國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

應用於LZW壓縮序列之高效能字串比對機制

Efficient Pattern Matching Scheme in LZW Compressed Sequences

研究生　　：黃迺倫

指導教授：李程輝 教授

中 華 民 國 九 十 五 年 六 月

應用於 LZW 壓縮序列之高效能字串比對機制

Efficient Pattern Matching Scheme in LZW Compressed Sequences

研 究 生： 黃迺倫　　Student: Nai-Lun Huang

指導教授： 李程輝 教授　　Advisor: Prof. Tsern-Huei Lee

國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

A Thesis
Submitted to Department of Communication Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering
June 2006
Hsinchu, Taiwan, Republic of China.

中 華 民 國 九 十 五 年 六 月

應用於LZW壓縮序列之高效能字串比對機制

學生: 黃迺倫　　　　　　　　　　指導教授: 李程輝 教授

國立交通大學電信工程學系碩士班

中文摘要

壓縮字串比對(CPM; compressed pattern matching)乃一新興之研究領域，著眼於此類問題：給定一壓縮序列及一字串，在最少量之解壓縮作用（或無解壓縮作用）下，於序列中尋找字串的出現(pattern occurrences)。可應用於直接在壓縮檔案中做電腦病毒及機密資訊外洩漏與否的偵查。LZW 為一廣受應用且壓縮效率高的壓縮演算法，本論文即記述了我們在壓縮字串比對上，對於處理 LZW 壓縮序列的研究成果。著名的 Amir-Benson-Farach 演算法是最早能在 LZW 壓縮序列中尋得字串的首次出現位置的演算法，我們針對此演算法提出一以位元串映像為基礎的(bitmap-based)實現方式，同時亦將其推廣為可尋得所有的字串出現，並回報它們在序列未壓縮前所處的絕對位置。程式模擬結果顯示，相較於解壓縮後再於解開之序列中搜尋的機制，我們採用的機制具有較高的時間效能；而相較於另一套同樣以位元串映像實現的壓縮字串比對演算法——Navarro-Raffinot 機制，我們的機制對於中等長度以上的字串，在所需的記憶體空間上，是較為節省的。

Efficient Pattern Matching Scheme in LZW Compressed Sequences

Student: Nai-Lun Huang Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering

National Chiao Tung University

Abstract

Compressed pattern matching (CPM) is an emerging research field addressing the problem: Given a compressed sequence and a pattern, find the pattern occurrence(s) in the (uncompressed) sequence with minimal (or no) decompression. It can be applied to detection of computer virus and confidential information leakage in compressed files directly. In this thesis, we report our work of CPM in LZW compressed sequences. LZW is one of the most effective compression algorithms used extensively. We propose a simple bitmap-based realization of the well-known Amir-Benson-Farach algorithm. We also generalize the algorithm to find all pattern occurrences (rather than just the first one) and to report their absolute positions in the uncompressed sequence. Experiments are conducted to compare the performance of our proposed generalization with the decompress-then-search scheme. We found that our proposed generalization is much faster than the decompress-then-search scheme. The memory space requirement of our proposed generalization is compared with that of the Navarro-Raffinot scheme, an alternative CPM algorithm which can also be realized with bitmaps. Results show that our proposed generalization has better space performance than the Navarro-Raffinot scheme for moderate and long patterns.

誌謝

由衷感謝我的指導教授——李程輝教授。在研究的過程中，您給予充分的信任，讓我有足夠的空間思考，並選擇感興趣的題目著手研究。在您的教誨下，我學到做研究的正確態度與方法，此外，您適時的鼓勵也讓我增加自信心。尤其感謝您的悉心指導以及在本論文研究上所提供的各項建議與協助，讓我有機會將研究成果投稿至IEE期刊，這對我來說是莫大的鼓舞。

感謝網路技術實驗室的魏震榮學長、郭耀文學長和謝景融學長在課業及研究上的指教，也要感謝一起奮鬥的同窗夥伴——郁文、政家和紹瑜，不論在課業、研究或生活上，都能不吝分享、相互勉勵，能成為你們之中的一份子，我感到幸運而且珍惜。

最後，我要特別感謝我的父親黃金宗先生與母親柯祝女女士，感謝您們的栽培與教養，以及無限的關懷和鼓勵，每當遭遇挫折，是您們給予我溫暖的避風港，也是您們再度教會我勇敢，讓我能重拾信心，接受挑戰。同時，感謝我的男友羅木榮先生所給予我的一切支持與包容。

謹將此論文獻給我的父親與母親。

西元2006年6月 於風城交大

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the population of communication networks users grows at a rapid rate, it is expected that the networks be capable of delivering data more effectively. In other words, how to utilize the transmission bandwidth efficiently is a key upon which the success of the communication networks heavily relies. Obviously, an economic way to utilize limited bandwidth efficiently is to send smaller amount of data by using data compression mechanisms. Accordingly, compressed pattern matching (CPM) that performs pattern search directly on the compressed data without initial decompression gains more and more attention. The CPM problem is often defined as: Given a compressed sequence and a pattern, find the pattern occurrence(s) in the (uncompressed) sequence with minimal (or no) decompression. Possible applications of CPM include detection of computer virus and confidential information leakage in compressed files directly.

