพิวเตอร์ วัตถุประสงค์แรก คือ การพัฒนาโครงสร้างข้อมูลใหม่ที่มีประสิทธิภาพเรียกว่า *รายการผกผัน* ด้วยการรวมหลักการของดัชนีผกผันและตารางการแฮชอย่างสมบูรณ์ วัตถุประสงค์ที่สอง คือ การพัฒนาอัลกอริทึมการเปรียบคู่พจนานุกรมแบบพลวัต จำนวนสามอัลกอริทึมโดยอาศัยโครงสร้างข้อมูลที่พัฒนาขึ้น ผลการพัฒนาเชิงทฤษฎีพบว่าโครงสร้างที่พัฒนาขึ้นใหม่มีประสิทธิภาพทั้งในเชิงทฤษฎีและการนำไปใช้งาน โดยเฉพาะอย่างยิ่งในด้านเวลาเนื้อที่ และความยืดหยุ่น โดยมีความซับซ้อนทางด้านเวลาและเนื้อที่จัดเก็บน้อยกว่าโครงสร้างในอดีต อีกทั้งยังสนับสนุนการเพิ่มและลบอักขระแบบได้ตลอดเวลา และเมื่อนำเอาโครงสร้างนี้ไปพัฒนาเป็นอัลกอริทึมในการค้นหา สามารถพัฒนาอัลกอริทึมการค้นหาได้จำนวนสามอัลกอริทึม ได้แก่ อัลกอริทึมแบบพื้นฐานที่สุด อัลกอริทึมแบบค้นหาโดยใช้ส่วนหน้า และอัลกอริทึมแบบค้นหาโดยใช้ปัจจัยส่วนหลัง โดยที่อัลกอริทึมแบบใช้ส่วนหน้าและใช้ปัจจัยส่วนหลังมีความซับซ้อนน้อยกว่าอัลกอริทึมในอดีต ผลการทดลองพบว่า โครงสร้างนี้สร้างได้ง่าย ใช้เวลาน้อยและใช้เนื้อที่จัดเก็บน้อยกว่าโครงสร้างอื่นที่นำมาเปรียบเทียบ ผลของการค้นหาพบว่า อัลกอริทึมแบบพื้นฐานที่สุดทำงานได้ช้าที่สุด แต่มีประสิทธิภาพเมื่อค้นหาในพจนานุกรมขนาดเล็กและข้อความจำนวนน้อย อัลกอริทึมแบบค้นหาโดยใช้ส่วนหน้าค้นหาได้แบบเชิงเส้นและมีประสิทธิภาพเหมือนกับอัลกอริทึมแบบแรกแต่เร็วกว่าทุกกรณี ส่วนอัลกอริทึมแบบค้นหาโดยใช้ปัจจัยส่วนหลังมีประสิทธิภาพมากที่สุด ค้นหาได้รวดเร็วกว่าสองอัลกอริทึมแรก นอกจากนั้นยังสามารถค้นหาได้รวดเร็วใกล้เคียงกับอัลกอริทึมที่มีประสิทธิภาพที่สุดของการเปรียบคู่พจนานุกรมแบบคงที่

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Thesis	Matching Algorithms Using Inverted Lists
Student	Mr. Chouvalit Khancome
Student ID.	49062905
Degree	Doctor of Philosophy
Programme	Computer Science
Year	2012
Thesis Advisor	Assoc.Prof.Dr.Veera Boonjing

ABSTRACT

This thesis aims to discover new solutions of two main areas of computer science. The primary purpose is to create a new crucial data structure named *inverted lists* based on the combinations of an inverted index and a perfect hashing table. The secondary purpose is to develop new three algorithms of dynamic dictionary matching by means of the inverted lists. As theoretical results, the inverted lists structure comprises both theoretical results and practical implementations especially in terms of simplicity and scalability. A dictionary that employs an inverted lists structure takes time and space less than a traditional dictionary. More importantly, this structure is very flexible when the proposed dictionary is updated with an individual pattern over time. With this discovered structure, an exhaustive approach, a prefix approach, and a suffix factor approach of dynamic dictionary matching algorithms can be developed. Among them, the prefix and the suffix factor solutions took less theoretical time for scanning the given text than the traditional algorithms even in worst case scenarios. In experimental results, the inverted lists dictionary was constructed easier and faster than the traditional data structures, while the accommodated space was more economical than earlier structures. The searching results showed that the exhaustive solution worked as a naïve search, but it was efficient in the cases of small dictionary and small text sizes. The prefix approach showed a linear searching time; in addition, it was faster than the exhaustive solution. As the fastest searching results, the suffix factor solution was the most efficient when comparing with the two proposed algorithms; furthermore, it searched almost as fast as the best solution of static dictionary algorithm.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ACKNOWLEDGEMENTS

There are some people to whom I am deeply indebted for their help in this process that culminates with a PhD dissertation. I take this opportunity to acknowledge their never faltering encouragement and support.

To my advisor Assoc.Prof.Dr.Veera Boonjing for the excellent supervision and for being an example of professionalism and dedication to solve my research problems; furthermore, his extensive experience in academic research, and particularly in the areas of information retrieval and algorithms have been of extreme importance to realize this thesis. His excellent supports this process and also in my academic and professional life.

To my dear wife Wichian, for the love, listening to me during the roughest time of my studies, understanding she deserves when I could not give her the attention, companionship and encouragement during moments in which I desired to give up everything. Thank you for sharing your life with me and the victories won during the entire doctorate. To my children Nuke and New for motivating me and giving me willpower to do the thesis methods, they are encouragement and mean everything in my life. With the grace of Buddha in our lives we will continue to be very happy.

Over the years I have received financial support from Rajabhat Rajanagarindra University. What is left of my sanity, I owe to the many friends I have made during my years at KMITL. I am also grateful to the Computer Science Department of Rajabhat Rajanagarindra University for providing the facilities and the community that have formed the major part of my life for the past several years.

The most important, to my dear parents; my father, my mother, my sister Nang, and my brothers Wit and Kid, their sacrifices made have given support for this achievement and provided love to complete my work.

Besides, I feel lucky for the good friends I have: Joe and Calan, and I would also like to give special thanks to them for their great help to read and to edit my papers and my thesis during these years.

Chouvalit Khancome.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

TABLE OF CONTENTS

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

LIST OF TABLES

Tables Page

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

LIST OF FIGURES

Figures Page

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

CHAPTER 1

INTRODUCTION

1.1 Statement of Problems

The matching principle is the most widely important principle that deals with data structures in efficient searching algorithms. This field relies on the excellent data structures that are provided for several searching aspects. There are plenty of new applications in computer science that apply this principle to solve their problems (e.g., [64], [65], [66], [71]) including the operating system commands (Unix grep command using Commentz-Walter [9] and agrep using Wu-Manber[25]), intrusion detection systems (e.g., SNORT using Aho-Corasick[1], Commentz-Walter [9], and Wu-Manber[25]), and so on.

Traditionally, single string pattern matching, multiple string pattern matching (often called a static dictionary matching), and dynamic dictionary matching are the emerging branches from the matching principle. In particular, dictionary matching simultaneously searches for all occurrences of patterns (known as dictionary) which appear in a large given text. This is called static dictionary matching when patterns do not enable the ability to update. It is called dynamic dictionary matching when the dictionary enables the ability to insert or delete individual patterns over time.

In general, there are two standard phases to solve the dictionary matching problems, which are the preprocessing phase and the searching phase. The preprocessing phase is that the target patterns are generated to an efficient data structure, and the searching phase scans the given text to find all occurrences of patterns in the data structure. Up until now, newer algorithms to be developed should be as fast as possible to ensure pattern occurrences in a large given text; additionally, the dynamic ability of the dictionary remains. Therefore, efficient data structures and supreme searching algorithms are important in solving these problems. Basically, Trie, Bit-parallel, Hashing table, and Suffix tree are the structures employed to accommodate the patterns in the preprocessing phase.

Trie is the well known data structure that employs an automaton structure for containing the set of states which are labeled by the characters. The static dictionary applied Trie to accommodate the set of patterns found in Aho-Corasick algorithm [1], Commentz-Walter [3] and SetHorspool [9]. Other Trie-based solutions [18], [19], MultiBDM[20], SBOM[5], and SDBM

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

(shown in [9], [15], and [11]) improved on Trie in that they decreased the searching time (detailed by [9]).

Bit parallel structure employs the binary digit to represent the set of target patterns. Several static dictionary algorithms used the Bit parallel such as Multiple Shift-And [8], the Multiple-BNDM (shown in [9]), and the other one shown in [10]. In other words, the first idea of hashing was presented by Karp and Rabin [14] in single string matching. The efficient solution, which was presented by Wu and Manber [23], created the shift table and implemented the hashing table to store the block of patterns. The faster solution was presented in solution [25], which improved the Wu and Manber [23] algorithm.

A more advanced well known data structure, which can be applied to the dynamic dictionary matching, is a suffix tree. The first algorithm, which employed the suffix tree, was presented by Amir and Farach [27] working in an adaptive aspect; however, it consumed time and space. Subsequently, [28], [29], [30], [31], [32], [33], [34] and [43] used the suffix tree to create their algorithm.

On the practical side, implementing Trie to applications usually takes a large amount of memory; moreover, Trie does not directly support updating the patterns over time. Thus, existing Trie dictionaries are needed to re-generate all patterns when only one pattern in the dictionary needs to be inserted or to be deleted. Meanwhile, suffix tree supports the insertion and deletion of individual patterns over time, although it needs other structures to help in updating patterns, which requires a long time. Indeed, the most widely implemented parts of this structure always take a larger memory than Trie. Bit-parallel algorithms are restricted by the computer word and need to deal with the complexity of the bit-conversion method; furthermore, it is also very hard to update the pattern when implementing the dictionary. Besides, applying the hashing principle to algorithms is more time exhaustive and does not support the insertion or deletion of the patterns. So the main challenge in the traditional dictionary is to develop new and better technique that allows the flexibility of a dictionary and retains the fast searching properties.

From a theoretical point of view, static dictionary matching algorithms always search faster than the dynamic dictionary matching algorithms. For instance, Aho-Corasick algorithm [1], Wu and Manber [23], and Hong- Ke-Yong[25] are more efficient than Shinalp and Vishkin [45], *AFGGP* [33]. Briefly, several searching algorithms of previous design take into account common efficient and flexible structures. In particular, the performance of a faster search

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

remains; while the dynamic patterns in dictionary are needed. Therefore, it can be said that new dynamic dictionary matching algorithms are always required.

In order to overcome those disadvantages, it is natural to ask whether a new data structure can be created to facilitate faster construction and more economical space usage than the traditional data structures. Additionally, the new data structure can be applied to several fields of the matching principles. Most important, the proposed algorithms which employ this structure are more efficient than the traditional dynamic dictionary matching algorithms. Still, the searching performances are faster than or equal to the static dictionary matching algorithms.

The most known motivation for this thesis comes from an inverted index and a perfect hashing table. Since the inverted index structure has been used as the main method of information retrieval to solve the problem, it has been applied to many applications (shown in [13], [37] and [39]). Furthermore, there is an excellent data structure which always takes a constant time and uses minimal space, called a perfect hashing table. Motivated by both data structures, this thesis combines them to explore a simpler data structure for accommodating a dictionary.

A primary focus of this thesis relies on a new data structure and new dynamic dictionary matching algorithms, including all reviewed matching principles above. The beginning of this thesis follows a recent trend in data structure, especially an inverted index and hashing principle. The key point is taking advantage of the inverted index and the perfect hashing table for accommodating the dictionary. The first methodology is to adapt the inverted index to create a new data structure called *inverted lists*. Then, this new structure employs the perfect hashing table to accommodate the patterns, which works similar to a preprocessing phase. Most importantly, it proposes developing new searching algorithms for the dynamic dictionary matching. Furthermore, these algorithms search efficiently when comparing to traditional algorithms.

1.2 Research Objectives

This thesis mainly aims to study the traditional data structures: an inverted index and a hashing table on how to create a new data structure for dynamic dictionary matching. The new proposed data structure, called *inverted lists*, is based on combining an inverted index principle and a perfect hashing principle. This structure provides an extreme ability to accommodate a set of patterns, supports an individual pattern insertion, and encourages a deletion of the target pattern. With the proposed data structure, new algorithms of dictionary matching are designed;

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

moreover, these algorithms are flexible and more efficient than the traditional algorithms in both theory and practice.

1.3 Scopes of Thesis

The scopes of the study are:

1. The new proposed data structure accommodates the set of target patterns for dynamic dictionary matching algorithms. Also, this structure enables the ability to insert and to delete the individual pattern at the user's discretion.

2. Three searching algorithms of dynamic dictionary matching, which are called an exhaustive approach: E-IVL, a prefix approach: PF-IVL, and suffix factor approach: SF-IVL, are developed by means of the proposed data structure.

3. The proposed data structure is proven by the time and the space complexity including the correctness of the algorithm. Then, the proposed searching algorithms are also proven by time and the correctness. Additionally, the time and space complexity of the new structure is compared with the traditional data structures and new algorithms are compared with the previous algorithms.

4. The proposed data structure and the proposed searching algorithms are implemented. For the sake of evaluations, the experimental results compare searching time efficiency with the earlier algorithms.

1.4 Hypothesis of Study

The inverted lists structure, adapted from the inverted index and the perfect hashing table, is easy to construct, requires less space to store, and is fast to access. Moreover, it is able to be applied to design the new algorithms of dynamic dictionary matching in an efficient manner.

1.5 Research Methodology

This thesis defines two tasks: explore a new data structure and develop new algorithms. The performance evaluations of the proposed data structure and the proposed algorithms are shown by proving time complexity, space complexity, and the algorithm correctness. Then,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

experiments investigate the performances by implementing programs from the proposed algorithms and running them on a variety of datasets with different parameters.

1.6 Advantage Results

The benefits of the research are as follows:

1. Different from the traditional paradigm, a new and better data structure is established in both less space and faster time to access. The inverted lists structure is able to insert and to delete individual patterns over time. Moreover, this structure takes minimal time complexity in construction, pattern insertion, and pattern deletion.

2. With the new data structure three new algorithms, which are called exhaustive approach (E-IVL), prefix approach (PF-IVL), and suffix approach (SF-IVL), are created. In theoretical results, these algorithms are more efficient than the traditional algorithms when time complexities were compared.

3. Experimental results showed that the inverted lists structure took significantly less space and less time for creating the dictionary and updating the dictionary. Furthermore, the searching times of the proposed algorithms were faster than the traditional algorithms, especially in a small dictionary and small text size.

1.7 Organization

The following four chapters are devoted to the contributions of this thesis, including some appendices and the biography.

Chapter 2 reviews the details that are related to the dictionary principles and their related works, which are used as the original sources in this thesis. Then the traditional algorithms are described, including the inverted index and the perfect hashing principle.

Chapter 3 presents the special technique for the inverted lists creation, dynamic technique of inverted lists structure, and all new dynamic dictionary matching algorithms. Additionally, all of them are explained by the algorithm details, proofs of time and proof of space complexity. Furthermore, the correctness of all algorithms is also proven.

Chapter 4 starts by showing the results of theoretical evaluations among the time and the space complexity, as well as the theoretical performances of searching comparison. Next, implementation details and experimental setup are described. Then, the experimental results of

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

preprocessing phase and all results of searching algorithms to be implemented are illustrated by tables and figures. In the final section, the resulting discussions are shown.

Chapter 5 sums up the contributions in this thesis and highlights some recommendations for future works.

Finally, appendix A shows all published and submitted research papers and articles that are outputs of this thesis. In addition, appendix B presents the research papers and the article that are selected for the thesis fulfillment. Lastly, the author biography is shown.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

CHAPTER 2

LITERATURE REVIEWS

This thesis is motivated by the need to create a new data structure for new algorithms of dynamic dictionary matching. The basic related principles and the knowledge of dictionary matching principles are integrated with those of dictionary accommodation, inverted index and perfect hashing. In this chapter, comprehensive details of significant static and dynamic dictionary principles are illustrated; moreover, available paradigms, approaches, techniques, and best practices (i.e., the state-of-the-art) are reviewed. Then, Trie and suffix tree are detailed. Importantly, the inverted index and the perfect hashing principle which are combined for a new data structure are explained. The following sections are organized as follows. Section 2.1 shows the static dictionary matching and the dynamic dictionary matching. Section 2.2 illustrated data structures for dictionary accommodation. Section 2.3 presents the methodology of inverted index. Section 2.4 highlights the overview of the hashing principle.

2.1 Dictionary Matching Principles

As previously discussed, dictionary matching is the principle that fulfills the need of locating all occurrences of several patterns in the same time. Usually this principle is separated into two areas; static dictionary and dynamic dictionary. Static dictionary focuses on efficiency of data structure and offers a faster search; meanwhile, dynamic dictionary focuses on the flexible pattern structures and requires frequent updates. This section summarizes certain solutions. The following details are dedicated to history and related research of static and dynamic dictionary algorithms, and the methodology for pattern updating in dynamic dictionary, including algorithm details.

2.1.1 Static Dictionary Matching

The static dictionary matching problem (i.e., multiple pattern matching) is always inherited from the single string pattern matching. This principle searches for all occurrences of patterns $P=\{p^1, p^2,...,p^r\}$ which appear in a large given text $T=\{t_1,t_2,t_3...t_n\}$ within a finite alphabet Σ. Traditionally, this solves the problem in two phases; the preprocessing for generating P to a suitable data structure and the searching for occurring the patterns of P in the text T.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The simplest way to search is to read the pattern p^l to p^r and compare each pattern with the character in the text one by one. This method leads to a time complexity of O($|P|$) in the preprocessing phase; furthermore, the search takes the longest time (O($|t||P|$)) where $|P|$ is the sum of pattern lengths from p^l to p^r and $|t|$ is a size of given text. Therefore, the static dictionary matching requires the efficient data structure to store the collection of patterns and offers a powerful search. Then, the next sub-sections show the static dictionary reviews, the algorithm of Aho-Corasick [1] and SetHorspool [9].

2.1.1.1 Static Dictionary Reviews

With the direction of comparison in each window search, Navaro [9] divided the basic search strategy of a static dictionary search into three directions: the prefix approach, the suffix approach and the factor based approach. The prefix approach searches from the left to the right of the window. The suffix approach searches from the right to the left of the window; finally, the factor approach uses prefix, suffix and word roots to differentiate the search. However, all search windows of all solutions above are slid from the beginning of text to the end of text.

The classic and well known algorithm of prefix approach is the Aho-Corasick[1] algorithm, which was implemented in *fgrep* command of a Unix system. This algorithm inherited the single pattern matching of *Knuth-Morris-Pratt* algorithm to create an automaton structure for generating the Trie of prefix patterns. This is the best solution working in a linear time. The other solutions are the multiple Shift-AND algorithm and the bit-parallel algorithms in [9].

The suffix approach searches from the right to the left of the search window. Normally, this principle inherited the single pattern matching of the Boyer-Moore algorithm to decrease the comparable time. The classic well known algorithms are the Commentz-Walter algorithm and the setHorspool [9] that created the reverted Trie for comparison. The better solution is the Wu-Manber algorithm which was implanted in *agrep* command of a Unix system. In implementing details, the Wu-Manber algorithm uses the block of strings and the hashing table for comparison (shown in [23]).

For factor approach searches, this principle takes the advantages of two previous approaches above to apply to the new algorithm. The factor approach search is inefficient because it is embedded with data structures. Therefore, the more time is required for practical applications. The other solutions are the multiple BNDM factor, the set Backward Dawg Matching, the Backward Oracle Matching algorithm, and so on (detailed in [9]).

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

On the other hand, when considering on techniques of data structure, Trie, Bit-parallel, and combined structures are the most widely used. There are a large number of algorithms that apply Trie; for example, the first linear-time algorithm Aho-Corasick [1] (inherited from KM [7]), the sub-linear time algorithms Commentz-Walter [3] (extended from [4]) and SetHorspool [9] (adapted from [3]). In addition, other Trie-based solutions are [18], [19], MultiBDM[20], SBOM[5], SDBM (shown in [9]), [15], and [11]. All of them improved the searching time (mentioned in [9]). However, implementing Trie to applications takes a large amount of memory in practice.

Alternatively, Navarov [9] showed how to apply the Bit-parallel in the single string Shift-Or and the single Shift-And to the Multiple Shift-And [8], the Multiple-BNDM[9], and algorithm in [10]. Unfortunately, Bit-parallel algorithms are restricted by the computer word and need to deal with the complexity of the bit-conversion methods. Moreover, it is also hard to update the pattern when implementing the dictionary.

Employing a hashing table for dictionary was an important choice; Karp and Rabin [14] were the first to find a solution with this structure. This algorithm took much more time in a worst case scenario: O(mn) time where m is the single pattern length. The disadvantage of this principle is the lengthy processing time when directly extended to a static dictionary. An efficient algorithm was presented by Wu and Manber [23], which created the shift table and implemented the hashing table to store the block of patterns. An efficient solution [25] improved the Wu and Manber method in [23] and saved a searching time in an average case, but the worst case time was not improved.

Other techniques are the algorithms that combine several structures to improve the time complexity such as the *q-gram* structure and the partitioning technique, but the worst case time is again not improved. More details on the development of the new standard can be found in [9], [24], [16] and [17].

Recently developed, solutions [68], [69] and [70] improved the Trie structure to accommodate the patterns especially [69] which has minimal space of solution. For other solutions, there exists a wealth of literature devoted to employ those classic data structures (e.g., Trie, Bit-parallel, and Hashing), can be found in [67], [72] and [73].

2.1.1.2 Aho-Corasick Algorithm

As it was mentioned earlier, this algorithm is called the Prefix Based Approach in [9] because it scans the given text T with P from left to right of the search window. In the

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

preprocessing phase, Aho-Corasick algorithm applies the data structure of KMP [7] to AC automaton Trie (AC-Trie) for accommodating patterns. Consequently, the searching phase needs to scan all characters in the given text. Thus, the time complexity often depends on the data structure implementation of AC-Trie and the time to traverse in Trie.

The AC-Trie is the generalization of Trie which consists of the states and their paths. The state is called the node and contains the number of state in normal or terminal condition to indicate the last characters of patterns. Each node is used to find the matching character in searching phase. The preprocessing phase need to compute two functions: the GOTO function and the Failure function. The main AC-Trie algorithm (detailed by Navaro[9]) is shown as follows.

Algorithm 2.1: AC-Trie Creation

1. Build_AC(P={p^1,p^2,p^3,...,p^r})
2. AC_trie←Trie(P) , Initial_state ← root of AC_trie, S_{AC}(Initial_state) ← θ
3. For Current in transversal order Do
4. Parent← parent of Current in AC_trie
5. σ ←label of the transition from Parent to Current
6. Down ← S_{AC}(Parent)
7. While Down ≠ θ AND $\delta_{AC}(Down,\sigma) = \theta$ Do
8. Down←S_{AC}(Down)
9. End of while
10. If Down ≠ θ Then
11. S_{AC}(Current)← $\delta_{AC}(Down,\sigma)$ //(GOTO function)
12. If S_{AC}(Current) is terminal Then
13. Mark Current as terminal
14. $F(Current) \leftarrow F(Current) \cup F(S_{AC}(Current))$
15. End of if
16. Else S_{AC}(Current)←Initial_state
17. End of if
18. End of for

For implementing the *GOTO* function, all characters of patterns are generated to be forwarded and given the path that are labeled by characters. The new states and the labeled paths of pattern are made if their prefixes are unique from the existing pattern in Trie. This step takes $O(|P|)$ time complexity. After calculating the *GOTO* function, S_{AC} is established for the failure state calculation that computes a backward transition of each node. *Trie(P)* is an input for S_{AC} for the backward transition. This transition will be used for considering the mismatch state while the input text is scanned. Then it will be lined or pointed to an appropriate state for the next

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

comparison. This step uses $O(|P|)$ similar to calculation of the *GOTO* function. For example, Figure 2.1 shows *AC-Trie* of $P=\{aab,aabc,aade\}$.

Example 2.1 Trie of $P=\{aab,aabc,aade\}$.

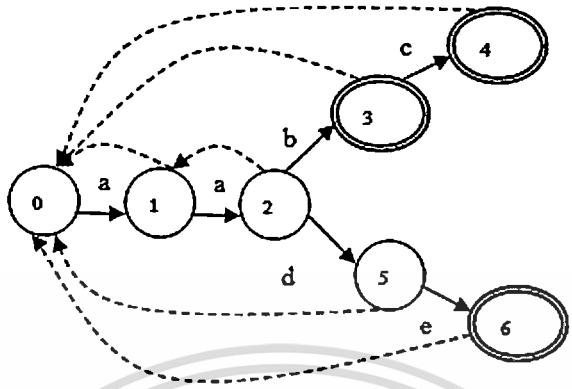


Figure 2.1 *AC-Trie* of $P=\{aab,aabc,aade\}$

In the searching phase, the first character of given text T is initiated with state 0 as the current point. The commencement of matching is activated on T, and the first step checks the finite state. If the current state is the finite, then the current position is reported in the pattern occurrence. In the next step, the failure function is considered for initiating the new state. Meanwhile, the next character in the given text will be compared. Then, the output is finally set into the occurrence patterns that appear in the text T. The time complexity of searching is considered by scanning the text and traversing in the *AC Trie*; that is, it takes $O(|t|)$ for scanning text T and *tocc* time for traversal in Trie. This following algorithm shows the searching idea.

Algorithm 2.2 : AC Search

1. Aho-Corasick($P=\{p^1,p^2,\ldots,p^r\}$,$T=t_1t_2\ldots tn$)
2. **Preprocessing**
3. AC←Build_AC(P)
4. **Searching**
5. Current ← Initial state of the automation AC
6. For pos $\in 1\ldots n$ Do
7. While $\delta_{AC}(Current,t_{pos}) = \theta$ AND S_{AC}(Current) $\neq \theta$ Do
8. Current←S_{AC}(Current)
9. End of while
10. If $\delta_{AC}(Current,t_{pos}) \neq \theta$ Then
11. Current← $\delta_{AC}(Current,t_{pos})$
12. Else Current←initial state of AC
13. End of if
14. If Current is terminal Then
15. Mark all the occurrences (F(Current),pos)
16. End of if
17. End of for

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Time complexity of this algorithm depends on the *GOTO* function and Failure function which computes in *O(2|P|)* time, known as *O(|P|)*. Although the searching phase takes *O(|t|+occ)* time (a linear time), in the practical implementation is not good enough. Unfortunately, *AC-Trie* does not directly support the insertion or deletion of individual patterns; hence, the pattern updating needs to re-generate a new Trie. Therefore, SetHorspool algorithm is expected to be better.

2.1.1.3 SetHorspool Algorithm

Since Commentz and Walter [3] had adapted Boyer-Moore [4] to static dictionary matching. The name of Suffix Approach algorithm was established. This solution is viewed as an original of a backward algorithm. The easier version is called SetHorspool [9]. This employed reverted-Trie and the Shift table for searching from left to right in each search window. This algorithm takes a sub-linear time complexity in practical implementation.

The reverted-Trie of SetHorspool uses an automaton idea to generate the nodes from each last character backward to the beginning of each pattern. Then, the shift table will be calculated from every characters provided for shifting. The shift table is used for referring to the next search window if the comparison is mismatched. The following algorithm details show the main idea of this structure.

Algorithm 2.3: HP Preprocessing

1. HO←Trie(P^{rv}={$(p^1)^{rv}$,..., $(p^r)^{rv}$})
2. δHO is its transition function
3. For c∈ $\sum$ Do d[c]←lmin
4. For j $j \in 1...r$ Do
5. For k $\in 1...m_j - 1$ Do $d[p_k^j] \leftarrow \min(d[p_k^j], m_j - k$
6. End of for

As shown above, reverted Trie ($P^{rv} = \{p_{rv}{}^1, p_{rv}{}^2, p_{rv}{}^3, ..., p_{rv}{}^r\}$) is created as a normal Trie. The time complexity and computing method are similar to *AC-Trie* of Aho-Corasick algorithm. Thus, it equals *O(|P|)* time while the shift table of $P = \{p^1, p^2, p^3, ..., p^r\}$ is also computed in *O(|P|)* time. Example 2.2 shows the reverted-Trie of *P={aab, aabc, aade}*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Example 2.2. If P is *{aab, aabc, aade}*, then the reverted Trie is P^r=*{baa,cbaa,edaa}* , and it is illustrated on Trie as follows.

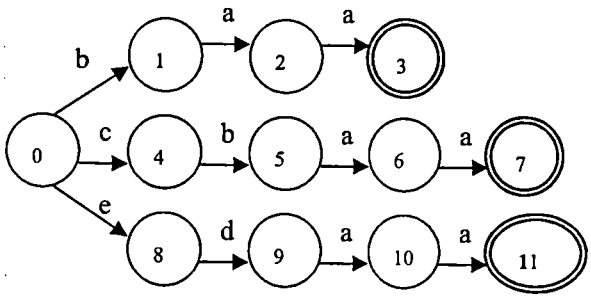


Figure 2.2 The revert Trie of P={aab,aabc,aade}

Table 2.1 The shift table window of P={aab, aabc, aade}

a	b	c	d	e	*
1	1	3	1	3	3

The search is initiated at state 0 and sets the first comparison at the last character in the first search window as a reference point to be matched. Then, matching is compared with the search window followed by the smallest pattern length. The matching needs to check three following operations: the first operation checks whether the match is a finite state; the second operation checks whether the forward transitions are in reverted-Trie or not, and the third operation checks if the matching has failed then uses the shift table to a new search window. The algorithm is shown below.

Algorithm 2.4 : HP Searching

1. pos←lmin
2. While pos $\leq n$ Do
3. j←0,Current←initial state of HO
4. While pos-j > 0 AND $\delta_{HO}(t_{pos-j}, Current) \neq \theta$ Do
5. If Current is terminal Then
6. Mark all the occurrences (F(Current),pos)
7. End of if
8. Current $\leftarrow \delta_{HO}(t_{pos-j}, Current)$
9. j←j+1
10. End of while
11. pos←pos+d[t_{pos}]
12. End of while

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

SetHorspool takes $O(|P|)$ time in the preprocessing phase and takes $O(|t|l_{max})$ time in the searching phase; where l_{max} is the maximum length of pattern in P. Similar to AC-Trie, the preprocessing phase takes two steps; the first is construction the revert-Trie, and the second is the calculation of the Shift table that leads to $O(2|P|)$ time. Shift table and the suffix approach are the advantage of SetHorspool; furthermore, searching is fast in practice, especially working in a sub-linear time. This algorithm is not efficient in searching if the minimum length is small, and the shift value is 1, which is equal to $O(|t||P|)$ time.

2.1.2 Dynamic Dictionary Matching

The dynamic dictionary matching is to search all occurrences of patterns $P=\{p^1, p^2,..,p^r\}$ (i.e., called the dictionary) appeared in the given text $T=\{t_1,t_2,t_3...t_n\}$ over a finite alphabet $\sum$, with the ability to insert or delete the individual patterns over time. Therefore, the dictionary requires an efficient and flexible data structure to accommodate the collection of patterns for a more powerful search.

2.1.2.1 Traditional Dynamic Dictionary Matching Algorithms

According to the earlier mentions, the Trie is the traditional structure to accommodate the static dictionary, and the suffix tree is the most common data structure to accommodate the dynamic dictionary. A large majority of the previous works rely heavily on the suffix tree. The first suffix tree based algorithm presented by Amir and Farach [27] and this is the first adaptive algorithm that requires high time complexity. The suffix tree offers dynamical patterns for the dictionary, though there is a trade off inefficiency of time and space. Subsequently, many of the logarithmic algorithms were based on the same principle i.e., $O(|t| \log |t|)$, shown in [28], [29], [30], [31], [32], [33], [34] and [43]. All of them require the dynamical method of McCreight [36], *DS-List* [35], or Weiner [44] to accommodate and to maintain the dictionary.

The main challenge of this principle is to recommend bypassing the factor of logarithmic time. Some efforts and some improvements were presented by AFGGP [33] that showed the first algorithm nearly took the same time as the Aho-Corasick algorithm [1]. Nevertheless, it did not collapse the *log n* factor. It can be said that all suffix tree approaches fall into the trap of $O(|t|log|t|)$ time. Even if other algorithms, such as [32] and [41], tried to improve DS-List [35] for storing patterns, the results of time complexity are in the logarithmic time.

Fortunately, the latest edition of an algorithm is shown by Shinalp and Vishkin [45], which takes the time complexity equivalent to the static dictionary algorithms. However, it employs the fat-tree, which leads to a larger space for accommodating the dictionary. Lately,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

there are good surveys and reviews in [24], [46], [47], and [16], which provide the relevant literatures. The following sub-sections show the classic dynamic dictionary algorithms of AFGGP[8], which highlight dictionary construction and pattern insertion and deletion.

2.1.2.2 AFGGP Algorithm

This thesis focuses on the *AFGGP* [8], which is the first algorithm implementing the suffix tree of McCreight [36]. The main idea of the algorithm is to create the dynamic suffix tree using the dynamic tree of Sleator and Tarjan [14]. This algorithm, known as a classic algorithm for dynamic dictionary matching, took an exhaustive time in searching phase. The following sub-section shows the pre-processing phase, updating pattern, and searching phase, respectively.

2.1.2.2.1 Preprocessing phase of AFGGP

This phase is separated into two steps; creating the suffix tree and creating the forest tree. Let $D_i = \{p_1,...,p_s\}$ be the dictionary where D_i consists of the suffix tree and the forest tree in the same structure. The suffix tree needs to add the special symbol as \$ in every pattern. DS_i represents the suffix tree T. Let $DS_i = p_1\$,...p_s\$$. The method works as follow.

1) read the pattern one by one to create the suffix tree of McCreight [36]. The procedure *STI* shows the algorithm details. Let β be the transition of nodes, and α be the transition which relates the node x to the child node $\int$.

Algorithm 2.5 : Procedure STI(v,i)

1. Step A : If v is root y← root . Go to Step D
2. Step B1 : Else x←SL(parent(u)). GO to Step C
3. Step B2 :If parent(v) is not root x←root; β ←$head_{i-1}$ – S[i-1] Go to Step C
4. Step C : If $|\alpha| < |\hat{\beta}|$,set $\hat{\beta} \leftarrow \hat{\beta} - \alpha$ and x←$\int$ and repeat until $|\alpha| \geq |\hat{\beta}|$
5. Step C1 :. If $|\alpha| > |\hat{\beta}|$, $\int$ ←$head_{i-1}$ – S[i-1] , L(d)= $head_{i-1}$ – S[i-1],set SL(v)←d
6. Step C2 : If $|\alpha| = |\hat{\beta}|$, $\int$ ←$head_{i-1}$ – S[i-1] , set SL(v)← $\int$;y←$\int$
7. Step D : $head_i$=L(y); L(v)=$head_i$; Create L(w)=S[i,m] as a child of v ; Return nod v

---

2) create the forest of trees. After creating the suffix tree, the dynamic structure can be calculated. The dynamic of the tree is to separate the suffix tree into each smaller unit called the forest of tree. Each is denoted by T_{D_i} and all suffix tree by F_{D_i}. The method creates the virtual node $\hat{v}$, which is created on the node v and marked at root of T_{D_i} for updating the pattern. The algorithm details are shown as below.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Algorithm 2.6 : Create Forest tree

1. For each node v in T_{D_i} there is a corresponding node $\hat{v}$ in F_{D_i}
2. v is marked in v T_{D_i} if and only if $\hat{v}$ is the root of some tree in F_{D_i}
3. $\hat{v}$ is in the tree with root $\hat{r}$ in F_{D_i} if and only if , among the marked nodes, L(r) is the longest pattern that is prefix of L(v)

2.1.2.2.2 Searching phase of AFGGP

Let *t[1,n]* be the text T to be scanned. This search needs to lookup the longest suffix D_i. Let h_j represent *t[j,n]\$* which is the substring of D_i, where $1 \leq j \leq n$. The search steps are shown below.

1) Searching a sub-pattern

Search h_j in the couple of clocus of β where clocus is the current character and $\beta = h_j$ - L(clocus) which can be found h_j by increasing j in the suffix tree T_{Di}. The searching method is shown as Algorithm 2.7.

Algorithm 2.7 :Procedure SEARCH(clocus, β)

1. Step A : case clocus is root : y←root . Go to Step D.
2. Step B : case clocus is not root : x←SL(clocus). Go to Step C.
3. Step C: set $\hat{\beta} \leftarrow \hat{\beta}$. If $|\alpha| < |\hat{\beta}|$,set $\hat{\beta} \leftarrow \hat{\beta} - \alpha$ and x← ∫ and repeat until $|\alpha| \geq |\hat{\beta}|$
4. Step C1 : If $|\alpha| > |\hat{\beta}|$, h_j=L(x) $\hat{\beta}$, since the first character of $\alpha - \hat{\beta}$ mismatches in step j-1
   Stop and return (x, $\hat{\beta}$)
5. Step C2 : If $|\alpha| = |\hat{\beta}|$,set y← ∫ . Go to Step D.
6. Step D : from node y searches down the suffix tree by scanning all character. If not match contract locas of hj and Return locus v and β =h_j-L(v)

2) Searching all occurrences

The searching method looks for each T_{D_i} which is not a root of T_{D_i} and compares them with the marked node that is not the root of T_{D_i} . There are two conditions to be considered.

1. If β is not null and $h_j = p_r$, then the matching is successful whenever h_j is the leaf node where L(w) = p_r\$r.

2. Each suffix of h_j needs the position of v in T_{Di} and v is the path from clocus to the root of T_{D_i} , and v is not the root, which is matched.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

Algorithm 2.8 :Procedure FINDALL(clocus, β)

1. report occurrence when if β not empty and if L(w) =$p_r\$_r$
2. u←clocus
3. While u ≠ root of T_{D_i}
4. $\hat{v}$ ←root($\hat{u}$)
5. If v ≠ root of T_{D_i} then
6. report occurrence
7. u←parrent(v)
8. End of if
9. End of while

2.1.2.2.3 Pattern Insertion

There are two step for pattern insertion; insert pattern $p_j\$$ into $T_{D_{i-1}}$ and mark the dynamic node in $F_{D_{i-1}}$ which needs to follow 2 step.

1. After insertion $p_j\$$, if the node v does not create a new node, then $L(v) = p_j$ and mark at the node v.

2. After insertion of $\alpha\$$, if there is a new node v where L(v) = α and α is the pattern, then the node v is marked. The algorithm below shows the pattern insertion details.