Since LZW [1] is one of the most effective and popular lossless compression algorithms, CPM in LZW compressed sequences is quite important. In the last decade, many related researches have been conducted. The first CPM algorithm which finds the first pattern occurrence in an LZW compressed file was presented in [2]. The complexity of the algorithm is $O(n+m^2)$ in both time and space, where n and m are, respectively, the lengths of the compressed text and the pattern. It was shown that, with different implementations, one can trade between the amount of extra space used and the algorithmm’s time complexity. This algorithm is now well-known and will be referred to as the Amir-Benson-Farach (ABF) algorithm in this thesis. Details of the ABF algorithm are presented in Chapter 2.2. In [3], the ABF algorithm is extended to find all pattern occurrences. The basic idea is to use a flag to indicate that complete pattern occurs inside a compressed data block, in addition to checking pattern occurrences across two consecutive blocks. However, the algorithm cannot tell how many occurrences are there inside a block. Moreover, there is no discussion about how to efficiently realize it. In [4], another CPM algorithm was proposed to do decompression and pattern matching on-the-fly. The drawback of the algorithm is its high computation complexity because it still needs partial decompression. Reference [5] presented a general scheme to find all pattern occurrences in sequential blocks and realized the scheme by using the technique of bit-parallelism. This scheme can be applied to LZ-family compression algorithms such as LZW and LZ77 and will be referred to as the Navarro-Raffinot (NR) algorithm in this thesis. A similar bitmap based implementation for pattern matching in LZW compressed sequences was independently proposed in [6]. The scheme was then generalized to match multiple patterns simultaneously [7].

In this thesis, we present our work of CPM in LZW compressed sequences. We propose a simple and efficient realization of the ABF algorithm. Moreover, a generalization of the ABF algorithm to find all pattern occurrences and report their absolute positions in the uncompressed sequence is presented. Experiments are conducted to compare the performance of our proposed generalization with the decompress-then-search scheme. We found that our proposed generalization significantly outperforms the decompress-then-search scheme. When compared with the NR scheme, our proposed generalization requires less memory space for moderate and long patterns with roughly the same throughput performance.

The rest of this thesis is organized as follows. Chapter 2.1 and Chapter 2.2 give brief reviews of LZW and ABF algorithms, respectively. Efficient realization of the ABF algorithm with bitmaps is presented in Chapter 3.1, followed by the generalization for all pattern occurrences in Chapter 3.2. Chapter 4 describes the most related work, i.e., the NR scheme. Experimental results and comparisons are shown in Chapter 5. Finally, Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 The LZW Compression Algorithm

In this chapter, we briefly review the LZW compression algorithm and the corresponding decompression procedure [1]. The notations used here are similar to those in [2]. Let $S = c_1c_2c_3...c_u$ be the uncompressed sequence (or text) of length u over alphabet $\Sigma = \{a_1, a_2, a_3, ..., a_q\}$, where q is the size of the alphabet. The LZW compressed format of S is $S.Z$ and each code in $S.Z$ is $S.Z[i]$, where $1 \leq S.Z[i] \leq n+q-1$ for $i = 1, ..., n$. The pattern being searched is $P = p_1p_2p_3...p_m$, where m denotes the length of P and $p_i \in \Sigma$ for $1 \leq i \leq m$. For convenience, we use the notation $S_1 S_2$ to denote the concatenation of two strings S_1 and S_2.

The LZW is a dictionary-based compression algorithm that uses a trie T_S to generate the compressed sequence. Each node on T_S contains:

- A node number: A unique number in the range $[0, n+q-1]$. ("node N" or "N" represents "the node numbered N" in this thesis.)
- A label: A symbol belonging to Σ.
- A chunk: The string that the node represents. It is simply the concatenation of the labels on the path from root to the node.

T_S and the compressed sequence are constructed as follows:

1. T_S is initialized as a $(q+1)$-node trie consisting of a root node numbered 0 and labeled *NULL* and q child nodes numbered $1, ..., q$. Child node i is labeled a_i.

2. During compression, the LZW algorithm finds the longest substring in the uncompressed sequence that is a chunk represented by some node *N* on T_S and outputs *N* to *S.Z*. T_S is then grown by adding a new node as a child of *N*. The new node's label is the next unencoded symbol in the sequence.

At the end of the compression, there are $n+q$ nodes on T_S.

The decompression procedure constructs the same trie T_S and uses it to decode *S.Z*. It is obvious that both compression and decompression can be done in time $O(u)$. The following observation makes it possible to construct T_S from *S.Z* in time $O(n)$ without decoding *S.Z* [2]. Note that, in order to construct T_S from *S.Z*, an additional symbol is stored in each node. This additional symbol is the first symbol of the string represented by the node.

Observation. Each code $S.Z[l]$ $(1 \le l \le n-1)$ causes creation of a new node numbered $l+q$ as a child of node $S.Z[l]$.

- The first symbol of node $l+q$ is the first symbol of $S.Z[l]$'s chunk.
- The last symbol or the label of node $l+q$ is the first symbol of $S.Z[l+1]$'s chunk if $S.Z[l+1]$ is different from $l+q$ or the first symbol of $S.Z[l]$ otherwise.

2.2 The Amir-Benson-Farach Algorithm

The Amir-Benson-Farach (ABF) algorithm [2] is an effective scheme which finds the first pattern occurrence in LZW compressed sequence without decompression. To facilitate pattern matching, the following terms of a node on T_S are defined with respect to pattern *P*.

Definition 1: A chunk is a prefix chunk if it ends with a nonempty prefix of *P*. Similarly, a chuck is a suffix chunk if it begins with a nonempty suffix of *P*.

Definition 2: A chunk is an internal chunk if it is an internal substring of *P*. That is, the substring $p_i...p_j$ is an internal chunk if $1 \le i \le j \le m$.

Definition 3: The prefix number of a chunk is the length of the longest pattern prefix the chunk ends with. Similarly, the suffix number of a chunk is the length of the longest pattern suffix the chunk begins with.

Definition 4: The internal range $[i, j]$ $(1 \le i \le j \le m)$ of a chunk indicates that the chunk is the internal substring $p_i...p_j$.

If a node's chunk is a prefix chunk, a suffix chunk, or an internal chunk, the node is called a prefix node, a suffix node, or an internal node, respectively. Moreover, prefix number = 0, suffix number = 0, or internal range = [0, 0] means that the node is not a prefix node, a suffix node, or an internal node, respectively.

The ABF algorithm consists of the Pattern Preprocessing part and the Compressed Text Scanning part which are described separately below.

A. Pattern Preprocessing

The pattern is pre-processed to allow answering the following queries:

1 Let S_1 be a pattern prefix with prefix number P_x (which is 0 if S_1 is a null string) and S_2 be a string with internal range *I* (which is [0, 0] if S_2 is not an internal substring of *P*).

$Q_1(P_x, I)$ = the length of the longest pattern prefix that is a suffix of $S_1 S_2$.

2 Let S_1 be a pattern prefix with prefix number P_x (which is 0 if S_1 is a null string) and S_2 be a pattern suffix with suffix number S_x.

$$\boldsymbol{Q}_2(\boldsymbol{P}_x, \boldsymbol{S}_x) = \begin{cases} \boldsymbol{i}, & \boldsymbol{i} \text{ is the smallest index of } \boldsymbol{S}_1\boldsymbol{S}_2 \text{ where the pattern occures.} \\ 0, & \text{no pattern occurs in } \boldsymbol{S}_1\boldsymbol{S}_2. \end{cases}$$

3 Let S_1 be an internal substring of P and $\alpha \in \Sigma$.

$$\boldsymbol{Q}_3(\boldsymbol{S}_1, \alpha) = \begin{cases} [\boldsymbol{i}, \boldsymbol{j}], & \boldsymbol{S}_1\alpha \text{ is the internal substring } \boldsymbol{p}_i \ldots \boldsymbol{p}_j. \\ [0,\ 0], & \boldsymbol{S}_1\alpha \text{ is not an internal substring of } \boldsymbol{P}. \end{cases}$$

B. Compressed Text Scanning

The compressed text scanning part is further divided into two components: the LZW Trie Construction and the Pattern Search. When constructing T_S, each node is assigned a node number, a first symbol, a label, a prefix number, a suffix number, and an internal range. The Pattern Search part keeps track of the largest partial match and finds out if the partial match can be extended to a complete match. The compressed text scanning procedure is described below.

```
Initialize: variable $Prefix \leftarrow 0$
for $l = 1$ to $n$ do
(Let node $S.Z[l]$'s prefix number = $P_x$, suffix number = $S_x$ and internal range = $I$.)
1    LZW Trie Construction
    1.1    Add a new node numbered $l+q$ to $T_S$ as a child node of $S.Z[l]$. Let $\alpha$ be
           the label of node $l+q$.
    1.2    The first symbol of node $l+q$ is that of node $S.Z[l]$.
    1.3    The label of node $l+q$ is the first symbol of node $S.Z[l+1]$. (If $S.Z[l+1]$ =
           $l+q$ then the label is $S.Z[l]$'s first symbol.)
    1.4   If $S.Z[l]$ is an internal node with corresponding string $S_1$
              Set $l+q$'s internal range $[i, j]$ as $Q_3(S_1, \alpha)$.
          Else
              Set $l+q$'s internal range $[i, j]$ as [0, 0].
    1.5   If $j = m$, set $l+q$'s suffix number as $m-i+1$. Otherwise, set $l+q$'s suffix
          number as $S_x$.
    1.6   Set $l+q$'s prefix number as $Q_1(P_x, I_\alpha)$, where $I_\alpha$ is the internal range of $\alpha$.
2   Pattern Search
If $Prefix = 0$
    $Prefix \leftarrow P_x$
Else        // $Prefix \neq 0$
    If $S.Z[l]$ is a suffix node        // $S_x \neq 0$
        // Check the pattern occurrence with $Q_2(Prefix, S_x)$
        If $Q_2(Prefix, S_x) \neq 0$
            a pattern occurrence is found
    If $S.Z[l]$ is an internal node     // $I \neq [0, 0]$
        $Prefix \leftarrow Q_1(Prefix, I)$
    Else        // $S.Z[l]$ is not an internal node
        $Prefix \leftarrow P_x$
```

To answer query Q_3, we need to construct the suffix trie of P, denoted by ST_P. Note that there are m nonempty suffixes of P and the number of nodes in ST_P is $O(m^2)$. Moreover, there is a unique node on ST_P which represents a specific substring of P (even if the substring appears multiple times in P). Query $Q_3(S_1, \alpha)$ can be easily answered by tracing ST_P. If there is a node representing substring S_1

on ST_P which has an outgoing edge labeled *α*, then $S_1\alpha$ is an internal substring of *P*. If no such outgoing edge exists, then $S_1\alpha$ is not an internal substring of *P* and its internal range is [0, 0].

Note that it is possible to reduce the space complexity of ST_P. A node on ST_P is said to be explicit if and only if (iff) either it represents a suffix of *P* or it has more than one child node. The nodes that are not explicit are said to be implicit. One can construct the compacted ST_P which contains only explicit nodes of the uncompacted ST_P by eliminating all implicit nodes in between two explicit nodes. As a result, the label on each edge becomes a substring of *P*. The space complexity can be reduced because the number of explicit nodes on the uncompacted ST_P is $O(m)$ [2].

Query Q_3 can be answered with the compacted ST_P as follows. Let S_1 be an internal substring of *P*. If S_1 is represented by a node, say node *N*, on the compacted ST_P, then $S_1\alpha$ is an internal substring iff *α* is the first symbol of a label on some outgoing edge of node *N*. Suppose that there is no node on the compacted ST_P which represents S_1. In this case, one can find two nodes on the compacted ST_P, say nodes N_1 and N_2, such that node N_1 represents the longest prefix (could be empty) of S_1 and node N_2 represents the shortest internal substring of *P* which contains S_1 as a prefix. Note that node N_1 is actually a parent node of node N_2 on the compacted ST_P. Assume that node N_1 represents substring S_1'. As a result, $S_1\alpha$ is an internal substring iff the $(|S_1|-|S_1'|+1)^{th}$ symbol of the label on the edge connecting nodes N_1 and N_2 is equal to *α*.