Algorithm 2.9: Procedure FTI(suffix tree node u)

1 Step A : If u is a leaf ; newnode($\hat{u}$); link($\hat{v}$,$\hat{u}$)

2 Step B : u is marked, but not a new node: cut($\hat{u}$) ; $\hat{u}$ is not the root and not marked , and $\hat{w}$ is descendant of $\hat{u}$,L(u) is the longest pattern that is prefix of L(u).

3 Step C : u is a new internal node : newnod($\hat{u}$) and take Step C1-C4

4 Step C1 : If u not marked and root($\hat{v}$)=root($\hat{w}$) ; cut($\hat{w}$); link($\hat{v}$,$\hat{u}$); link($\hat{u}$,$\hat{w}$)

5 Step C2 : If u not marked and root($\hat{v}$) ≠ root($\hat{w}$);link($\hat{v}$,$\hat{u}$)

6 Step C3 : u marked and root($\hat{v}$)=root($\hat{w}$); cut($\hat{w}$); link($\hat{u}$,$\hat{w}$)

7 Step C4 : u marked and root($\hat{v}$) ≠ root($\hat{w}$) ; Do nothing

2.1.2.2.4 Pattern Deletion

The pattern deletion is needed to remove p_j from $D_{i-1} = \{p_1,...,p_s\}$. This method needs to convert $T_{D_{i-1}}$ to T_{D_i} for $DS_i = p_{j-1}\$_{j-1}p_{j+1}\$_{j+1}...p_s\$_s$. Then, all suffix $p_j\$_j$ needs to be deleted and

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

137186

$F_{D_{i-1}}$ is changed. The algorithm uses *STD* for deleting the suffix tree node and *FTD* for updating the forest of $F_{D_{i-1}}$. The algorithm detail is shown below.

Algorithm 2.10: Procedure STD(suffix tree node v)

1. Step A : If v is the root : delete u.
2. Step B : If v is not the root take two step B1,B2
3. Step B1 : If v has more than two children : delete u. v still has at least two children.
4. Step B2 : If v has exactly two children : delete u; delete v; and assign (parent(v),v),(v,w)

Algorithm 2.11: Procedure FTD(suffix tree node u)

1. Step A : If u is a leaf : cut($\hat{u}$)
2. Step B : If u is an internal node,unmarked but not deleted : link($\hat{v}$,$\hat{u}$)
3. Step C : If u is an internal node, which has been deleted ; Let q be the only child of u : cut($\hat{u}$); cut($\hat{q}$);link($\hat{v}$,$\hat{q}$)

To sum up, the time complexity of the preprocessing phase is $O(|P| \log |P|)$; meanwhile, the searching time is $O((|t|+tocc) \log |P|)$ where *tocc* is the number of occurrences. The pattern insertion takes $O(|p| \log|P|)$ time where $|P|$ is the dictionary after insertion, and $|p|$ is the length of pattern to be inserted. The pattern deletion takes $O(|p| \log|P|)$ time.

2.2 Data Structures for Dictionary Accommodation

As mentioned in the previous chapter, Trie and Suffix tree are the classic structures to accommodate the patterns in the preprocessing phase. Trie is the well known data structure provided for solving the problems in string processing. This employs an automaton structure for containing the set of states which are labeled by the characters. The well known data structure, which can be applied to the dynamic dictionary matching, is the suffix tree. This structure was presented by Weiner in 1973. Then, this structure was improved by McCreight [36] in 1976. The following sub-sections show the details of Trie and Suffix tree, which are the main data structure to be compared with the inverted lists structure.

2.2.1 Trie Structure

Trie, a nickname that comes from "Retrieval", is an efficient dictionary data structure for string processing. This structure can quickly search a large text such as the Oxford English dictionary in large Giga byte units. Moreover, Trie can be applied in text compression, dictionary matching, approximate string matching, and so on. Trie is the prefix tree structure for containing

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

the string (shown in [23] and [24]). Trie can be more efficient if all paths of single-child nodes have at least two children. The path compressed trie has the minimum number of nodes among the Trie representing a certain set of string. Briefly, it can be said that Trie consists of the nodes of state and the edges lined by the characters, while every node must have the suffix.

Trie is constructed by using the string $S_1,S_2,...,S_n$ where S_1 to S_n are the individual strings which use the symbol *$* for termination. The well known Trie is non-compact Trie, which is applied to the static dictionary implemented by the automaton for the preprocessing phase. The Aho-Corasick[1] and the setHorspool (shown in [9]) are the algorithms employing the Trie to accommodate the set of patterns.

Non-compact Trie is the tree structure where each node is lined by character, each node is related to the next level of tree and the leaf node stores the special symbol $. The example 2.3 shows the Trie, which stores the string BIG, BIGGER, BILL, GOOD, and GOSH

Example 2.3 the Trie structure of string: BIG, BIGGER, BILL, GOOD, GOSH.

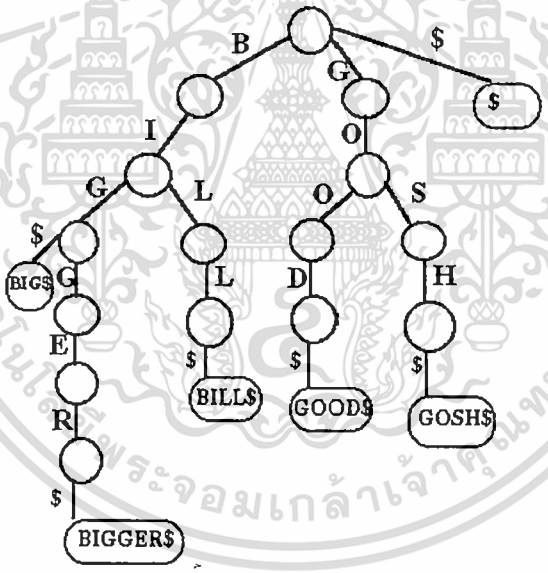


Figure 2.3 Non-compact Trie of BIG, BIGGER, BILL, GOOD, GOSH

Applying non-compact Trie to application employs the automaton string without the leaf node. Example 2.4 represents the Trie when the pattern *P={ATATATA, TATAT, ACGATAT}*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Example 2.4 the Trie of *P={ATATATA,TATAT,ACGATAT}*.

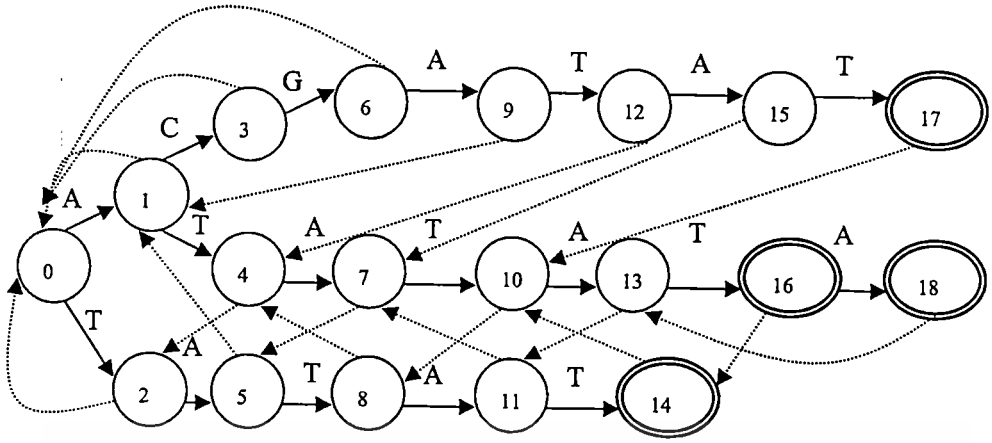


Figure 2.4 Trie of *P={ATATATA,TATAT,ACGATAT}*

Above is an illustrated where the characters are read one by one from the first pattern to the last pattern, while insertion methods maneuver the Trie to a suitable position. The pseudo code is shown as algorithm 2.12 where P is the set of patterns, *Current* is the current position to consider for adding the node and $\delta(Current, p_j^i)$ is the function to search the position to add the node of pattern i^{th} at character j^{th}. $F(Current)$ is the function to search the path of current position to the root of Trie.

Algorithm 2.12: Trie Creation

1. Trie($P = \{p^1, p^2, ..., p^r\}$)
2. Create an initial non terminal state 0
3. For $i \in 1...r$ Do
4. Current ← initial state 0
5. j← 1
6. While $j \le m_i$ AND $\delta(Current, p_j^i) \neq \theta$ Do
7. $Current \leftarrow \delta(Current, p_j^i)$
8. $j \leftarrow j+1$
9. End of while
10. While $j \le m_i$ Do
11. Create a new non terminal state State
12. $\delta(Current, p_j^i) \leftarrow State$
13. $Current \leftarrow State$
14. $j \leftarrow j+1$
15. End of while
16. If Current is already terminal Then $F(Current) \leftarrow F(Current) \cup \{i\}$
17. Else mark Current as terminal, $F(Current) \leftarrow \{i\}$
18. End of for

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The time complexity of Trie is $O(|P|)$ where $|P|$ is the sum of all length of pattern in P. In addition, the complexity depends on the implementing technique for programming. The Aho-Corasick and the setHorspool algorithm take this principle to create the automaton to store the pattern in the preprocessing phase.

In other words, Commentz-Walter[3] and SetHorspool [18] used Trie in the reverted-Trie to accommodate the pattern as well. The example above can generate to the reverted-Trie as shown in an example 2.5 below.

Example 2.5 Reverted-Trie of $P=\{ATATATA,TATAT,ACGATAT\}$ when the reverted-patterns of P is $P^{rv}=\{ATATATA,TATAT,TATAGCA\}$.

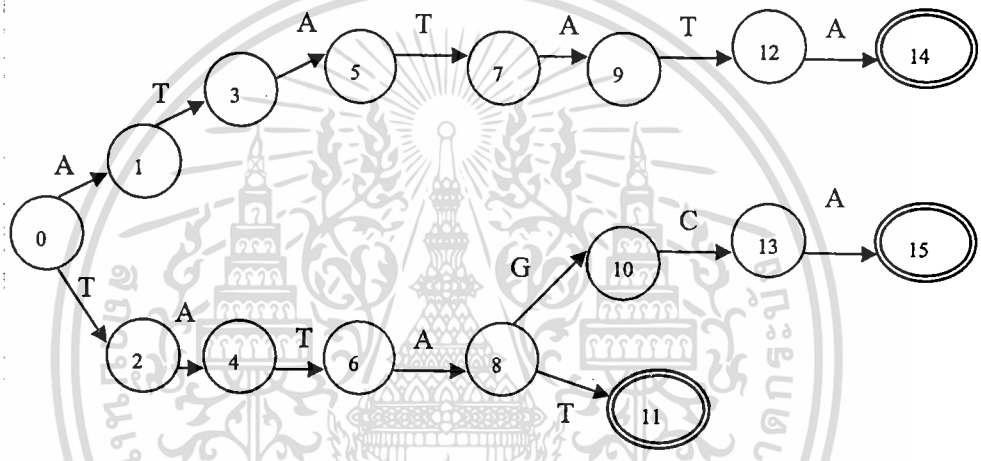


Figure 2.5 the revert Trie of $P=\{ATATATA,TATAT,ACGATAT\}$.

A search method is trying to match from the root of the tree until the searched string is completely matched or a leaf is reached. Thus, successfully searching a given text of length n with the matching time *locc* required $O(n+locc)$ time. The searching details can be found in the previous section.

2.2.2 Suffix Tree

Suffix tree is a very powerful tool for many tasks in dictionary matching. This structure is actually developed from the suffix Trie. The suffix tree was presented by Weiner in 1973 and was improved by McCreight [36] in 1976. Afterwards, it was developed as an on-line algorithm by Ukkonen in 1995. This principle, which applies to the dynamic dictionary matching, was employed in the algorithm of McCreight [36] and Ukkonen in 1995.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The time complexity of Trie is $O(|P|)$ where $|P|$ is the sum of all length of pattern in P. In addition, the complexity depends on the implementing technique for programming. The Aho-Corasick and the setHorspool algorithm take this principle to create the automaton to store the pattern in the preprocessing phase.

In other words, Commentz-Walter[3] and SetHorspool [18] used Trie in the reverted-Trie to accommodate the pattern as well. The example above can generate to the reverted-Trie as shown in an example 2.5 below.

Example 2.5 Reverted-Trie of $P=\{ATATATA,TATAT,ACGATAT\}$ when the reverted-patterns of P is $P^{rv}=\{ATATATA,TATAT,TATAGCA\}$.

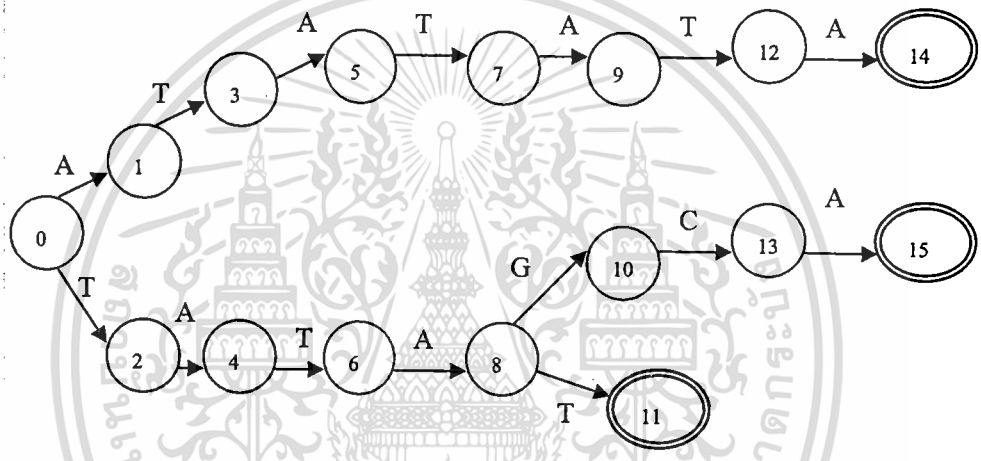


Figure 2.5 the revert Trie of $P=\{ATATATA,TATAT,ACGATAT\}$.

A search method is trying to match from the root of the tree until the searched string is completely matched or a leaf is reached. Thus, successfully searching a given text of length n with the matching time *locc* required $O(n+locc)$ time. The searching details can be found in the previous section.

2.2.2 Suffix Tree

Suffix tree is a very powerful tool for many tasks in dictionary matching. This structure is actually developed from the suffix Trie. The suffix tree was presented by Weiner in 1973 and was improved by McCreight [36] in 1976. Afterwards, it was developed as an on-line algorithm by Ukkonen in 1995. This principle, which applies to the dynamic dictionary matching, was employed in the algorithm of McCreight [36] and Ukkonen in 1995.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

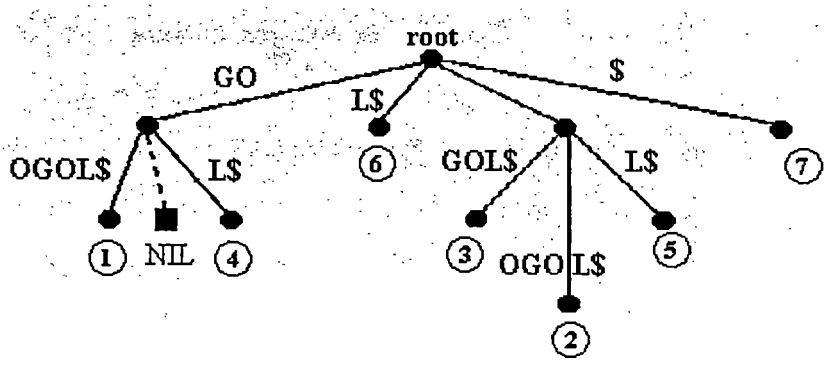


Figure 2.7 The suffix Tree of GOOGOL$

The pseudo code for constructing the suffix Trie of McCreigt [36] is shown in the algorithm 2.13 where ε is the root of tree. In this case, if there is only one string, then the time complexity is $O(|P|)$, which is similar to the normal Trie. In principal, the pruning algorithm works as follows.

Algorithm 2.13 : Tree Construction

```
1.  Construct tree for x[1..n]
2.  for i = 1 to n do
3.      if head(i)= ε then
4.          head(i+1) = slowscan( ε ,s(tail(i)))
5.          add i+1 and head(i+1) as node if necessary
6.          continue
7.      u = parent(head(i)) ; v = label(u,head(i))
8.      if u≠ε then w = fastscan(s(u),v)
9.      else w = fastscan( ε ,v[2..|v|])
10.     if w is an edge then
11.         add a node for w
12.         head(i+1) = w
13.     else if w is a node then
14.         head(i+1) = slowscan(w,tail(i))
15.         add head(i+1) as node if necessary
16.         s(head(i)) = w
17.     add leaf i+1 and edge between head(i+1) and i+1
```

2.3 Inverted Index

The inverted index is the data structure that reverses the words and their positions for supporting the data in information retrieval systems. This structure represents the data structure in the form of <*document ID, word:pos*> where '*document ID*' is the indicated number of documents, '*word*' represents the vocabulary keywords and '*pos*' is the occurrence position of '*word*' in the *document ID*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The original inverted index sources [13], [37] and [39] define all documents as $D=\{D_1...D_n\}$ where D_i is each document containing the various keywords in the various positions and $1 \le i \le n$.

In order to give evidence of the above statements, the inverted index is created in three steps; analyze the keywords occurred in target documents, format the keywords and apply to a suitable data structure. The following steps express these rules of an inverted index construction.

1. Assign the number of documents and analyze the keywords in each document.

Let D be the entire document containing any keywords $w_a, w_b, w_c, ...$and $D=\{ D_1...D_n\}$. If there are the documents $D_1...D_n$, then the keywords can be analyzed as follows:

D_1: $w_a, w_b, w_c, ...$

D_2: $w_a, w_d, w_e, ...$

D_3: $w_a, w_b, w_d, ...$

...

D_n: $w_x, w_y, w_z, ...$

2. Format the keywords to the index construction form.

Rewrite the keywords in the inverted form; the form of words, positions, and documents. Each document indicates a unique number from D_1 to D_n, and the keywords w in each document are analyzed and given their occurring positions. For example, if there are the keywords w_a:*1*,w_b:*2*,w_c:*3*,..., then w_a:*1* means the keyword w_a appears at position 1.

3. Take the keywords and their position to store in a suitable data structure such as tree structure, Trie, or B^+ tree. Example 2.7 illustrates the steps of an inverted index construction.

Example 2.7 The inverted index construction in the case of several documents.

The Project Gutenberg Etext of Webster's

OPTED is a public domain English word list dictionary

All the Gutenburg EText tags

OPTED v0.03 by Letter

Webster's Unabridged Dictionary

including OPTED, Project Gutenburg

The following steps are performed as below.

1. Assign numbers to each line of the document like in table 2.2

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 2.2 The number of all documents.

Document ID	Lists of Words
D_1	The Project Gutenburg Etext of Webster's
D_2	OPTED is a public domain English word list
D_3	All the Gutenburg Etext tags
D_4	OPTED v0.03 by Letter
D_5	Webster's Unabridged Dictionary
D_6	including OPTED, Project Gutenburg

2. Analyze the keywords and give the position occurrences in terms of the keywords and their *document ID*, as shown below.

Table 2.3 The number of each document.

Document ID	Lists of Words
D_1	The:1 Project:2 Gutenburg:3 Etext:4 of:5 Webster's:6
D_2	OPTED:1 is:2 a:3 public:4 domain:5 English:6 word:7 list:8
D_3	All:1 the:2 Gutenburg:3 Etext:4 tags:5
D_4	OPTED:1 v0.03:2 by:3 Letter:4
D_5	Webster's:1 Unabridged:2 Dictionary:3
D_6	Including:1 OPTED:2, Project:3 Gutenburg:4

Keywords : (Occurrences in Document ID: Positions)

a : (2:3)

All : (3:1)

by : (4:3)

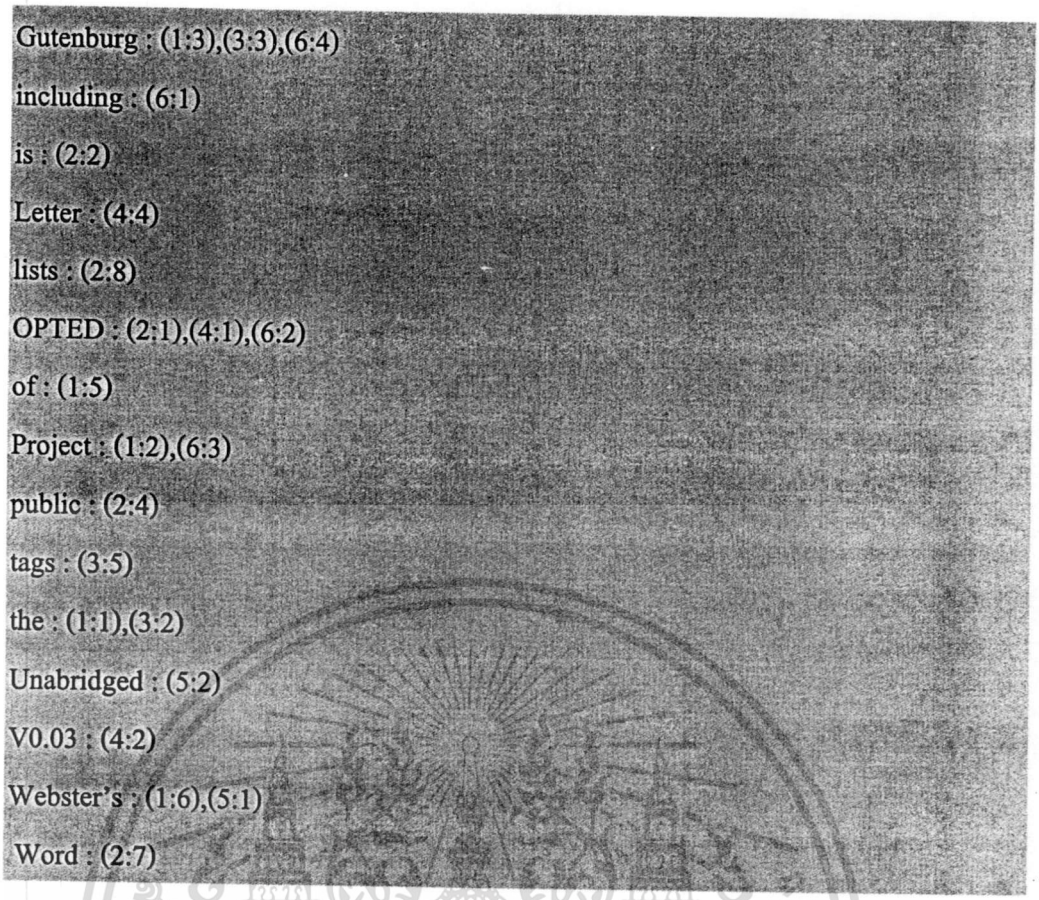
Dictionary : (5:3)

domain : (2:5)

English : (2:6)

Etext : (1:4), (3:4)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



3. Take the keywords and their positions for storing in a suitable data structure; for instance, the suffix tree, B^+ Tree, the suffix array, etc.

In the context of this thesis, all keywords are rewritten in the form of *word: posting lists* where '*posting lists*' is the form of *documentID: word position in that document*. Chapter 3 details how to apply this idea for accommodating the set of patterns.

2.4 Perfect Hashing Principle

The hashing table is commonly known as the data structure that searches quickly in *O(1)* time. Normally, this structure accommodates data to the set of tables and accesses data by calculating the data address by a hashing function. There are the universal keys which can collide and the related set provides all data. The universal keys might be generated into the individual key provided they have mapped the data search. The data in the set will be accessed if the correct key is mapped. Figure 2.10 shows this idea.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

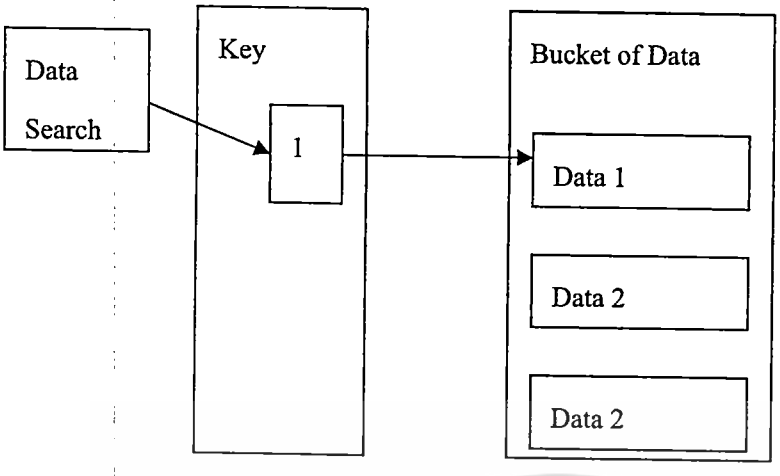


Figure 2.8 The idea of the hashing table

Fundamentally, data to be searched is required to be calculated and accessed through the key for retrieving the matched data in the bucket. In general, a hashing function maps several different keys to the same index. As a consequence, the major problem of the hashing principle is the collision of keys when mapping the data search into the bucket of the data structure.

Fortunately, a perfect hashing table is the excellent solution when keys collide. This principle is suitable for static keys e.g., a set of reserved words in a programming language. A hashing technique is called perfect hashing if and only if the worst-case time for a search is *O(1)*.

Implementing the perfect hashing table uses two-levels of data structure. The first level is the same as hashing with chaining: n keys are hashed into $m = n$ slots using a hash f_n. h from a universal collection. In the second level, the chaining keys hashes to the same slot j, and a small *secondary hash table* S_j with an associated hash f_n. h_j.

The uniqueness of the hash function also yields the capability of accessing the parent of a node without using extra storage. With the proposed expecting structure, the perfect hashing takes O(n) space and *O(1)* time (shown in [37], [38], [39] and [40]) for the searching phase where n is the size of data. This idea is exemplified below.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

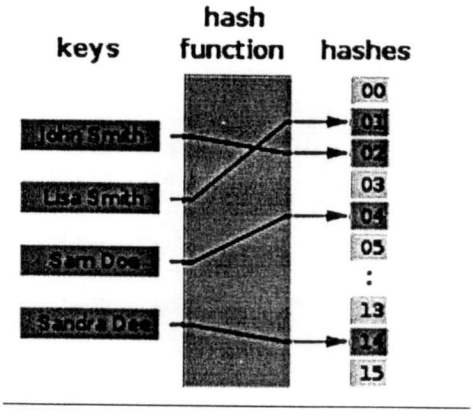


Figure 2.9 The idea of the perfect hashing function

As mentioned above, the perfect hashing principle has:

1) The universal key U to accommodate all keys for accessing all data in the table, and

2) Two levels to implement the table.

The first level is the n keys for accessing to the second level by function $f(n)$ and the second level is the data items associated the corresponding key of n.

This thesis assigns $\sum$ as the universal key U and $f(\lambda)$ as $f(k)$ for the first level of the perfect hashing table. In the second level of the table, the inverted lists are assigned as data items in the bucket of the hashing table. This rule can be roughly applied in the example below.

Example 2.8 If $P=\{aab, aabc, aade\}$, then the patterns are assigned as $D_1=aab$, $D_2=aabc$, and $D_3=aade$.

$U=\{a,b,c,..z\}$

$\lambda=\{a,b,c,d,e\}$

Then $a,b,c,d,$ and e are analyzed for their position as individual keywords in an inverted index. After analyzing, the results are:

D_1: a:1,a:2,b:3,

D_2: a:1,a:2, b:3, c:4, and

D_3: a:1,a:2, d:3, e:4.

In the final step, each position and its occurrence in all documents are assigned for accommodation in two levels of the hashing table. When '*a*' is considered; therefore, '*a*' is occured at the positons 1 and 2, and in the document 1, 2, and 3. In this case, the postions 1 and 2 are represented as the first level of the hashing table, as well as the document numbers 1, 2, and 3

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

are represented as the second level of the hashing table. Meanwhile, the univeral keys are the constant and can be represented as fixed locations when they are implemented.

With this idea, the hashing table maps only the first and the second levels (as shown above) into the bucket that leads to take double the time for accessing. However, it takes only *O(1)* time. Because of the perfect hashing is suitable for accommodating the proposed data structure. The next chapter presents the granular details of an applied structure.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

CHAPTER 3

INVERTED LISTS DICTIONARY MATCHING

This chapter details the main study methodologies that consist of 1) analysis of the related principles, 2) adaptation of the inverted index and perfect hashing table to create the inverted lists, 3) development of algorithms of dictionary construction, dictionary updating, and three search algorithms, and 4) proof of the complexity of time, space, and correctness of all developed algorithms. The first section explains principle derivations on how to adapt an inverted index and a perfect hashing table to create the inverted lists. Then, the inverted lists data structure for dynamic dictionary matching are illustrated. The next sections show dictionary construction algorithms, pattern insertion and pattern deletion. Then, an exhaustive solution (called E-IVL), a prefix approach (called PF-IVL), and a suffix factor solution (called SF-IVL) are illustrated respectively. In addition, each algorithm is also analyzed and proven for time complexity and correctness. In particular, the inverted lists dictionary needs to be analyzed and to be proven for space complexity.

3.1 Principle Derivations

As previously mentioned, this research has two principles with the goal of creating a new structure. In this section, the details of inverted lists in the perfect hashing aspect are deepened in the two following sub-sections.

3.1.1 Adapting Inverted Index to Inverted Lists

The inverted index, shown in chapter 2, contains the words in the target documents with the form of <*document ID, word:pos*> where '*document ID*' is the indicated number of documents, '*word*' represents the keywords or the vocabulary, and '*pos*' is the occurrence position of '*word*' in the *document ID*. The original inverted index (shown in [13], [20], and [22]) defines all documents as $D=\{D_1...D_n\}$ where D_i is each document containing various keywords in the various positions where $1 \le i \le n$. The inverted index can be constructed in two steps: converting characters and assigning position form.

Firstly, each document is indicated to a unique number from D_1 to D_n. The keywords of w are represented in each document. For instance, if the keywords are $w_a{:}1, w_b{:}2, w_c{:}3, \ldots$ then $w_a{:}1$ it means the keyword w_a appears at position 1. Thus, if the documents are $D_1, D_2, D_3, \ldots, D_n$, they can

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

be defined as the spot ❶ in Figure 3.1. The words are grouped to the form of *word: (posting lists)* where '*posting list*' is the form of *(documentID: word position in that document)*.

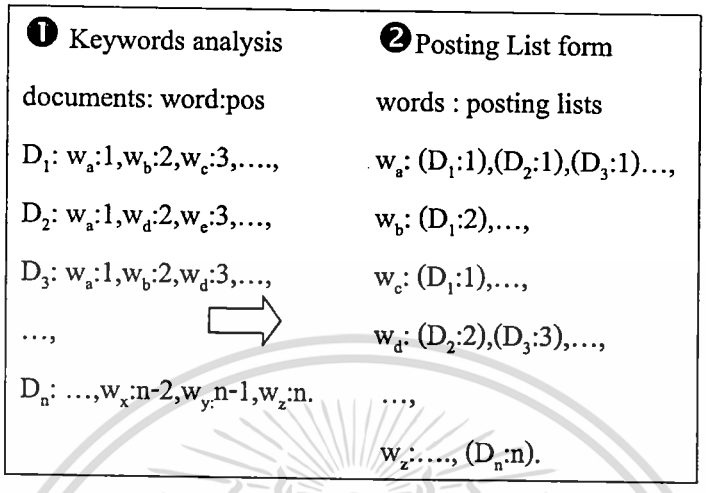


Figure 3.1 Keyword representation in posting list form

Secondly, the idea behind creation to this structure is to replace the document D by the pattern P, and p^i with each D_i. For example, if we have the given pattern P=*{aab, aabc, aade}* then the patterns are assigned as D_1=*aab*, D_2=*aabc*, and D_3=*aade*. Afterwards, the keywords in each document are analyzed and their positions are located. This idea is further examined in Figure 3.2.

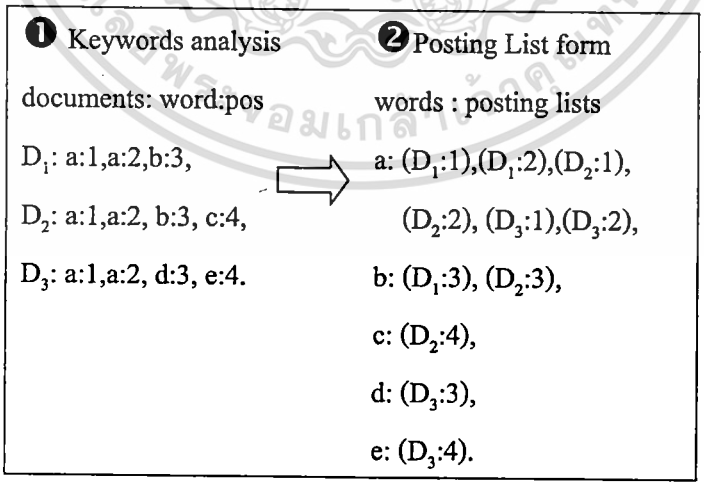


Figure 3.2 The generalized idea of inverted index in our data structure

Finally, the pairs of characters between the alphabets and their position in P are represented and implemented to the form of *character :<the occurrence position of character in*

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

pattern: the indicated status of the last character of pattern: the number of pattern in P>; e.g., *a:<1:0:1*>, <*2:0:1*>, <*1:0:2*>, <*2:0:2*>,... . This form is called the *individual posting list*. Using this method, the individual posting lists can be applied to many aspects of positions. One example is shown in 3.1.

Example 3.1 If P consists of *{ram, run, running}*, then $p^1=r_1a_2m_3$, $p^2=r_1u_2n_3$, and $p^3=r_1u_2n_3n_4i_5n_6g_7$. All individual posting lists of P are represented below.

$p^1 = r :< 1 : 0 : 1 >, a :< 2 : 0 : 1 >, m :< 3 : 1 : 1 >,$
$p^2 = r :< 1 : 0 : 2 >, u :< 2 : 0 : 2 >, n :< 3 : 0 : 2 >,$ and
$p^3 = r :< 1 : 0 : 3 >, u :< 2 : 0 : 3 >, n :< 3 : 0 : 3 >,$
$n :< 4 : 0 : 3 >, i :< 5 : 0 : 3 >, n :< 6 : 0 : 3 >, g :< 7 : 1 : 3 >.$

3.1.2 Accommodating Inverted Lists in Perfect Hashing

As it was already mentioned, this research assigns $\sum$ as the universal key U and $f(\lambda)$ as $f(n)$ for the first level of the perfect hashing table and the groups of posting lists as the data items of the second level where $\lambda \subseteq \sum$. In the detail of table, there are two parts: keys and the levels of inverted lists. The keys represent all characters which occur in the set of patterns to be used. There are two levels of inverted lists as well: positions and corresponding IVL. A position part represents the positions of character which occur in patterns; meanwhile, the corresponding IVL represents the patterns that are related to the occurred positions. The perfect hashing table is created as in Figure 3.3.

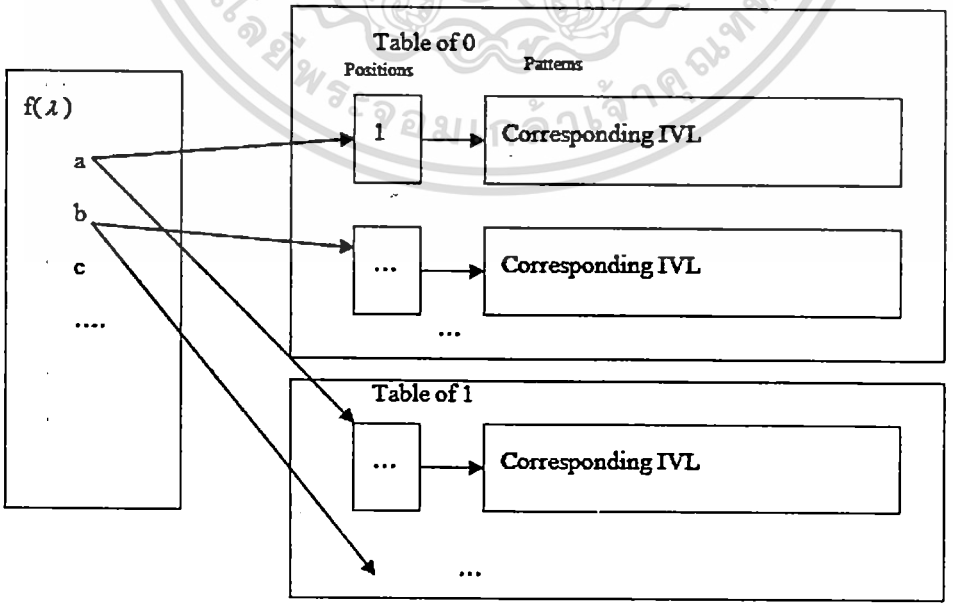


Figure 3.3 the idea for creating the table

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

With this idea, all individual inverted lists are grouped into new forms to be implemented in the hashing table. Example 3.2 shows the method from the conversion of a single character to the individual inverted list. Afterwards, they are stored in the perfect hashing table as Figure 3.4.

Example 3.2 Implementation of individual inverted lists from example 3.1.

Firstly, all individual posting lists are grouped by the form of *character : position :{set of number of patterns}*. Therefore, all individual posting lists above can be grouped to the new form below.

$a :< 2 : 0 : \{1\} >$
$g :< 7 : 1 : \{3\} >$
$i :< 5 : 0 : \{3\} >$
$m :< 3 : 1 : \{1\} >$
$n :< 3 : 0 : \{3\} >,< 3 : 1 : \{2\} >,< 4 : 0 : \{3\} >, < 6 : 0 : \{3\} >$
$r :< 1 : 0 : \{1, 2, 3\} >$
$u :< 2 : 0 : \{2, 3\} >$

A general outlook of inverted lists in such a hashing table is demonstrated in figure 3.4.