Queries Q_1 and Q_2 can be answered in constant time during text scanning if two tables of space complexity $O(m^2)$ are constructed in advance [2]. Obviously, when m is large, these two tables require significant amount of memory. In the following chapter, we present a novel realization which requires only $O(m)$ storage.

Chapter 3

An Efficient Pattern Matching Scheme in LZW Compressed Sequences

3.1 An Efficient Realization of Amir-Benson-Farach Algorithm

Let us consider the implementation of query Q_2 first. Given a pattern P = $p_1p_2p_3 \dots p_m$ of length m, we need two sets of bitmaps where each bitmap has m bits. The first set, called prefix bitmaps, consists of m bitmaps that correspond to the m possible prefix numbers 0, 1, 2, …, m-1. Let $A_i = a_i^1 a_i^2 \dots a_i^m$ denote the i^{th} prefix bitmap which corresponds to prefix number i-1. We assign a_i^k = 1 iff $k \le i$ and $p_{i-k+1} \dots p_{i-1}$ is a nonempty prefix of P, i.e., $p_{i-k+1} \cdots p_{i-1} = p_1 \cdots p_{k-1}$. Note that $p_{i-k+1} \dots p_{i-1}$ represents a null string if k =1. Clearly, with the assignment, we have a_i^1 = 0 for all i, $1 \le i \le m$, a_i^i = 1 if $1 < i \le m$, and a_i^j = 0 if $j > i$.

The second set of bitmaps, called suffix bitmaps, consists of m-1 bitmaps which correspond to the m-1 possible suffix numbers 1, 2, …, m-1. Again, the size of each suffix bitmap is m bits. Let B_i = $b_i^1 b_i^2 \dots b_i^m$ be the i^{th} suffix bitmap which corresponds to suffix number i. Assign b_i^k = 1 iff $k \ge m$-i+1 and $p_{m-i+1} \cdots p_{2m-i-k+1}$ is a nonempty suffix of P, i.e., $p_{m-i+1} \cdots p_{2m-i-k+1}$ = $p_k \cdots p_m$. In other words, b_i^k = 1 iff the length-(m-k+1) prefix of $p_{m-i+1} \dots p_m$ is a nonempty suffix of P. Similarly, with the assignment, we have $b_i^{m-i+1} = 1$ and $b_i^j = 0$ if $j < m$-i+1.

We now show that query $Q_2(P_x, S_x)$ can be answered with the two sets of bitmaps. Let $P_x = i$-1 and $S_x = k$. In other words, we have $S_1 = p_1 \ldots p_{i-1}$, $S_2 = p_{m-k+1} \ldots p_m$, and $S_1 S_2 = p_1 \ldots p_{i-1} p_{m-k+1} \ldots p_m$. Note that $S_1 = p_1 \ldots p_{i-1}$ represents a null string if i = 1. To answer query Q_2, we first perform the bitwise AND operation of A_i and B_k. Let $R = r_1 r_2 \ldots r_m$ denote the result, i.e., $R = A_i \otimes B_k$, where $\otimes$ represents the bitwise AND operation. If $i > 1$ and there is a cross-boundary pattern occurrence starting at the j^{th} position of S_1, then it must hold that $p_j \ldots p_{i-1}$ is a prefix of P and $p_{m-k+1} \ldots p_{2m-k-i+j}$ is a suffix of P. Since $p_j \ldots p_{i-1}$ is a prefix of P, we have $a_i^{i-j+1} = 1$. Similarly, $p_{m-k+1} \ldots p_{2m-k-i+j}$ is a suffix of P implies $b_k^{i-j+1} = 1$. Consequently, the pattern occurrence can be detected because it holds that $r_{i-j+1} = 1$. To determine the first pattern occurrence, we need only identify the rightmost 1 of R. Assume that the rightmost 1 of R occurs in the l^{th} position, i.e., $r_l = 1$ and $r_i = 0$ for l+1$\le i \le m$, then the first pattern occurrence is found starting at the $(|S_1| - l + 2)^{th}$ position of S_1. There is no pattern occurrence crossing the boundary of S_1 and S_2 if $r_x = 0$ for all x, $1 \le x \le m$. In the case that i = 1, i.e., S_1 is a null string, $a_i^x = 0$ for all x, $1 \le x \le m$ implies $r_x = 0$ for all x, $1 \le x \le m$ as expected. Note that the implementation can actually find all cross-boundary pattern occurrences. This function will be used in the generalization to find all pattern occurrences presented in Chapter 3.2.

To implement query Q_3, we need to construct the compacted suffix trie ST_P.

The answer of Q_3 can be obtained by tracing the compacted ST_P, as mentioned before. It is obvious that the implementation can result in correct answer for query Q_3 and thus its proof is omitted.

Let us consider the implementation of query Q_1. A third set of *m*-bit bitmaps are required. For convenience, we number the nonempty suffixes of *P* so that suffix *i* is of length *i*, $1 \le i \le m$. We need a bitmap to be associated with each node on the compacted ST_P. Consider the bitmap C_N = $c_N^1 c_N^2 \ldots c_N^m$ associated with a particular node *N*. Assign c_N^i = 0 for all *i*, $1 \le i \le m$, if node *N* is the root node. The bitmap associated with the root node is for the internal range [0, 0]. Assume that node *N* is not the root node. It is clear that node *N* represents a unique nonempty substring of *P*. Assign c_N^{m-k+1} = 1 iff node *N* represents suffix *k* or the node which represents suffix *k* is a descendent node of *N*. Note that the above assignment results in c_N^{m-k+1} = 1 iff the substring represented by node *N* is a nonempty prefix of suffix *k*.