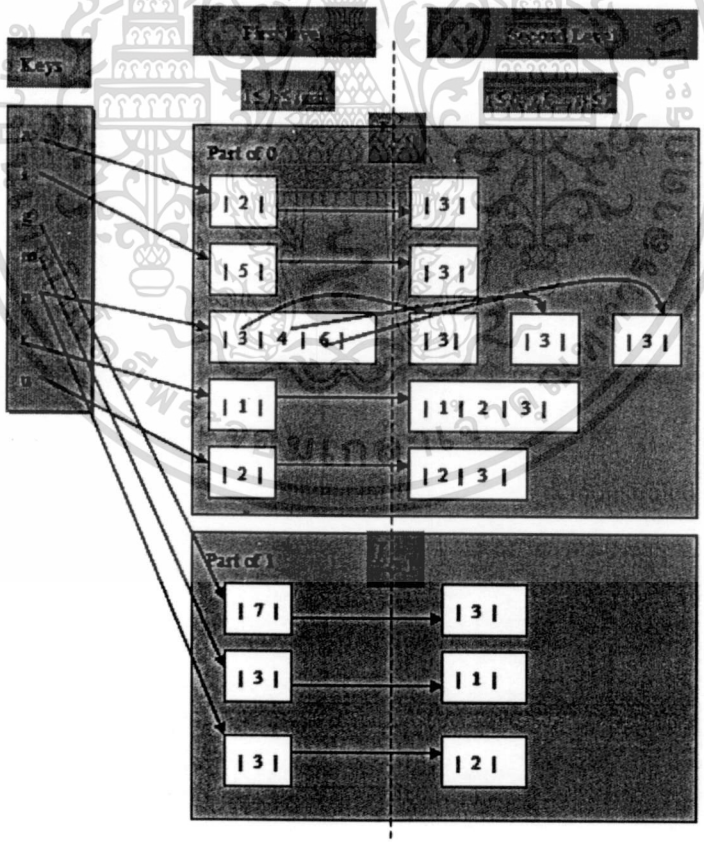


Figure 3.4 The idea of dictionary implementation

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

As depicted in Figure 3.4, the next section shows their related definitions and their examples of these details. Then, all searching algorithm to be explored use this idea to accommodate the dictionary in the preprocessing phase.

3.2 Basic Definitions

In this section, all ideas of inverted lists and all important notations to be used in the next sections are declared. Emphatically, the posting lists are the pairs of indices between all characters in $\sum$ and their positions in P. The individual posting lists are grouped to the form called the inverted lists form. They are defined and carefully exemplified in the light of the significant examples below.

Definition 1. Let $P=\{p^1, p^2, p^3, ..., p^r\}$ be the set of patterns where p^i is the individual pattern i^{th} of m characters performed by $\{c_1c_2c_3...c_m\}$, and $1 \le i \le r$. An individual posting list of a character c_k is defined as c_k:<k:0:i> if $k<m$, or c_k:<k:1:i> if $k=m$. Symbolically, the individual posting list of c_k:<k:0:i> is denoted by $\varphi_0^{k_i}$, and c_k:<k:1:i> is denoted by $\varphi_1^{k_i}$, where $1 \le k \le m$.

Example 3.3 The individual posting lists of $P=\{aab, aabc, aade\}$.

As per the definition above, P can be assigned to the documents as $p^1=a_1a_2b_3$, $p^2=a_1a_2b_3c_4$, and $p^3=a_1a_2d_3e_4$. Thus, they are defined as follows.

p^1 = a:<1:0:1>, a:<2:0:1>, b:<3:1:1>,

p^2 = a:<1:0:2>, a:<2:0:2>, b:<3:0:2>, c:<4:1:2>, and

p^3 = a:<1:0:3>, a:<2:0:3>, d:<3:0:3>, e:<4:1:3>.

It should be noted that all individual posting lists above can be grouped to the new form as a:<1:0:{1,2,3}>, <2:0:{1, 2, 3}>, b:<3:1:{1}>, <3:0:{2}>, and so on. Using this technique, the groups of posting lists are defined by Definition 2 below.

Definition 2. Let l_{max} be the maximum length of patterns in P and ε be the position of unique character λ, which appears in the various patterns of P where $1 \le \varepsilon \le l_{\max}$ and $\lambda \subseteq \sum$. The posting lists of λ are $\{\varphi_0^{\varepsilon_i}, \varphi_0^{\varepsilon_l}, ..., \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ and/or $\{\varphi_1^{\varepsilon_i}, \varphi_1^{\varepsilon_l}, ..., \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ where $1 \le \{i,l,..., p,q\} \le r$. A group of posting lists of λ can be defined in two aspects below.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

1. If the posting lists are $\{\varphi_0^{\varepsilon_1}, \varphi_0^{\varepsilon_1}, \ldots, \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$, then a group of posting lists of λ is denoted by $\lambda_{\varepsilon,0}$.

2. If the posting lists are $\{\varphi_1^{\varepsilon_1}, \varphi_1^{\varepsilon_1}, \ldots, \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$, then a group of posting lists of λ is denoted by $\lambda_{\varepsilon,1}$.

Example 3.4 The posting lists of *P={aab, aabc, aade}* follow Definition 2 and each individual inverted list will be arranged to its own group as follows

Posting lists	$\lambda_{\varepsilon,0}$ / $\lambda_{\varepsilon,1}$
a:<1:0:{1,2,3}>, a:<2:0:{1,2,3}>,	$a_{1,0}$, $a_{2,0}$,
b:<3:1:{1}>, b:<3:0:{2}>,	$b_{3,1}$, $b_{3,0}$,
c:<4:1:{2}>,	$c_{4,1}$,
d:<3:0:{3}>,	$d_{3,0}$,
e:<4:1:{3}>.	$e_{4,1}$.

Definition 3. The inverted lists structure (i.e., IVL) of character λ is defined as $I_{\lambda_{\varepsilon,0}}$ if and only if the group of posting lists is $\lambda_{\varepsilon,0}$. Similarly, the inverted list of alphabet λ is denoted as $I_{\lambda_{\varepsilon,1}}$ if and only if the group of posting lists are $\lambda_{\varepsilon,1}$.

Example 3.5 The groups of posting lists from example 2 are $I_{a_{1,0}}, I_{a_{2,0}}, I_{b_{3,1}}, I_{b_{3,0}}, I_{c_{4,1}}$, $I_{d_{3,0}}$, and $I_{e_{4,1}}$.

Definition 4. The hashing table, which is provided for the storing all alphabets λ and their corresponding inverted lists $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$, is called the inverted lists table and is denoted by τ.

Example 3.6 Table 3.2 shows the table τ of *P={aab, aabc, aade}*.

Table 3.1 Table τ of *P={aab, aabc, aade}*

λ (keys)	Inverted lists	Inverted lists details
a	$I_{a_{1,0}}$, $I_{a_{2,0}}$	*<1:0:{1,2,3}>, <2:0:{1,2,3}>*
b	$I_{b_{3,1}}$, $I_{b_{3,0}}$	*<3:1:{1}>, <3:0:{2}>*
c	$I_{c_{4,1}}$	*<4:1:{2}>*
d	$I_{d_{3,0}}$	*<3:0:{3}>*
e	$I_{e_{4,1}}$	*<4:1:{3}>*

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

For the next implementation, example 3.7 shows the perfect hashing table from section 3.1.

Example 3.7 Table τ when the example 3.5 is implemented.

Table 3.2 The part of table τ

λ (keys)	Part 0	Part 1
a	*<1:0:{1,2,3}>, <2:0:{1,2,3}>*	-
b	*<3:0:{2}>*	*<3:1:{1}>,*
c	-	*<4:1:{2}>*
d	*<3:0:{3}>*	-
e	-	*<4:1:{3}>*

Table 3.3 shows the hashing implementation of inverted lists table 3.2.

Table 3.3 Table τ in the perfect hashing table

λ (keys)	Part 0		Part 1	
	First level (Positions)	Second Level (Pattern numbers)	First level (Positions)	Second Level (Pattern numbers)
a	*1* *2*	1,2,3 1,2,3	-	-
b	*3*	2	*3*	1
c	-		*4*	2
d	*3*	3	-	-
e	-	-	*4*	3

Definition 5. The hashing space, which is provided for any inverted lists $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$, is called the *SET*.

Implementing *SET*, all inverted lists above can be applied and stored in any SET. For instance, if there are the inverted lists *<1:0:{1,2,3}> and <2:0:{1,2,3}>*, then they can be stored in *SET1=<1:0:{1,2,3}>* and *SET2=<2:0:{1,2,3}>*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.3 Dictionary Construction

The dictionary construction relies on the following steps. In the first step, the hashing structure needs to be initiated. The empty table τ must be constructed for all alphabets of $\sum$. Then, all characters of P must be generated into the individual inverted list one by one. Next step is that these inverted lists in each pattern are added to the table τ.

As soon as the empty table τ is built for $\sum$, all patterns are then generated to the inverted lists and are added into the tables. If one or more inverted lists of target character are already stored in the table, only the number of pattern is added to the corresponding inverted lists; otherwise, a new inverted list is created and added into the table. Algorithm 3.1 shows the details of the dictionary construction.

Algorithm 3.1 Pre-processing phase

Input: $P=\{p^1, p^2, ..., p^r\}$

Output: table τ of P

1. **Create** empty table τ
2. **For** $i=1$ **To** r **Do**
3. **For** $j=1$ to m of p^i **Do**
4. **If** τ does not exist $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ **Then**
5. $\tau \leftarrow \varphi_0^{j_i}$ if $j<m$ or $\tau \leftarrow \varphi_1^{j_i}$ if $j=m$
6. **Else**
7. $I_{char(j)_{j,0}} \leftarrow i$ if $j<m$ or $I_{char(j)_{j,1}} \leftarrow i$ if $j=m$
8. **End of If**
9. **End of For**
10. **End of For**
11. **Return** τ

Time and space complexity proofs are commonly described by Lemmas and Theorems. Lemma 1 shows how to get the inverted lists in constant time. Theorem 1, 2, and 3 show the correctness, time, and space proofs, respectively.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Lemma 1. *If there are the inverted lists* $I_{\lambda_{\varepsilon,0}}$ *or* $I_{\lambda_{\varepsilon,1}}$ *of* λ *in* τ*, then access to all inverted lists of* λ *uses O(1) time.*

Proof. Each alphabet λ is a unique character in $\sum$, and λ is implemented as the first level of the perfect hashing table taking *O(1)* time. The inverted lists $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ are implemented as the second level of the perfect hashing table; therefore, each data item takes *O(1)* time, and all items in the second level of the table take *O(1)* as well. □

Theorem 1. *Let* $P=\{p^1,p^2,p^3,...p^r\}$ *be the given patterns, and let* τ *be the provided hashing table for accommodating P. Algorithm 3.1 can generate all patterns of P into the table* τ *correctly.*

Proof. The presented algorithm correctly proves when p^1 to p^r are generated to the inverted lists, and all inverted lists are added to the table τ. The proofs are organized by 1) proving the initial step, 2) proving *for* of the inner loop, and 3) proving *for* of the outer loop. For the initial step, line 1 needs to be true, and the table must be created for running the other steps of the proof.

Regarding the inner loop, the proof is by the induction on j for $j = 1$ to $j = m$. The invariants are still at the end of each j^{th} iteration on $1 \le j \le m$ and $1 \le i \le r$ for $j = 1$ to $j = m$. The pre-condition is that p^i does not exist in the table, and the length of each p^i is m. Also, the variable of m can be changed when each p^i is changed. The post-condition is that each p^i is formed by the sequence of $\{c_1c_2c_3 \ldots c_m\}$. All characters $c_1c_2c_3 \ldots c_m$ are generated to inverted lists and are added to the table. Since the *for* loop is executed by a fixed number, this therefore guarantees the termination of the loop. In the base case, c_1 of p^i is converted to $\varphi_0^{l_1}$ and added to the table as a new inverted list. This result is true, and the invariants remain. Assuming the proposed invariants are true after m-1 iteration, proof can be demonstrated using the two following cases.

In the first case, if there are no inverted lists of p_j^i, then a new inverted list $\varphi_0^{j_i}$ if $j < m$ or $\varphi_1^{j_i}$ if $j = m$ is generated and the table at $I_{char(j)_{j,0}} \leftarrow i$ or $I_{char(j)_{j,1}} \leftarrow i$ is created. Then $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ is stored in the table, and the invariants are unchanged. In the second case, if there are the inverted lists of p_j^i, the number of m-1 is stored in τ. This then implies that the variable j unchanged where $1 \le j \le m-1 \le m$ and $1 \le i \le r$ for *j=1* to *j=m-1*. Also by induction, the variable j and $1 \le j \le m-1 \le m$ and $1 \le i \le r$ for *j=1* to *j=m-1*. Adding $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ to the table implies an iteration of $j = m$ as the hypothesis induction, and the post-condition is shown when c_m is added to the inverted list. In either case the proposed invariants remain and the termination is guaranteed by a fixed number of j; therefore, the inner loop is correct.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The outer loop is proved by induction on *i*. The pre-condition is that there are *P* and τ , the post-condition is all patterns in *P* are generated to the inverted lists and are added to τ . The proposed invariant is $1 \le i \le r$. For the base case, if $i = 1$ then it is true by the inner loop and the proposed invariant $1 \le i \le r$, remains. The inverted lists from c_1 to c_m are followed by the inner loop, and the loop is run on the fixed number of *i*, this also guarantees its termination. In the induction step, the iteration of *i=r-1* must be proved; the pattern p^{r-1} is formed by $\{c_1c_2c_3...c_m\}$, and all of the characters are sent to the inner loop, which are proven true after running the inner loop. The termination is guaranteed by the fixed number of *m*, and when all inverted lists are stored in τ . The invariant still remains while τ stores the inverted lists from pattern p^1 to p^{r-1} after running the inner loop. By induction, the hypothesis is reached, and the correctness is proved. □

Theorem 2. Generating *the patterns* $\{p^1,p^2,p^3,...p^r\}$ *to the inverted lists and adding them into the table* τ *takes* $O(|P|)$ *time where* $|P|$ *is the sum of all pattern lengths.*

Proof. The hypothesis is that all characters of $\{p^1,p^2,p^3,...p^r\}$ are generated to inverted lists, and they are added into τ . Referring to Algorithm 3.1, all pattern lengths are denoted by $|p^1|$, $|p^2|$, $|p^3|$,..., $|p^r|$. For the initial step, the table τ is built in $O(1)$ time. Each round processes the inner loop to execute line 5 or line 7 until they equal the length of each pattern. The summation is $|p^1|+ |p^2|+ |p^3|+... +|p^r|$ which equal $|P|$, and it reaches the hypothesis step by the last character of p^r. Therefore, the inverted lists are constructed in $|P|$ time; this is called $O(|P|)$ time complexity. Meanwhile, line 4, 5, and 7 access the table in $O(1)$ by Lemma 1. Hence, the preprocessing time is proved in $O(|P|)$ time. □

For space complexity, the table τ containing the alphabet λ over a finite alphabet Σ is referred to for the proof. The inverted lists of λ are $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$, and they exist in the second column of τ . The space depends on the number of λ and the posting lists in $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$. However, the table τ only takes O($|P|$) space for accommodating the dictionary.

Theorem 3. *The table* τ *requires* $O(|P|)$ *space for accommodating whole inverted lists* $\{p^1,p^2,p^3,...p^r\}$*; where* $|P|$ *is the sum of pattern lengths.*

Proof. The space is proved when all characters of *P* are generated to inverted lists and are added into the table τ taking $|P|$ space. The pattern lengths in *P* are $|p^1|$, $|p^2|$, $|p^3|$, ..., $|p^r|$, and each p^i contains the sequence string $\{c_1c_2c_3...c_m\}$ which has the length *m*. The length *m* is denoted by p^i.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

In the initial step, the first column of table τ is created for all patterns. Each inverted list is created by the pre-processing phase for all patterns of P; therefore, each inverted list of string $\{c_1c_2c_3...c_m\}$ in each p^i only takes one space per one list. Thus, the space is equal to $|p^1|+|p^2|+|p^3|+...+|p^r|=|P|$ for the second level of the perfect hashing table. As mentioned earlier, the perfect hashing table required $O(n)$ space to accommodate the data items; hence, the maximum space of τ is $O(|P|)$. □

3.4 Pattern Insertion

Obviously, converting and inserting the inverted lists are easy, and is similar to that of the inner loop of Algorithm 3.1. Let p^ϕ be a new pattern which does not appear in τ, and contains the sequence string $\{c_1c_2c_3...c_m\}$. Then all inverted lists of p^ϕ are generated and added into the table in the pre-processing phase. This method is illustrated by Algorithm 3.2.

Algorithm 3.2 Pattern Insertion

Input : $p^\phi=\{c_1c_2c_3...c_m\}$, and τ

Output : Inverted lists of $\{c_1c_2c_3...c_m\}$ are stored in τ

1. **If** p^ϕ does not appear in τ **Then**
2. **For** $j=1$ **To** m **Do**
3. **If** $\varphi_0^{j_\phi}$ or $\varphi_0^{j_\phi}$ does not exist in τ **Then**
4. $\tau \leftarrow \varphi_0^{j_\phi}$ if $j<m$ or $\tau \leftarrow \varphi_1^{j_\phi}$ if $j=m$
5. **Else**
6. $I_{char(j)_{j,0}} \leftarrow \phi$ if $j<m$ or $I_{char(j)_{j,1}} \leftarrow \phi$ if $j=m$
7. **End of If**
8. **End of For**
9. **End of If**
10. **Return** τ

Example 3.8 Inserting the pattern $p=rap$ to the dictionary of $P=\{aab, aabc, aade\}$.

As an illustration, the inverted lists of example 2, the table 3.2 and the table 3.3 are referred to for explaining the insertion methods. For insertion, $p^4=rap$ is the new pattern, and line 1 seeks the existence of p^4. Thus p^4 does not exist in the dictionary and the insertion is started.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The character *r* is considered for generating the new inverted list or adding the position of pattern. In this case, '*r*' and '*p*' do not occur in the table then *r:<1:0:{4}>* and *p:<3:1:{4}>* are added into the table by line 4. If the inverted lists of *'a'* already exist in the table, then the inverted list of *a:<2:0:{4}>* is added by line 6. With the results, only the number 4 is added to τ. Therefore, the final result is *a:<2:0:{1,2,3,4}>*.

Apart from the work performed by the update routine, the insertion algorithm comprises only the pattern to be inserted. Theorem 4 shows the correctness of Algorithm 3.2, and Theorem 5 proves the time complexity of the individual pattern insertion.

Theorem 4. *Let* $p^{\phi} = \{c_1c_2c_3...c_m\}$ *be the new pattern which is not contained in the table* τ. *Algorithm 3.2 inserts the inverted lists of pattern* p^{ϕ} *into the table* τ *correctly.*

Proof. It is crucial for the correctness of the algorithm that the following property holds when all inverted lists of p^{ϕ} are added into the existing table τ. Let τ_{old} be the table before adding the new pattern, and let τ_{new} be the table after adding the new pattern.

The pre-condition is that τ_{old} contains the inverted lists $|P|$, and the post-condition is $|P|+|p^{\phi}|(\tau_{new})$. The invariants are $1 \le j \le m-1 \le m$ and $1 \le i \le r$ for $j=1$ to $j=m$ where m is the length of $\{c_1c_2c_3...c_m\}$. The proof is by induction on j as the inner loop of Algorithm 3.1.

Obviously then, the pattern p^{ϕ} is similar to the pattern p^i in P. Algorithm 3.2 is run as the inner loop of Algorithm 3.1. Therefore, the proof in the loop of Algorithm 3.2 is claimed as well. Also, the invariants remain because there is nothing to change them. Hence, the post-condition shown at c_m, and τ_{new} is finally shown. □

Theorem 5. *Inserting* $p^{\phi} = \{c_1c_2c_3...c_m\}$, *which does not appear in the table* τ, *takes* $O(|p|)$ *where* $|p|$ *is the length of* p^{ϕ}.

Proof. Algorithm 3.2 is referred to for straightforward proof. The length of p^{ϕ} is m and is denoted by $|p|$. The loop *for* reads all characters and converts them to the inverted lists. Then each individual inverted list is added into τ one by one. This loop creates the inverted lists from c_1 to c_m, and it takes m operations. Thus, $O(|p|)$ time is shown and then the hypothesis is also proved after c_m operated. The other lines (3, 4, 6) access the table taking $O(1)$ by Lemma 1. Therefore, to insert all characters of p^{ϕ} into the existing dictionary takes only $O(|p|)$ time. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.5 Pattern Deletion

There are two methods for deleting the pattern in this phase. The first is by looking for the required pattern to be deleted, and the second is by removing the inverted lists of the target pattern from the dictionary.

For the sake of simplicity, assuming that p^{σ} is the existing pattern in τ, and p^{σ} is formed by the sequence string $\{c_1c_2c_3...c_m\}$, then all characters from p^{σ} are read one by one, and each inverted list is removed from the dictionary. Driving the deleting mechanism, if the corresponding inverted list exists in only one posting list, it will be immediately deleted. Otherwise, only the inverted lists where the pattern number equals σ are deleted. The method is described by Algorithm 3.3.

Algorithm 3.3 Pattern Deletion

Input: $p=\{c_1c_2c_3...c_m\}$, and τ

Output : Inverted lists of p are deleted.

1. **If** *ExistDel(p)*= σ **Then**
2. **For** j=1 To *m* **Do**
3. **If** posting lists char(p_j^{σ}) in $\varphi_0^{j_\sigma}$ or $\varphi_1^{j_\sigma} > 1$ **Then**
4. Delete the posting lists equal σ
5. **Else**
6. Delete $\varphi_0^{j_\sigma}$ if $j<m$ or $\varphi_1^{j_\sigma}$ if $j=m$ of char(p_j^{σ})
7. **End of If**
8. **End of For**
9. **End of If**
10. **Return** τ

Example 3.9 Taking *p=aab* off *P={aab, aabc, aade}*.

For deletion, table 3.2 is referred to the existing dictionary in *P*. Line 1 inspects the existence of pattern and returns the number σ for forwarding the deletion. The mechanism of line 1 finds the pattern *aab* and the result is the pattern number *1*. The pattern *aab* is formed to *a:<1:0:{1}*, *a:<2:0:{1}>*, and b:*<3:1:{1}>*. Line 4 takes the inverted list of *a:<1:0:{1}>* from *a:<1:0:{1,2,3}>* then the result is *a:<1:0:{2,3}>*. Also, when *a:<2:0:{1}>* is taken from

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

a:<2:0:{1,2,3} then the result is *a:<2:0:{2,3}>;* meanwhile, the inverted lists of *b:<3:1:{1}>* are removed from the table by line 6.

Theorem 6. *Deleting the existing pattern* $p^{\sigma} = \{c_1c_2c_3...c_m\}$ *from the existing table* τ *by Algorithm 3.3 is correct.*

Proof. τ contains the inverted lists of *P* with the size |*p*|. Let τ_{aft} be the inverted lists table after deleting the pattern $p^{\sigma} = \{c_1c_2c_3...c_m\}$, and the size of τ_{aft} is $|P| - |p^{\sigma}|$ where σ is the pattern number which appears in the table. The proof needs to show all inverted lists of $\{c_1c_2c_3...c_m\}$ that are removed from the table. The pre-condition is that p^{σ} exists in table τ, and the post-condition is τ_{aft}. The proposed invariant is $1 \le j \le m$ for *j=1* to *j=m*.

For an illustration of this proof, the explanations are by induction on *j*. The base case is in $j = 1$ and the character c_1 is converted to the inverted list. Then, the inverted list $\varphi_0^{j\sigma}$ is formed by line 3. The proof needs to show both conditions of *if*. In the first case, if the number of posting list c_1 in the table is more than 1, then the number of $\varphi_0^{j\sigma}$ is removed from τ. In the second case, if there is only one inverted list in τ, then $I_{char(j)_{j,0}}$ is removed by line 5. Thus, $\varphi_0^{j\sigma}$ is removed from the table after the first iteration, and the size of the table is decreased by 1. In both cases, the invariant $1 \le j \le m$ remains. According to the fixed number of loops, the termination of loop is guaranteed by the value of *m*.

In the inductive step, the invariant needs to be true after the iteration of $j = m-1$. The character of c_{m-1} is created as φ_0^{m-1}. If the number of inverted lists of c_{m-1} is more than 1 then the number of φ_0^{m-1} is removed from τ. If there is only one inverted list then $I_{char(m-1)_{m-1,0}}$ is removed. The invariant still remains. The number of inverted lists in the table equals $|P| - |p^{\sigma} - 1|$ while $1 \le j \le m-1 \le m$ for *j=1* to *j=m*, and τ_{aft} is shown. By induction, the hypothesis is implied. Therefore, Algorithm 3.3 is correct. □

Theorem 7. *Deleting the pattern* p^i *from the dictionary of P takes O(|p|) time where* p^i *is the target pattern to be deleted and |p| is the length of pattern* p^i.

Proof. Assuming that p^i is the existing pattern to be deleted, and p^i is formed by the sequence string $\{c_1c_2c_3...c_m\}$. The length of p^i is *m* and is denoted by |*p*|, and *i* is the number of the pattern i^{th} in *P*. The hypothesis is that all inverted lists of p^i are removed from the dictionary of *P*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The deletion process repeats to remove the inverted lists from c_1 to c_m. Each operation for accessing the inverted list uses *O(1)* by Lemma 1. The operations remove the matched inverted lists from the table one by one. All operations take $|p|$ time while line 4 or line 6 takes the constant time according to Lemma 1. The time to create inverted lists and scan the proposed pattern are the same as the length of pattern. Thus, to delete all characters of pattern p^i from the dictionary P takes $|p|$ which is $O(|p|)$ time. □

3.6 Exhaustive Inverted Lists Dictionary Matching Algorithm: E-IVL

This solution is an original approach which works as a naïve search, which is the first view point of applying the inverted lists structure. The algorithm initiates a window search at the beginning of a given text after the preprocessing phase. The character comparisons are done by converting each character in the given text to inverted lists and accessing the perfect hashing table, while scanning from left to right. The new window search uses the special factor of the shift variable that can be repositioned only once in the given text. With this method, the comparison and the text scanning time are redundant.

For convenience, the search employs the variables '*N*', '*SHIFT*', '*pos*' and '*n*' to propel the searching window where '*N*' is the target position in the text, '*SHIFT*' is the initial position of the next searching window, '*pos*' is the required position of inverted lists to be matched, and '*n*' is the length of the text *T*. In addition, '*SET1*' and '*SET2*' are the temporary variables used to operate the continuity and the matching during the search.

In initial searching, the variables *N* and *SHIFT* are initiated to enforce the searching window, and the variable '*pos*' is used to control the required position in the text *T*. Afterwards, the text is scanned and searched from the left to the right. The target of the scan is the inverted lists in τ. The positions equal '*pos*' at the row of λ by *text[N]*, storing to *SET1* or *SET2*. Algorithm 3.4 illustrates this methodology.

Algorithm 3.4 Exhaustive Inverted Lists Dictionary Matching Algorithm: E-IVL

Input : $P = \{p^1, p^2 p^3, \ldots, p^r\}(\tau)$ and $T = t_1 t_2 t_3 \ldots t_n$

Output : all occurrences are reported, and *T* is scanned.

1. *N=1, SHIFT=2, pos=1, SET1=SET2={}, RESULTS={}*

2. **While** *N* <= *n* and *SHIFT* <= *n* **Do**

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3. **If** *pos = 1* **Then**
4. *SET1*⟵ τ *(text[N], 1)*
5. **Else**
6. *SET2*⟵ τ *(text[N], pos)*
7. **End of If**
8. *SET1* ⟵ *SET1* ∩ *SET2*
9. **Store** the matched position to *RESULTS* if *SET1* contains $\varphi_{(N^{pos}),1}$
10. **If** *SET1*<>*{}* **Then**
11. *N*++ and *pos*++
12. **Else**
13. *N = SHIFT, SHIFT* ++ and *pos = 1*
14. **End of If**
15. **End of While**
16. **Return** *RESULTS*

With careful attention to Algorithm 3.4, the special function, which drives the continuity of search, is *INTERSECTION*(∩). This function considers the position in *SET1* and *SET2* and searches for the matched positions and returns the continuity of inverted lists into *SET1* for the next comparison. The intersection function can be efficiently implemented through a simple procedure, which is indicated in an Algorithm 3.5.

Algorithm 3.5 *INTERSECTION (SET1, SET2, N, pos)*

1. Report the successful matching at *N* if *IVL* in *SET2* containing $\varphi_1^{(pos)}$
2. Add every *IVL* in *SET2* that are continued (i.e., $\varphi_0^{(pos)}$) from *SET1* to *TEMP*
3. Return *TEMP*

Every comparison takes the inverted lists from the table τ to the sub-hash variable *SET1* or *SET2*. Whenever the inverted lists are taken, the *INTERSECTION* is invoked to operate the continuity and occurrence of patterns. For instance, if *SET1={<1:0:{1,2}>}* and *SET2={<2:0:{1,3}>}* operate, then the intersection is ordered by the positions *1* to *2* between *SET1* and *SET2*. In this case, the first consideration is by the sequence of inverted lists in *SET1* which are

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

described by *SET2*. Thus, the pattern number *{1}* in *SET1* is described by position *{1}* in *SET2*, while the required position is '*2*' in *{<2:0:{1,3}>}*. If the inverted lists are considered, the indicated number '*0*' and '*1*' are also considered, and the occurrence is reported if the indicated number is '*1*'. Consequently, the indicated number of *SET2* is *{<2:0:{1,3}>}*, which is not the last character of the pattern: it does not match at this position. Therefore, the result is *SET1={<2:0:{1}>}*.

Lemma 2. If *SET1* and *SET2* are the instances of *SET* as in Definition 5, then *SET1*← *SET1* ∩ *SET2* is correct.

Proof. The substantiation is that all inverted lists in $\varphi_0^{(pos)}$ or $\varphi_1^{(pos)}$ of SET2 which continue from *SET*1 are to *SET*1. The pre-conditions are that $N \leq 2$ and the inverted lists be stored in *SET*1 and *SET*2. The post-condition is that all inverted lists of $\varphi_0^{(pos)}$ or $\varphi_1^{(pos)}$ in *SET*2, which continue from *SET*1, are returned to *SET*1.

The required position is *pos*, and the continuity is that all inverted lists $\varphi_0^{(pos-1)}$ or $\varphi_1^{(pos-1)}$ of *SET*1 are described by $\varphi_0^{(pos)}$ or $\varphi_1^{(pos)}$ in *SET*2. *SET*1 and *SET*2 contain any $I_{\lambda_{\varepsilon,0}}$ *or* $I_{\lambda_{\varepsilon,1}}$ of the hashing set in definition 4 and 5, and they are the second level of the perfect hashing table. It can be said that every inverted list of *SET*2 must be inspected and compared with the inverted lists in *SET*1 by the properties of intersection. Thus, the results are $\varphi_0^{(pos)}$ and/or $\varphi_1^{(pos)}$ and φ_0^1. The post-condition is then fulfilled. □

Example 3.10 Searching *P={aab,aabc,aade}* in the given text *T=thisisaabcnd*. The search is initiated with the variables in Figure 3.5, and table 3.2 is referred to for the inverted lists structure.

N=1, SHIFT=2

T	t h i s i s a a b c n d	
	1 2 3 4 5 6 7 8 9 10 11 12	
p^1	a a b	pos=1
p^2	a a b c	SET1={}
p^3	a a d e	SET2={}

Figure 3.5 Initiating the variables of searching window.

1. Compare the inverted lists with *text[1]='t'*. Set *SHIFT=2* and *pos=1*, and the results of *SET1* and *SET2* are *{}*.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2. Skip to *text[2]='h'*. Set *SHIFT=3*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
3. Skip to *text[3]='i'*. Set *SHIFT=4*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
4. Skip to *text[4]='s'*. Set *SHIFT=5*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
5. Skip to *text[5]='i'*. Set *SHIFT=6*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
6. Skip to *text[6]='s'* . Set *SHIFT=7*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
7. Skip to *text[7]='a'*. Set *SHIFT=8*, *pos=1*, and the result of *SET1= {<1:0:{1,2,3}}* and *SET2={}*.
8. Skip to *text[8]='a'*. Set *SHIFT=8*, *pos=2*, and the result of *SET1= {<1:0:{1,2,3}}* and *SET2={<2:0: {1,2,3}>}*. Then after operating *SET1* and *SET2*, *SET1={<2:0:{1,2,3}>}*.
9. Skip to *text[9]='a'*. Set *SHIFT=8*, *pos=3*, and the result of *SET1= {<2:0:{1,2,3}>}* and *SET2={<3:1: {1}><3:0:{2}>}*. Then after operating SET1 and SET2, *SET1={<3:0:{2}>}*, and the matched pattern *{aab}* is reported.
10. Skip to *text[10]='a'*. Set *SHIFT=8*, *pos=4*, and the result of *SET1={<3:0:{2}>}* and *SET2={<4:1:{2}>}*. After operating *SET1* and *SET2*, *SET1={}*, and the matched pattern *{aabc}* is reported.
11. Go to *text[8]='a'*. Set *SHIFT=9*, *pos=1*, and the result of *SET1={<1:0:{1,2,3}}* and *SET2={}*.
12. Skip to *text[9]='b'*. Set *SHIFT=10*, *pos=2*, and the result of *SET1={<1:0:{1,2,3}}* and *SET2={}*. After operating *SET1* and *SET2*, *SET1={}*.
13. Skip to *text[10]='b'*, Set *SHIFT=11*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
14. Skip to *text[11]='c'*. Set *SHIFT=12*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
15. Skip to *text[12]='n'*. Set *SHIFT=13*, *pos=1*, and the result of *SET1* and *SET2* are *{}*.
16. Skip to *text[13]='d'*. Set *SHIFT=14*, *pos=1*, and the result of *SET1* and *SET2* are *{}* and the search is finished.

It should be noted that comparisons of the text were replicated at positions 8, 9, and 10, where the time to scan the characters in the text was more than the length of *T* by 4 times. This number is called '*locc*' and denoted as the number of the matched characters. This includes the mismatched time. However, many redundant symbol comparisons might be scanned, especially if there are several overlapping patterns.

The complexity of Algorithm 3.4 is dominated by the variables *SET1* and *SET2*. Definition 5 is the additional property for the proofs. The variable *SET* is an instance of the row

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

of τ and contains only one row of inverted lists. It can be said that *SET* is the sub-hash table of table τ, which are *SET1* and *SET2*. The time required for the best intersection is proportional to *O(1)* time, which is worked with hashing properties. Lemma 3 shows this operation.

Lemma 3. *The intersection between SET1 and SET2 takes O(1) time.*

Proof. Let *SET1* and *SET2* be the instances of *SET*. *SET1* contains the inverted list groups $I_{\lambda_{\varepsilon 1,0}}$ and/or $I_{\lambda_{\varepsilon 1,1}}$, and *SET2* contains the inverted list $I_{\lambda_{\varepsilon 2,0}}$ and /or $I_{\lambda_{\varepsilon 2,1}}$. Then every operation of *SET* can be solved by Lemma 2 in O(*1*) time. Hence, every operation to access $I_{\lambda_{\varepsilon 1,0}}, I_{\lambda_{\varepsilon 1,1}}, I_{\lambda_{\varepsilon 2,0}}$ and $I_{\lambda_{\varepsilon 2,1}}$ takes O(*1*) time by Lemma 1. □

As shown above, the intersection between *SET1* and *SET2* finds a consecutive set of numbers in *SET2* that originated from *SET1*. Importantly, this method reports the matched position whenever the terminate status equals *1*. The continuity is concentrated on the posting lists in *SET1*, described by *SET2*. If the numbers of positing lists in *SET2* are superior to *SET1* in one position, they are kept in *SET1* for the next operation.

Theorem 8. *Algorithm 3.4 is correct for searching P in the given text T.*

Proof. The proof is in the induction on n for t_1 to t_n. The proposed invariants are $1 \le N \le n$, $1 \le pos \le l_{max}$, and $1 < SHIFT \le n$. Pre-conditions are that τ is computed and *RESULTS* is empty. Post-conditions are that *T* is scanned and all occurrences are stored in *RESULTS*.

The algorithm is worked by assuming line 1 is true. The proof needs to show all invariants on variable *n*, which explain the *while* loop (line 2). The next step needs to prove two conditions of *else* (line 3 and line 10).

On line 3, if *pos=1*, then the inverted lists are taken to *SET1*. In this case, all invariants are unchanged; therefore, the algorithm is true. In the case of *else*, the inverted lists are put into *SET2* and the invariants remain; hence, it is true.

Line 8 is always true by Lemma 2; in addition, line 9 does not relate to the invariants. Therefore, all of them are unchanged. The algorithm is still true because this line only inspects and puts $\varphi_1^{(pos)}$ into *RESULTS*.

The second condition of *if* states that every invariant is still true when *SET1={}* , *N++*, and *pos++*. The invariants are unchanged until the *while* loop is checked in line 2, which does not

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

change all invariants after increasing those variables. This case is continued whenever the inductive step is not yet reached; in addition, $1 \leq N \leq n$, $1 \leq pos \leq l_{max}$, and $1 < SHIFT \leq n$ remain. In contrast, if the *else* condition is activated, then *SET1!={}*, *N=SHIFT*, *SHIFT++*, and *pos=1*. In this case, *pos* and *N* are always within invariants if and only if $SHIFT \leq n$. All invariants depend on the *while* loop of line 2 which is run in a fixed number. *SHIFT* is increased one by one and *N=SHIFT*; then, each *SHIFT* will reach the inductive step by a fixed number of *n* and all invariants will remain.

In the inductive step, the iteration *n*-1 needs to be proven when the case of *SET*1 *!= null*. This algorithm runs the variable *n* by the fixed number and then the iteration *n* - 1 is reached. All iterations from t_3 to t_{n-1} are true; reporting the matched patterns in line 4 proves in both the correctness and the pattern continuity. It can then be claimed the iteration t_n is true by induction and the correctness of the algorithm is also proven. □

Theorem 9. *Searching for all occurrences of patterns in* $P=\{p^1, p^2, p^3,...,p^r\}$ *which occur in the text* $T=\{t_1t_2t_3...t_n\}$ *takes O(n+locc) time where n is the length of T, and locc is the numbers of matched characters, which includes the mismatched time.*

Proof. The hypothesis is that all characters of $t_1t_2t_3...t_n$ are scanned, and all matched patterns are reported. The initial step takes O(*1*) time by line 1. The time complexity is dominated by the variables *SHIFT*, *N*, *SET1*, and *SET2*, and these following cases give an explanation of the time complexity.

In the first case, loop *while* is run from t_1 to t_n. All operations are dominated by the variable *N* and *SHIFT*, and the variable *SHIFT* orders to inspect all characters in the text *T*. It can be said that line 3 takes O(*n*) time because this step is processed from the initial step to *n* times.

In the second case, the variable *N* drives line 4 and line 6 to operate in *locc* time at most. Each domination of *N* drives line 4 and line 6 to take O(*1*) time by Lemma 1. This stimulates line 10 to equal *locc* time as well. However, each operation of line 8 takes O(*1*) time by Lemma 3. The variable *N* orders the loop and returns to line 3, and at most equals the number of the characters to be matched with the inverted lists in the table τ. This takes *locc* time. Line 3 and line 10 take a constant time to control the other steps. Thus, the hypothesis is reached by line 13 and line 6, and the searching time is computed in O(|*t*|+*locc*) time. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.7 Prefix Inverted Lists Dictionary Matching Algorithm: PF-IVL

The motivation for this algorithm is to avoid scanning the previous solution; in particular, overlapping patterns. This solution scans the given text in a single pass even in the worst case scenario, which equals the Aho-Corasick solution [1]. As noted by section 3.6, the previous solution needs to backtrack while scanning text because the shifting value is fixed for next search window. The prefix inverted lists search avoids initiating a new search window in the backward position. This algorithm scans all characters of *T*; meanwhile, matching is checked and intersection operations and the overlap handling are activated in the same time.

Before describing the searching methodology in depth, this section refers back to the basic definitions which are used for running the search algorithm. Let *N* be the target position in the given text to be compared; *pos* is the required position of the inverted lists to be matched; and *n* is the length of the text *T*. In addition, *SET*1 and *SET*2 are the variables that are operated for continuity during the search. *RESULTS* is the special variable for keeping the pattern occurrences.

Assume for simplicity that the variables *N pos*, *SET*1, and *SET*2 are set to enforce the searching window, and the variable *pos* is used to control the required position in the text *T*. This search is based on reading from left to right along the text *T* in single round. While reading, the inverted lists that equal *pos* in the row of *text*[*N*] are taken to *SET*1 or *SET*2. When considering continuity, the intersection is used for checking patterns in *SET*1 and *SET*2. The intersection between *SET*1 and *SET*2 finds a set of numbers in *SET*2 that originated from *SET*1. Importantly, it reports the matched position whenever the terminate status equals *1*. The continuity is concentrated on the posting lists in *SET*1 that are described by *SET*2, as well as in previous sections. If the numbers of positing lists in *SET*2 are superior to *SET*1, these are kept in *SET*1 for the next operation. Thereby, the indicated number '*0*' or '*1*' of *SET*1 is considered for reporting all occurrences. If the indicated number in *SET*1 is '*1*', then the matched position is reported, and the matched result is stored into *RESULTS*.

In the case of the overlapping patterns, the inverted lists which are equal <1 : 0 : *{. . .}* > must be attached to *SET*2 when accessing the inverted lists of any positions. For instance, if the patterns are '*ram*' and '*amazing*', the inverted lists of '*a*' are *<2:0:{1}>* and *<1:0:{2}>*. In this case they are taken together when the character '*a*' in the given text is scanned.

To sum up, this solution works in a linear searching time. An algorithm, an illustrative example, the proofs of the correctness and the proof of time complexity are shown as below, respectively.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Algorithm 3.6 Prefix Inverted Lists Dictionary Matching Algorithm: PF-IVL

Input : $P=\{p^1, p^2 p^3, \ldots, p^r\}(\tau)$ and $T = t_1 t_2 t_3 \ldots t_n$

Output : all occurrences are reported, and T is scanned.

1. $N = 1$, $pos = 1$, $SET1 = SET2 = null$, $RESULTS = \{\}$
2. $SET1 \leftarrow (IVL(text[N]), pos), N++$
3. **While** ($N <= n$) **Do**
4. **Store** the matched position into *RESULTS* set if *SET*1 contains φ^{pos},1
5. **If** $SET1 <> null$ **Then**
6. $pos++$
7. $SET2 \leftarrow (IVL(text[N]), pos \text{ or } 1)$
8. $SET1 \leftarrow SET1 \cap SET2$
9. **Else**
10. $pos = 1$
11. $SET1 \leftarrow (IVL(text[N]), pos)$
12. **End of If**
13. $N++$
14. **End of While**
15. **Return** *RESULTS*

Example 3.11 Searching *P={aab,aabc,aade}* in the given text *T=aabcdgaadef*

1. Set the search to the initial Step that *N=1*, *SET1=SET2=RESULTS={}*.
2. Skip to the line 2 and *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}*, and *N=2*.

a	a	b	c	d	g	a	a	d	e	f
1	2	3	4	5	6	7	8	9	10	11

3. Take the first loop '*while*', *pos=2*, and *SET2={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a	a	b	c	d	g	a	a	d	e	f
1	2	3	4	5	6	7	8	9	10	11

SET1 ← *INTERSECTION (SET1, SET2, N, pos)* thus *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}* and *N=3*.

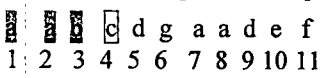
4. Skip to the next loop of *while*, *pos=3*, *SET2={<3:1:{1}><3:0:{2}>}*

a	a	b	c	d	g	a	a	d	e	f
1	2	3	4	5	6	7	8	9	10	11

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

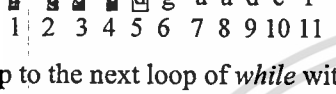
SET1 ← *INTERSECTION(SET1, SET2, N, pos)* thus *SET1={<3:1:{1}><3:0:{2}>}* and '*b*' is matched at *<3:1:{1}>* in the pattern 1. After matching report, set *N=4* and *SET1={<3:0:{2}>}*.

5. Skip to the next loop of *while*, *pos=4*, and *SET2={<4:1:{2}>}*.

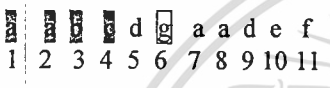


SET1 ← *INTERSECTION(SET1, SET2, N, pos)* thus *SET1={<4:1:{2}>}* and '*c*' is matched at *<4:1:{2}>* in the pattern 2. After matching report, set *N=5* and *SET1={}*, and keep the result into *RESULTS*.

6. Skip to the next loop of *while* with the condition of *else* that *pos=1*, *SET1={}*, and *N=5*.

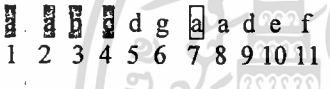


7. Skip to the next loop of *while* with the condition of *else* that *pos=1*, *SET1={}*, and *N=6*.

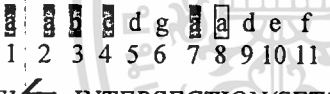


8. Skip to the next loop of *while* with the condition of *else* that *pos=1*,

SET1={<1:0:{1,2,3}><2:0:{1,2,3}, and *N=7*.



9. Skip to the next loop of *while* with *pos=2*, *SET2={<1:0:{1,2,3}><2:0:{1,2,3}*, and *N=8*.

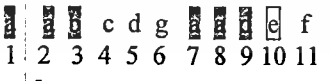


SET1 ← *INTERSECTION(SET1, SET2, N, pos)* thus *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}* and *N=9*.

10. Skip to the next loop of while, and set pos=3, SET2={<3:0:{2}>}, and N=8.

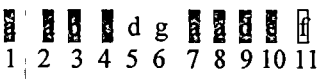
SET1 ← *INTERSECTION(SET1, SET2, N, pos)* thus *SET1={<3:0:{3}>}* and *N=10*.

11. Skip to the next loop of *while*, and set *pos=4*, *SET2={<4:1:{3}>}*, and *N=10*.



SET1 ← *INTERSECTION(SET1, SET2, N, pos)* thus *SET1={<4:1:{2}>}* and the '*e*' is matched at *<4:1:{3}>* in the pattern 3. After the matching report, set *N=11* and *SET1={}*, and keep the result in *RESULTS*.

12. Skip to the next loop of *while*, and it takes the condition of else that *pos=1*, *SET1={}*, and *N=11*.



Therefore, *N=12* and *N>n*, and the searching is finished.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Theorem 10. *Algorithm 3.6 is correct for searching P in the given text T.*

Proof. The correctness of the algorithm is easily verified. The crucial point is that the induction on n for t_1 to t_n. The proposed invariants are $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$.

Assuming that line 1 and line 2 are true; then the base case, the variable N is increased by 1 before getting to the *while* loop. This step needs to be proved in the case of *SET*1 <> *null* and *SET*1 = *null*. Then if *SET*1 <> *null* the inverted lists of *text*[*N*] are taken to *SET*2, and $1 \leq pos \leq$ *lmax*, the correctness is proved by the intersection in Lemma 1. If *SET*1 = *null*, then the *text*[*N*] is taken to *SET*1. After the end of this iteration $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$. In both cases, the invariants remain unchanged and thus this step is true. In the inductive step, the iteration n-1 needs to prove when the case of *SET*1 <> *null*. This algorithm runs the variable n by the fixed number and then the iteration n - 1 is reached. All iterations from t_3 to t_{n-1} are true by reporting the matched patterns in line 4 which prove both the correctness and the pattern continuity. It can then be claimed the iteration t_n is true by induction and the algorithm is correct. □

Theorem 11. *Searching the occurrences of* $P = \{p^1, p^2, p^3, \ldots, p^r\}$ *which appear in the given text* $T = \{t_1 t_2 t_3 \ldots t_n\}$ *takes* $O(|t|)$ *time where* $|t|$ *is the length of T.*

Proof. The proof is that all characters of $t_1 t_2 t_3 \ldots t_n$ are scanned, and all occurrences are reported in n time. Referring back to the searching algorithm, the time complexity is dominated by the variables *N*, *SET1*, and *SET2*. The while loop in line 3 is repeated to inspect the inverted lists of t_2 to t_n. Each iteration of the loop definitely stores all occurrences in line 4 with *O(1)*. It can be said that the loops of line 3 take $O(|t|)$ time because this step is processed from the initial step to $|t|$ time. Meanwhile, line 5, 6, and 8 take a constant time by Lemma 1 and Lemma 3, respectively. Therefore, the time complexity takes only $O(|t|)$ time. Also, this algorithm is able to perform in both an average case and a worst case scenario. Beside, the best case shows this complexity as well. □

3.8 Suffix Factor Inverted Lists Dictionary Matching Algorithm: SF-IVL

This solution is the best approach shown in this thesis. It utilizes the inverted lists of each comparison to avoiding redundance of comparison. The window search is initiated, as well as the previous solutions. The comparison begins at the end of search window, which is set by the

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

minimum pattern length. If the last character in that the search window is matched, then the text scanning will be worked from the beginning of the search window. With this method, each attempt can scan beyond the current window search, whilst the shifting value is always set at least two-fold of the minimum pattern length. This is certainly optimal in many common cases; in particular, in the cases of non-patterns overlapping, which often leads to better (or at least good) performance.

In granular search, the variables *N*, *SET1*, *SET2*, *SETE* and *SHIFT* are initiated to enforce the searching window. The variable *min_length* is the minimum length of patterns in *P*. Each comparison takes the inverted lists from the table τ to *SET1*, *SET2*, or *SETE*, which are the hashing set. Then, the continuity and the occurrences are analyzed by considering the current position (defined as *pos*). The searching phase works as described in Algorithm 3.7. Initially, the variables are set in three spots as shown Figure 3.6.

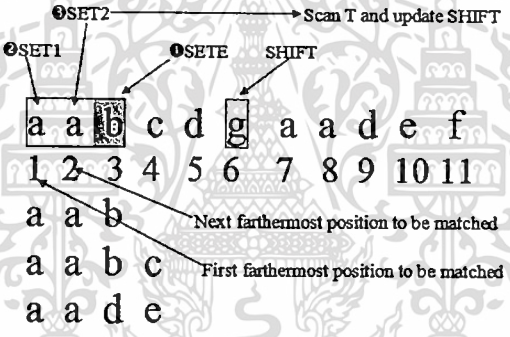


Figure 3.6 Initiating the first window search.

Algorithm 3.7 Suffix Factor Inverted Lists Dictionary Matching Algorithm: SF-IVL

Input : $P=\{p^1, p^2, p^3, ..., p^r\}$ and $T=t_1t_2...\ t_n$

Output: occurrences are reported.

1. *N=min_length, SHIFT=2x(min_length), SET1={}, SET2={}, SETE={}, RESULTS={}*
2. **While** *(N<=n)* and *(SHIFT<=n+min_length)* **Do**
3. *SETE* ← *IVL(text[N])*
4. **If** *SETE!={}* **Then**
5. *N=farthermost position from SETE*
6. *SET1* ← *IVL(text[N])*
7. **While** (*SET1!={}*) and *(N<n)* **Do**

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

8. **If** *N=position of SETE* **Then**
9. *SET1* ←*SET1* ∩ *SETE*
10. **Else**
11. *SET2*← *IVL(text[N])*
12. *SET1*← *SET1* ∩ *SET2*
13. **End of If**
14. **Store** φ_1^{pos} to in *SET1* to *RESULTS*, and remove it
15. *N* ← *N+1*
16. **If** (*N>=pos of SETE*) and (*SET1!={}*) **Then**
17. *SHIFT* ← *SHIFT+1*
18. **End of If**
19. **If** (*SET1={}*) and (*N=position of SETE*) **Then**
20. Check the overlapping patterns and add them to *SET1*
21. **Else**
22. *SET1=Next farthermost(SETE)*
23. **End of If**
24. **End of While**
25. **End of If**
26. *N=SHIFT,SHIFT=SHIFT+min_length, SET1={}, SET2={}, SETE={}*
27. **End of While**
28. **Return** *RESULTS*

Firstly, the variables *N* and *SHIFT* are initialized to enforce the searching window. Then, the given text is scanned from left to right along the text *T*. While scanning, the inverted lists at the row of *text[N]* are taken to *SET1* or *SET2* or *SETE* and the intersection is activated.

Secondly, the intersection ∩ is used for considering the continuity of the patterns in *SET1* and *SET2* even in *SETE*. Afterwards, the continuity of the inverted lists are returned and stored to *SET1*. The next comparison takes *SET1* for the next inspection with the next farthermost position.

Every matched comparison takes the inverted lists from the table τ to *SET1* or *SET2*. Whenever the inverted lists are taken, then the intersection of *SET1* and *SET2* is started for analyzing the continuity of the pattern. For instance, supposing that *SET1={<1:0:{1,2}>}* and

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

SET2={<2:0: {1,3}>} then the intersection is ordered by the position *1* to *2* between *SET1* and *SET2*. In this case, the first consideration is the sequence of inverted lists in *SET1* that are described by *SET2*. Thus, the pattern number *{1}* of *SET1* is described by the position of *{1}* in *SET2* while the required position is '*2*' in *{<2:0:{1,3}>}*. Therefore, the result is *SET1={<2:0:{1}>}*. For reporting the occurrences, line 14 of Algorithm 3.7 considers the indicated number '*0*' or '*1*' of *SET1*. If the indicated number in *SET1* is '*1*', then the matched position is reported. In this case, *SET1={<2:0:{1}>}* is not the last character of pattern, and it is not matched.

For the pattern overlapping, the inverted lists that equal *<1:0:{...}>* must be attached with *SET2* as well. For instance, if the patterns are '*ram*' and '*amazing*', the inverted lists of '*a*' are *<2:0:{1}>* and *<1:0:{2}>*. In this case they are attached together when the character '*a*' in the given text is scanned. One advantage of this strategy leads to an optimal scanning time because the algorithm deduces only all characters to be compared in the given text.

The rest of this section is devoted to prove the correctness and the time complexity, including an example illustration.

Theorem 12. *Algorithm 3.7 is correct for searching P in the given text T.*

Proof. Algorithm 3.7 needs to prove the corrections of two *while* (line 2 and line 7) and three conditions of *if* (line 8, line 16, and line 19). Considered the *while* loops, line 2 and line 7 need to show all invariants, and all post-conditions are reached. Line 2 has pre-condition in line 1 of algorithm and its post-conditions are the output when *N>n* and *SHIFT>n+min_length*. Meanwhile, the invariants are *N<=n*, *SHIFT<=n+min_length*. The pre-conditions of line 7 are *SETE* and *SET1*!={}, $1 \le N \le n$ and its post-conditions are *SET1={}*. If there are the matched positions, then they will be reported. The invariants of line 7 are *N<=n*, *SHIFT<=n+min_length*, and *SET1* and *SETE!*={}. The pre-conditions and the post-conditions of all *if* conditions are explained as follows.

Operating line 4, the pre-conditions are *SETE!={}*, *SET1!={}*, and the post-conditions are all matched positions to be reported, and *SET1={}*. In line 8, the pre-conditions are *N=position of SETE*, and *N<=post of SETE*, and the post-conditions are *SET1=SET1* $\cap$ *SET2* or *SET1=SET1* $\cap$ *SETE*. In line 16, the pre-conditions are *N>position of SETE*, *SET1!={}*, and the post-conditions are *SHIFT=SHIFT+1*. Considered the line 19, the pre-conditions are *SET1!={}* and *N=position of SETE*, and the post-conditions are *SET1* ← *IVL* that overlap or *SET1* ←*IVL* of

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

the next farthermost position to be matched.

In *while* loop of line 2, there is the pre-conditions of *if SETE!={}* to initiate the searching phase in the inner loop of *while* (*SET1!={}* and (*N<=n*)). Also, it consists of three *if* conditions, which need to be proved. When initiating the outer loop and *while* is activated. The first inverted lists structure from the table are put to *SETE* for driving the conditions of *if* (*SETE!={}*). The correctness of algorithm must be proved in three possible cases as follows.

In the first case (can not reach *if* in line 4), if there is no pattern to be matched, then it guarantees the termination of loop for reaching the post-conditions when *N>=n* or *SHIFT>n+min_length*. The post-conditions are *N>n* or *SHIFT>n+min_length* and the invariants are *N<=n*, *SHIFT<=n+min_length*. Let *i=min_length*. The proof is by induction on *i*. In the base case, *i=1*; the result is true for forwarding the next proof. This case shows *i=min_length*, *SHIFT=2xmin_lenght*, while the invariants are unchanged. In the inductive step, *i=n-min_length* which is comparable as step of *n-1* of inductive step, and *SHIFT=n-min_length*. When testing *SETE={}*, the variable *SHIFT=(n-min_length)+ min_length* which equal *SHIFT=n*. Meanwhile *i* equals *n-1* (*n-min_length*). By induction, the hypothesis of *i=n* when *SHIFT=n+min_length*. Therefore, the hypothesis can be reached.

In the second case (line 4 can be reached), *SETE!={}* and there is at least one pattern to be matched. In this case, the proof needs to show the post-condition of all *if* conditions to be reached for correctness of the inner *while*. First of all, the condition *if* of line 8, line 16, and line 19 must be true. The proofs show as follows.

For operating line 8, the pre-conditions are *N=position of SETE*, and *N<=position of SETE*, and the post-conditions are *SET1=SET1* $\cap$ *SET2* or *SET1=SET1* $\cap$ *SETE*. The proof is started that if *N=position of SETE* or otherwise cases, then it is true by Lemma 2 (shown post-conditions). Afterwards the matched positions must be reported by line 14 and the variable *N* is driven by line 15 for forwarding to line 16.

Considered the line 16, the pre-conditions are N>*position of SETE*, *SET1!={}*, and the post-condition is *SHIFT=SHIFT+1*. In this case, if *N>position of SETE*, then it definitely increases the variable *SHIFT* and the post-condition is reached.

In line 19, the pre-conditions are *SET1!={}* and *N=position of SETE*, and the post-conditions are *SET1*$\leftarrow$*IVL* that overlap or *SET1*$\leftarrow$*IVL* of the next farthermost to be matched. The results of *SET1* are possible in *SET1!={}* if there are the overlapping positions or there is the next farthermost position to be matched. Otherwise, *SET1* is empty because the condition of *if* in line

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

16 returns the corresponding position(s) in *SET1* and line 15 increases the variable *N*. Therefore, the post-conditions guarantee the correctness. If *SET1* does not empty and *N=position of SETE*, then the overlapping of pattern is processed and the result is added to *SET1*, which the post-conditions can be reached. For the condition *else* of *if*, there are the inverted lists, or there are no inverted lists in *IVL(text[N])*; both of them are also driven by *SET1* to the post-conditions. Thus, *if* is correct.

In the *while* loop of line 7, the pre-conditions are *SETE* and *SET1!={}*, *1<N<n* and its post-conditions are *SET1={}*, and the matched positions are reported and kept into *RESULTS*. All invariants are continued in the external loop, especially *SETE!={}* and *N<=n*. Supposing that the *if* conditions of line 8, line 16, and line 19 are true (shown as the following proofs). The proof is by the induction. Let *j=N*; the variable *j* is used for straightforward proof. In the base case, the variable *j=N* needs to be true. If there is only one farthermost position, the base case is true. Otherwise, the proof must be shown as the external loop. Assuming that *if* of line 8, line 16 and line 19 are true, then the post-conditions can be reached and the matched position can be reported. This loop will terminate *if* of the post-condition in line 19, which is *SET1={}*, or continue if the post-condition of line 19 does not empty.

In the case of *SET1* not being empty; the inductive step needs to prove in both the following cases. The first case is that the variable *j<=position of SETE*, which means the variable *N* increases 1 until equal to *position of SETE*. In the inductive step, *j=position of SETE-1* (*j=position of SETE-N*), which *N=N+1*. However, it is definitely proved true by the above assumptions. Thus, the hypothesis step is reached. In the second case, the variable of *N>position of SETE* (the overlapping patterns must be inspected). Also, line 19 is true with *SET1!={}* or *SET1={}*. Like the first case, if *SET1* is empty, then this loop is terminated. If *SET1* does not empty, then the proof follows by the variable *N=N+1* which *j=N*. This loop will terminate at the end of the given text (variable *N=n*), which is applied to the first *while* in line 2 for the inductive hypothesis. Thus, *while* is correct.

Whenever the inner loop is terminated, the post-conditions of *while* are reached to *N=SHIFT*, and the loop is terminated. The third case of this algorithm is that *SET1!={}* or *SETE!={}* in some cases. If *N* and *SHIFT* are not taken, then the loop is thus correct as the first case or the second case of the above proof. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Example 3.12 Searching *P={aab,aabc,aade}* in the given text *T=aabcdgaadef*

1. Set the search window to the initial step that *N=3*, *SET1={}*, *SET2={}*, *SETE={}*, and *SHIFT=6* (2 folds of minimum pattern length).

1.1 *N=3*, *SET1={}*,*SET2={}*, *SETE={<3:1:{1}><3:0:{2}>}*

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

1.2 The farthest position to be found from *SETE* is 3, and then *N=1*, *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}*, *SET2={}*, and *SETE={<3:1:{1}><3:0:{2}>}*.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

After operation, the result is *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

1.3 *N=2*, *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}*, and *SET2={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

This case is that *SETE={<3:1:{1}><3:0:{2}>}*, *SET1*← *SET1* ∩ *SET2* and *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

1.4 *N=3*, *SET1={<1:0:{1,2,3}>}*, *SET2={}*, *SETE={<3:1:{1}><3:0:{2}>}*, this case does not access the table but using *SET1*← *SET1* ∩ *SETE*.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

Then *SET1* operates *SETE* that *SET1 = {<3:1:{1}><3:0:{2}>}*, and the matching is succeed on *<3:1:{1}>*. After matching, *<3:1:{1}>* is removed from *SET1* then *SET1={<3:0:{2}>}* and the *SET1* is not empty. Then, the internal while loop is continued.

1.5 *N=4*, *SET1={<3:0:{2}>}*, *SET2={<4:1:{2}>}*, *SETE={<3:1:{1}><3:0:{2}>}*, and *SHIFT=7*, which uses the '*if*' condition in line 16.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

After operation, the search is matched on *<4:1:{2}>*, and this inverted list is removed from *SET1* then *SET1={}*. Then, this window search is finished and exits to external *while* for next window search .

2. Set the next window with *N=7*, *SET1={}*, *SET2={}*, *SETE={}*, and *SHIFT=10*.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

N=7, *SET1={}*, *SET2={}*, *SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

2.1 The farthest position from *SETE* is *3* then *N=5*, *SET1={<3:0:{3}>}*, *SET2={}*, and *SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a a b c d g a a d e f

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

1 2 3 4 5 6 7 8 9 10 11

In this case, *SET1* is not a sequence to *SETE* so that 'g' is not compared, and the next farthest position from SETE is considered and the list *<1:0:{1,2,3}>* is found.

2.2 *N=7, SET1={<1:0:{1,2,3}>}, SET2={}, SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

In this case, the inverted list that equal to the farthest position to *SET1* is taken, then *SET1={<1:0:{1,2,3}>}* and the search is continued to next position. Then *N=8*.

2.3 *N=8, SET1={<1:0:{1,2,3}>}, SET2={<1:0:{1,2,3}><2:0:{1,2,3}>}*, and *SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a a b c d g a a d e f
1 2 3 4 5 6 7 8 9 10 11

SET1← *SET1* ∩ *SET2*, which *SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}* and *SHIFT* is updated.

2.4 *N=9, SET1={<1:0:{1,2,3}><2:0:{1,2,3}>}, SET2={<3:0:{3}>}*, and *SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a a b c d g a a d e f -
1 2 3 4 5 6 7 8 9 10 11

SET1← *SET1* ∩ *SET2*, which *SET1={<3:0:{3}>}* and SHIFT is updated again.

2.5 *N=10, SET1={<3:0:{3}>}, SET2={<4:1:{3}>}, SETE={<1:0:{1,2,3}><2:0:{1,2,3}>}*.

a a b c d g a a d e f -
1 2 3 4 5 6 7 8 9 10 11

The search is matched on *<4:1:{3}>* and *SET1={}* and this widow search is finished.

Notice that the characters of positions 6 and 11 are not compared, the sum of these positions is called the skipping number (represented by α). In this case, the sum of skipping number is 2 from the length of text $|t|=11$. By this way, the algorithm takes the characters to scan as $11-2=9$, which is $O(|t|-\alpha)$ time. Unfortunately, if there are no characters which can be skipped, the comparing time only equals the length of the given text.

Theorem 13. *All occurrences of $P=\{p^1, p^2, p^3,..., p^r\}$ appearing in the given text $T=\{t_1t_2t_3...t_n\}$ are reported in $O(|t|-\alpha)$ time in an average case, $O(|t|)$ in a worst case, and $O(|t|/min\_length)$ time in a best case scenario where $|t|$ is the length of T, α is the skipping number, and min_length is the minimum of the pattern lengths in P.*

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proof. In a best case scenario, if all searching windows are mismatched, then there is no character to be matched once. It takes $O(|t|/min\_length)$ time. In a worst case scenario, all characters of $\{t_1t_2t_3...t_n\}$ are scanned and all occurrences are reported in $|t|$ time. The time complexity of Algorithm 3.7 is dominated by the variable *N*, *SET1*, and *SET2*. Line 6 is replicated to inspect the inverted lists which are necessary to compare in *T*.

Unfortunately, if there is no character to skip for comparison, then this loop needs to scan from t_2 to t_n. This case could happen if all characters in the given text are matched. It can be said that the loops of line 6 scan for all characters in the given text, and the time complexity is $O(|t|)$. The other lines in the inner loop take the constant time by Lemma1, Lemma 2, and Lemma 3 respectively.

In an average case scenario, line 1 takes the constant time. For outer loop, if there is no character to be matched, then the best case scenario is shown. If there is at least one character which is matched, then the inner loop is driven to the average case. The average case needs to be proved in the inner loop (line 6 to line 23). This loop needs to scan the given text from t_2 to t_n in a worst case, but if there is one character that can skip then the time for comparing the text is $|t|-1$ (α is shown). If there are many positions to be skipped, then the sum of the skipping number equals α. In other lines of the inner loop, they take a constant time; therefore, the final average time of the searching phase equals $O(|t|-\alpha)$. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

CHAPTER 4

EVALUATIONS

This chapter presents the theoretical measurements and practical implementation results that demonstrate the favorable performance of proposed algorithms, supported the algorithm complexities and the experimental results. There are essentially two main methods to evaluate the performances. The first measurement is by the theoretical results of dictionary construction, those of pattern updating (insertion and deletion), and those of searching algorithms. The second evaluation is by experimental results of preprocessing time, space accommodation, and searching results. The following sections are organized as follows. Section 4.1 shows theoretical results. Section 4.2 is implementation details and experiment setup. Section 4.3 shows the preprocessing results. Section 4.4 shows the experimental results of searching and section 4.5 is discussions about results.

4.1 Theoretical Results

The following notations refer to the theoretical results; in addition, the following tables show the complexities of new algorithms and well known algorithms that are used for comparison. The notations to be used in this chapter are $|P|$, $|p|$, $|t|$, $tocc$, l_{min}, $locc$, and α where:

$|P|$ is the sum of patterns length in P,

$|p|$ is the length of pattern p,

$|t|$ is the length of text T,

$tocc$ is the number of occurrences of pattern set of P in text T,

l_{min} is the minimum length of patterns in P,

$locc$ is the number of the matched characters that includes the mismatched time, and

α is the number of matched position that lead to mismatch.

Additionally, the following notations denote all algorithms, E-IVL represents the exhaustive searching algorithm, PF-IVL represents the prefix searching algorithm, and SF-IVL represents the suffix factor searching algorithm.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.1.1 Theoretical Results of the Preprocessing Phase

The preprocessing phase constructs the dictionary by means of the inverted lists data structure. The method reads each character in each pattern and converts it to the individual inverted list. Then, each inverted list will be inserted in to the table one by one. Each insertion takes $O(1)$ time; therefore, for all character, the time complexity is $O(|P|)$, as shown in Table 4.1. When comparing with the well known data structures, new solutions are more efficient than these structures. All new solutions take only $O(|P|)$ time, which matches the best static data structure Trie of Aho-Corasick[1] and setHorspool [2]. The comparison results are shown in table 4.2.

Table 4.1 Dimensions of the preprocessing phase

Algorithms	Preprocessing-Time	Space accommodations
E-IVL	$O(\|P\|)$	$O(\|P\|)$
PF-IVL	$O(\|P\|)$	$O(\|P\|)$
SF-IVL	$O(\|P\|)$	$O(\|P\|)$

Table 4.2 Time complexities of preprocessing compared to well known algorithms

Algorithms	Preprocessing time
Aho-Corasick [1]	O($\|P\|$)
setHorspool [in 17]	O($\|P\|$)
Amir-Farach [2]	O($\|P\|log\|P\|$)
Amir et al. [8]	O($\|P\|log\|P\|$)
Amir et al. [9]	O($\|P\|$)
Sahinalp-Vishkin [28]	O($\|P\|$)
E-IVL	O($\|P\|$)
PF-IVL	O($\|P\|$)
SF-IVL	O($\|P\|$)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.1.2 Theoretical Results of Pattern Updating

In this phase, each character of the new pattern is to be inserted or the existing pattern is to be removed in optimal time $O(|p|)$. Each character is converted to inverted list and accessed in $O(1)$. The results are shown as in table 4.3.

Table 4.3 Time complexities of pattern updating

Algorithms	Pattern Insertion	Pattern Deletion
E-IVL	$O(\|p\|)$	$O(\|p\|)$
PF-IVL	$O(\|p\|)$	$O(\|p\|)$
SF-IVL	$O(\|p\|)$	$O(\|p\|)$

According to the new data structure, the time complexity when compared with other algorithms is optimal. It is clear that the new structure is more efficient than the classic algorithms. Table 4.4 shows the results.

Table 4.4 Time complexities of pattern updating when compared to well known algorithms

Algorithms	Insertion	Deletion
Aho-Corasick [1]	-	-
setHorspool [in 17]	-	-
Amir-Farach [2]	$O(\|p\|log\|P\|)$	$O(\|p\|log\|P\|)$
Amir et al. [8]	$O(\|p\|log\|P\|)$	$O(\|p\|log\|P\|)$
Amir et al. [9]	$O(\|p\|(log\|P\|/loglog\|P\|)$	$O(\|p\|(log\|P\|/loglog\|P\|)$
Sahinalp-Vishkin [28]	$O(\|p\|)$	$O(\|p\|)$
E-IVL	$O(\|p\|)$	$O(\|p\|)$
PF-IVL	$O(\|p\|)$	$O(\|p\|)$
SF-IVL	$O(\|p\|)$	$O(\|p\|)$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.1.3 Theoretical Results of the Searching Phase

The new algorithms scan only characters that need to be compared. They avoid scanning all characters in the given text, especially in an SF-IVL solution. Among them, the minimum time complexity is $O(|t|-\alpha)$ in the best case, $O(|t|)$ in an average, and $O(|t||P|)$ (only in E-IVL) in a worst case scenario. Table 4.5 shows these results.

Table 4.5 Time complexities of searching phase

Algorithms	Best Case	Average Case	Worst Case
E-IVL	$O(\|t\|)$	$O(\|t\|+locc)$	$O(\|t\|\|P\|)$
PF-IVL	$O(\|t\|)$	$O(\|t\|)$	$O(\|t\|)$
SF-IVL	$O(\|t\|/l_{min})$	$O(\|t\|-\alpha)$	$O(\|t\|)$

With respect to these results, the proposed algorithms are more efficient than the well known algorithms, especially the dynamic dictionary matching algorithms. Considered by types, the new approaches are more efficient than all dynamic dictionary matching algorithms even when compared with the static dictionary matching such as Aho-Corasick [1], and setHorspool [2]. A comparison of the main solutions related to the traditional algorithms is given in table 4.6.

Table 4.6 Time complexities of searching phase when compared to well known algorithms .

Algorithms	Searching
Aho-Corasick [1]	$O(\|t\|+tocc)$
setHorspool [in 17]	$O(\|t\|\|l_{max}\|)$
Amir-Farach [2]	$O((t+tocc)log\|P\|)$
Amir et al. [8]	$O((t+tocc)log\|P\|)$
Amir et al. [9]	$O((t+tocc)(log\|P\|/(loglog\|P\|))$
Sahinalp-Vishkin [28]	$O(\|t\|+tocc)$
E-IVL	$O(\|t\|+locc)$
PF-IVL	$O(\|t\|)$
SF-IVL	$O(\|t\|-\alpha)$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.2 Implementation Details and Experiment Setup

4.2.1 Implementation Details

The Netbeans 6.9.1 attached the Java compiler version 1.6.22 was used to write the programs. Then the abstract data type (ADT) of java.util.Vector was employed to accommodate all structures compared. AC-Trie and Reverted-Trie structures were created by the special classes to represent the nodes of Trie and Reverted-Trie, before being put into the java.util.Vector. The following sub-sections show the details of dictionary construction and the searching phase.

The inverted lists table was created by the java.util.Vector as well. In each instance of the second level of the perfect hashing table, which is a set of positions and set of pattern numbers, was implemented by the java.util.HashTable structure and the java.util.HashSet structure respectively.

The second level utilizes java.util.HashTable to store the positions of characters which occur in the patterns just as java.util.HashSet is used for storing the pattern numbers. The IVLtableZero and the IVLtableOne are instaces of java.util.Vector represented as attributes in inverted lists class.

The IVLtableZero was used to store the inverted lists, which represents the terminate status equal to 0 (as shown in "0" of Figure 3.1). In the searching phase, the variables *SET*1, *SET*2 and *SETE* were also implemented by java.util.HashTable. All results were kept in the instances of java.util.ArrayLists named *RESULTS*.

4.2.2 Experiment Setup

All experiments were performed on a Dell Vostro 3400 notebook with Intel(R) CORE(TM) i5 CPU, M 560 @2.67 GHz, 4 GB of RAM, running Windows 7 Professional (32-bits) as an application machine. All of them were implemented in Java with JavaTM 2 SDK, Standard Edition Version 1.6.22 built in the Netbeans 6.9.1.

With regard to performance measures, the size of data tests ($|\Sigma|$) were the 52 letters of the English alphabet; 'A' to 'Z' and 'a' to 'z'. Then, programs randomized each pattern with the various lengths of 3 to 20 characters, where the average length was 12 characters. The proposed numbers of patterns were 10; 100; 1,000; 10,000; 50,000; 100,000 and 300,000 (only for the

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

inverted lists algorithm). Each of the pattern numbers was randomly built in 10 files. The texts were randomized from the size of 1 KB, 10 KB, 100 KB, 1 MB, 5 MB, and 10 MB. Also, each of the text sizes was performed in 10 files.

In the pre-processing tests, each file in each group was read and generated to the data structures one by one. Then the processing time of each file was captured in nano-seconds. Afterwards, each file again was built and both the data structure and the memory usages were captured in kilo-bytes. Performing the searching experiments, every pattern file was paired with each text file. To get significant insight on searching time, the first file of 10 patterns was paired with the first file of 1 KB, the second file of 10 patterns was paired by the second file of 1 KB, and the other cases were performed in the same way. When the search in each pair of patterns and text sizes was completed, the processing time in nano-seconds was captured. Then, when the 10 pairs of each group of text finished processing, the average time was given. All comparisons were made against the well known algorithms as shown in the previous sections.

4.3 Preprocessing Results

The inverted lists structure was constructed faster and used smaller space than the earlier structures (Aho-Corasick [1] called AC-Trie, SetHorspool in [17] called Reverted-Trie, and the Suffix tree[13]). Creating an inverted lists structure was faster than AC-Trie by 3.75 fold, the Reverted-Trie by 2.33 fold, and the suffix tree by 15.69 fold. The resulting details are shown in table 4.7, which converts nano-time to seconds.

Concerning the storage needs, the overall space of the inverted lists structure on average used less space than the AC-Trie (18.42)%, the Reverted-Trie (20.05)%, and the sufffix tree (92.38)%. In case of pattern numbers more than 1,000; the suffix tree could not create the structure (represented by '-') because the computer was out of heap memory in java while generating the structure. As well as, AC-Trie and RT-Trie were similar to suffix tree when using the pattern number 300,000. The results are shown in table 4.7 and table 4.8.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 4.7 Processing time (seconds)

pattern	AC-Trie	RT-Trie	ST	IVL
10	0.161	0.095	0.154	**0.030**
50	0.151	0.201	0.235	**0.113**
100	0.401	0.466	0.467	**0.278**
500	0.566	0.710	19.276	**0.351**
1,000	1.023	1.905	708.274	**0.767**
5,000	10.791	8.001	-	**5.519**
10,000	45.431	20.728	-	**6.918**
50,000	532.518	110.561	-	**43.623**
100,000	3,598.131	5,745.879	-	**851.156**
300,000	-	-	-	**1,132.651**

Table 4.8 Memory usage (KB)

pattern	AC-Trie	RT-Trie	ST	IVL
10	4.71	4.93	24.88	4.56
50	4.82	4.96	48.35	4.81
100	4.90	5.10	896.11	**4.89**
500	5.59	5.67	2,512.46	**5.12**
1,000	6.21	6.29	5,371.879	**5.33**
5,000	11.10	11.22	-	**7.56**
10,000	15.83	16.13	-	**9.84**
50,000	54.57	55.11	-	**23.36**
100,000	155.84	131.14	-	**47.63**
300,000	-	-	-	**169.58**

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.4 Searching Results

For searching measurements, five algorithms were implemented and the processing times were compared. Firstly, algorithms called Exhaustive-IVL (E-IVL), Prefix-IVL (PF-IVL), and Suffix Factor-IVL(SF-IVL) were implemented by Java language. Then, Aho-Corasick algorithm[1] called AC, and SetHorspool algorithm (mentioned in [18]) called HP were also implemented in Java. The searching times of the inverted lists Suffix Factor-IVL were more efficient than the SetHorspool algorithm in an average case but took longer than the Aho-Corasick. In the case of small pattern numbers and small text sizes, the inverted lists algorithms were more efficient than the others.