With the prefix bitmaps and the bitmaps associated with the nodes on the compacted ST_P, one can now answer query $Q_1(P_x, I)$. Let *M* be the node on the LZW trie T_S which represents the substring S_2 with internal range *I*. Also, let *N* be the node on the compacted ST_P which either represents the substring S_2 or the substring it represents is the shortest substring represented by any node on the compacted ST_P which contains S_2 as a prefix. Node *M* contains a pointer which points to the bitmap associated with node *N*. To answer query $Q_1(P_x, I)$, we perform the bitwise AND operation of the prefix bitmap corresponding to prefix number P_x and the bitmap pointed to by the pointer stored in node *M*. Let *R* = $r_1r_2...r_m$ denote the result of the bitwise AND operation. If r_i = 0 for all *i*, $1 \le i \le m$, then $Q_1(P_x, I)$ returns the prefix number of node *M*. Assume that r_i = 1 for at least one *i*. The answer of $Q_1(P_x, I)$ equals (*k*-1) + Dep(*M*) if r_k = 1 and r_i = 0, $k+1 \le i \le m$, where Dep(*M*), the depth of node *M*, denotes the length of the chunk represented by node *M*.

The correctness of the above implementation for query Q_1 can be proved as follows. Assume that P_x = *i*-1 so that S_1 = $p_1...p_{i-1}$. If *i* > 1 and the longest pattern prefix that is a suffix of S_1 S_2 starts at the j^{th} position of S_1, then it holds that $p_j...p_{i-1}$ is a prefix of *P* and suffix *m*-*i*+*j* contains S_2 as a prefix. As a result, we have a_i^{i-j+1} = 1 and c_N^{i-j+1} = 1 which implies r_{i-j+1} = 1. In other words, such a prefix can be detected by the bitwise AND operation. Since we are looking for the longest pattern prefix, the rightmost 1 of *R* is selected. If it happens in the k^{th} position, then the symbol p_{i-k+1} starts the longest pattern prefix whose length is equal to (*k*-1) + Dep(*M*). Of course, if r_i = 0 for all *i*, $1 \le i \le m$, then the longest pattern prefix is completely contained in S_2, which implies the length of the longest pattern prefix is equal to the prefix number of node *M*. Therefore, the above implementation does result in correct answer for query Q_1.

Below are two examples.

Example 1. Let *P* = *abcab*. Table 3-1 and Table 3-2 show the prefix bitmaps and the suffix bitmaps of *P*, respectively. As an example of query *Q*2, assume that S_1 =

abca and S_2 = *bcab*. Consequently, we have P_x = 4, S_x = 4, and *R* = 01001. For this example, the first pattern occurrence starts at the first position of S_1. In fact, as indicated by the two 1's appeared in *R*, there are two pattern occurrences in $S_1 S_2$.

Table 3-1. Prefix bitmaps of P = *abcab*

P_x	Prefix	Bitmap #	Bitmap
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*abc*	4	00010
4	*abca*	5	01001

Table 3-2. Suffix bitmaps of P = *abcab*

S_x	Suffix	Bitmap #	Bitmap
1	*b*	1	00001
2	*ab*	2	00010
3	*cab*	3	00100
4	*bcab*	4	01001

Example 2. Let *P* = *ababc*. Table 3-3 shows the prefix bitmaps of *P*. For ease of description, we use the uncompacted suffix trie ST_P of *P* as illustrated in Figure 3-1. The bitmaps associated with the explicit nodes of ST_P are given in Table 3-4. As an example of query Q_1, assume that S_1 = *abab* and the internal range *I* = [1, 2] (or [3, 4]) which represents substring S_2 = *ab*. In our implementation, *I* = [1, 2] (or [3, 4]) is represented by node 2 of the uncompacted suffix trie ST_P. Since the prefix number of S_1 is 4 with corresponding prefix bitmap 00101 and the bitmap associated with node 2 on ST_P is 10100, we have *R* = 00100. In other words, one pattern prefix starting at the third position of S_1 is found. As a result, the answer of query Q_1(4, [1, 2]) (or Q_1(4, [3, 4])) is (3-1) + $|S_2|$ = 2 + 2 = 4. As another example, if S_1 = *ab* and *I* = [2, 4] (the bitmap to be used is the one associated with node 9 of ST_P) which represents substring S_2 = *bab*, then we have P_x = 2 and *R* = 00100 $\otimes$ 01000 = 00000. In this case, the answer of query Q_1(2, [2, 4]) is 2, which is the prefix number of *bab*.

Let us consider now examples of query Q_3. Assume that S_1 = *ab* which is represented by node 2 of the uncompacted ST_P. If $\alpha = b$, then we have $Q_3(S_1, \alpha)$ = [0, 0] because there is no transition from node 2 to any node with label *b*. However, if $\alpha = c$, then we have $Q_3(S_1, \alpha)$ = [1, 3] which is represented by node 10 of ST_P.

Table 3-3. Prefix bitmaps of *P* = *ababc*

P_x	Prefix	Bitmap #	Bitmap
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*aba*	4	01010
4	*abab*	5	00101

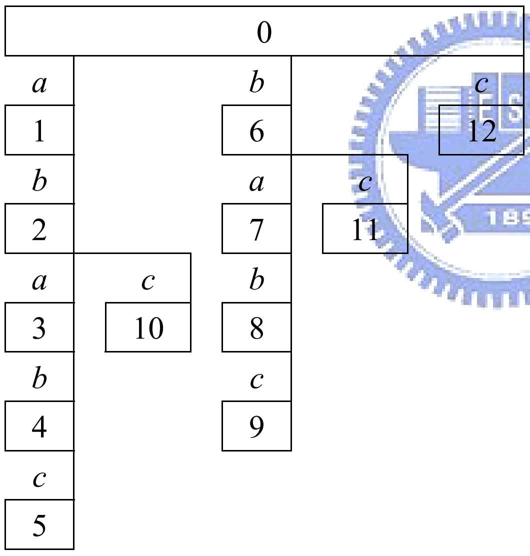

Figure 3-1. The uncompacted suffix trie ST_p of *P* = *ababc*

Table 3-4. Bitmaps associated with explicit nodes

Explicit node	Bitmap
0	00000
2	10100
5	10000
6	01010
9	01000
10	00100
11	00010
12	00001

3.2 Generalization to All Pattern Occurrences

As mentioned before, the pattern occurrence checking in the original ABF algorithm is only performed cross two consecutive data blocks. Moreover, only the first occurrence is reported. To generalize the ABF algorithm to find all pattern occurrences, we need to consider all pattern occurrences cross two consecutive data blocks and those inside a data block as well. Our implementation presented in Chapter 3.1 allows detection of all pattern occurrences cross two consecutive data blocks. Therefore, the remaining work is to detect all pattern occurrences inside a data block. The generalization is designed to also report the absolute positions of pattern occurrences. Reporting the absolute positions of all occurrences may be desirable to some applications.