The experimental results showed that Suffix Factor-IVL took less time than the others. They were particularly significant in the pattern numbers 10, 50 and 100 with the given text 1 KB, 10 KB, 100 KB, 1 MB, 5 MB, and 10 MB respectively. Suffix Factor-IVL was slower than the others when using a large number of patterns. The results are shown in table 4.9 to table 4.14.

Table 4.9 Searching time (seconds) in 1 KB text size

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	0.011	0.045	0.050	0.040	**0.003**
50	0.018	0.102	0.053	0.042	**0.014**
100	0.020	0.134	0.072	0.045	**0.019**
500	0.017	0.201	0.182	0.147	0.172
1,000	0.039	0.212	0.203	0.165	0.233
5,000	0.042	0.186	0.883	0.167	0.394
10,000	0.033	0.165	0.994	0.994	0.771
50,000	0.029	0.734	5.981	0.393	1.021
100,000	0.038	0.817	7.498	0.498	1.347
300,000	-	-	8.987	0.987	1.548

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 4.10 Searching time (seconds) in 10 KB text size

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	0.066	0.200	0.098	0.098	**0.055**
50	0.102	0.755	0.274	0.168	**0.100**
100	0.104	1.574	0.298	0.141	**0.009**
500	0.112	1.698	0.863	0.365	0.713
1,000	0.120	1.733	1.321	1.001	0.722
5,000	0.161	1.695	4.722	1.276	3.191
10,000	0.210	1.889	8.623	1.301	6.345
50,000	0.705	2.034	11.984	1.696	9.120
100,000	1.667	2.701	22.501	2.019	10.124
300,000	-	-	30.542	3.089	17.984

Table 4.11 Searching time (seconds) in 100 KB text size

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	0.301	1.272	1.378	1.004	**0.300**
50	0.554	8.253	1.243	1.621	**0.504**
100	0.588	13.101	2.145	1.589	**0.548**
500	0.634	17.892	6.231	4.984	4.365
1,000	0.579	15.605	8.843	7.312	7.276
5,000	1.405	18.243	47.842	6.988	33.419
10,000	1.464	19.454	72.456	9.002	56.827
50,000	2.185	20.798	82.586	10.102	71.657
100,000	4.387	25.391	132.421	15.235	92.241
300,000	-	-	252.978	20.680	101.279

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 4.12 Searching time (seconds) in 1 MB text size.

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	3.143	24.231	28.633	23.623	**3.012**
50	6.286	108.732	34.457	30.380	**6.117**
100	5.012	131.076	35.911	29.766	**5.010**
500	7.501	155.205	50.442	45.291	28.671
1,000	7.003	190.532	88.642	46.665	71.611
5,000	26.784	175.989	135.665	49.571	128.181
10,000	58.860	179.859	258.932	51.651	167.530
50,000	90.725	249.101	763.159	94.290	289.689
100,000	102.198	321.761	851.421	134.872	298.764
300,000	-	-	976.789	356.712	362.432

Table 4.13 Searching time (seconds) in 5 MB text size

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	15.118	103.881	111.961	98.612	**14.988**
50	30.889	400.266	446.547	126.493	**23.443**
100	31.997	766.786	576.351	136.431	**29.671**
500	35.150	804.195	651.244	139.498	149.137
1,000	42.807	869.241	766.112	153.521	133.761
5,000	129.047	900.480	891.122	161.541	160.178
10,000	159.598	905.586	913.141	213.094	217.353
50,000	201.003	1,521.231	1,214.14	300.691	297.514
100,000	302.677	1,941.345	1,476.195	341.861	472.121
300,000	-	-	1,674.712	399.765	753.448

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 4.14 Searching time (seconds) in 10 MB text size

pattern	AC	setHP	E-IVL	PF-IVL	SF-IVL
10	40.907	250.583	210.661	189.778	**39.798**
50	69.397	882.667	546.647	287.018	**43.543**
100	72.976	1,534.896	676.751	291.781	**49.667**
500	85.781	1,688.574	757.244	320.745	179.135
1,000	89.481	2,457.665	868.182	331.905	183.671
5,000	290.882	2,561.901	995.129	348.921	**210.517**
10,000	361.175	2,551.012	1,013.181	449.005	**351.351**
50,000	425.236	2,631.422	1,415.148	631.448	495.513
100,000	649.133	2,752.843	1,891.789	739.094	657.349
300,000	-	-	2,135.987	876.339	987.924

Some common interesting experiments have been performed with the small alphabet sizes and small dictionaries. In these investigations, the experiments were tested on the patterns from 10 to 100 and several given texts that are less than 1 KB. The searching times were more efficient than the Aho-Corasick and the SetHorspool in the cases of 10, 20, 30, 40, 50 and 60 patterns, especially when the given text sizes are 100 and 500 bytes. For the number of pattern 70 to 100; new algorithms took less time than the SetHorspool, but longer than the Aho-Corasick. Figure 4.1 to Figure 4.4 illustrate the searching time where the x-axis is the pattern number and the y-axis is the processing time in the second unit.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

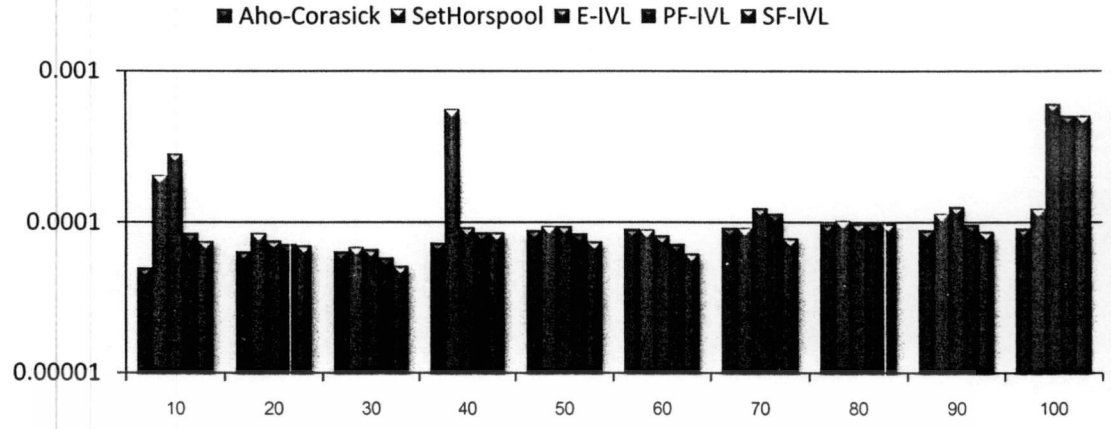


Figure 4.1 searching results when the given text 10 bytes.

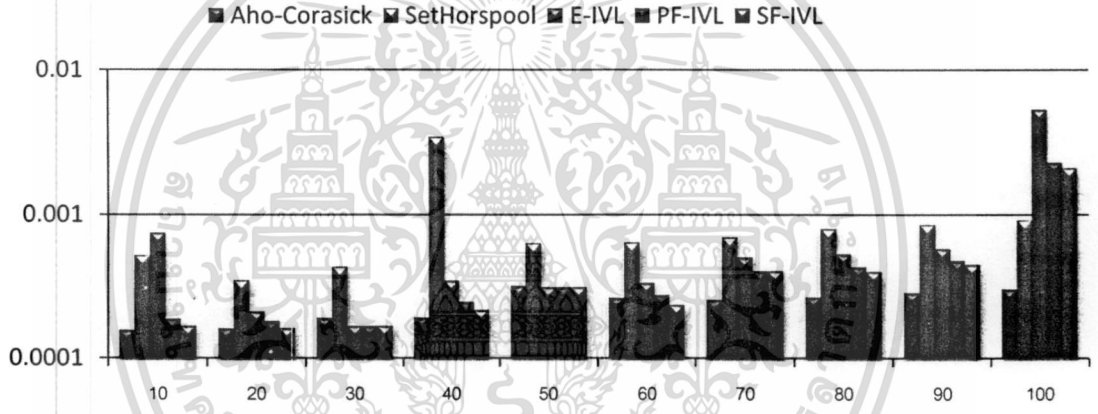


Figure 4.2 searching results when the given text 50 bytes.

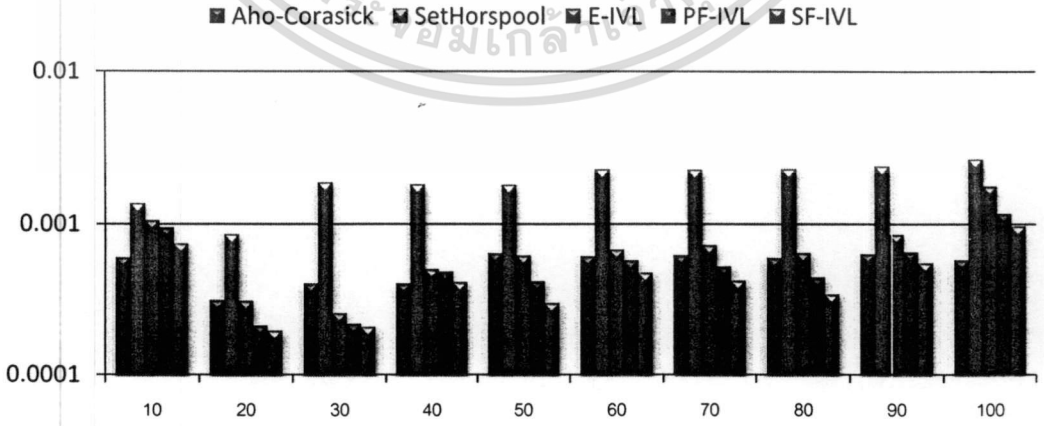


Figure 4.3 searching results when the given text 100 bytes.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

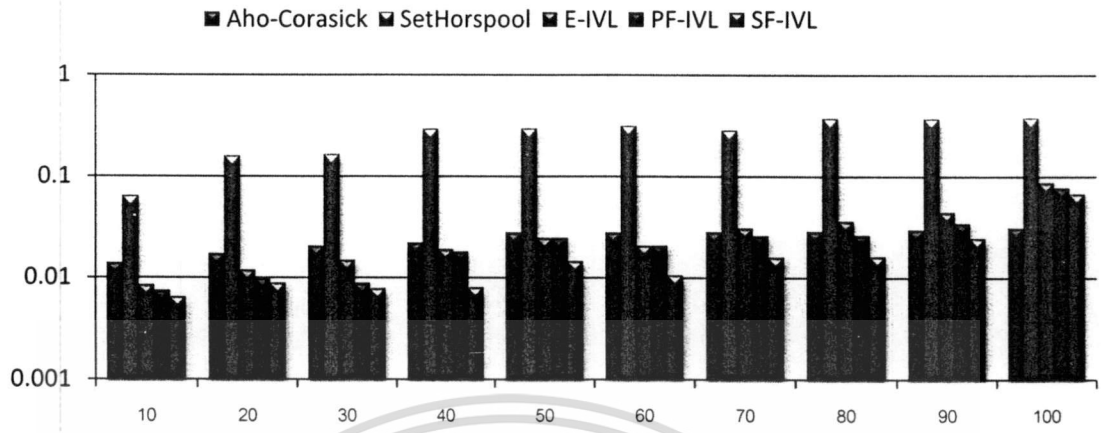


Figure 4.4 searching results when the given text 500 bytes.

4.5 Discussion and Analysis

This section is reserved to analyze data results. Apart from the theoretical results, the dictionary construction takes less time and space than the traditional dictionaries, which are implemented by Trie and Suffix tree. It can be said that the new data structure is more efficient than the previous data structures in terms of time and space efficiency. The reason why the inverted lists structure is superior to the others is that each inverted list is an individual item without any nodes or paths; meanwhile, the AC-Trie and the Suffix tree need to create the nodes and calculate their paths.

However, there were many cases that did not outperform the traditional structures. The methods to implement, the speed of CPU, and the computer processes being run may also make the differences from the theoretical results.

The first advantage is that the search algorithms can scan an input text in less than a single pass. The proposed dictionary and search algorithms are ideal structures for any small size input text and dictionary.

The second advantage of the inverted lists structure is that the expected pattern to be matched can be reported over time because this structure keeps the positions and the numbers of patterns. Then the searching results are based not only on yes or no answers, but can also report

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

all numbers and patterns to be matched over time. In contrast, the traditional data structures are not able to handle this aspect because they need to access by sequencing from the root of structures.

A final remark is that the type of each window search, which is by Navarro and Raffinot [16], divides the searching approach to a prefix approach (comparing from left to right), a suffix approach (comparing from right to left) and a factor approach (comparing by calculating the special positions). To distinguish deference of data structure, the inverted lists structure is not sequenced to access and to compare the inverted lists in the target text. Thus the target text can be scanned by all approaches even in parallel scanning or simultaneous access.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

CHAPTER 5

CONCLUSION

5.1 Conclusion

Dictionary matching areas is mostly concerned with solving problems of computer science applications, shown in the introductory and the related works chapters. In this thesis, both a new data structure and non-trivial algorithms have been presented for dynamic dictionary matching principles. This new data structure, called *inverted lists*, is based on combining an inverted index principle and a perfect hashing principle. The structure enables to accommodate a set of patterns called a dictionary and also supports the ability of individual patterns to be updated overtime. Most important, three new dictionary matching algorithms, which employ this structure, have been developed.

With the methodology of computer science, the proposed data structure was proven by the time and the space complexity, including the correctness of the algorithm. Then, these times and spaces were compared with the traditional data structure and algorithms.

As theoretical results, the thesis hypothesis was fulfilled that the inverted lists structure adapted from the inverted index and the perfect hashing table was easy to construct, required less space to store and was fast to access. Moreover, it is able to be applied to the new algorithms of dynamic dictionary matching efficiently. The performances of the proposed algorithms are shown by proving time complexity and space complexity. In addition, experiments investigate the performances by implementing programs from the proposed algorithms and running them on a variety of datasets with different parameters.

In major theoretical results, the inverted lists structure is employed for three proposed algorithms of dynamic dictionary matching: exhaustive solution (E-IVL), prefix solution (PF-IVL), and suffix factor solution (SF-IVL).

The exhaustive algorithm takes $O(|P|)$ time and $O(|P|)$ space in dictionary construction where $|P|$ is the sum of the length of all patterns in P. This solution takes $O(|p|)$ time for insertion or deletion where $|p|$ is the length of pattern to be inserted or deleted. In the searching phase, this algorithm takes $O(|t|)$ time in the best case, $O(|t|+locc)$ time in average case, and $O(|t||P|)$ time in the worst case scenario, where $|P|$ is the number of patterns, $locc$ is the matching number, and $|t|$ is the length of input text.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The prefix algorithm takes $O(|P|)$ time and $O(|P|)$ space in dictionary construction. Meanwhile, the searching phase takes $O(|t|)$ time in all cases: the best case, an average case, and the worst case.

The suffix factor solution takes $O(|P|)$ time and $O(|P|)$ space in dictionary construction. The searching phase takes $O(|t|/l_{min})$ time in the best case, $O(|t|-\alpha)$ time in an average case and $O(|t|)$ time in the worst case, where l_{min} is the minimum pattern length and α is the number of matched position that lead to mismatch.

In the experimental results of preprocessing phase, the inverted lists structure was constructed faster and used a smaller space than the earlier structures. Creating an inverted lists structure was faster than AC-Trie, the Reverted-Trie, and the Suffix tree. Also, the inverted lists structure on average used less space than those structures.

The experimental results of all searching algorithms showed that E-IVL was slowest (working as a naïve algorithm); meanwhile, PF-IVL was better and worked in linear time in theoretical and experimental results. Compared to the traditional algorithms, E-IVL, PF-IVL and SF-IVL were faster than the SetHorspool, but slower than Aho-Corasick. In particular, E-IVL was slowest.

The best results were that SF-IVL took less time than the other solutions. Particularly, they were significant in the pattern numbers 10, 50 and 100 with the given text 1 KB, 10 KB, 100 KB, 1 MB, 5MB and 10 MB, respectively. SF -IVL was slower than the others when using a large number of patterns. Additionally, if small text sizes and small dictionary sizes were considered, then the new algorithms were more efficient than the earlier algorithms especially when there were 10 to 6o patterns and the text sizes were 100 and 500 bytes.

5.2 Recommendations

There are four possibilities for developing enhancement of the data structure and algorithms presented in this thesis.

1) A further improvement can be gained by an inverted lists structure could group or compress the lists of the pattern sets. For instance, if the pattern numbers are connected in the sequence such as <1:0:{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}>, then they can be grouped as <1:0:1-10> or in other ways. This contributes to a good (structure) performance.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2) Other interesting aspects offer insight into many fundamentals of computer science. This structure could be applied to solve other problems such as approximate matching, regular searching, two-dimensional matching, pattern recognition, and so on.

3) The ability to apply the inverted lists by separating the given texts could enhance the inverted lists to handle those sections as well as patterns. Then, the inverted lists can be employed for this context, called text indexing.

4) In order to solve more complex problems and more accelerative searches, algorithms could also be improved to a parallel search and/or a distributed search.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

REFERENCES

[1] Aho A. V. and Corasick M. J. "Efficient string matching. An aid to bibliographic search". **Comm. ACM**, 1975, Pp.333-340.

[2] Moffat A. and Zobel J. "Self-Indexing Inverted Files for Fast Text Retrieval". **ACM Transactions on Information Systems**, Vol. 14, No. 4, 1996, Pp.349-379.

[3] Commentz-Walter B. "A string matching algorithm fast on the average". **In Proceedings of the Sixth International Collogium on Automata Languagees and Programming**, 1979, Pp.118-132.

[4] Boyer R.S. And Moore J. S. "A fast string searching algorithm". **Communications of the ACM**, Vol. 20, No. 10, 1977, Pp. 762-772.

[5] Allauzen C. and Raffinot M. "Factor oracle of a set of words". **Technical report 99-11, Institute Gaspard Monge**, Universitĕ de Marne-la-Vallĕe, 1999.

[6] Monz C. and Rijke M. de. (11, 02, 2002). *Inverted Index Construction.* [Online]. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf.

[7] Knuth D. E., Morris J. H. and Pratt V.R. "Fast pattern matching in strings". **SIAM Journal on Computing**, Vol. 6, No. 1, 1997, Pp. 323-350.

[8] Navarro G. "Improved approximate pattern matching on hypertext". **Theoretical Computer Science**, 2000, 237:455-463.

[9] Navarro G. and Raffinot M. "Flexible Pattern Matching in Strings". **The press Syndicate of The University of Cambridge**. 2002.

[10] HYYRO H., SSON K. F. and NAVARRO G. "Increased Bit-Parallelism for Approximate and Multiple String Matching". **ACM Journal of Experimental Algorithms**, Vol.10, No. 2.6, 2005, Pp. 1-27.

[11] Fan J. J. and Su K. Y. "An efficient algorithm for match multiple patterns". **IEEE Trans. On Knowledge and Data Engineering**, Vol. 5 No. 2, 1993, Pp. 339-351.

[12] Zobel J. and Moffat A. "Inverted Files Versus Signature Files for Text Indexing". **ACM Transaction on Database Systems**, Vol. 23, No. 4, 1998, Pp. 453-490.

[13] Zobel J. and Moffat A. "Inverted Files for Text Search Engines". **ACM Computing Surveys**, Vol. 38, No. 2, 2006, Pp. 1-56.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[14] Karp K. M. and Rabin M.O. "Efficient randomized pattern-matching algorithms". **IBM Journal of Research and Development**, Vol. 31, No. 2, 1987, Pp. 249-260.

[15] Gongshen L., Jianhua L. and Shenghong L. "New multi-pattern matching algorithm". **Journal of Systems Engineering and Electronics**, Vol. 17, No. 2, 2006, Pp. 437-442.

[16] Salmela L., Tarhio J. and Kytöjoki J. "Multipattern string matching with q-grams". **ACM Journal of Experimental Algorithmics (JEA)** , Vol. 11, Article No. 1.1, 2006, Pp. 1-19.

[17] Ping L., Jian-Long T. and Yan-Bing L. "A partition-based efficient algorithm for large scale multiple-string matching". **In Proceeding of 12th Symposium on String Processing and Information Retrieval (SPIRE'05)**. Lecture Notes in Computer Science, Vol. 3772, Springer-Verlag, Berlin, 2005.

[18] Crochemore M., Czumaj A., gąsieniec L., Jarominek S., Lecroq T., Plandowski W. and Rytter W. "Fast practical multi-pattern matching". **Rapport 93-3, Institute Gaspard Monge**, Universitĕ de Marne-la-Vallĕe, 1993.

[19] Crochemore M., Czumaj A., gąsieniec L., Lecroq T., Plandowski W. and Rytter W. "Fast practical multi-pattern matching". **Information Processing Letters**, Vol. 71, No. 3/4, 1999, Pp. 107-113.

[20] Raffinot M. "On the multi backward dawg matching algorithm (MultiBDM)". In R. Baeza-Yates, editor, **Proceedings of the 4th South American Workshop on String Processing**, Valparaìso, Chile. Carleton University Press, 1997, Pp. 149-165.

[21] Fredriksson K. and Navarro G. "Average-Optimal Single and Multiple Approximate String Matching". **ACM Journal of Experimental Algorithmics**, Vol. 9, No. 1.4, 2004, Pp.1-47.

[22] Yates R. B. and Neto B. R. "Mordern Information Retrieval". **The ACM press. A Division of the Association for Computing Machinery, Inc**. 1999, Pp. 191-227.

[23] Wu S. and Manber U. "A fast algorithm for multi-pattern searching". **Report tr-94-17, Department of Computer Science**, University of Arizona, Tucson, AZ, 1994.

[24] Klein S. T., Shalom R. and Kaufman Y. "Searching for a set of correlated patterns". **Journal of Discrete Algorithm**, Elsevier, 2006, Pp. 1-13.

[25] Hong Y. D., Ke X. and Yong C. "An improved Wu-Manber multiple patterns matching algorithm". **Performance, Computing, and Communications Conference**, 2006. IPCCC 2006. 25th IEEE International 10-12, 2006, Pp. 675-680.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[26] Alqadi Z. A. A., Aqel M. and El Emary I. M.M. "Multiple skip Multiple pattern matching algorithm (MSMPMA)". **IAENG International Journal of Computer Science**, Vol. 34, No. 2, IJCS_34_2_03, 2007.

[27] Amir A. and Farach M. "Adaptive dictionary matching". **Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science**, 1991, Pp. 760-766.

[28] Amir A., Farach M., Idury R.M., La Poutr'e J.A. and Schäffex A.A. "Improved Dynamic Dictionary-Matching". **In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms.** 1993, Pp. 392-401.

[29] Amir A., Farach M., Idury R. M., La Poutré J. A. and Schäffex A. A.. "Improve dynamic dictionary matching". **Information and Computation**, Vol, 199, No. 2, 1995, Pp. 258-282.

[30] Amir A., Farach M. and Matias Y. "Efficient randomized dictionary matching algorithms", **In CPM: 3rd Symposium on Combinatorial Pattern Matching**, 1992.

[31] Amir A., Farach M., Galil Z., Giancarlo R. and Park K. "Dynamic dictionary matching". **Manuscript.** 1991.

[32] Amir A., Farach M., Galil Z., Giancarlo R. and Park K. "Dynamic dictionary matching". **Journal of Computer and System Sciences.** 1993.

[33] Amir A., Farach M., Galil Z., Giancarlo R. and Park K. "Dynamic dictionary matching". **Journal of Computer and System Science**, Vol. 49, No. 2, 1994, Pp. 208-222.

[34] Amir A., Farach M., Idury R. M., La Poutre J. A. and Schäffex A. A.. "Improved Dynamic Dictionary Matching". **Information and computation.** Vol. 119, 1995, Pp. 258-282.

[35] Sleator D. D. and Tarjan R. E. "A data structure for dynamic trees". **Journal of Computer and System Sciences**, Vol. 26, No. 3, 1983, Pp. 362-391.

[36] McCreight E. M. "A space-economical suffix tree construction algorithm". **Journal of Algorithms**, Vol. 23, No. 2, 1976, Pp. 262-272.

[37] Zaïane O. R.. "CMPUT 391: Inverted Index for Information Retrieval". **University of Alberta.** 2001.

[38] Dietz P. F. and Sleator D. D. "Two algorithm for maintaining order in a list". **In Proceeding of the Nineteenth Annual ACM Symposium on Theory of Computing.** 1997, Pp. 365-372.

[39] Chvez E. and Lonardi S. "Improved Fast Similarity Search in Dictionaries". **(Eds.):SPIRE 2010**, LNCS 6393. 2010, Pp.173-178.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[40] Melnik S., Raghavan S., Yang B. and Garcia-Molina H. "Building a Distributed Full-Text Index for the Web". **ACM Transactions on Information Systems**, Vol. 19, No. 3, 2001, Pp. 217-241.

[41] Mazeika A., Bohlen H. M., Koudas N. and Srivastava D. "Estimating the Selectivity of Approximate String Queries". **ACM Transactions on Database Systems**. Vol. 32, No. 2, 2007, Pp.1-40.

[42] Lam T.W. and To K.K.. (03, 11, 2005). "The Dynamic Dictionary Matching Problem Revisited" . [Online]. Avaiable : http://citeseer.ist.psu.edu/413873.html.

[43] Crochemore M. and Perrin D. "Two-Way String-Matching". **Journal of the Association for Computing Machinery**, Vol. 38, No. 3, 1991, Pp.651-675.

[44] Weiner P. "Linear Pattern Matching Algorithms". **In Proceedings of Symposium on Switching and Automata Theory**, 1973, Pp. 1-11.

[45] Sahinalp S. and Vishkin U. "Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract)". **In 37th Annual Symposium on Foundations of Computer Science**, 1996, Pp. 320-328.

[46] Amir A., Landau G. M., Lewenstein M. and Sokol D. "Dynamic text and static pattern matching". **ACM Transactions on Algorithms**, Vol. 3, No. 2, Article 19, 2007, Pp. 1-24.

[47] Chan H.-L., Hon W.-K., Lam T.-W. and Sadakane K.. "Dynamic dictionary matching and compressed suffix trees". **SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms**, 2005, Pp. 13-22.

[48] wikipedia, (2006, November 15), *Hash table*. [Online]. Available: http://en.wikipedia.org /wiki/Hash_table.

[49] Escardo M., (2008, October 15), *Complexity considerations for hash tables*. [Online]. Available: http://www.cs.bham.ac.uk/~mhe/foundations2/node92.html.

[50] Loudon K. (2008, November 24), *Hash Tables*. [Online]. Available: www.oreilly.com /catalog/ masteralgoc/chapter/ch08.pdf.

[51] DINH V. H. (2006, November 24), *Hash Table*. [Online]. Available: http://libetpan.sourceforge.net /doc/API/API/x161.html.

[52] Chrochemore M. and Handcart C. "Automata for Matching Patterns", in **Handbook of Formal Languages**, Volume 2, Linear Modeling: Background and Application, G. Rozenberg and A.Salomaa ed.,Springer-Verlag,Berlin. 1997, Ch. 9, Pp. 399-462.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[53] Chrochemore M. "Off-line serial exact string searching". in Pattern Matching Algorithms, A. Apostolico and Z. Galil ed., **Oxford University Press**. Chapter 1, Pp. 1-53.

[54] Crochemore M., Gasieniec L. and Rytter W. "Constant-space string-matching in sublinear average time". **Compression and Complexity of Sequences 1997**, 1997, Pp. 230 – 239.

[55] Charras C. and Lecroq T. (2008, October 10). *Handbook of Exact String Matching*. [Online]. Available: www-igm.univ-mlv.fr/~lecroq/string/string.pdf.

[56] Galil Z. and Giancarlo R. "On the exact complexity of string matching upper bounds". **SIAM Journal on Computing**, Vol. 21, No. 3, 1992, Pp. 407-437.

[57] Kesong H., Yongcheng W. and Guilin C. "Research on A Faster Algorithm for Pattern Matching". **Proceedings of the fifth International workshop on Information retrieval with Asian languages**. 2000, Pp. 119-124.

[58] Law J. "Book reviews: Review of "Flexible pattern matching in strings: practical on-line algorithms for text and biological sequences by Gonzolo Navarro and Mathieu Raffinot". Cambridge University Press 2002". **ACM SIGSOFT Software Engineering Notes**, Vol. 28 Issue 2. 2003, Pp. 1-36.

[59] Navarro G. and Raffinot M. "Fast and flexible string matching by combining bit-parallelism and suffix automata". **Journal of Experimental Algorithmics (JEA)**, Vol. 5, 2000.

[60] Ager M. S., Danvy O. and Rohde H. K. "Fast partial evaluation of pattern matching in strings". **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Vol. 28 Issue 4. 2006, Pp. 3-9.

[61] Ager M. S., Danvy O. and Rohde H. K. "On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation". **Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation**, 2002, Pp. 32-46.

[62] Morris J. R. and Pratt J. H. "A linear pattern-matching algorithm". **Technical Report 40**, University of California, Berkeley. 1970.

[63] Simon I. "String matching and automata". in Results and Trends in Theoetical Computer Science", Graz, Austria, J. Karhumaki, H. Maurer and G. Rozenerg ed., **Lecture Notes in Computer Science 814, Springer- Verlag, Berlin**. 1994, Pp. 386-395.

[64] Cisar, P. Bosnjak, and S. Maravic Cisar, S. "EWMA Based Threshold Algorithm for Intrusion Detection". **Computing and Informatics**, Vol. 29 No. 6+, 2010, Pp. 1089–1101.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[65] Lu, P. Che, Y. and WangK, Z. "UMDA/S: An Effective Iterative Compilation Algorithm for Parameter Search". **Computing and Informatics**, Vol. 29, No. 6+, 2010, Pp. 1159–1179.

[66] Makula, M. and Be nu skova L. "Interactive visualisation of oligomer frequency in DNA". **Computing and Informatics**, Vol. 28, No. 5, 2009, Pp. 695–710.

[67] Hu, Y. Wang, P.-F. and Hwang, Kai. "A Fast Algorithm for Multi-String Matching Based on Automata Optimization". **C2010 2nd International Conference on Future Computer and Communication**, Vol. 2, 2010, Pp. 379–383.

[68] Askitis, N. and Zobel, J. "Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache". **ACM Journal of Experimental Algorithmics**, Vol. 15, No. 1, Article 1.7, 2011, Pp. 1–61.

[69] Belazzougui, D. "Worst Case Efficient Single and Multiple String Matching in the RAM Model". **21st International Workshop on Combinatorial Algorithms (IWOCA 2010)**, LNCS 6460, 2011, Pp. 90–102.

[70] Haapasalo, T. Silvasti, P. Sippu, S. and Soisalon-Soininen, E. "Online Dictionary Matching with Variable-Length Gaps". **10th International Symposium on Experimental Algorithms (SEA 2011)**, LNCS 6630, 2011, Pp. 76–87.

[71] Kuruppu, S. Beresford-Smith, B. Conway, T. and Zobel J. "Iterative Dictionary Construction for Compression of Large DNA Data Sets". **IEEE/ACM Transactions on Computational Biology and Bioinformatics**, Vol. 9, No. 1, 2012, Pp. 137–149.

[72] Jin Kim, H. Kim, H.-S. and Kang, S. "A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems". **IEEE Transaction on Parallel and Distributed Systems**, Vol. 22, No. 11, 2011, Pp. 1904–1911.

[73] Dai, L. and Xia, Y. "A Lightweight Multiple String Matching Algorithm". **International Conference on Computer Science and Information Technology 2008 (ICCSIT'08)**, Singapore, Aug. 29-Sept. 2, 2008, Pp. 611–615.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

APPENDICES

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Appendix A

Published Papers in Proceedings

International Conferences

1. C. Khancome and V. Boonjing. **String Matching Using Inverted List.** XIX International Conference on Computer, Information and Systems Science and Engineering (CISE 2007). Enformatika Transcations on Engineering, Computer and Technology Quarterly volumn 19, January, 2007, Pp.108-112. **(Selected for publishing in International Journal of Mathematics and World Academy of Science, Engineering and Technology)**
2. C. Khancome and V. Boonjing. **Dynamic Dictionary Matching Using Inverted Lists.** Proceeding of the Third IASTED International Conference ADVANCEDS IN COMPUTER SCIENCE AND TECHNOLOGY (ACST2007). April 2-4, 2007, Phuket,Thailand. Pp.397-401.
3. C. Khancome and V. Boonjing. **An Improved Dynamic Dictionary Matching Using Inverted Lists.** Proceeding The 9th International DSI Conference in conjunction with The 12th APDSI Conference July 11-15, 2007, Bankok Thailand.
4. C. Khancome and V. Boonjing. **Inverted Lists String Pattern Matching.** Proceedings 2009 2rd IEEE International Conference on Computer Science and Information Technology (ICCSIT2009). Vol. 2, 2009, Pp. 623-627.
5. C. Khancome and V. Boonjing. **A Character-Based Indexing Using Inverted Lists.** Proceeding of 2009 International Conference on Computer Technology and Development, ICCTD 2009. 2009, Pp. 221-224. **(Received a reward from PERDO)**
6. C. Khancome and V. Boonjing. **Data Structure for Dynamic Pattern.** International MultiConference of Engineers and Computer Scientists 2010, HK, 17-19 March 2010, Pp. 399-404.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

7. C. Khancome and V. Boonjing. **Dynamic Multiple Pattern Detection Algorithm**. 2010 the 2nd International Conference on Computer and Network Technology (ICCNT 2010), TH, 23-25 April 2010. **(Received a reward from PERDO)**
8. C. Khancome and V. Boonjing. **New Hashing-Based Multiple String Pattern Matching Algorithms**. 2012 Ninth International Conference on Information Technology- New Generations (ITNG 2012), LasVegas, USA, 2-4 April 2012, Pp.195-200.

Local Conference

1. C. Khancome and V. Boonjing. **String Matching Using Inverted List**. *The 1st Conference on Graduate Research, SRRU*. Surin Rajabhat University, February, 2007, 104-115. **(Received reward in Best Paper Awards)**
2. C. Khancome and V. Boonjing. **An Improved Dynamic Dictionary Matching Using Inverted Lists**. The 3rd Naresuan Research Conference 2007, July, 28-29, 2007, Thailand.
3. C. Khancome and V. Boonjing. **A Character-Based Indexing Using Inverted Lists**. 4th Mahasarakam University Research Conference, 2008, (Abstract) Pp. 92.

Published Papers in Journals

International Journals

1. C. Khancome and V. Boonjing. **String Matching Using Inverted List**. International Journal of Mathematics. Vol. 1, Number 3, 2007, Pp. 191-195.
2. C. Khancome and V. Boonjing. **String Matching Using Inverted List**. World Academy of Science, Engineering and Technology, Vol. 25, 2007, Pp. 108-112.
3. C. Khancome and V. Boonjing. **Optimal Linear-time Multi-string Pattern Matching Algorithm**. International Journal of Computational Science, Vol. 3 No. 6, December 2009, Pp. 629-641.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4. C. Khancome and V. Boonjing. **Inverted Lists String Matching Algorithms.** International Journal of Computer Theory and Engineering (IJCTE). Vol. 2, No. 3, June, 2010, Pp.352-357.
5. C. Khancome and V. Boonjing. **New Dynamic Dictionary Matching Algorithms Based on Inverted Lists.** International Journal of Mathematical Models and Method in Applied Science. (Submitted)
6. C. Khancome and V. Boonjing. **A New Linear-Time Dyanamic Dcitionary Matching Algorithm.** Computing and Informatics (CAI). (Accepted)
7. C. Khancome and V. Boonjing. **New Multiple String Patterns Matching Algorithm Using Inverted Lists.** The IAENG International Journal of Applied Mathematics. (Accepted)

Local Journals

1. C. Khancome and V. Boonjing. **String Matching Using Inverted List.** Journal of Graduate School, Surindra Rajabhat University. Vol. 1, No. 1, 2007, Pp. 80-87.
2. C. Khancome and V. Boonjing. **An Improved Dynamic Dictionary Matching Using Inverted Lists.** Journal of Rajanagarindra, Vol. 5, No. 13, 2008, Pp. 61-70.
3. C. Khancome and V. Boonjing. **A Character-Based Indexing Using Inverted Lists.** Journal of Science (selected paper from 4th Mahasarakam University Research Conference), Special issue 2008, Pp. 92-97.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Appendix B

Selected Papers for Fulfilment of Study

1. C. Khancome and V. Boonjing. **Dynamic Dictionary Matching Using Inverted Lists.** Proceeding of the Third IASTED International Conference ADVANCEDS IN COMPUTER SCIENCE AND TECHNOLOGY (ACST2007). April 2-4, 2007, Phuket, Thailand, Pp. 397-401.

2. C. Khancome and V. Boonjing. **Data Structure for Dynamic Pattern.** International MultiConference of Engineers and Computer Scientists 2010, HK, 17-19 March 2010, Pp. 399-404.

3. C. Khancome and V. Boonjing. **A New Linear-Time Dyanamic Dcitionary Matching Algorithm**. Computing and Informatics (CAI). (Accepted.)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the Third IASTED International Conference
ADVANCES IN COMPUTER SCIENCE AND TECHNOLOGY
April 2-4, 2007, Phuket, Thailand
ISBN CD: 978-0-88986-656-0

DYNAMIC DICTIONARY MATCHING USING INVERTED LISTS

Chouvalit Khancome and Veera Boonjing
Software Systems Engineering Laboratory
Department of Mathematics and Computer Science
Faculty of Science
King Monkut's Institute of Technology at Ladkrabang(KMITL)
Ladkrabang,Bankok 10520,THAILAND
E-mail: chouvalit@hotmail.com , kbveera@kmitl.ac.th

ABSTRACT

This paper proposes a new solution to the problem of dynamic dictionary matching. It employs inverted lists as data structures accommodating string patterns. The new solution takes (1) O(|P|) time for preprocessing, where |P| is a sum of the length of all patterns in set of pattern P; (2) O(|p|) time for insertion or deletion, where |p| is the length of pattern to be inserted or deleted; and (3) a search O(|t|+locc) time, where |t| is the length of input text and locc is the number of occurrences of matching between a character in the input text and in the inverted list.

KEY WORDS

Multiple string matching, dictionary matching, inverted list, trie, pattern

1. Introduction

The problem of dynamic dictionary matching is to efficiently locate a set of patterns occurring in an input text. In this problem, the set of patterns can change over time because of insertion and deletion of individual patterns. It calls for a data structure accommodating this set, which (1) allows quick insertions of patterns into the dictionary as well as deletions of patterns from the dictionary and (2) supports efficient searching for pattern strings in the input text. A trie, used by fast dictionary matching solutions such as [1], [9], [12], is an example of data structure supporting such an efficient searching. Unfortunately, insertions and deletions of patterns require reconstruction of the trie [2], [3], [4], [5], [6], [7], [8]. Solutions to these problems are the modifications of trie such as a suffix tree [2], [3], [11], [14] and a combination of a compact Trie and a fat tree [16].

In this paper, we propose to use an inverted list, a new data structure derived from an inverted index used in information retrieval field [10], [13], [15], instead of using a trie or a trie-based data structure. This structure is very simple and highly efficient. Furthermore, it well supports dynamic patterns.

The rest of paper is organized as follows. Section 2 gives details of inverted lists and dictionary preprocessing. Section 3 describes the new algorithm as well as its proof of time complexity. Conclusion is in section 4.

2. Inverted Lists

Let $P=\{p^1,p^2,...,p^r\}$ where p^i is a string from $c_1c_2,...,c_m$ under $\sum$ and $\sum$ is the set of the character in P.

2.1 Basic Definitions

Definition 1 A keyword ω^i of pattern p^i contains $w_{a_1,0,j}\ w_{b_2,0,j}\ w_{c_3,0,j}\ ...\ w_{-,-,1,j}$; where $w_{a_1,0,j}$ or $w_{a_1,1,j}$ is a_1 and k = 1, 2, ..., m; 1 indicates a status of last character in p^i and 0 otherwise. Therefore,

$$\omega^i = w_{a_1,0,j}\ w_{b_2,0,j}\ w_{c_3,0,j}\ ...\ w_{-,-,1,j} \quad (1)$$

Example 1 Given P={aab, aabc, aade}. We have ω^1=aab, ω^2=aabc and ω^3=aade. Therefore,

$\omega^1=a_{1,0,1}a_{2,0,1}b_{3,1,1}$,
$\omega^2=a_{1,0,2}a_{2,0,2}b_{3,0,2}c_{4,1,2}$, and
$\omega^3=a_{1,0,3}a_{2,0,3}d_{3,0,3}e_{4,1,3}$.

Definition 2 An inverted list L of ω^i, denoted by L_{ω^i}, is defined as

$$L_{\omega^i}=w_a:<1:0:i>, w_b:<2:0:i>, w_c:<3:0:i>,...,w_-:<m:1:i> \quad (2)$$

Example 2 From example 1, we have
L_{ω^1} = a:<1:0:1>, a:<2:0:1>, b:<3:1:1>,
L_{ω^2} = a:<1:0:2>, a:<2:0:2>, b:<3:0:2>, c:<4:1:2>, and
L_{ω^3} = a:<1:0:3>, a:<2:0:3>, d:<3:0:3>, e:<4:1:3>.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Definition 3 An index w_λ of invert list is $<\varepsilon:0:\{i,j,...\}>$ or $<\varepsilon:1:\{i\}>$. Therefore,

$$w_\lambda : <\varepsilon:0:\{i,j,...\}> \text{ or } <\varepsilon:1:\{i\}> \quad (3)$$

Example 3 From example 2, we have
a: <1:0:{1,2,3}>, <2:0:{1,2,3}>,
b: <3:1:{1}>, <3:0:{2}>,
c: <4:1:{2}>,
d: <3:0:{3}>, and
e: <4:1:{3}>.

Definition 4 Let $I_{\lambda_{i,0}}$ and $I_{\lambda_{i,1}}$ be $<\varepsilon:0:\{i,j,...\}>$ and $<\varepsilon:1:\{i\}>$, respectively. Therefore,

$$w_\lambda : I_{\lambda_{i,0}} \text{ or } w_\lambda : I_{\lambda_{i,1}}. \quad (4)$$

Definition 5 An inverted list table τ is a hash table with 2 columns: w_λ and $I_{\lambda_{i,0}}/I_{\lambda_{i,1}}$; where w_λ contains w_α, and $I_{\lambda_{i,0}}/I_{\lambda_{i,1}}$ contains

$$I_{\alpha_{i,0}}, I_{\alpha_{i,0}}, I_{\alpha_{i,1}}, \ldots \quad (5)$$

Example 4 The table τ constructed from example 3 is as shown in table1.

Table 1 the table of pattern P={aab,aabc,aade}

w_λ	$I_{\lambda_{i,0}}/I_{\lambda_{i,1}}$
A	<1:0:{1,2,3}><2:0:{1,2,3}>
B	<3:1:{1}><3:0:{2}>
C	<4:1:{2}>
D	<3:0:{3}>
E	<4:1:{3}>

Theorem 1 The access to $I_{\lambda_{i,0}}$ or $I_{\lambda_{i,1}}$ in the table τ take O(1) time.

Proof Let f(x) be a hash function, let $w_{\lambda_{i,0}}$ be the key for access $I_{\lambda_{i,0}}$ and $w_{\lambda_{i,1}}$ be the key for access $I_{\lambda_{i,1}}$.

Let the table τ implemented by the hash table that take O(1), therefore the access to $I_{\lambda_{i,0}}$ with $f(w_{\lambda_{i,0}})$ or access $I_{\lambda_{i,1}}$ with $f(w_{\lambda_{i,1}})$ take O(1) time#

2.2 Preprocessing phase

Let $\sum$ be a set of all characters in P and char(p_j^i) be a character 'j' of the pattern 'i'. Let |P| be a total length of all patterns in P. This algorithm must create the inverted list table before adding all character from $\sum$ to a character column. Afterwards, this algorithm reads a character one by one from each pattern and adds into the inverted list column. Before the addition, there is a process to check the inverted list char(p_j^i) at the same position. If it already exists the inverted list of the char(p_j^i) at the same position then we add only the number pattern to the invert list table. Otherwise, we must create a new inverted list and add the pattern number into the table. Figure 1 gives details of the algorithm.

```
Inverted-List Table(P={p^1,p^2,...,p^r})
Step A   Create table for alpha from Σ
Step B   for i:1:r
Step C     for j:1:m
Step D       if not Exist() inverted list of p_j^i
Step D1        Generate a new invert list and add to
               table at alphabet char(p_j^i)
             else
Step D2        Add a number i to the set of the
               represent part of the invert list at
               alphabet p_j^i
```

Figure 1 Algorithm for creating inverted list table

Theorem 2 The times complexity of the algorithm in figure 1 is O(|P|).

Proof Given $P=\{p^1,p^2,p^3,...p^r\}$ and the length of each of p^i is m_i such $m_1+m_2+m_3+...+m_r = |P|$

Step A Creating table for storing the inverted list take O(1) time.

Step B Repeating each of p^i takes r rounded. Each round must loop in Step C m_i times. Therefore, the inverted list takes $m_1+m_2+m_3+...+m_r = O(|P|)$ times.

Step D, Step D1 and Step D2 use for checking in table that take O(1) by theorem 1.

Therefore, the time complexity of this algorithm is O(|P|). #

2.3 Pattern Insertion

Let p^i be a new pattern for insertion where i is a number refer to unique symbol. The insertion must check the non-existence of p^i in the table. Afterwards, reading and inserting a character from pattern into table which is similar to the preprocessing phase. This algorithm is illustrated in the figure 2.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
InsertPattern(p^l)
Step A  if not Exist(p^l)
Step B    for j:1:m
            if not Exist() of char( p'_j )
Step B1       Generate a new invert list and add
              to table at alphabet char( p'_j )
            else
Step B2       Add a number l to the set of the
              represent part of inverted list at
              alphabet p'_j
```

Figure 2 Algorithm for pattern insertion

Theorem 3 Time complexity of the algorithm for pattern insertion is O(|p|).

Proof Let p^l be a new pattern for insertion where p^l contain a string $p^l = c_1c_2c_3...c_m$ such that the length m represent by |p|.

Step A Repeating from c_1 to c_m, use m time that is |p| or take O(|p|) times. The access to the inverted list table takes O(1) followed by theorem 1.

Step B Repeating for adding pattern one by one from c_1 to c_m that use |p|, therefore that take O(|p|) times.

Step B1 or B2 access to table taking constant time by theorem 1. That all of Step B takes O(|p|) times.

Therefore, the insertion algorithm takes O(|p|) times #

2.4 Pattern Deletion

Let p be a pattern for deletion and 'Numberpattern' be a number of pattern. ExistDel(p) is a function to detect the existence of pattern for deletion. The deletion must search for pattern p using ExistDel(p) function and the result is the pattern number for deletion. It then searches one by one for deletion until finish. The importance of deletion is that we need to check the inverted list in the same position of a number pattern that we want to delete. If it has only one, we can delete that inverted list immediately. Otherwise, we must delete an inverted list only 'Numberpattern'. The algorithm is illustrated in figure 3.

```
DeletePattern(p)
Step A  Numberpattern = ExistDel(p)
          if Numberpattern != θ
Step B    for i:1:m
            Search inverted list of p_i^Numberpattern
            if number of the items in represent
            pattern p_i^Numberpattern >1
Step B1       Delete items in represent part =
              Numberpattern
            else
Step B2       Delete inverted list of p_i^Numberpattern
```

Figure 3 Algorithm for pattern deletion

Theorem 4 The deletion algorithm takes O(|p|) times.

Proof Let p^l be a pattern for deletion where p^l contains a string $p^l = c_1c_2c_3...c_m$ with length m = |p|.

Step A repeat to read from c_1 to c_m takes O(|p|) times. Each time we access an inverted list use O(1) by theorem 1. Therefore, this step takes O(|p|) times.

Step B repeat for read a character one by one from c_1 to c_m takes O(|p|) times.

Step B1 or B2 accesses the inverted list with constant time from theorem 1. It takes O(|p|) time.

Therefore, the deletion algorithm takes O(|p|) time #

3. Searching phase

The searching phase employs the navigator variable N and the shift window search SHIFT. At first, we must initialize N, SHIFT and the control position variable at the beginning text T. This algorithm scans from left to right of text T. Every comparison takes the inverted list to the temporary variable SET1 or SET2. Meanwhile taking the inverted list to the temporary variable must intersect ($\cap$) SET1 and SET2 also. The purpose of intersection is to search for the sequence of pattern and check the matching. Afterwards, the algorithm shifts a window search. We illustrate the algorithm in figure 4.