To detect all pattern occurrences inside a data block, we add two fields, called pattern inside flag (PIF) and pattern inside pointer (PIP), to every node of the LZW trie T_S. The PIF flag is an indication of existence of patterns inside the chunk and the PIP pointer is used for backtracking to find the positions of all pattern occurrences inside the chunk. For the root node, its PIF is 0 and its PIP pointer points to the node itself, which is also 0. Assume that a new node M is to be added as a child node of node N. The PIP pointer of node M inherits the PIP value of node N if N is not a final node, i.e., a node whose chuck ends with the complete pattern P. To identify final nodes, we let the prefix number of a final node equal m. In case node N is a final node, the PIP pointer of node M points to node N. Similarly, the PIF of node M inherits the PIF value of node N unless the PIF of node N is 0 and node M is a final node. In this case, we set the PIF of node M to 1. With these additional fields, one can trace back the LZW trie to find all pattern occurrences inside a chuck. The trace-back ends once a node with PIP pointer points to the root node, i.e., PIP = 0, is reached. Note that although PIF can be replaced by the PIP pointer and the prefix number (PIF = 1 is equivalent to PIP $\neq$ 0 or prefix number = m), we suggest to use PIF to simplify the checking of pattern existence inside a chunk.

Note that, since we allow the prefix number of a node to be equal to *m*, we need to add to the set of prefix bitmaps an additional prefix bitmap corresponding to prefix number = *m*. The contents of the bitmap are assigned with the same algorithm described in Chapter 3.1. It is clear that the value of the variable *Prefix* may equal *m* too. However, it does not cause any error because the bitmap corresponding to prefix number = *m* is the same as the bitmap corresponding to prefix number = *k*, where $p_{m-k+1}\ldots p_m$ is the longest suffix of *P* which is also a proper prefix of *P*, i.e., a prefix which is not *P* itself.

For convenience, we also allow the suffix number of a node to be equal to *m*. As a consequence, another bitmap corresponding to suffix number = *m* is added to the set of suffix bitmaps. Again, the contents of the added suffix bitmap are assigned according to the algorithm described in Chapter 3.1 and the additional suffix bitmap does not cause any error because $a_i^1 = 0$ for all *i*, $1 \le i \le m+1$.

To report absolute positions of pattern occurrences, we can rely on the depth fields of nodes on the LZW trie T_S and a global variable *COUNT* which stores the number of bytes in text S that have been scanned. Computation of the depth field is simple. The depth of the root node is 0. When node *M* is added as a child node of node *N*, the depth of node *M* equals that of node *N* plus one. Clearly, with the depth fields, one can compute the position of a node inside a chuck, which, together with the global variable *COUNT*, can be used to determine the absolute position of any pattern occurrence. The overall generalized algorithm is described below.

A. Pattern Preprocessing

The prefix bitmaps and the suffix bitmaps are computed. Also, the compacted suffix trie ST_P of pattern *P* with the associated bitmaps are determined.

B. Compressed Text Scanning

When constructing the LZW trie T_S, each node's node number, label, prefix number, suffix number, internal range, the first symbol, depth, PIF, and PIP are computed and stored. The compressed text scanning procedure is described below.

```
Initialize: $Prefix \leftarrow 0$, $COUNT \leftarrow 0$
for $l = 1$ to $n$ do
(Let node $S.Z[l]$'s prefix number = $P_x$, suffix number = $S_x$, internal range = $I$, PIF = $F$ and depth = $D$.)
1    LZW Trie Construction
    1.1    Add a new node numbered $l+q$ to $T_S$ as a child node of $S.Z[l]$. Let $\alpha$ be the label of node $l+q$.
    1.2    The first symbol of node $l+q$ is that of node $S.Z[l]$.
    1.3    The label of node $l+q$ is the first symbol of node $S.Z[l+1]$. (If $S.Z[l+1] = l+q$ then the label is $S.Z[l]$'s first symbol.)
    1.4    If $S.Z[l]$ is an internal node with corresponding string $S_1$
               Set $l+q$'s internal range $[i, j]$ as $Q_3(S_1, \alpha)$.
           Else
               Set $l+q$'s internal range $[i, j]$ as [0, 0].
    1.5    If $j = m$, set $l+q$'s suffix number as $m-i+1$. Otherwise, set $l+q$'s suffix number as $S_x$.
    1.6    Set $l+q$'s prefix number as $Q_1(P_x, I_\alpha)$, where $I_\alpha$ is the internal range of $\alpha$.
    1.7   If $F = 0$ and $l+q$'s prefix number = $m$, then $l+q$'s PIF $\leftarrow$ 1.
          Else, $l+q$'s PIF $\leftarrow F$.
    1.8   Set the depth of node $l+q$ as $D+1$.
    1.9   If $P_x = m$, then $l+q$'s PIP $\leftarrow S.Z[l]$.
          Else, $l+q$'s PIP $\leftarrow S.Z[l]$'s PIP.
2   Pattern Search
If $S_x \neq 0$
    Check cross-boundary occurrences with the bitwise AND operation for query
    $Q_2(Prefix, S_x)$.   Let $R = r_1 r_2 \ldots r_m$ be the result of the bitwise AND operation.
    for $k$ = 1 to $m$ do
        If $r_k = 1$
            Report the position: $COUNT - k + 2$
If $F = 1$   // Pattern is inside $S.Z[l]$
    If $P_x = m$
        Report the position: $COUNT + D - m + 1$
    $N \leftarrow S.Z[l]$'s PIP
    While $N \neq 0$
        Report the position: $COUNT + \mathrm{Dep}(N) - m + 1$
        $N \leftarrow N$'s PIP
$Prefix \leftarrow Q_1(Prefix, I)$   // Note that the answer of $Q_1(Prefix, I)$ is $P_x$, if the result of
                                     bitwise AND operation for $Q_1(Prefix, I)$ is all-zero
$COUNT \leftarrow COUNT + D$
```