```
Inverted-List Mutiple-Pattern Search
(P={p^1,p^2,p^3,...,p^r},T=t_1t_2...t_n)
Preprocessing Phase :
   Create Inverted-List Table(P={p^1,p^2,p^3,...,p^r})
Searching Phase
Step A  N=1 ,SHIFT=2,pos=1
Step B  while N<=n and SHIFT<=n
          if pos=1
Step B1     Store all member of row(text[N]) where
            position of Inverted List = pos to SET1
            and pos←pos+1
          else
Step B2     Store all member of row(text[N]) where
            position of Inverted List = pos to SET2
            and pos←pos+1
Step C    SET1 ← SET1 ∩ SET2 and meanwhile
          Check terminate and mark if terminate
          status = 1
          if SET1 != Empty
Step D      N← N+1, pos←pos+1
          else
Step E      N=SHIFT, SHIFT←SHIFT+1 ,pos =1
```

Figure 4 Algorithm for searching

Lemma 1 Let SET be the sub table with keys $w_{\lambda_{r,0}}$ and $w_{\lambda_{r,3}}$ for accessing $I_{\lambda_{r,0}}$ and $I_{\lambda_{r,3}}$,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

respectively. The access to $I_{\lambda_{x,0}}$ and $I_{\lambda_{x,1}}$ in SET using $f(w_{\lambda_{x,0}})$ or $f(w_{\lambda_{x,1}})$ function takes O(1) times.

Proof Let SET be the hashing a table with keys $w_{\lambda_{x,0}}$ and $w_{\lambda_{x,1}}$.

Therefore, accessing to $I_{\lambda_{x,0}}$ and $I_{\lambda_{x,1}}$ using $f(w_{\lambda_{x,0}})$ and $f(w_{\lambda_{x,1}})$ takes O(1) times by theorem 1 #

Lemma 2 To take to the inverted list from τ matching text[N] into SET takes O(1) times.

Proof Let text[N] be a character from string T with keys key $w_{\lambda_{pos,0}}$ and $w_{\lambda_{pos,1}}$. The access to $I_{\lambda_{pos,0}}$ and $I_{\lambda_{pos,1}}$ in table τ takes O(1) times.

Therefore, to take $I_{\lambda_{pos,0}}$ and $I_{\lambda_{pos,1}}$ into SET takes O(1) times by lemma 1 #

Definition 6 An intersection for continuity from position $\varepsilon 1$ to $\varepsilon 2$ of $I_{\lambda_{q\varepsilon 1,0}}$ and/or $I_{\lambda_{q\varepsilon 1,1}}$ in SET1 and $I_{\lambda_{b\varepsilon 2,0}}$ and/or $I_{\lambda_{b\varepsilon 2,1}}$ in SET2 is a set of pattern numbers that the character described by SET2 follows the character described by SET1. (6)

Example 5 Suppose SET1 ={<1:0:{1,2}>} and SET2={<2:0:{1,3}>}. The intersection for continuity from position 1 to 2 of SET1 and SET1 is {1}. Therefore, the character described by SET2 follows the character described by SET1 in pattern number 1.

Lemma 3 The intersection between SET1 and SET2 takes O(1) time.

Proof Let SET1 be a set of $I_{\lambda_{x1,0}}$ and/or $I_{\lambda_{x1,1}}$, and SET2 be a set of $I_{\lambda_{x2,0}}$ and /or $I_{\lambda_{x2,1}}$.

The access to $I_{\lambda_{x1,0}}, I_{\lambda_{x1,1}}, I_{\lambda_{x2,0}}$ and $I_{\lambda_{x2,1}}$ for intersection takes O(1) times by lemma 1 #

Theorem 5 The search algorithm takes O(|t|+locc) times.

Proof Let |t| be the sum of length of $T=t_1t_2t_3...t_n$ and locc be a number of comparisons that found a character in T and matches with the inverted list.

Step A take O(1) times.

Step B take O(|t|) times, because it repeats |t| times.

Step B1 and B2 take O(1) times by lemma 1.

Step C takes O(1) times by lemma 3.

Step D takes O(1) times by lemma 1. It then returns to step B2 which equals to a number of the occurrence of match between a character and the inverted list in the hash table that takes O(locc) time.

Step E takes O(1) times, because checking a variable for proceeding to other steps.

Therefore, the time complexity of a searching phase is O(|t|+locc)#

4. Conclusion

This paper presents a new solution to dynamic dictionary matching using an inverted list. We show that this solution takes (1) O(|P|) time for preprocessing, where |P| is a sum of the length of all patterns in set of pattern P; (2) O(|p|) time for insertion or deletion, where |p| is the length of pattern to be inserted or deleted; and (3) a search O(|t|+locc) time, where |t| is the length of input text and locc is the number of occurrences of matching between a character in the input text and in the inverted list. The search could improve to O(|t|) time when the number of navigator variables equal to the length of input text.

References

[1] A. V. Aho and M. J. Corasick. "Efficient string matching. An aid to bibliographic search". *Comm. ACM*, 1975. 333-340.

[2] A. Amir and M. Farach. "Adaptive dictionary matching". *Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science*, 1991. 760-766.

[3] A. Amir, M. Farach, R.M. Idury, J.A. La Poutr'e, and A.A. Schaffer. "Improved Dynamic Dictionary-Matching". *In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*. 1993, 392-401.

[4] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré , and A. A. Schäffex. "Improve dynamic dictionary matching". *Information and Computation, 199(2)*. 1995, 258-282.

[5] A, Amir, M. Farach, and Y. Matias. "Efficient randomized dictionary matching algorithms". *In CPM: 3rd Symposium on Combinatorial Pattern Matching*, 1992.

[6] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Manuscript*. 1991.

[7] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Sciences*. 1993.

[8] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Science, 49(2)*. 1994, 208-222.

[9] B. Commentz-Walter. "A string matching algorithm fast on the average". *In Proceedings of the Sixth International Collogium on Automata Languagees and Programming*. 1979, 118-132.

[10] C. Monz and M. de Rijke. (2002) *Inverted Index Construction*. [Online]. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf .

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[11] D. D. Sleator and R. E. Tarjan. "A data structure for dynamic trees". *Journal of Computer and System Sciences 26(3)*. 1983, 362-391.
[12] G. Navarro and M. Raffinot. "*Flexible Pattern Matching in Strings*". The press Syndicate of The University of Cambridge. 2002.
[13] O. R. Zaïane. "CMPUT 391: Inverted Index for Information Retrieval". *University of Alberta*. 2001.
[14] P. F. Dietz and D. D. Sleator. "Two algorithm for maintaining order in a list". *In Proceeding of the Nineteenth Annual ACM Symposium on Theory of Computing*. 1997, 365-372.
[15] R. B. Yates and B. R. Neto. "Mordern Information Retrieval". *The ACM press.A Division of the Association for Computing Machinery,Inc.* 1999, 191-227.
[16] S. Sahinalp and U. Vishkin. "Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract)". *In 37th Annual Symposium on Foundations of Computer Science*. 1996, 320-328.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol I,
IMECS 2010, March 17 - 19, 2010, Hong Kong

Data Structure for Dynamic Patterns

Chouvalit Khancome and Veera Boonjing, *Member, IAENG*

***Abstract*—String matching and dynamic dictionary matching are significant principles in computer science. These principles require an efficient data structure for accommodating the pattern or patterns to be searched for in a large given text. Moreover, in the dynamic dictionary matching, the structure is able to insert or delete the individual patterns over time. This research article introduces a new dynamic data structure named *inverted lists* for both principles. The inverted lists data structure, which is derived from the inverted index, is implemented by the perfect hashing idea. This structure focuses on the position of characters and provides a hashing table to store the string patterns. The new data structure is more time efficient than traditional structures. Also, this structure is faster to construct and consumes less memory than others.**

***Index Terms*—Data Structure, String Matching, Multiple String Matching, Suffix Tree, Trie, Bit-parallel, Hashing Table, Dictionary matching, Inverted Index, Inverted list.**

I. Introduction

There are many principles emerging from string processing such as string pattern matching, multiple string pattern matching called static dictionary, dynamic multiple patterns string pattern matching called dynamic dictionary matching. All of them deal with the pattern or patterns of string to be searched for in a large given text. Basically, pattern or patterns are generated to suitable data structures which are then provided for searching.

For solving the problem, string pattern matching deals with single string pattern $p=c_1c_2c_3...c_m$, while dynamic dictionary matching deals with multiple pattern strings $P=\{p^1, p^2,..,p^r\}$. And the patterns in P enable the ability to update individual patterns over time. Traditionally, Trie, Bit-parallel, Hashing table, and Suffix tree are the data structures used for accommodating p or P.

Trie, known as classic data structure for static dictionary, has been used for accommodating patterns for a long time. This structure employs an automaton to contain the set of states labeled by characters of patterns. Many algorithms are based on Trie such as the first linear time (Aho-Corasick [1]—extended from [15]), the sub-linear time (Commentz-Walter[11]—extended from [12]) and SetHorspool (mentioned in [17]). However, when implementing the Trie to applications a large amount of memory is consumed. And when the pattern is updated Trie needs to regenerate the structure with $O(|P|)$ time where $|P|$ is the sum of all pattern lengths.

Bit-parallel data structure employs the sequence of bit to store the patterns. Navarro and Raffinot [17] showed how to apply the single string Shift-Or and Shift-And to Multiple Shift-And[8], Multiple-BNDM[9], and [10]. Nevertheless, this structure is restricted by the word length of computer architecture; furthermore, it requires special methods which are more complex in converting the patterns to the bit form.

The first hashing idea was presented by Karp and Rabin [14] in single string matching. This algorithm takes the worst case scenario in $O(mn)$ time where m is the pattern length. Unfortunately, the dictionary matching algorithms which directly extend from [14] take $O(n|P|)$ time (exhaustive solution) where n is the length of the given text. A more efficient algorithm presented by Wu and Manber [24] creates the reverted Trie, the shift table, and implements the hashing table for storing the block of patterns to solve the problem. The last solution [25] improves Wu and Manber [24], but it does not support updating the patterns.

Suffix tree is implemented for accommodating the dynamic patterns. Generally, this structure does not directly support the dynamic patterns because it needs to employ the dynamic mechanism of McCreight [16], DS-List [14], or Weiner [23]. Thus, implementing suffix tree to algorithms must attach $O(log|P|)$ time because the tree structure is embedded by $log|P|$ for data accessing. The first suffix tree algorithm presented by Amir and Farach [2] is the first adaptive algorithm that displayed exhaustive time consumption. Subsequently, [3], [4], [5], [6], [7], [8], [9] and [22] showed the logarithmic algorithms of suffix tree generalization. Therefore, this structure is immediately challenged by how to escape from the factor of logarithmic time ($nlog|P|$). Moreover, the applications which are implemented by suffix tree take more memory than Trie as well.

For solving the problem of information retrieval, the inverted index has been applied; this structure can be adapted to several data structures. The motivation of this research is from the inverted index which focuses on the keywords. But the new structure is based on characters.

This research article proposes to adapt the inverted index [13], [18], and [19] to create a new data structure called inverted lists for accommodating the pattern or patterns. This structure uses $O(m)$ time and $O(m+|\lambda|)$ space for managing the single pattern string where m is the length of pattern p. For dynamic dictionary matching, the inverted lists structure takes $O(|P|)$ time and $O(|\lambda|+|P|)$ space where λ are any characters which are exactly used in a finite alphabet Σ

Manuscript received December 8, 2009.

Chouvalit Khancome is a PhD student in Computer Science at King Mongkut's Institute of Technology Ladkrabang, Thailand. He is a teacher in the Department of Computer Science, Rajanakarindra Rajabhat University, Thailand; (e-mail: chouvalit@ hotmail.com).

Veera Boonjing is an Associate Professor in Computer Science, Department of Mathematics and Computer Science at King Mongkut's Institute of Technology Ladkrabang, Thailand. Also, he is working in National Centre of Excellence in Mathematics, PERDO, Bangkok, Thailand 10400, (e-mail: kbveera@kmitl.ac.th).

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol I,
IMECS 2010, March 17 - 19, 2010, Hong Kong

and $\lambda \subseteq \sum$. Importantly, this structure can handle the inserting or deleting of the individual pattern in O(|*p*|) time where *p* is the individual pattern to be inserted or deleted. In experimental results, this structure is constructed faster and uses less memory than the traditional data structures.

The remaining sections are organized as follows. Section II shows how to derive the inverted index and the perfect hashing principle for accommodating the new data structure. Section III describes the inverted lists for string pattern matching. Section IV shows the inverted lists for dictionary matching. Section V illustrates the implementations and the experimental results, and section VI is a conclusion.

II. Deriving the Principles

A. Deriving the Inverted Index

The inverted index structure represents the words in the target documents by the form of *<documentID, word:pos>* where '*documentID*' is the indicated number referring to the number of documents, '*word*' is the keywords in the document, and '*pos*' is the occurrence position of '*word*' in the *documentID*. The original inverted index [13], [18], and [19] assign all documents as $D=\{D_1...D_n\}$ where D_i is any document that contains the various keywords with different positions and $1 \le i \le n$.

Each document is analyzed for keeping keywords and their positions. For instance, if the document D_1 has the keywords w_a:*1*,w_b:*2*,w_c:*3*, it can be said that the keyword w_a appears at position 1, w_b appears at position 2, and w_c appears at position 3. Then, all keywords can be rewritten by the form of *word: (posting lists)* where '*posting list*' is *(documentID: word position in that document)*. Thus, all keywords in document D_1 can be rewritten as w_a: (1:1), w_b: (1:2), and w_c: (1:3) respectively.

Afterwards, the keywords and posting lists are converted into the suitable data structures such as B$^+$tree, suffix tee, and suffix array. This research focuses on the position of characters instead of the keywords. Initially, the document *D* is replaced by the pattern *P*, and each D_i is replaced by p_i. For instance, if there are the patterns $P=\{aab, aabc, aade\}$ then the patterns are assigned as D_1=*aab*, D_2=*aabc*, and D_3=*aade*. Then, they are defined by the form of *character :<the occurrence position of character in pattern: the indicated status of the last character of pattern: the number of pattern in P>*; e.g., *a:<1:0:1>, <2:0:1>, <1:0:2>, <2:0:2>,...* . Each list in this form is called the "*individual posting list*". Using this method, all of the individual posting lists can be applied for accommodating the dictionary.

B. Deriving the Perfect Hashing

The most powerful hashing principle is the perfect hashing which takes O(1) time in worst-case performances (shown in [26], [27] and [28]) where *n* is the size of data. This structure is suitable for the set of static keys such as the reserved words in the programming language. For this reason, the perfect hashing is chosen for implementing the inverted lists.

The perfect hashing uses the universal key *U* to accommodate all keys for accessing all data in the table and two-level schemes for implementation. The first level is the *n* keys for hashing with chaining to the second level by function *f(n)*, and the second level is the data items associated with the corresponding key of *n*. This research assigns $\sum$ as the universal key *U* and $f(\lambda)$ as *f(n)* for the first level of the perfect hashing table and the groups of posting lists as the data items in the second level where $\lambda \subseteq \sum$.

III. Inverted Lists for Single Pattern

Definition 1 Let $p=c_1c_2c_3...c_m$ be the pattern input, and c_k is any alphabet that occurs in *p* at position *k* where $k=1,2,3,...,m$. The inverted list of c_k can be written by c_k: *<k:0>* if only if $k<m$, or c_k:*<k:1>* if only if $k=m$. Symbolically, c_k: *<k:0>* is represented by c: I_{k_0} , and c_k:*<k:1>* is represented by c: I_{k_1} .

Example 1. The inverted lists of *p=aabcz*. We have c_1=*a*, c_2=*a*, c_3=*b*, c_4=*c*, and c_5=*z*. The whole inverted list of *p* are *a:<1:0>*, *a:<2:0>*, *b:<3:0>*, *c:<4:0>*, and *z:<5:1>*.

Definition 2 The perfect hashing table which is provided for storing the inverted lists of the single pattern *p* is called the inverted lists table for single patterns and is denoted by τ_s .

Example 2. The table τ_s of pattern *p=aabcz*.

Table 1. The inverted list table τ_s of *p=aabcz*.

Σ	I_{k_0} / I_{k_1}	(i.e., the inverted list)
a	I_{1_0}, I_{2_0}	*<1:0>*,*<2:0>*
b	I_{3_0}	*<3:0>*
c	I_{4_0}	*<4:0>*
z	I_{5_1}	*<5:1>*

Lemma 1 Let I_{k_0} and I_{k_1} be the inverted list of $p=c_1c_2c_3...c_m$. If I_{k_0} and I_{k_1} are stored in the hash table τ_s then accessing I_{k_0} or I_{k_1} takes O(*1*) time where $k=1,2,3,...,m$.

Proof Each inverted list I_{k_0} or I_{k_1} can be retrieved from the table τ_s in O(*1*) time. Given *f(x)* be a hashing function which c_{k_0} is a key to access I_{k_0}, and c_{k_1} is the key to access I_{k_1} where *c* is the character in λ. The table τ_s is implemented by the perfect hashing table. Thus, to retrieve the inverted list I_{k_0} by $f(c_{k_0})$ or to retrieve the inverted list I_{k_1} by $f(c_{k_1})$ takes O(*1*) times by the hashing properties. □

This phase creates the inverted list table τ_s for all alphabets in λ. The next method reads the characters one by one and generates to inverted lists, and each inverted list is added into the table τ_s. Algorithm 1 shows this method.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol I,
IMECS 2010, March 17 - 19, 2010, Hong Kong

Algorithm 1.
Input : $p=c_1c_2c_3...c_m$
Output: Table τ_s of p
1. Create table τ_s
2. j=1
3. while (j<=m) do
4. Create the inverted list of $c_j \rightarrow \tau_s$ at char(c_j)
5. j←j+1
6. end while

Algorithm 1 takes O(m) time and O($m+|\lambda|$) space shown as the proof in Theorem 1.

Theorem 1 Let $p=c_1c_2c_3...c_m$ be the pattern input, and given τ_s be the inverted list table of single pattern. Generating all characters of p to the inverted lists and adding all inverted lists into τ_s takes O(m) time, and the table τ_s uses O($m+|\lambda|$) space.

Proof For time complexity, the hypothesis is that the whole characters of p are generated into the table τ_s. Line 1 creates the table, and line 2 initializes variables, which take O(*1*). Line 3 needs to repeat m rounds, and it also takes O(m) time. Line 4 is O(*1*) by Lemma 1 while line 5 takes O(*1*) as line 2. Therefore, the preprocessing time take O(m) time. □

For space complexity, the table τ_s is created with the size $|\lambda|$ for all alphabet of $\lambda \subseteq \sum$. Each inverted list of c_k of $p=c_1c_2c_3...c_m$ uses one space per one inverted list then for $k=1$ to $k=m$ take m space as well. Hence, all required spaces of τ_s is O($m+|\lambda|$).□

IV. INVERTED LISTS FOR DYNAMIC PATTERNS

Definition 3 Let $P=\{p^1, p^2, p^3,..., p^r\}$ be the set of patterns where p^i is the pattern i^{th} of m character $\{c_1c_2c_3...c_m\}$, and $1 \le i \le r$. An individual posting list of a character c_k from the pattern p^i is defined as c_k:<*k*:0:*i*> if $k<m$, or c_k:<*k*:1:*i*> if $k=m$. Symbolically, c_k:<*k*:0:*i*> is $\varphi_0^{k,i}$, and c_k:<*k*:1:*i*> is $\varphi_1^{k,i}$ where $1 \le k \le m$.

As in Definition 3, if $P=\{p^1=aab, p^2=aabc, p^3=aade\}$ then the documents are $p^1=a_1a_2b_3$, $p^2=a_1a_2b_3c_4$ and $p^3=a_1a_2d_3e_4$. The individual posting lists of P are defined as below:

p^1= a:<1:0:1>, a:<2:0:1>, b:<3:1:1>,
p^2=a:<1:0:2>, a:<2:0:2>, b:<3:0:2>
c:<4:1:2>, and
p^3= a:<1:0:3>, a:<2:0:3>, d:<3:0:3>,
e:<4:1:3>.

Notice that all individual posting lists above can be grouped to a new form such as *a*:<1:0:{1,2,3}>, <2:0:{1,2,3}>, *b*:<3:1:{1}>, <3:0:{2}>, and so on. Definition 2 shows how to group the posting lists to a new form.

Definition 4 Let l_{max} be the maximum length of patterns in $P=\{p^1,p^2,p^3,...,p^r\}$, and let ε be the position of the same character λ which appears in the various patterns of P where $1 \le \varepsilon \le l_{max}$ and $\lambda \subseteq \sum$. The posting lists of λ are $\{\varphi_0^{\varepsilon_1}, \varphi_0^{\varepsilon_2},..., \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ or $\{\varphi_1^{\varepsilon_1}, \varphi_1^{\varepsilon_2},..., \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ where $1 \le \{i,l,...,p,q\} \le r$. A group of posting lists of λ can be defined as follows.

1. If the posting lists are $\{\varphi_0^{\varepsilon_1}, \varphi_0^{\varepsilon_2},..., \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ then a group of posting lists of λ is $\lambda_{\varepsilon,0}$.

2. If the posting lists are $\{\varphi_1^{\varepsilon_1}, \varphi_1^{\varepsilon_2},..., \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ then a group of posting lists of λ is $\lambda_{\varepsilon,1}$.

Definition 5 The inverted list of alphabet λ is defined as $I_{\lambda_{\varepsilon,0}}$ if only if the group of posting lists is $\lambda_{\varepsilon,0}$. Similarly, the inverted list of alphabet λ is denoted as $I_{\lambda_{\varepsilon,1}}$ if only if the group of posting lists is $\lambda_{\varepsilon,1}$.

With Definitions 4 and 5, if the posting lists are *a*:<1:0:{1,2,3}>, *a*:<2:0:{1,2,3}> then the groups of posting lists must be written as $I_{a_{1,0}}$ and $I_{a_{2,0}}$ respectively (shown in table 2).

Definition 6 The perfect hashing table which provides for all alphabets over $\sum$ and their corresponding inverted lists of P is called the inverted lists table for the dynamic dictionary; this table is denoted as τ_d.

From Definition 6, it can be said that all characters λ are stored in the first column of τ_d, and $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$ are stored in the second column of τ_d. For instance, if there is $P=\{aab, aabc, aade\}$ then P can be implemented to the perfect hashing table as the table 2.

Example 3. Table 2 shows the table τ_d of $P=\{aab, aabc, aade\}$.

Table 2. The inverted list table τ_d of $P=\{aab, aabc, aade\}$.

f(λ)	Inverted lists	i.e., the granular inverted lists
a	$I_{a_{1,0}}$, $I_{a_{2,0}}$	<1:0:{1,2,3}>, <2:0:{1,2,3}>
b	$I_{b_{3,1}}$, $I_{b_{3,0}}$	<3:1:{1}>, <3:0:{2}>
c	$I_{c_{4,1}}$	<4:1:{2}>
d	$I_{d_{3,0}}$	<3:0:{3}>
e	$I_{e_{4,1}}$	<4:1:{3}>

A. Inverted lists table construction

First of all, the empty table τ_d is built for λ, and the entire patterns are then generated to the inverted lists and are added into the table τ_d after the table is constructed. If the inverted lists of target character are already stored in the table, only the number of pattern is added to the corresponding inverted lists; otherwise, a new inverted list is created and added into the table. Algorithm 2 shows this method.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol I,
IMECS 2010, March 17 - 19, 2010, Hong Kong

Algorithm 2.
Input: $P=\{p^1,p^2,...,p^r\}$
Output : the table τ_d of P
1. initiate τ_d
2. for $i \leftarrow 1$ to r do
3. for $j \leftarrow$ to m do
4. if Exist(p_j^i)=null then
5. $\tau_d \leftarrow \varphi_0^{j,i}$ or $\varphi_1^{j,i}$
6. else
7. $I_{char(J)_{j,0}}$ or $I_{char(J)_{j,1}} \leftarrow i$
8. end if
9. end for
10. end for

For analyzing time and space, Algorithm 2 is referred as proof. Theorem 2 shows the time complexity and Theorem 3 illustrates the space complexity.

Lemma 2 If there are the inverted lists $I_{\lambda_{j,0}}$ or $I_{\lambda_{j,1}}$ of λ in τ_d then to access all inverted lists of λ uses $O(1)$ time.
Proof Each alphabet λ is a unique character in $\sum$, and λ is implemented as the first level of the perfect hashing table taking O(1) time. The inverted lists $I_{\lambda_{j,0}}$ or $I_{\lambda_{j,1}}$ are implemented as the second level of the perfect hashing table; therefore, each data item takes O(1) time, and all items in the second level of the table can be applied to O(1) as all individual items. □

Theorem 2 Let $P=\{p^1,p^2,p^3,...p^r\}$ be the given patterns. All patterns in P are generated into the table τ_d in $O(|P|)$ time where $|P|$ is the sum of all pattern lengths in P.
Proof The proof is that all characters of P are generated to inverted lists and are added into τ_d in $O(|P|)$ time. All of the pattern lengths are denoted as $|p^1|, |p^2|, |p^3|,..., |p^r|$. For the initial step, the table τ_d is built in $O(1)$ time. As soon as the table τ_d is built completely, each pattern is scanned by individual character from the first character to the last character. Thus the time to scan equals the number of pattern length; therefore, all patterns are scanned from the pattern 1 to pattern r. This step takes the processing time as $|p^1|+|p^2|+|p^3|+...+|p^r|=|P|$, and it reaches to the hypothesis step by the last character of p^r. Therefore the inverted lists construction takes $O(|P|)$ time. Meanwhile, to access the table τ_d for storing the inverted list takes $O(1)$ time by Lemma 2. Hence, the preprocessing time is proved in $O(|P|)$ time. □

Theorem 3 The table τ_d requires $O(|\lambda|+|P|)$ space for accommodating the whole inverted lists of P; where $|P|$ is the sum of pattern lengths of P, and τ_d is the inverted lists table.
Proof All patterns in P contain the various characters over λ by the size $|\lambda|$ where $\lambda \subseteq \sum$, and the table τ_d is implemented as the perfect hashing table. The algorithm is proved as all characters of P are generated to inverted lists and are added into the table τ_d with $|P|$ space. The lengths of P are $|p^1|, |p^2|, |p^3|, ..., |p^r|$, and each p^i contains the string $\{c_1c_2c_3...c_m\}$ where $1 \le i \le r$. The length of this string is denoted by $|p^i|$. For the initial step, the first column of table τ_d is created by $|\lambda|$ size. Each inverted list is created by the preprocessing phase for all patterns of P; therefore, each inverted list of string $\{c_1c_2c_3...c_m\}$ in each p^i only takes one space per one list. Thus, the space is equal to $|p^1|+|p^2|+|p^3|+...+|p^r|=|P|$ for the second level of perfect hashing table. Hence, the space of τ_d is $O(|\lambda|+|P|)$. □

B. Pattern Insertion

The pattern insertion deals with the problem of adding all characters of the individual pattern into the inverted table τ_d and maintains the stable dictionary. Let p^ϕ be a new pattern for insertion where ϕ is a unique number that does not appear in the dictionary before inserting the pattern. We begin to search for the existence of p^ϕ in the table τ_d by the function PExist(). The insertion method will be executed if only if the result is null. Next, all inverted lists of p^ϕ are generated and are added into the table as Algorithm 2. This algorithm is illustrated by Algorithm 3 below.

Algorithm 3.
Input : $p^\phi=\{c_1c_2,...,c_m\}$
Output : p^ϕ is stored in τ_d
1. if PExist(p^ϕ) =null then
2. for j $\leftarrow 1$ to m do
3. if Exist(p_j^ϕ)=null then
4. $\tau_d \leftarrow \varphi_0^{j,\phi}$ or $\varphi_0^{j,\phi}$
5. else
6. $I_{char(J)_{j,0}}$ / $I_{char(J)_{j,1}} \leftarrow \phi$
7. end if
8. end for
9. end if

Theorem 4 Let $p=\{c_1c_2c_3...c_m\}$ be the new individual pattern to be inserted into the existing dictionary $P=\{p^1,p^2,p^3,...p^r\}$. The insertion time is $O(|p|)$ where $|p|$ is the length of p.
Proof Algorithm 3 is referred for the proof. Given p^ϕ be a new pattern which contains a string $\{c_1c_2c_3...c_m\}$ where ϕ is an non-existing number of the pattern in P. The length of p^ϕ is m and represented by $|p|$. In the initial step, line 1 repeats the search for each character from c_1 to c_m, and it takes m operation which is $O(|p|)$ time. The accessing of the inverted list takes $O(1)$ time by Lemma 2. For the inner loop, if a new pattern does not exist in P, line 2 will insert the inverted lists from c_1 to c_m. All operations use $|p|$ time, and the hypothesis is proved as well. Meanwhile, line 2, line 4, or line 6 also access the table, and they take $O(1)$ by Lemma 2. It can be said that all operations of line 2 take $O(|p|)$ time; therefore, all characters of p^ϕ are converted and added into the existing table in $O(|p|)$ time. □

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol I,
IMECS 2010, March 17 - 19, 2010, Hong Kong

C. Pattern Deletion

Pattern deletion consists of two methods: (1) looks for the required pattern to delete, and (2) repeats and removes the inverted lists of the target pattern one by one. For deletion mechanism, if the corresponding inverted lists have only one posting list, it will be deleted immediately. Otherwise we will delete only an inverted list in the inverted lists group when the pattern number equals σ. The deletion algorithm is described by Algorithm 4 below.

Algorithm 4.
Input: $p=\{c_1c_2c_3...c_m\}$
Output : p is removed from τ_d
1. if(ExistDel(p) = σ) then
2. for $j \leftarrow$ 1 to m do
3. if posting lists in $I_{char(j),0} / I_{char(j),1} > 1$ then
4. Delete posting lists number equal to σ
5. else
6. Delete $I_{char(j),0} / I_{char(j),1}$ of char(p_j^σ)
7. end if
8. end for
9. end if

Example 4. Take p=*aab* off from P=*{aab,aabc,aade}*.
As the example before, the table 2 is referred, and the deletion is begun in line 1. This inspects and returns the number σ for deletion. The characters in the pattern '*aab*' are converted to *a:<1:0:{1}>*, *<2:0:{1}>*, and *b:<3:1:{1}>*. Then the deleting mechanism is started for deleting the inverted lists one by one. By line 4, the inverted list of *a:<1:0:{1}>* and *<2:0:{1}>* are updated as *<1:0:{2,3}>* and *<2:0:{2,3}>* (by taking the number '*1*' off). Otherwise, the inverted list *b:<3:1:{1}>* is deleted by line 6.

Theorem 5 The deletion pattern p from the existing table of P takes O($|p|$) time where p is the target pattern to be deleted, and $|p|$ is the length of pattern p.
Proof Referring to Algorithm 3, let p^i be a pattern to be deleted, which p^i contains a string $\{c_1c_2c_3...c_m\}$ with the length m. The length m is denoted by $|p|$, and 'i' is the number of the existing pattern i^{th} in P. The hypothesis is that all characters of p^i are removed from the inverted list table of P. Line 2 loops to remove c_1 to c_m. Each operation for accessing the inverted list uses O(1) by Lemma 2. Thus the main step is initiated by line 2. This step repeats to read one by one from c_1 to c_m and removes the matched inverted lists from τ_d. All operations take $|p|$ time while line 4 or line 6 uses the constant time by Lemma 2. Therefore, to delete all characters of pattern p^i from τ_d takes O($|p|$) time. □

V. Implementation and Experimental Results

A. Implementations

We used the Dell Latitude D500 notebook with Intel Pentium M 1.3 GHz, 512 MB of RAM, and running on Windows XP Home as a running application machine. We implemented Aho-Corasick Trie [1] (named AC Trie), Reverted Tire of SetHorspool in [9], dynamic Suffix tree of [16], and our inverted lists. And the abstract data type (ADT) of java.util.Vector in Java language was employed for accommodating all structures.

The $|\Sigma|$ was 52 letters of English alphabet; 'A' to 'Z' and 'a' to 'z'. We programmed to randomize each pattern with the various lengths of 3 to 20 characters. The programs randomized the pattern groups of 10, 50, 100, 500, 1000, 5000, 10000, and 50000. Each group contained 10 files, and each file was performed to test 10 times and the average was given.

B. Experimental Results

The tests measure the processing time in seconds, and the memory usages in Kilo-Bytes. The processing time of inverted lists construction was faster than the traditional data structures shown in table 3.

Table 3. Comparing processing time (Seconds).

Pattern Number	AC Trie	Reverted Trie	Suffix Tree	Inverted List
10	0.41	0.62	0.35	0.29
50	0.19	0.10	0.16	0.05
100	0.17	0.27	0.59	0.15
500	0.66	0.98	21.67	0.39
1000	1.34	2.08	-	0.86
5000	13.79	9.02	-	4.55
10000	49.43	26.28	-	7.91
50000	550.74	121.11	-	47.63

For using memory, the inverted lists structure used less memory than the others. However, the suffix tree structure was not able to generate in the case of pattern numbers over 500 patterns because Java language used excessive memory. The results are shown by table 4.

Table 4. Comparing memory usage (KB).

Pattern Number	AC Trie	Reverted Trie	Suffix Tree	Inverted List
10	4.74	4.95	24.86	4.87
50	4.83	4.98	48.34	4.89
100	4.92	5.02	896.12	4.90
500	5.60	5.66	2512.56	5.11
1000	6.29	6.30	-	5.32
5000	11.07	11.23	-	7.57
10000	15.86	16.15	-	9.83
50000	54.56	55.15	-	23.38

VI. Conclusion

This research has presented the new data structure called inverted lists for dynamic patterns of string matching and dynamic dictionary matching. The inverted lists structure uses O(m) time and O($m+|\lambda|$) space for accommodating the single pattern string where m is the length of pattern. In the dynamic patterns, this structure takes O($|P|$) time and O($|\lambda|+|P|$) space for accommodating multi-patterns string where P is the sum of pattern lengths, and λ represents any characters which are exactly used in a set of a finite alphabet Σ. Furthermore, this structure is able to insert or delete the individual pattern in O($|p|$) time where p is the individual pattern to be inserted or deleted. In experimental results, this

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

structure is faster and uses less memory than the traditional data structures.

REFERENCES

[1] A. V. Aho and M. J. Corasick. "Efficient string matching: an aid to bibliographic search". *Comm. ACM*, 1975, 333-340.

[2] A. Amir and M. Farach. "Adaptive dictionary matching". *Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science*, 1991, 760-766.

[3] A. Amir, M. Farach, R.M. Idury, J.A. La Poutr'e, and A.A. Schaffer. "Improved Dynamic Dictionary-Matching". *In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*. 1993, 392-401.

[4] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré , and A. A. Schäffer. "Improved dynamic dictionary matching". *Information and Computation, 199(2)*. 1995, 258-282.

[5] A, Amir, M. Farach, and Y. Matias. "Efficient randomized dictionary matching algorithms". *In CPM: 3rd Symposium on Combinatorial Pattern Matching*, 1992.

[6] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Manuscript*. 1991.

[7] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Sciences*. 1993.

[8] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Science, 49(2)*. 1994, 208-222.

[9] A. Amir, M. Farach, R. M. Idury, J. A. La Poutre and A. A. Schäffer. "Improved Dynamic Dictionary Matching". *Information and computation*. 119, 1995, 258-282.

[10] A. Moffat, and J. Zobel. "Self-Indexing Inverted Files for Fast Text Retrieval". ACM Transactions on Information Systems, Vol. 14, No. 4, 1996, 349-379.

[11] B. Commentz-Walter. "A string matching algorithm fast on the average". *In Proceedings of the Sixth International Colloqium on Automata Languages and Programming*. 1979, 118-132.

[12] R.S. Boyer. And J.S. Moore. "A fast string searching algorithm". Communications of the ACM. 20(10), 1977, pp. 762-772.

[13] C. Monz and M. de Rijke. (11, 02, 2002). *Inverted Index Construction*. [Online]. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf.

[14] D. D. Sleator and R. E. Tarjan. "A data structure for dynamic trees". *Journal of Computer and System Sciences 26(3)*. 1983, 362-391.

[15] D.E. Knuth, J.H. Morris, V.R. Pratt, "Fast pattern matching in strings". SIAM Journal on Computing 6(1), 1997, 323-350.

[16] E.M. McCreight. "A space-economical suffix tree construction algorithm". Journal of Algorithms, 1976, 23(2):262-272.

[17] G. Navarro and M. Raffinot. "*Flexible Pattern Matching in Strings*". The Press Syndicate of The University of Cambridge. 2002.

[18] O. R. Zaïane. "CMPUT 391: Inverted Index for Information Retrieval". *University of Alberta*. 2001.

[19] R. B. Yates and B. R. Neto. "Mordern Information Retrieval". *The ACM press.A Division of the Association for Computing Machinery,Inc*. 1999, 191-227.

[20] S. Melnik, Sriram Raghavan, Beverly Yang and Hector Garcia-Molina. "Building a Distributed Full-Text Index for the Web". ACM Transactions on Information Systems, Vol. 19, No. 3, 2001, 217-241.

[21] T.W. Lam, K.K. To. (03, 11, 2005). "The Dynamic Dictionary Matching Problem Revisited". [Online]. Available : http://citeseer.ist.psu.edu/413873.html.

[22] H-L. Chan, W-K. Hon, T-W. Lam, and K. Sadakane. "Dynamic dictionary matching and compressed suffix trees". SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms. 2005, 13-22.

[23] P. Weiner. "Linear Pattern Matching Algorithms". In Proceedings of Symposium on Switching and Automata Theory. 1973, 1-11.

[24] S.Wu and U. Manber. "A fast algorithm for multi-pattern searching". Report tr-94-17, Department of Computer Science, University of Arizona, Tuscon, AZ, 1994.

[25] Y. D. hong, X. Ke and C. Yong. "An improved Wu-Manber multiple patterns matching algorithm". Performance, Computing, and Communications Conference, 2006. IPCCC '2006. 25th IEEE International 10-12, 2006, 675-680.

[26] F. C. Botelho. "Near-Optimal Space Perfect Hashing Algorithms". The thesis of PhD. in Computer Science of the Federal University of Minas Gerais, 2008.

[27] R. Pagh. (11, 08, 2009). "Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions". [Online]. Available: www.it.-c.dk/people/pagh/papers/hash.pdf.

[28] Wikipedia (10,07,2009). "Hash function". [Online]. Available: en.wikipedia.org/wiki/Hash_fuction.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Computing and Informatics, Vol. 22, 2003, 1001–1026, V 2012-Jun-18

A NEW LINEAR-TIME DYNAMIC DICTIONARY MATCHING ALGORITHM

Chouvalit Khancome[1]

[1]*SoftwareSystemsEngineeringLaboratory*
Department of Mathematics and Computer Science
Faculty of Science, King Monkut's Institute of Technology at Ladkrabang(KMITL)
Ladkrabang,Bangkok 10520,Thailand
e-mail: chouvalit@hotmail.com, chouvalit.kha@csit.rru.ac.th

Veera Boonjing[1,2]

[2]*NationalCentreofExcellenceinMathematics, PERDO*
Bangkok, Thailand 10400
e-mail: kbveera@kmitl.ac.th

Abstract. This research presents *inverted lists* as a new data structure for the dynamic dictionary matching algorithm. The inverted lists structure, which derives from the inverted index, is implemented by the perfect hashing table. The dictionary is constructed in optimal time and the individual patterns can be updated in minimal time. The searching phase scans the given text in a single pass, even in a worst case scenario. In experimental results, the inverted lists used less time and space than the traditional structures; the searches were processed and showed an efficient linear time.

Keywords: Dynamic Dictionary Matching; Static Dictionary Matching; Multiple Pattern String Matching; Inverted Index; Inverted Lists; Trie; Bit-parallel; Hashing Table.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

1 INTRODUCTION

Dictionary matching is one of the main priciples in classical string processing. This principle deals with a large number of string patterns $P=\{p^1,p^2,..,p^r\}$ that can be searched simultaneously in a given text $T=\{t_1t_2t_3...t_n\}$. There are plenty of new applications in computer science that apply this principle to solve their problems (e.g.,[32, 33, 34, 39]) including the operating system commands (Unix grep command using Commentz-Walter [9] and agrep using Wu-Manber[25]), intrusion detection systems (e.g., SNORT using Aho-Corasick[1], Commentz-Walter [9], and Wu-Manber[25]), and so on. Traditionally, the target patterns are generated to a suitable data structure (called dictionary) in the pre-processing phase. Then the searching phase scans and compares the given text with data in the dictionary for finding all pattern occurrences. Typically, if the dictionary can not support the updating of individual patterns,it is called the static dictionary. In contrast, if the dictionary has the ability to delete or insert the individual patterns over time, it is called the dynamic dictionary.

Since dictionary matching is a fundamental problem; Trie and Bit-parallel data structures have been used to accommodate the static dictionary. Trie-based algorithms (Aho-Corasick [1], Commentz-Walter [9], and SetHorspool(mentioned in [18])) take the linear time or sub-linear time to solve problems. On the other hand, the suffix tree has also traditionally been used for creating the dynamic dictionary. Naturally, suffix tree-based algorithms work in logarithmic time ($O(nlog|P|)$) where $|P|$ is the sum of pattern lengths from p^1 to p^r and n is the length of T. Although suffix tree-based algorithms are able to handle the mechanism of dynamic dictionary matching, all of them need some flexible structures such as McCreight [14], DS-List [22], or Weiner [23]. Unfortunately, these structures are also embedded with a logarithmic time ($log|P|$) which often drives the searching phase to O($nlog|P|$). It can therefore be said that all suffix tree-based algorithms are limited by the $log|P|$ time trap. The key to solving the dynamic dictionary problem can be found in reducing the space and resolving the $log|P|$ time complexity.

Considered by efficiency, the algorithms using static dictionary are more efficient than the algorithms implementing the dynamic dictionary, but the dynamic dictionary can be updated by the individual patterns in an optimal time while the static dictionary takes an exhausive time (regenerates all patterns). Although there were several static dictionary matching solutions [35, 36, 37, 38, 40, 41] shown recently, they still tried to improve the classic data structures for accommodating the dictionary. For example, solutions [36, 37, 38] improved Trie structure, solution [35] used tree, solution [40] employed Bit-parallel; as well as the solution [41] used the hashing of Wu-Manber[25] and a quick sort principle. Therefore, a new superior data structure and a new faster algorithm will always need to incorporate both efficiency and flexibility.

Up until now, the inverted index and the perfect hashing table have been popular data structures used for solving a variety of problems. The inverted index has been viewed as an excellent data structure in solving information retrieval problems such

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

as [16, 28, 29, 15]. The principle of the inverted index found in [17, 27, 26] focused on the keywords and their positions. This principle can be applied to several data structures for offering a fast search. As well as, the perfect hashing table is used to accommodate the data in minimal space $O(n)$ and to provide the search method in minimal time as $O(1)$. It must then asks if there is a way to combine both these excellent data structures to create a new dynamic dictionary matching algorithm.

The first solution [12] presented a new method of dynamic dictionary matching using the inverted lists. This solution combined an inverted index idea and a normal hashing table to create the inverted lists structure. This algorithm takes $O(n + locc)$ time in an average case where $locc$ is the number of characters to be matched while comparing characters in the given text. Unfortunately, this solution leads to backtracking which takes an exhausive time in $O(n|P|)$. The next solution [30] showed a new static dictionary matching algorithm. This solution concentrated on the perfect hashing table to implement the inverted lists structure. Importantly, it takes the linear time ($O(n)$) to search the given text even in a worst case scenario. Surprisingly, when the inverted lists were adapted to accommdate the dynamic dictionary; the searching algorithm still works in a linear time. Furthoremore, deriving the inverted lists structure to the other field of computer science, the solution [31] illustrated two algorithms of single string matching that were more efficient in the case of small alphabet sizes especially when searching on binary digits.

This paper proposes to adapt a linear time static dictionary matching [30] to create a new solution of linear time dynamic dictionary matching, which avoids the backtracking of text scaning of [12]. This new approach concentrates on the inverted lists structure which is implemented by the perfect hashing table and explores different ways to utilize flexible updating and efficient linear-time usage. In theoretical results, the inverted lists structure is constructed in $O(|P|)$ time and space, and the insertion or the deletion of the individual pattern take $O(|p|)$ time where $|p|$ is the length of pattern to be inserted or deleted. The searching phase takes only $O(n)$ time even in a worst case scenario. In experimental results, the inverted lists structure consumes less time and space than the traditional data stuctures. The new algorithm takes a linear-time to process the searching phase. Compared to previously well known static dictionary algorithms:Aho-Corasick[1] and SetHorspool(mentioned in [18]), the inverted lists algorithm searches faster than SetHorspool but slower than Aho-Corasick.

Section 2 summaries the related algorithms and shows the derivations from the inverted index and the perfect hashing principle. Section 3 explains the details of the inverted lists structure and the dynamic mechanism. Section 4 illustrates the searching algorithm and its example. Section 5 details the implementations; as well as the experimental results are reported. Section 6 is the discussion and the suggestions for improving the algorithm and the data structure. Section 7 is the conclusion and planned future works.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2 RELATED WORKS AND DERIVING PRINCIPLES

This section presents a history on related works of static and dynamic dictionary matching algorithms. Furthermore, the ideas that derived the inverted index and the perfect hashing table are described.

2.1 Related works

2.1.1 Static dictionary matching algorithms

Basically, Trie, Bit-parallel and Hashing table have been employed for storing the dictionary. The static dictionary algorithms always work best in linear time or sub-linear time, but in a worst case scenario often take an exhausive time in $O(n|P|)$. An overview of this principle is described below.

Trie has been used for accommodating patterns for a long time. The first linear time (Aho-Corasick [1]–extended from [13] using $O(n + nocc)$), the sub-linear time (Commentz-Walter [9]) and SetHorspool(mentioned in [18] taking $O(n|P|)$ in a worst case scenario) are the solutions using Trie where $nocc$ is the number of pattern occurences. Although the existing solution [10] tries to improve the static dictionary, the patterns still need to regenerate when the dictionary is updated. The main disadvantage is that when implementing Trie to applications a large amount of memory is cosumed.

Alternatively, Bit-parallel is also popular in accommodating the static dictionary. Bit-parallel-based algorithms employ the sequences of bit to store the patterns. Navarro and Raffinot [18] showed how to apply the single string Shift-Or and Shift-And to Multiple Shift-And[4] and Multiple-BNDM[18]. This principle is restricted by the word length of computer architecture; furthermore, it requires special methods which are more complex in converting the patterns to the bit form.

On the other hand, the first hashing algorithm was presented by Karp and Rabin [22] in single string matching. This algorithm takes the worst case scenario in $O(mn)$ time where m is the pattern length. Unfortunately, the dictionary matching algorithms which directly extend from [22] take $O(n|P|)$ time (comparable to the exhaustive solution). A more efficient algorithm presented by Wu and Manber [25] creates the shift table and implements the hashing table for storing the block of patterns to solve the problem. The new solution [11] improves Wu and Manber [25] and provides a faster solution to this principle.

Recently developed, solutions [36, 37, 38] improved Trie structure to accommodate the patterns especially [37] shown minimal space of solution. Other solutions, which employed those classic data structures(e.g.,Trie,Bit-parallel, and Hashing), can be found in [35, 40, 41].

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2.1.2 Dynamic dictionary matching algorithms

Dynamic dictionary matching algorithms are scalable in terms of the flexibility of their patterns, but they are disadvantaged in time and memory consumption. Suffix tree-based algorithms are able to handle the mechanism of dynamical patterns. This structure has led the dynamic dictionary research community to explore new solutions. The first suffix tree based dictionary presented by Amir and Farach [2] was the first adaptive algorithm that displayed a consuming time. Subsequently, [4, 3, 5, 8] showed the logarithmic algorithms of suffix tree generalization. All of them required one of the dynamic data structures such as McCreight [14], DS-List [22], and Weiner [23] for managing the dictionary.

This principle was straight away challenged by how to escape from the factor of logarithmic time ($nlog|P|$). Although, AFGGP [4] was the first algorithm with almost linear time efficiency, the $log|P|$ factor still remained problematic. It can be said that all suffix tree approaches fall into the $log|P|$ time trap. Furthermore, when implementing the suffix tree to applications it takes more space than Trie. Nevertheless, [20] tried to improve DS-List [22] for storing patterns, but the time complexity is still affected by logarithmic time. For a clearer understanding, there are many sources [6, 8, 21] which provide good information on this principle.

2.2 Deriving the inverted index

The inverted index is the method for creating the index of keywords which appear in D=$\{D_1 \ldots D_n\}$ where D_i is any individual document which contains the various keywords over $\sum$, and $1 \leq i \leq n$. Then, the keywords are represented by $< documentID, word : pos >$ where $documentID$ is the indicated number referring to the number of documents, *word* is the keywords in the document, and *pos* is the occurrence position of *word* in the $documentID$.

For example, assume that there are the documents D_1:sun of sun, D_2:moon of moon, and D_3:star of star. Then, each document is analyzed for keeping keywords and their positions. Thus the keywords in the documents are *sun*, *of*, *moon* and *star*. Then, all keywords can be rewritten by the form of *word* : (*posting lists*) where *posting list* is *(documentID: position of words in that document)*. In this case, all keywords in these documents are re-written as sun:$(D_1 : 1)$,$(D_1 : 3)$; of:$(D_1 : 2)$,$(D_2 : 2)$,$(D_3 : 2)$; moon:$(D_2 : 1)$,$(D_2 : 3)$, and star:$(D_3 : 1)$,$(D_3 : 3)$.Afterwards, the keywords and posting lists are converted into the suitable data structures such as B^+tree, suffix tree, and suffix array.

Motivated by the positions of keywords, this research focuses on the position of characters instead of the keywords. For deriving the principle, the document D is first replaced by the pattern P, and each D_i is replaced by p^i. For instance, if there are the patterns $P = \{ram, run, running\}$ then the patterns are assigned as $D_1 = ram$, $D_2 = run$, and $D_3 = running$. In the next step, they are re-written by the form of *character :<the occurrence position of character in pattern: the indicated status of the last character of pattern: the number of pattern in P>*; e.g.,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

r :< 1 : 0 : 1 >, < 1 : 0 : 2 >, < 1 : 0 : 3 >,.... Each item of this form is called the ***individual posting list.*** Then, the context is determined using the individual posting lists that accommodate the dictionary(shown in section 3).

2.3 Deriving the perfect hashing

The perfect hashing principle is the most powerful hashing table because it is completely devoid of collision. Importantly, this priciple takes $O(1)$ time in worst-case performances (shown in [7, 19, 24]). Moreover, it takes $O(n)$ space where n is the size of data. This structure is suitable for the set of static keys such as the reserved words in the programming language. Similary, the alphabets($\sum$), which are used in all languages, are as limited as the static keys. This is the reason why perfect hashing should be chosen for implementing the inverted lists.

Fundamentally, the perfect hashing table consists of 2 levels. The first level is the universal key U to accommodate all keys for accessing all data in the table. This level has the n keys for hashing to access the second level by the function $f(n)$. The second level contains the data items associated with the corresponding key of n. This level splits into 2 buckets which avoid collision when accessing data. By using this method, accessing data may need re-hashing 2 times.

This research assigns $\sum$ as the universal key U and $f(\lambda)$ as $f(n)$ for the first level of the perfect hashing table and represents the groups of posting lists as the data items in the second level where $\lambda \subseteq \sum$.

The first level has the hashing function $h(key) \rightarrow (data\ in\ level\ 2)$. If there are collisions then they need to re-hash by $h(key\ of\ level\ 2)$. However, the time complexity still takes $O(1)$. For implementing, $\sum$ and λ are unnecessary to store in the memory because they can be calculated using the special function $f(character, pos)$ (shown in definition 3.6) when accessing the inverted lists in the second level. This method decreases the space in the first level of hashing table while the accessing of the items takes a constant time.

3 INVERTED LISTS DATA STRUCTURE

The main ideas to accommodating dictionary are highlighted in this section. Initially, all characters of each pattern are analyzed and given their positions. Then the positions are grouped to a new form and are arranged into the perfect hashing table. The following sub-sections present all basic definitions for the next sections, the pre-processing algorithm, pattern insertion, and pattern deletion.

3.1 Basic definitions

As mentioned earlier, this paper adapts the inverted lists structure and the searching algorithm presented in [30] to improve the approach outlined in [12] and thus some definitions and notations are the same in both [30, 12]. In representing the

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

characters by position, defintion 3.1 and 3.2 illustrate the individual lists and their form. Definition 3.3 represents the inverted lists in new context. Definition 3.4 to 3.7 are for creating the table, keeping the inverted lists by temporary variables, the functions for accessing the table, and a theorem called intersection for analyzing the continuity of patterns.

Definition 3.1 Given P is a set of patterns $\{p^1, p^2, .., p^r\}$ where p^i denotes a pattern i^{th} which $1 \leq i \leq r$. The length of p^i is m and p^i is formed by the character sequence $\{c_1c_2c_3...c_m\}$. A single individual posting list of a character c_k is defined as $c_k :< k : 0 : i >$ if $k < m$ or $c_k :< k : 1 : i >$, if $k = m$ where $1 \leq k \leq m$. The individual posting list of $c_k :< k : 0 : i >$ is denoted by $\varphi_0^{k_i}$, and $c_k :< k : 1 : i >$ is denoted by $\varphi_1^{k_i}$.

Example 1. If there are the set of $\{ram, run, running\}$ then each pattern can be assigned as $p^1 = r_1a_2m_3$, $p^2 = r_1u_2n_3$ and $p^3 = r_1u_2n_3n_4i_5n_6g_7$. All individual posting lists are represented as below.

p^1 =*r:<1:0:1>, a:<2:0:1>, m:<3:1:1>,*
p^2 =*r:<1:0:2>, u:<2:0:2>, n:<3:1:2>,* and
p^3 =*r:<1:0:3>, u:<2:0:3>, n:<3:0:3>,*
n:<4:0:3>, i:<5:0:3>, n:<6:0:3>, g:<7:1:3>.

The next step is that all individual posting lists are grouped to a new form as *character:<position:terminate status:{set of patterns which occur in the same position}>*. Then, the groups of all characters can be shown as *r:<1:0:{1,2,3}>, n:<3:0:{2,3}>, <3:1:{2}>, <4:0:{3}>,* <6:0:{3}>, and so on. Definition 3.2 shows how to group the posting lists to the new form.

Definition 3.2 Let l_{max} be the maximum length of patterns in $\{p^1, p^2, p^3, \ldots, p^r\}$, and let ε be the position of any character λ which appears in the various patterns at the same position where $1 \leq \varepsilon \leq l_{max}$ and $\lambda \subseteq \Sigma$. Then the posting lists are $\{\varphi_0^{\varepsilon_i}, \varphi_0^{\varepsilon_l}, \ldots, \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ or $\{\varphi_1^{\varepsilon_i}, \varphi_1^{\varepsilon_l}, \ldots, \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ where $1 \leq \{i, l, \ldots, p, q\} \leq r$. A group of posting lists of λ can be defined as follows.

1. If the posting lists of λ are $\{\varphi_0^{\varepsilon_i}, \varphi_0^{\varepsilon_l}, \ldots, \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ then a group of posting lists is defined by $\lambda_{\varepsilon,0}$.
2. If the posting lists of λ are $\{\varphi_1^{\varepsilon_i}, \varphi_1^{\varepsilon_l}, \ldots, \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ then a group of posting lists is definded by $\lambda_{\varepsilon,1}$.

Example 2. The posting lists of *P={ram, run, running}*.

Posting lists	$\lambda_{\varepsilon,0}/\lambda_{\varepsilon,1}$
$a :< 2 : 0 : \{3\} >$,	$a_{2,0}$,
$g :< 7 : 1 : \{3\} >$,	$g_{7,1}$,
$i :< 5 : 0 : \{3\} >$,	$i_{5,0}$,
$m :< 3 : 1 : \{1\} >$,	$m_{3,1}$,
$n :< 3 : 0 : \{3\} >, < 3 : 1 : \{2\} >,$	$n_{3,0}, n_{3,1},$
$< 4 : 0 : \{3\} >, < 6 : 0 : \{3\} >,$	$n_{4,0}, n_{6,0},$
$r :< 1 : 0 : \{1,2,3\} >$,	$r_{1,0}$,
$u :< 2 : 0 : \{2,3\} >$.	$u_{2,0}$.

Definition 3.3 Let I be the inverted list structure of any group of the posting lists. For any inverted lists structure of alphabet λ; if the posting lists group is $\lambda_{\varepsilon,0}$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

then the inverted lists structure is defined as $I_{\lambda\varepsilon,0}$. Similarly, if the posting lists group is $\lambda_{\varepsilon,1}$ then the inverted lists structure is denoted as $I_{\lambda\varepsilon,1}$.

Example 3. The groups of posting lists shown in example 2 can be re-written as $I_{a2,0}$, $I_{g7,1}$, $I_{i5,0}$, $I_{m3,1}$, $I_{n3,0}$,$I_{n3,1}$, $I_{n4,0}$, $I_{n6,0}$, $I_{r1,0}$, and $I_{u2,0}$.

Definition 3.4 The perfect hashing table which provides for all alphabets over $\sum$ and their corresponding inverted lists, is called the inverted lists table and denoted by τ .

Example 4. The groups of posting lists shown in example 3 can be stored in the table τ as shown in table 1. It is unnecessary to store the first column in the real table because it can be calculated by the code of ASCII or Unicode when implemented, but the second column is stored in the memory which is split into two parts. These are described in the third and the fourth columns.

Table 1: Table of the inverted lists of P

$f(\lambda)$ (first level)	Second level	set of positions	set of pattern numbers
a	$I_{a2,0}$	2 : 0	{3}
g	$I_{g7,1}$	7 : 1	{3}
i	$I_{i5,0}$	5 : 0	{3}
m	$I_{m3,1}$	3 : 1	{1}
n	$I_{n3,0}$,$I_{n3,1}$	3 : 0, 3 : 1	{3},{2},
	$I_{n4,0}$,$I_{n6,0}$	4 : 0, 6 : 0	{3},{3}
r	$I_{r1,0}$	1 : 0	{1,2,3}
u	$I_{u2,0}$	2 : 0	{2,3}

Definition 3.5 Two hashing sets which are provided for storing any inverted lists $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ are called $SET1$ and $SET2$.

Definition 3.6 A hashing function which takes $I_{\lambda pos,0}$ and/or $I_{\lambda pos,1}$ from τ is called the inverted lists hashing function, denoted by $IVL(\lambda, pos)$ where $\lambda \subseteq \sum$ and pos is the required position of posting lists which are stored in the second level of τ.

Definition 3.7 If $SET1$ and $SET2$ contain the inverted lists groups, then the continuity of patterns is operated by the intersecting function which is denoted by $SET1 \cap SET2$.

Example 5. Supposing that $SET1$={<*1:0:*{*1,2*}>} and $SET2$={< 2 : 0 : {1,3} >} then $SET1 \cap SET2$ is ordered by the position 1 to 2. The first consideration is that the sequence of inverted lists in $SET1$ are described by $SET2$. In this case, the pattern number {*1*} of $SET1$ is described by the position of {*1*} in $SET2$ while the required position is '*2*' in {<*2:0:*{*1,3*}>} (prior to the positions in $SET1$ 1). Therefore, the result is $SET1$={<*2:0:* {*1*}>}.

3.2 Pre-processing phase

This section shows the algorithm for generating the table τ. Lemma 1 shows how to get the inverted lists in constant time. Theorem 1, Theorem 2, and Theorem 3 define the correctness, time, and space of Algorithm 1, respectively.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Pre-processing represents the steps for creating the inverted lists which take $O(|P|)$ *time. The first step is creating the empty table* τ. *The second step is* reading p^1 to p^r. Whenever each pattern is read, the character is converted to the inverted lists. Then if an inverted list of the considering character exists in the table, the number of pattern is added into the part of {*set of patterns which occur in the same position*}. Otherwise, a new inverted list is created and added into the table.

Algoirthm 1: Pre-processing phase
Input: $P = \{p^1, p^2, \ldots, p^r\}$
Output: table τ of P
1. Create table τ
2. for $i = 1$ to τ do
3. for $j = 1$ to m of p^i do
4. if $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ does not exist in τ then
5. $\tau \leftarrow \varphi_0^{j_i}$ if $j < m$ or $\tau \leftarrow \varphi_1^{j_i}$ if $j = m$
6. else
7. $I_{char(j)j,0} \leftarrow i$ if $j < m$ or $I_{char(j)j,1} \leftarrow i$ if $j = m$
8. end if
9. end for
10. end for
11. return table τ

Lemma 1 If there are the groups of inverted lists $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ in τ then accessing all inverted lists of $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ uses $O(1)$ time.

Proof. Since $\lambda \subseteq \sum$ then each alphabet is a unique character, and λ is implemented as the first level of the perfect hashing table taking $O(1)$ time. The inverted lists $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ are implemented as the second level of the perfect hashing table; therefore, each data item takes $O(1)$ time, and all items in the second level of table are taken in $O(1)$ as well as the individual item. □

Theorem 1 Algorithm 1 can generate all patterns $\{p^1, p^2, p^3, \ldots p^r\}$ to the inverted lists and store them into τ correctly.

Proof. The correctness is proved when p^1 to p^r are generated to the inverted lists, and all inverted lists are added to the table τ. The proofs are organized by 1) *proving the initial step*, 2) *proving for of inner loop, and* 3) *proving for of outer loop*. For the *initial step*, line 1 needs to be true, and the table must be created for running the other steps of proof.

Regarding the inner loop, the proof is by the induction on j for $j = 1$ to $j = m$. The invariants are still at the end of each j^{th} iteration on $1 \le j \le m$ and $1 \le i \le r$ for $j = 1$ to $j = m$. The pre-condition is that p^i does not exist in the table, and the length of each p^i is m. Also, the variable of m can be changed when each p^i is changed. The post-condition is that each p^i is formed by the sequence of $\{c_1c_2c_3 \ldots c_m\}$. All characters $c_1c_2c_3 \ldots c_m$ are generated to inverted lists and are

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

added to the table. Since the *for* loop is executed by a fixed number, this therefore *guarantees the termination of the loop. In the base case, c_1 of p^i is converted to $\varphi_0^{j_1}$* and added to the table as a new inverted list. This result is true, and the invariants remain. Assuming the proposed invariants are true after $m-1$ iteration, proof can be demonstrated using the two following cases.

In the first case, if there are no inverted lists of p_j^i, then a new inverted list $\varphi_0^{j_1}$ if $j < m$ or $\varphi_1^{j_1}$ if $j = m$ is generated and the table at the $I_{char(j),0} \leftarrow i$ or $I_{char(j),1} \leftarrow i$ is created. Then $\varphi_0^{j_1}$ or $\varphi_1^{j_1}$ is stored in the table, and the invariants are unchanged. *In the second case, if there are the inverted lists of p_j^i, the number of $m-1$ is stored in τ. This then implies that the variable j and $1 \le j \le m-1 \le m$ and $1 \le i \le r$ for $j = 1$ to $j = m-1$. Also by induction, the variable j and $1 \le j \le m-1 \le m$ and* $1 \le i \le r$ for $j = 1$ to $j = m-1$. Then after adding $\varphi_0^{j_1}$ or $\varphi_1^{j_1}$ to the table it implies an iteration of $j = m$ as the hypothesis induction, and the post-condition is shown when c_m is added to the inverted list . In either case the proposed invariants remain *and the termination is guaranteed by a fixed number of j; therefore, the inner loop* is correct.

The outer loop is proved by induction on i. The pre-condition is that there are P and τ, the post-condition is all patterns in P are generated to the inverted lists and are added to τ. The proposed invariant is $1 \le i \le r$. For the base case, if $i = 1$ then it is true by the inner loop and the proposed invariant $1 \le i \le r$ remains. The inverted lists from c_1 to c_m are followed by the inner loop, and the loop is run on the fixed number of i, this also guarantees its termination. In the induction step, the iteration of $i = r-1$ must be proved; the pattern p^{r-1} is formed by $\{c_1c_2c_3 \ldots c_m\}$, and all of the characters are sent to the inner loop which are then true after running the inner loop. The termination is guaranteed by the fixed number of m, and when all inverted lists are stored in τ. The invariant still remains while τ stores the inverted lists from pattern p^1 to p^{r-1} after running the inner loop. By induction, the hypothesis is reached, and the correctness is proved. □

Theorem 2 Generating the patterns $\{p^1, p^2, p^3, \ldots p^r\}$ to the inverted lists and adding them into τ takes $O(|P|)$ time where $|P|$ is the sum of all pattern lengths.

Proof. The hypothesis is that all characters of $\{p^1, p^2, p^3, \ldots p^r\}$ are generated to inverted lists, and they are added into τ. Referring to Algorithm 1, all pattern lengths are denoted by $|p^1|$, $|p^2|$, $|p^3|$,..., $|p^r|$. For the initial step, the table τ is built in $O(1)$ time. Each round processes the inner loop to execute line 3 or line 8 until they equal the length of each pattern. The summation is $|p^1| + |p^2| + |p^3| + \ldots + |p^r|$ which equal $|P|$, and it reaches to the hypothesis step by the last character of p^r. Therefore the inverted lists are constructed in $|P|$ time; this is called $O(|P|)$ time complexity. Meanwhile, line 4, 5, and 7 access the table in $O(1)$ by Lemma 1. Hence, the preprocessing time is proved in $O(|P|)$ time. □

Theorem 3 The table τ requires $O(|P|)$ space for accommodating whole inverted lists of $\{p^1, p^2, p^3, \ldots p^r\}$ where $|P|$ is the sum of pattern lengths.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Proof. The space is proved when all characters of P are generated to inverted lists and are added into the table τ taking $|P|$ space. The pattern lengths in P are $|p^1|, |p^2|, |p^3|, \ldots, |p^r|$, and each p^i contains the sequence string $\{c_1c_2c_3 \ldots c_m\}$ which has the length m. The length m is denoted by $|p^i|$. For the initial step, the first column of table τ is created for all patterns. Each inverted list is created by the pre-processing phase for all patterns of P; therefore, each inverted list of string $\{c_1c_2c_3 \ldots c_m\}$ in each p^i only takes one space per one list. Thus, the space is equal to $|p^1|+|p^2|+|p^3|+\ldots+|p^r| = |P|$ for the second level of the perfect hashing table. As mentioned earlier, the perfect hashing table required $O(n)$ space to accommodate the data items; hence, the space of τ is $O(|P|)$. □

3.3 Pattern insertion

The method of pattern insertion is similar to that of the inner loop of Algorithm 1. Let p^ϕ be a new pattern which does not appear in τ, and contains the sequence string $\{c_1c_2c_3 \ldots c_m\}$. Then all inverted lists of p^ϕ are generated and added into the table as the pre-processing phase. This method is illustrated by Algorithm 2. Theorem 4 shows the correctness of Algorithm 2, and Theorem 5 proves the time of the individual pattern insertion.

Algorithm 2: Pattern Insertion
Input : $p^\phi = \{c_1c_2, c_3, \ldots, c_m\}$
Output : τ after insertion the inverted lists of $c_1c_2, c_3, \ldots, c_m$

1. for $j = 1$ to m do
2. if $\varphi_0^{j_\phi}$ or $\varphi_1^{j_\phi}$ of p^ϕ does not exist in τ then
3. $\tau \leftarrow \varphi_0^{j_\phi}$ if $j < m$ or $\tau \leftarrow \varphi_1^{j_\phi}$ if $j = m$
4. else
5. $I_{char(j),0} \leftarrow \phi$ if $j < m$ or $I_{char(j),1} \leftarrow \phi$ if $j = m$
6. end if
7. end for
8. return table τ

Example 6. If there is the new pattern $p^\phi = rap$ to be inserted into the table τ of $P = \{ram, run, running\}$. In this case, all characters are formed to inverted lists as $r :< 1 : 0 : 4 >$, $a :< 2 : 0 : 4 >$, and $p :< 3 : 1 : 4 >$. Then, the result is $r :< 1 : 0 : \{1,2,3,4\} >$. Similarly, the character 'a' can be generated and added in $a :< 2 : 0 : \{3,4\} >$. But then the character 'p' is a new character that does not exist in the table. The inverted list of 'p' is generated as $p :< 3 : 1 : \{4\} >$ by line 3.

Theorem 4 Let $p^\phi = \{c_1c_2, c_3, \ldots, c_m\}$ be the new pattern which is not contained in the table τ. Algorithm 2 inserts the inverted lists of pattern p^ϕ into the table τ correctly.

Proof. The correctness of Algorithm 2 is proved when all inverted lists of p^ϕ are added into the existing table τ. Let τ_{old} be the table before adding the new pattern,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

and let τ_{new} be the table after adding the new pattern.

The pre-condition is that τ_{old} contains the inverted lists $|P|$, and the post-condition is $|P| + |p^{\phi}|(\tau_{new})$. The invariants are $1 \leq j \leq m$, and $1 \leq i \leq r$ for $j = 1$ to $j = m$ where m is the length of $\{c_1c_2, \ldots, c_m\}$. The proof is by induction on j as the inner loop of Algorithm 1.

Obviously then, the pattern p^{ϕ} is similar to the pattern p^i in P. Algorithm 2 is run as the inner loop of Algorithm 1. Therefore, the proof in the loop of Algorithm 2 is claimed as well. Also, the invariants remain because there is nothing to change them. Hence, the post-condition is shown at c_m, and τ_{new} is shown. □

Theorem 5 Inserting $p^{\phi} = \{c_1c_2, c_3, \ldots, c_m\}$, which does not appear in the table τ, takes $O(|p|)$ where $|p|$ is the length of p^{ϕ}.

Proof. Algorithm 2 is referred to for straightforward proof. The length of p^{ϕ} is m and is denoted by $|p|$. The loop *for* reads all characters and converts them to the inverted lists. Then each individual inverted list is added into τ one by one. This loop creates the inverted lists from c_1 to c_m, and it takes m operations. Thus, $O(|p|)$ time is shown and then the hypothesis is also proved. The other lines (2,3,5) access the table taking $O(1)$ by Lemma 1. Therefore, to insert all characters of p^{ϕ} into the existing dictionary takes $O(|p|)$ time. □

3.4 Pattern deletion

Assumming that p^{σ} is the existing pattern in τ, and p^{σ} is formed by the sequence string $\{c_1c_2c_3 \ldots c_m\}$. Then, all characters from p^{σ} are read one by one, and each inverted list is removed from the dictionary. For the mechanism of deletion, if the corresponding inverted list exists in only one posting list, it will be immediately deleted. Otherwise, only the inverted lists which the pattern number equals to σ are deleted. The method is described by Algorithm 3.

Algorithm 3: Pattern Deletion
Input : $p^{\sigma} = \{c_1c_2c_3 \ldots c_m\}$
Output : τ after deletion the inverted lists of $c_1c_2c_3 \ldots c_m$.

1. for $j = 1$ to m do
2. if the posting lists in $I_{char(j)j,0}$ or $I_{char(j)j,1} > 1$ then
3. Delete $\varphi_0^{j\sigma}$ if $j < m$ or $\varphi_1^{j\sigma}$ if $j = m$
4. else
5. Delete $I_{char(j)j,0}$ if $j < m$ or $I_{char(j)j,1}$ if $j = m$
6. end if
7. end for
8. return table τ

An illustrative example is shown below, followed by the proof of correctness and the time complexity.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Example 7. Taking p=*ram* off P={*ram, run, running*}. In this case, the *target pattern is 1, and all characters of ram are formed to* $r :< 1 : 0 : \{1\} >$, $a :< 2 : 0 : \{1\} >$, and $m :< 3 : 1 : \{1\} >$. Line 3 takes the inverted list of $r :< 1 : 0 : \{2\} >$ from $r :< 1 : 0 : \{1,2,3\} >$ then the result is $r :< 1 : 0 : \{2,3\} >$. But the inverted lists of $a :< 2 : 0 : \{1\} >$ and $m :< 3 : 1 : \{1\} >$ are removed from the table by line 5.

Theorem 6 Deleting the existing pattern $p^\sigma = \{c_1c_2, c_3, \ldots, c_m\}$ from the table τ by Algorithm 3 is correct.

Proof. τ contains the inverted lists of P with the size $|P|$. Let τ_{aft} be the inverted lists table after deleting the pattern $p^\sigma = \{c_1c_2, c_3, \ldots, c_m\}$, and the size of τ_{aft} is $|P| - |p^\sigma|$ where σ is the pattern number which appears in the table. The proof needs to show all inverted lists of $\{c_1c_2, c_3, \ldots, c_m\}$ that are removed from the table. The pre-condition is that the p^σ exists in table τ, and the post-condition is τ_{aft}. The proposed invariant is $1 \le j \le m$ for $j = 1$ to $j = m$.

The proof is by induction on j. *The base case is in* $j = 1$ *and the character* c_i *is* converted to the inverted list. Then, the inverted list $\varphi_0^{j_\sigma}$ is formed by line 3. The proof needs to show both conditions of *if*. In the first case, if the number of posting list c_i in the table is more than 1 then the number of $\varphi_0^{j_\sigma}$ is removed from τ. In the second case, if there is only one inverted list in τ then $I_{char(j),0}$ is removed by line 5. Thus, $\varphi_0^{j_\sigma}$ is removed from the table after the first iteration, and the size of the table is decreased by 1. In both cases, the invariant $1 \le j \le m$ remains. According to the fixed number of loops, the termination of loop is guaranteed by the value of m.

In the inductive step, the invariant needs to be true after the iteration of $j = m-1$. The character of c_{m-1} is created as $\varphi_0^{\sigma_{m-1}}$. If the number of inverted lists of c_{m-1} is more than 1 then the number of $\varphi_0^{\sigma_{m-1}}$ is removed from τ, and if there is only one inverted list then $I_{char(m-1)m-1,0}$ is removed. The invariant still remains. The number of inverted lists in the table equals $|P| - |p^\sigma - 1|$ while $1 \le j \le m-1 \le m$ for $j = 1$ to $j = m$, and τ_{aft} is shown. By induction, the hypothesis is implied. Therefore, Algorithm 3 is correct. □

Theorem 7 Deleting the existing pattern p^i from the dictionary P takes $O(|p|)$ time where p^i is the target pattern to be deleted and $|p|$ is the length of pattern p^i.

Proof. Assumming that p^i is the existing pattern to be deleted, and p^i is formed by the sequence string $\{c_1c_2c_3 \ldots c_m\}$. The length of p^i is m and is denoted by $|p|$, and i is the number of the pattern i^{th} in P. The hypothesis is that all inverted lists of p^i are removed from the dictionary of P.

The process of deleting repeats to remove the inverted lists from c_1 to c_m. Each operation for accessing the inverted list uses $O(1)$ by Lemma 1. The operations remove the matched inverted lists from the table one by one. All operations take $|p|$ *time while line 3 or line 5 takes the constant time by Lemma 1. Thus, to delete* all characters of pattern p^i from the dictionary P takes $|p|$ which is $O(|p|)$ time. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4 SEARCHING PHASE

Before describing the searching methodology in depth, this section refers back to the basic definitions which are used for running the searching algorithm. Let N be the target position in the given text to be compared; pos is the required position of the inverted lists to be matched; and n is the length of the text T. In addition, $SET1$ and $SET2$ are the variables that are operated for continuity during the search.

Initially, the variables N pos, $SET1$, and $SET2$ are set to enforce the searching window, and the variable pos is used to control the required position in the text T. This search is based on reading from left to right along the text T. While reading, the inverted lists that equal pos in the row of $text[N]$ are taken to $SET1$ or $SET2$. When considering continuity, the intersection ($\cap$) is used for checking patterns in $SET1$ and $SET2$.

The intersection between $SET1$ and $SET2$ finds a set of numbers in $SET2$ that continue from $SET1$. Importantly, it reports the matched position whenever the terminate status equals *1*. The continuity is concentrated on the posting lists in $SET1$ that are described by $SET2$. If the numbers of positing lists in $SET2$ are superior to $SET1$, these are kept in $SET1$ for the next operation. For reporting the occurrences, the indicated number '0' or '1' of $SET1$ is considered; if the indicated number in $SET1$ is '1' then the matched position is reported.

In the case of the overlapping patterns, the inverted lists which are equal $< 1 : 0 : \{\ldots\} >$ must be attached to $SET2$ when accessing the inverted lists of any positions. For instance, if the patterns are 'ram' and 'amazing', the inverted lists of 'a' are <2:0:{1}> and <1:0:{2}>. In this case they are taken together when the character 'a' in the given text is scanned.

The algorithm, an illustrative example, the proofs of the correctness and the time complexity are shown below respectively.

Algorithm 4: Searching Algorithm
Input : P and T
Output : all occurrences are reported, and T is scanned.

1. $N = 1 pos = 1, SET1 = SET2 = null, RESULTS = \{\}$
2. $SET1 \leftarrow (IVL(text[N]), pos), N++$
3. while $(N \leq n)$ do
4. Store the matched position into $RESULTS$ set if $SET1$ contains φ_1^{pos}
5. if $SET1 <> null$ then
6. $pos++$
7. $SET2 \leftarrow (IVL(text[N]), pos or 1)$
8. $SET1 \leftarrow SET1 \cap SET2$
9. else
10. $pos = 1$
11. $SET1 \leftarrow (IVL(text[N]), pos)$
12. end if
13. $N++$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

14. end while
15. report all occurrences in *RESULTS*

Example 8. Searching P={*ram, run, running*} in the given text T= *run as running on ram* by searching algorithm.

1. Initiate the variables N=1, $SET1 = SET2 = \{\}$.

2.Skip to the line 2 and $SET1$ ={<1:0:{1,2,3}>}, and $N = 2$.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

3. Take the first loop of *while*, *pos*=2, and $SET2$={<2:0:{1,2,3}>}.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1$ ={<2:0:{1,2,3}>} and N=3.

4. Skip to the next loop of *while*, *pos*=3, $SET2$={<3:1:{1}>,<3:0:{2}>}

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1$={<3:1:{2}>,<3:0:{3}>} and n is matched at <3:1:{2}> in the pattern 2. After matching report, set N=4 and $SET1$ = {< 3 : 0 : {3} >}.

5. Skip to the next loop of *while*, *pos*=4, and $SET2$={}

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

6. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, $SET1$={}, and N=5.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

7. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, $SET1$={}, and N=6.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

8. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, $SET1$={}, and N=7.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

9. Skip to the next loop of *while*, and set *pos*=2,$SET2$={<1:0:{1,2,3}>}, and N=8.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

10. Skip to the next loop of *while*, and set *pos*=2, $SET2$={<2:0:{1,2,3}>}, and N=9.