The following example illustrates the process to detect all pattern occurrences and report their absolute positions.

Example 3. As in Example 2, let P = *ababc*. The prefix bitmaps and the suffix bitmaps are shown in Tables 3-5 and 3-6, respectively. Since the suffix trie of pattern P and the bitmaps associated with the explicit nodes are not changed, they are not reproduced here. Assume that some of the compressed text had been processed and the current value of *COUNT* = 100. Assume further that the last three chunks that had been processed are *xxx*, *xxxx*, and *xaba*, and the current chunk to be processed is N_p = *bcababcxababcxx*.

Table 3-7 shows the contents of the nodes along the path from the root node to node N_p on the LZW tire. Note that there are two pattern occurrences inside the current chunk which can be determined by tracing back the PIP pointers. Table 3-8 shows a brief summary of the results when the last three chunks are processed. The procedure of pattern detection with report of absolute occurrence positions in processing N_p is sketched below.

- Reporting absolute positions of cross-boundary pattern occurrences:

  Since N_p's suffix number = 2 ≠ 0

  // Check cross-boundary occurrences with bitmaps:

  Prefix = 3 with corresponding bitmap 01010.

  N_p's suffix number = 2 with corresponding bitmap 00010.

  The result of bitwise AND operation $R = 01010 \otimes 00010 = 00010$.

  ➔ The absolute occurrence position $COUNT - 4 + 2 = 98$ is reported.

- Reporting absolute positions of inside-chunk pattern occurrences:

  Since N_p's PIF = 1

  Since N_p's PIP = $N_2 \neq 0$

  The absolute occurrence position $COUNT + \text{Dep}(N_2) - m + 1 = 109$ is reported.

Since N_2's PIP = $N_1 \neq 0$

The absolute occurrence position $COUNT$ + Dep(N_1) – m + 1 = 103 is reported.

Since N_1's PIP = 0

The trace-back ends.

Table 3-5. Prefix bitmaps of P = *ababc*

P_x	Prefix	Bitmap #	Bitmap
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*aba*	4	01010
4	*abab*	5	00101
5	*ababc*	6	00000

Table 3-6. Suffix bitmaps P = *ababc*

S_x	Suffix	Bitmap #	Bitmap
1	*c*	1	00001
2	*bc*	2	00010
3	*abc*	3	00100
4	*babc*	4	01000
5	*ababc*	5	10000

Table 3-7. Contents of nodes along the path from root to N_p on T_s

Node number	**Label**	**First symbol**	**Prefix number**	**Suffix number**	**Internal range**	**PIF**	**PIP**	**Depth**
0 (root)	*NULL*	-	-	-	-	0	0	0
	b	*b*	0	0	[2, 2] (or [4, 4])	0	0	1
	c	*b*	0	2	[4, 5]	0	0	2
	a	*b*	1	2	[0, 0]	0	0	3
	b	*b*	2	2	[0, 0]	0	0	4
	a	*b*	3	2	[0, 0]	0	0	5
	b	*b*	4	2	[0, 0]	0	0	6
N_1	*c*	*b*	5 (= m)	2	[0, 0]	1	0	7
	x	*b*	0	2	[0, 0]	1	N_1	8
	a	*b*	1	2	[0, 0]	1	N_1	9
	b	*b*	2	2	[0, 0]	1	N_1	10
	a	*b*	3	2	[0, 0]	1	N_1	11
	b	*b*	4	2	[0, 0]	1	N_1	12
N_2	*c*	*b*	5 (= m)	2	[0, 0]	1	N_1	13
	x	*b*	0	2	[0, 0]	1	N_2	14
N_p	*x*	*b*	0	2	[0, 0]	1	N_2	15

Table 3-8. Brief summary of the results when the last three chunks are processed

	The last three chunks that had been processed			N_p
S =…	*xxx*	*xxxx*	*xaba*	*bcababcxababcxx*
Depth =	3	4	4	
COUNT =	92	96	100	
Prefix =	0	0	3	

Chapter 4
Related Work

A different bitmap based implementation was independently developed by two groups of researchers [5] and [6]. The scheme proposed in [5] is more general than the one presented in [6] and thus we will follow its description and call it the Navarro-Raffinot (NR) scheme. As our generalization, the NR scheme can find all pattern occurrences and report their absolute positions. Below is a description of the NR scheme extracted from [5].

The NR scheme is a general technique to perform string matching when the text is presented as a sequence of atomic strings, called blocks, instead of a sequence of symbols. The blocks either have just one symbol or are formed by concatenating previously seen blocks. Let T' denote the text already processed at any moment of the search. When the search process is over, it holds that $T' = T$, the original text.