```
r u n   a s   r u n   n   i   n   g     o   n     r   a   m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1$={<2:0:{1,2,3}>} and N=10.

11. Skip to the next loop of *while*, and set *pos*=3, $SET2$={<3:1:{2}>,<3:0:{3}>}, and N=10.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
*SET*1 ← *SET*1 ∩ *SET*2 thus *SET*1={<3:1:{2}>,<3:0:{3}>} and *n* is matched at <3:1:{2}> in pattern 2. After matching report, set *N*=11 and *SET*1={<3:0:{3}>}.
12. Skip to the next loop of *while*, and *pos*=4, *SET*2={<4:0:{3}>}, and *N*=11.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
*SET*1 ← *SET*1 ∩ *SET*2 thus *SET*1={<4:0:{3}>} and set *N*=12.
13. Skip to the next loop of *while*, and *pos*=5, *SET*2={<5:0:{3}>}, and *N*=12.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
*SET*1 ← *SET*1 ∩ *SET*2 thus *SET*1={<5:0:{3}>} and set *N*=13.
14. Skip to the next loop of *while*, and *pos*=6, *SET*2={<6:0:{3}>}, and *N*=13.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
*SET*1 ← *SET*1 ∩ *SET*2 thus *SET*1={<6:0:{3}>} and set *N*=14.
15. Skip to the next loop of *while*, and *pos*=7, *SET*2={<7:1:{3}>}, and *N*=14.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
*SET*1 ← *SET*1 ∩ *SET*2 thus *SET*1={<7:1:{3}>} and *g* is matched at <7:1:{3}> in the pattern 3. After matching report, set *N*=15 and *SET*1={}.
16. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, *SET*1={}, and *N*=15.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
17. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, *SET*1={}, and *N*=16.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
18. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, *SET*1={}, and *N*=17.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
19. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, *SET*1={}, and *N*=18.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
20. Skip to the next loop of *while*, and it takes the condition of *else* that *pos*=1, *SET*1={<1:0:{1,2,3}>}, and *N*=19.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
21. Take the first loop of *while*, *pos*=2, and *SET*2={<2:0:{1}>}, *N*=20.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1$={<2:0:{1}>} and N=21.
22. Take the first loop of while, pos=2, and SET2={<3:1:{1}>},N=22.

r	u	n		a	s		r	u	n		n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	

$SET1 \leftarrow SET1 \cap SET2$ thus *SET1={<3:1:{1}>}* and the matched position is reported. Therefore, N=22 and $N > n$, and the searching is finished.

Lemma 2 If SET1 and SET2 contain the inverted lists then SET1 ← SET1 ∩ SET2 is correct.

Proof. The correctness is that all inverted lists in φ_0^{pos} or φ_1^{pos} of $SET2$ which continue from $SET1$ are returned and put into $SET1$. The pre-conditions are $N \geq 2$, and the inverted lists stored in $SET1$ and $SET2$. The post-condition is that all inverted lists of φ_0^{pos} or φ_1^{pos} in $SET2$ which continue from $SET1$ are returned and put into $SET1$.

The required position is *pos*, and the continuity is that all inverted lists φ_0^{pos-1} or φ_1^{pos-1} of $SET1$ are described by φ_0^{pos} or φ_1^{pos} in $SET2$. $SET1$ and $SET2$ contain any $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ of the hashing set in definition 3.3, and they are the second level of the perfect hashing table. It can be said that every inverted list of $SET2$ must be inspected and compared with the inverted lists in $SET1$ by the properties of intersection. Thus, the results are φ_0^{pos} and/or φ_1^{pos} and φ_0^1. The post-condition is then reached. □

Theorem 8 Algorithm 4 is correct for searching P in the given text T.

Proof. The correctness is proved by the induction on n for t_1 to t_n. The proposed invariants are $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$.

Assuming that line 1 and line 2 are true; then in the base case the variable N is increased by 1 before getting to the *while* loop. This step needs to be proved in the case of $SET1 <> null$ and $SET1 = null$. Then if $SET1 <> null$ the inverted lists of $text[N]$ are taken to $SET2$ and $1 \leq pos \leq l_{max}$, the correctness is proved by the intersection in Lemma 4.1. If $SET1 = null$, then the $text[N]$ is taken to $SET1$ and then after the end of this iteration $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$. In both cases, the invariants remain unchanged and thus this step is true. In the inductive step, the iteration $n-1$ needs to prove when the case of $SET1 <> null$. This algorithm runs the variable n by the fixed number and then the iteration $n-1$ is reached and all iterations from t_3 to t_{n-1} are true; (by reporting the matched patterns in line 4 which prove both the correctness and the pattern continuity). It can then be claimed the iteration t_n is true by induction and the algorithm is correct. □

Lemma 3 The time complexity for taking any $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ from SET is $O(1)$.

Proof. From definition 3.5, $SET1$ or $SET2$ contains only one row of inverted lists; Also, both of them are the perfect hashing set. Thus, it implies $O(1)$ time by their hashing properties. □

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Lemma 4 If SET1 and SET2 contain the inverted lists then $SET1 \cap SET2$ takes $O(1)$ time.

Proof. Let $SET1$ contain the inverted list groups $I_{\lambda c1,0}$ and/or $I_{\lambda c1,1}$. Let $SET2$ contain the inverted list groups $I_{\lambda c2,0}$ and/or $I_{\lambda c2,1}$. $SET1$ and $SET2$ are the perfect hashing set; then every operation can be solved in $O(1)$ time using Lemma 3. Therefore, every operation to access $I_{\lambda c1,0}$, $I_{\lambda c1,1}$, $I_{\lambda c2,0}$, and $I_{\lambda c2,1}$ also takes $O(1)$ time by Lemma 1. □

Theorem 9 Searching the occurrences of $P = \{p^1, p^2, \ldots, p^r\}$ which appear in the given text $T = \{t_1 t_2 t_3 \ldots t_n\}$ takes $O(n)$ time where n is the length of T.

Proof. The proof is that all characters of $t_1 t_2 t_3 \ldots t_n$ are scanned, and all occurrences are reported in n time. Referring back to the searching algorithm, the time complexity is dominated by the variables N, $SET1$, and $SET2$. The *while* loop (line 3) is repeated to inspect the inverted lists of t_2 to t_n. Each iteration of the loop definitely reports all occurrences in line 4 with $O(1)$ by Lemma 4. It can be said that the loops of line 3 take $O(n)$ time because this step is processed from the initial step to n time. And line 5, 6, and 8 take a constant time by Lemma 3 and Lemma 4, respectively. Therefore, the time complexity takes only $O(n)$ time. Also, this algorithm is able to perform in both an average case and a worst case scenario. □

5 EXPERIMENTAL METHODS AND THEIR RESULTS

The sub-sections below begin to explain the implementation details, then the first set of experiments shows the time and the space requirements of the inverted lists structure. Furthermore, the searching times in several patterns and several given text sizes are also presented.

5.1 Implementation

The experiments were performed on a Dell Vostro 3400 notebook with Intel(R) CORE(TM) i5 CPU, M 560 @2.67 GHz, 4 GB of RAM, and running on Windows 7 Professional (32-bits) as an application machine.

Implementing data structures for pre-processing phase, Aho-Corasick Trie [1] (named AC-Trie), Reverted Trie of SetHorspool(mentioned in [18]), and dynamic Suffix tree [14] were implemented for comparing with the inverted lists structure. In the searching phase, the searching algorithms of Aho-Corasick [1], SetHorspool (mentioned in [18]), and inverted lists algorithm were implemented. Additionally, the programs for randomize the pattern and the text were also implemented.

All of them were implemented in Java with JavaTM 2 SDK, Standard Edition Version 1.6.22 built in the Netbeans 6.9.1. The abstract data type (ADT) of java.util.Vector was employed for accommodating all structures which were compared. AC-Trie and Reverted-Trie structures were created by the special classes

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

to represent the nodes of Trie and Reverted-Trie, and then they were put into the instances of java.util.Vector. The table τ was created by the java.util.Vector as well, but each instance in the second level of the perfect hasing table (set of positions and set of pattern numbers) was implemented by the java.util.HashTable and the java.util.HashSet structure respectively. For the new proposed algorithm, the variables $SET1$ and $SET2$ were also implemented by java.util.HashTable; as well as, all results were kept in the instances of java.util.ArrayLists.

The data tests of $|\sum|$ were the 52 letters of the English alphabet; 'A' to 'Z' and 'a' to 'z'. Then, programs were randomized each pattern with the various lengths of 3 to 20 characters, where the average range was 12 characters. The proposed numbers of patterns were 10; 100; 1,000; 10,000; 50,000; and 100,000 and 300,000 (only for the inverted lists algorithm). Each of the pattern numbers was randomly built in 10 files. The texts were randomized from the size of 1 KB, 10 KB, 100 KB, 1 MB, 5 MB, and 10 MB. Also, each of the text sizes was performed in 10 files as well.

For pre-processing tests, each file in each group was read and generated to the data structures one by one. Then the processing time of each file was captured in nano-seconds. Afterwards, each file again was built and both the data structure and the memory usage was captured in Kilo-Bytes. Performing the searching experiments, every pattern file was paired with each text file. For instance, the first file of 10 patterns was paired by the first file of 1 KB, the second file of 10 patterns was paired by the second file of 1 KB, and the other cases were performed in the same way. When the search in each pair of pattern and text size completed, the processing time in nano-seconds was captured. Then, when the 10 pairs of each group of text finished processing, the average time was given.

5.2 Pre-processing results

The inverted lists structure was constructed faster and used smaller space than the earlier structures (Aho-Corasick [1] called AC-Trie, SetHorspool in [18] called Reverted-Trie,and the suffix tree[14]).

The inverted lists structure takes the shorter average time than AC-Trie 3.75 folds, the Reverted-Trie 2.33 folds, and the suffix tree 15.69 folds. The resulting details are shown in table 2, which converts the nano-time to the seconds where '-' means the data structure could not construct (out of the java heap memory).

Then, the inverted lists structure used less average space than the AC-Trie 18.42%, the Reverted-Trie 20.05%, and the suffix tree 92.38%. In the case of pattern numbers more than 1,000, the suffix tree could not create the structure because our computer was out of heap memory in java while generating the structure. The results are shown in table 3.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 2: Processing time (Seconds)

#patterns	AC-Trie	Reverted-Trie	Suffix Tree	Inverted Lists
10	0.161	0.095	0.154	0.030
50	0.152	0.201	0.235	0.113
100	0.401	0.466	0.467	0.278
500	0.566	0.710	19.276	0.351
1,000	1.023	1.905	708.274	0.767
5,000	10.791	8.001	-	5.519
10,000	45.431	20.728	-	6.918
50,000	532.518	110.561	-	43.623
100,000	3,598.131	5,745.879	-	851.156
300,000	-	-	-	1,132.651

Table 3: Memory usages(KB)

#patterns	AC-Trie	Reverted-Trie	Suffix Tree	Inverted Lists
10	4.71	4.93	24.88	4.56
50	4.82	4.96	48.35	4.81
100	4.90	5.10	896.11	4.890
500	5.59	5.67	2,512.46	5.12
1,000	6.21	6.29	-	5.33
5,000	11.10	11.22	-	7.56
10,000	15.83	16.13	-	9.84
50,000	54.57	55.11	-	23.36
100,000	155.84	131.14	-	47.631
300,000	-	-	-	169.584

5.3 Searching results

The searching times of the inverted lists algorithm(represented by IVL) were more efficient than the SetHorspool algorithm(represented by HP) in average, but took a longer time than the Aho-Corasick(represented by AC). In the case of small pattern numbers and the small text sizes, the inverted lists algorithm took an almost equal searching time to that of Aho-Corasick.

In the case of the large pattern numbers and the large text sizes, the proposed algorithm took an almost similar time to SetHorspool in some cases. It should also be noticed that the bottleneck of our algorithm occurs if there are a large number of patterns, in which case then the inverted lists break into serveral groups. Although the intersection can be operated one time per each intersection, it needs the time to analyze the sequence of continuity and needs the time to check the matching positions. These two points need the time to process and then the searching times are longer than the static algorithms such Aho-Corasick[1] which compares only once per character.

The following tables (4 to 9) show the experimental results which are crossed by the text sizes as 1 KB, 10 KB, 100 KB, 1 MB, 5 MB and 10 MB; and the pattern numbers as 10; 50; 100; 500; 1,000; 5,000; 10,000; 50,000; 100,000 and 300,000 (only

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

the inverted lists algorithm).

Table 4: Searching time (Seconds) in the given text 1 KB.

#patterns	AC	HP	IVL
10	0.011	0.045	0.040
50	0.018	0.102	0.042
100	0.020	0.134	0.045
500	0.017	0.201	0.147
1,000	0.039	0.212	0.165
5,000	0.042	0.186	0.167
10,000	0.033	0.165	0.159
50,000	0.029	0.734	0.393
100,000	0.038	0.817	0.498
300,000	-	-	0.987

Table 5: Searching time (Seconds) in the given text 10 KB.

#patterns	AC	HP	IVL
10	0.066	0.200	0.098
50	0.102	0.755	0.168
100	0.104	1.574	0.141
500	0.112	1.698	0.365
1,000	0.120	1.733	1.001
5,000	0.161	1.695	1.576
10,000	0.210	1.889	1.301
50,000	0.705	2.034	1.696
100,000	1.667	2.701	2.019
300,000	-	-	3.089

Table 6: Searching time (Seconds) in the given text 100 KB.

#patterns	AC	HP	IVL
10	0.301	1.272	1.004
50	0.554	8.012	1.621
100	0.588	13.101	1.589
500	0.634	17.892	4.984
1,000	0.579	15.605	7.312
5,000	1.405	18.243	6.988
10,000	1.464	19.454	9.002
50,000	2.185	20.798	10.102
100,000	4.387	25.391	15.235
300,000	-	-	20.680

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 7: Searching time (Seconds) in the given text 1 MB.

#patterns	AC	HP	IVL
10	3.143	24.231	23.623
50	6.286	108.732	30.380
100	5.012	131.076	29.766
500	7.501	155.205	45.291
1,000	7.003	190.532	46.665
5,000	26.784	175.989	49.571
10,000	58.860	179.859	51.651
50,000	90.725	249.101	94.290
100,000	102.198	321.761	134.872
300,000	-	-	356.712

Table 8: Searching time (Seconds) in the given text 5 MB.

#patterns	AC	HP	IVL
10	15.118	103.881	98.612
50	30.889	400.266	126.493
100	31.997	766.786	136.431
500	35.150	804.195	139.498
1,000	42.807	869.241	153.521
5,000	129.047	900.480	161.541
10,000	156.598	905.586	213.094
50,000	201.003	1,521.231	300.691
100,000	302.677	1,941.345	341.861
300,000	-	-	399.765

6 DISCUSSIONS AND SUGGESTIONS

This section describes the advantages of the inverted lists structure, opening the research, how to improve the inverted lists structure, and suggestions on how to apply this structure to other matching principles.

The primary advantage of the inverted lists structure is that the expected pattern

Table 9: Searching time (Seconds) in the given text 10 MB.

#patterns	AC	HP	IVL
10	40.907	250.583	189.778
50	69.397	882.667	287.018
100	72.976	1,534.896	291.781
500	85.781	1,688.574	320.745
1,000	89.481	2,457.665	331.905
5,000	290.882	2,561.901	348.921
10,000	316.175	2,551.012	449.005
50,000	452.236	2,631.422	631.448
100,000	649.133	2,752.843	739.094
300,000	-	-	876.339

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

to be matched can be reported over time because this structure keeps the positions and the numbers of patterns. Then the searching results are based not only on yes or no answers, but can also report all numbers and patterns to be matched over time. In contrast, the traditional data structures mentioned in section 1 and section 2 are not able to handle this aspect because they need to access by sequencing the root of structures.

Considered by the type of each window search, Navarro and Raffinot [16] divide the searching approach to prefix approach (comparing from left to right), suffix approach (comparing from right to left), and factor approach (comparing by calculating the spacial positions). Then the inverted lists structure is unsequnced to access and compare the characters in the target text. Then the target text can be scanned by all approaches even parallel scanning(simultaneous access).

For suggestions, the inverted lists structure could improve the space by grouping or compressing the lists of the pattern sets. For instance, if the pattern numbers are connected in the sequence such as $< 1 : 0 : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} >$, they can be grouped as $< 1 : 0 : 1 - 10 >$. Furthermore, this structure could be applied to solve other problems such as approximate matching, regular search, two-dimensional matching, pattern recognition, text indexing and so on.

7 CONCLUSION AND PLANNED FUTURE WORKS

A linear time dynamic dictionary matching algorithm, which improves the approach presented in [12], is proposed. This solution adapts the linear time static dictionary matching [30], and especially the inverted lists structure for accommodating the dynamic dictionary. The inverted lists structure is implemented by the perfect hashing table, and it is constructed in optimal time. Furthermore, it is able to insert or delete an individual pattern in minimal time. In theoretical results, this solution takes (1) $O(|P|)$ time for pre-processing where $|P|$ is the sum of all pattern lengths, (2) $O(|p|)$ time for inserting or deleting the pattern where $|p|$ is the length of pattern to be inserted or deleted, and (3) $O(n)$ time for searching in an average and a worst case scenario where n is the length of the given text. In experimental results, the inverted lists structure takes less time and space than the traditional structures; and, the searching time is processed in a linear time. In the near future, we will reduce the table space and create the dynamic dictionary matching algorithm using the suffix approach and the factor approach for improving the time complexity. Also, the approximate matching algorithm is being developed by the inverted lists structure.

REFERENCES

[1] AHO, A. V. AND CORASICK, M. J.: Efficient string matching:An aid to bibliographic search. *Comm. ACM*, 1975, pp. 333–340.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[2] AMIR, A. AND FARACH, M.: Adaptive dictionary matching. In *Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science*, 1991, pp. 760–766.

[3] AMIR, A., FARACH, M. IDURY, R.M., LA POUTRÉ, J.A. AND SCHAFFER, A.A.: Improved Dynamic Dictionary-Matching. In *Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*. 1993, pp. 392–401.

[4] AMIR, A., FARACH, M., GALIL, Z., GIANCARLO, R. AND PARK, K.: Dynamic dictionary matching. *Journal of Computer and System Science*, Vol. 49, No. 2, 1994, pp. 208–222.

[5] AMIR, A., FARACH, M., IDURY, R. M., LA POUTRÉ , J. A. AND SCHÄFFEX, A. A.: Improved dynamic dictionary matching. *Information and Computation*, Vol. 199, No. 2, 1995, pp. 258–282.

[6] AMIR, A. LANDAU, G. M., LEWENSTEIN, M. AND SOKOL, D.: Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, Vol. 3, No. 2, Article 19, 2007, pp. 1–24.

[7] BOTELHO, F. C.: Near-optimal space perfect hashing algorithms. *PhD. thesis Federal University of Minas Gerais, Brazil.*

[8] CHAN, H-L., HON, W-K., LAM, T-W. AND SADAKANE, K.: Dynamic dictionary matching and compressed suffix trees. *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 2005, pp. 13–22.

[9] COMMENTZ-WALTER, B.: A string matching algorithm fast on the average. In *Proceedings of the Sixth International Collegium on Automata Languages and Programming*, 1979, pp. 118–132.

[10] GONGSHEN, L., JIANHUA, L. AND SHENGHONG, L.: New multi-pattern matching algorithm. *Journal of Systems Engineering and Electronics*, Vol. 17, No. 2, 2006, pp. 437–442.

[11] HONG, Y. D., KE, X. AND YONG, C.: An improved Wu-Manber multiple patterns matching algorithm. *Performance Computing and Communications Conference, 2006 IPCCC 2006. 25th IEEE International 10-12*, 2006, pp. 675–680.

[12] KHANCOME, C. AND BOONJING, V.: Dynamic dictionary matching using inverted lists. *Proceeding of the Third IASTED International Conference ADVANCES IN COMPUTER SCIENCE AND TECHNOLOGY (ACST2007)*, 2007, pp. 397–401.

[13] KNUTH, D.E., MORRIS, J.H., PRATT, V.R.: Fast pattern matching in strings. *SIAM Journal on Computing*, Vol. 6 No. 1, 1997, pp. 323–350.

[14] MCCREIGHT, E.M.: A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 1976, Vol. 23, No. 2, pp. 262–272.

[15] MELNIK, S. RAGHAVAN, S. YANG, B. AND GARCIA-MOLINA, H.: Building a Distributed Full-Text Index for the Web. *ACM Transactions on Information Systems*, Vol. 19, No. 3, 2001, pp. 217–241.

[16] MOFFAT, A. AND ZOBEL, J.: Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems*, Vol. 14, No. 4, 1996, pp. 349–379.

[17] MONZ, C. AND DE RIJKE, M.: *Inverted Index Construction* (2002), Availaible on: `http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf`.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[18] NAVARRO, G. AND RAFFINOT, M.: *Flexible Pattern Matching in Strings.* The press Syndicate of The University of Cambridge. 2002.

[19] PAGH, R.: Hash and displace: Efficient evaluation of minimal perfect hash functions. (2009), Availaible on: `www.it.-c.dk/people/pagh/papers/hash.pdf`.

[20] SAHINALP, S. AND VISHKIN, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science*, 1996, pp. 320–328.

[21] SALMELA, L., TARHIO, J. AND KYTÖJOKI, J.: Multipattern string matching with q-grams. *ACM Journal of Experimental Algorithmics (JEA)* , Vol. 11, Article No. 1.1, 2006, pp. 1-19.

[22] SLEATOR, D. D. AND TARJAN, R. E.: A data structure for dynamic trees. *Journal of Computer and System Sciences*, Vol. 26, No. 3, 1983, pp. 362–391.

[23] WEINER, P.: Linear Pattern Matching Algorithms, In *Proceedings of Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[24] KNUTH, D.E.: *The Art of Computer Programming* . Addison-Wesley Publishing Company. Vol. 3, 1973, pp. 506–549.

[25] WU, S. AND MANBER, U.: A fast algorithm for multi-pattern searching. *Report tr-94-17*, Department of Computer Science, University of Arizona, Tucon, AZ, 1994.

[26] YATES,R. B. AND NETO, B. R.: Modern Information Retrieval. *The ACM press.A Division of the Association for Computing Machinery,Inc*, 1999, pp. 191–227.

[27] ZAÏANE, O. R.: CMPUT 391: Inverted Index for Information Retrieval, *University of Alberta*, 2001.

[28] ZOBEL, J. AND MOFFAT, A.: Inverted Files Versus Signature Files for Text Indexing. *ACM Transaction on Database Systems*, Vol. 23, No. 4, 1998, pp. 453–490.

[29] ZOBEL, J.: and Moffat, A.: Inverted Files for Text Search Engines. *ACM Computing Surveys*, Vol. 38, No. 2, 2006, pp. 1–56.

[30] KHANCOME, C. AND BOONJING, V.: Optimal Linear-time Multi-string Pattern Matching Algorithm. *International Journal of Computational Science*, Vol. 3 No. 6, 2009, pp. 629–641.

[31] KHANCOME, C. AND BOONJING, V.: Inverted Lists String Matching Algorithms. *International Journal of Computer Theory and Engineering*, Vol. 2 No. 3, 2010, pp. 1793–8201.

[32] ĆISAR, P. BOŠNJAK, S. MARAVIĆ ĆISAR, S. : EWMA Based Threshold Algorithm for Intrusion Detection. *Computing and Informatics*, Vol. 29 No. 6+, 2010, pp. 1089–1101.

[33] LU, P. CHE, Y. AND WANGK, Z.: UMDA/S: An Effective Iterative Compilation Algorithm for Parameter Search. *Computing and Informatics*, Vol. 29 No. 6+, 2010, pp. 1159–1179.

[34] MAKULA, M. AND BEŇUŠKOVÁ Ľ.: Interactive visualisation of oligomer frequency in DNA. *Computing and Informatics*, Vol. 28 No. 5, 2009, pp. 695–710.

[35] HU, Y. WANG, P.-F. AND HWANG, KAI.: A Fast Algorithm for Multi-String Matching Based on Automata Optimization. *C2010 2nd International Conference on Future Computer and Communication*, Vol. 2 , 2010, pp. 379–383.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

[36] Askitis, N. and Zobel, J.: Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache. *ACM Journal of Experimental Algorithmics*, Vol. 15 No. 1, Article 1.7, 2011, pp. 1–61.

[37] Belazzougui, D.: Worst Case Efficient Single and Multiple String Matching in the RAM Model. *21st International Workshop on Combinatorial Algorithms (IWOCA 2010)*, LNCS 6460, 2011, pp. 90–102.

[38] Haapasalo, T. Silvasti, P. Sippu, S. and Soisalon-Soininen, E.: Online Dictionary Matching with Variable-Length Gaps. *10th International Symposium on Experimental Algorithms (SEA 2011)*, LNCS 6630, 2011, pp. 76–87.

[39] Kuruppu, S. Beresford-Smith, B. Conway, T. and Zobel J.: Iterative Dictionary Construction for Compression of Large DNA Data Sets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 9 No. 1, 2012, pp. 137–149.

[40] Jin Kim, H. Kim, H.-S. and Kang, S.: A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems. *IEEE Transaction on Parallel and Distributed Systems*, Vol. 22 No. 11, 2011, pp. 1904–1911.

[41] Dai, L. and Xia, Y.: A Lightweight Multiple String Matching Algorithm. *International Conference on Computer Science and Information Technology 2008 (ICCSIT'08)*, Singapore, Aug. 29-Sept. 2, 2008, pp. 611–615.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

BIOGRAPHY

PERSONNEL INFORMATION

Thai Name: นายเชาวลิต ขันคำ

English Name: Mr. Chouvalit Khancome

Date of Birth: 24 March 1972

Permanent Address: 99/42 Ban Mai Autchara 2007, Chachoengsao-Bangnumpreaw Road, Tumbon Namuang, Amphur Muang Chachoengsao, 24000, Thailand.

Telephone: (+66) 086-8301458

E-mail: chouvalit@hotmail.com, sk_aran01@yahoo.com, chouvalit.kha@csit.rru.ac.th

EDUCATION

2005 – 2012 **Doctor of Philosophy in Computer Science.**
Department of Computer Science, Faculty of Science,
King Mongkut's Institute of Technology Ladkrabang,
Bangkok, Thailand 10520.

2003 – 2005 **Master Degree in Computer Science.**
Department of Computer Science, Faculty of Science,
King Mongkut's Institute of Technology Ladkrabang,
Bangkok, Thailand 10520.

1991 – 1994 **Bachelor Degree in Computer Eduction.**
Department of Computer Science,
Faculty of Science and Tecnology,
Rajabhat Mahasarakam University,
Mahasarakam, Thailand, 44000.

1895 – 1990 **Highschool in Science-Math Programme.**
Aranyaprathet School, Aranyaprathet District,
Sakeaw, Thailand, 27120.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้