The blocks are processed one by one. For each new block B, a description for B which has all the information of the block that is relevant for the search is computed. This description is denoted by $D(B) = (L, O, S, P, M)$, where

- $L = |B|$, the length of B in symbols
- $O = \mathrm{Offs}(B)$ = the length in symbols of the text we had processed when B appeared
- $S = \mathrm{Suff}(B)$ = all the pattern positions which either start a complete occurrence of B inside the pattern, or start a proper pattern suffix which matches with a prefix of B. Formally,

$$\mathrm{Suff}(B) = \{|x|,\ P = xBy\} \cup \{|x|,\ |x| > 0 \wedge |z| > 0 \wedge P = xz \wedge B = zy\}$$

- $P = \mathrm{Pref}(B)$ = all the pattern positions which either follow a complete occurrence of B inside the pattern, or follow a proper pattern prefix which matches with a suffix of B. Formally,

$$\mathrm{Pref}(B) = \{|xB|,\ P = xBy \wedge |y| > 0\} \cup \{|z|,\ |z| > 0 \wedge |y| > 0 \wedge P = zy \wedge B = xz\}$$

- M = Matches(B) = all the block positions where the pattern occurs (Ø if $|B| < |P|$). Formally,

$$\text{Matches}(B) = \{|x|,\ B = xPy\}$$

Note that, to simplify the notation, the pattern positions start at zero in the above description, while in previous chapters, the pattern positions start at one.

There are two cases for a new block B: (a) the block is a symbol or (b) the block is a concatenation of other blocks previously known. For case (a), the description $D(B)$ can be obtained directly and, for case (b), it can be derived from the descriptions of the previous blocks.

Once the description of the new block is computed, it is used to update the states of the search. This concludes the processing of a block and the search process moves to the next one. The states of the search contains the matches that have already occurred and the potential matches in progress, that is,

- Res(T') = the text positions that matched up to now. Formally,

$$\text{Res}(T') = \{|x|,\ T' = xPy\}$$

- Active(T') = the set of positions following the pattern prefixes which match a suffix of the current text. Formally,

$$\text{Active}(T') = \{|x|,\ |x| > 0 \wedge |y| > 0 \wedge P = xy \wedge T' = zx\}$$

Hence, when the text processing is complete and T' is the whole text, Res(T) is the answer. The initial state of the search is Res(ε) = Active(ε) = Ø, and $T' = \varepsilon$, where ε denotes the empty string.

Four operations which are used in the search process are defined below [5].

- Left$_i$, which receives a set of Suff() positions not smaller than i, subtracts i to all them and then adds new pattern positions filling the holes left by the shift. Formally,

$$\text{Left}_i(X) = \{x-i,\ x \in X\} \cup \{m-i,\ m-i+1,\ \ldots,\ m-1\}$$

- Right$_i$, which does the same for Pref() positions, in the other direction. Formally,

$$\text{Right}_i\,(X) = \{x+i,\ x \in X\} \cup \{1, 2,\ \ldots,\ i\}$$

- $\text{Add}_i(X) = \{i + x, x \in X\}$, which adds i to all the elements of the set.
- $\text{Subtr}_i(X) = \{i - x, x \in X\}$, which subtracts all the elements of the set from i.

The base case of the scheme is to obtain the description of a block which is a symbol a. We have

- $|B| = 1$
- $\text{Offs}(B) = |T'|$
- $\text{Suff}(B) = \{|x|, P = xay\}$
- $\text{Pref}(B) = \{|xa|, P = xay \wedge |y| > 0\}$
- $\text{Matches}(B) = \text{if } P = a \text{ then } \{0\} \text{ else } \varnothing$

which are direct applications of the general formulas.

Assume that block B is defined as the concatenation of one or more previous blocks. If B is identical to one previous block B', we just copy the description of B' for B. Assume that B is a concatenation of two blocks B_1 and B_2. Note that it suffices to study concatenation of two blocks because the case of more than two blocks is a simple iteration over this procedure. We have to obtain the description for their concatenation $D(B) = D(B_1B_2) = D(B_1) \cdot D(B_2)$ (where $\cdot$ is a notation for concatenation of block descriptions). The formulas were given in [5] as follows

- $|B| = |B_1| + |B_2|$
- $\text{Offs}(B) = |T'|$
- $\text{Suff}(B) = \text{Suff}(B_1) \cap \text{Left}_{|B_1|}(\text{Suff}(B_2))$
- $\text{Pref}(B) = \text{Pref}(B_2) \cap \text{Right}_{|B_2|}(\text{Pref}(B_1))$
- $\text{Matches}(B) = \text{Matches}(B_1) \cup \text{Add}_{|B_1|}(\text{Matches}(B_2))$

$$\cup\ (\text{Subtr}_{|B_1|}(\text{Pref}(B_1) \cap \text{Suff}(B_2)) \cap \{0, 1, 2, \ldots, |B|-m\})$$

We need to update the states of the search after processing a new block B. The formulas to obtain the new $\text{Res}(T'B)$ and $\text{Active}(T'B)$ values from the old $\text{Res}(T')$ and $\text{Active}(T')$ ones are

- $\text{Active}(T'B) = \text{Right}_{|B|}(\text{Active}(T')) \cap \text{Pref}(B)$
- $\text{Res}(T'B) = \text{Res}(T') \cup \text{Add}_{|T'|}(\text{Matches}(B))$

$$\cup\ \text{Subtr}_{|T'|}(\text{Active}(T') \cap \text{Suff}(B) \cap \{m-|B|, m-|B|+1, \ldots, m-1\})$$

The above searching technique can be easily realized with two sets of bitmaps, Pref(B) and Suff(B), for every block B. The length of every bitmap for Pref(B) and Suff(B) is equal to m, the pattern length. Obviously, for LZW compressed sequences, the number of bitmaps for Pref(B) and Suff(B) is the same as the number of nodes on the LZW trie. This number tends to be large for a big file. The states of the search, i.e., Res(T') and Active(T'), and the result of each block B, i.e., Matches(B), can be represented by either bitmaps or arrays of numbers. In the comparison presented in Chapter 5, we assume that Active(T') is represented by an m-bit bitmap and Res(T') and Matches(B) are represented by arrays of numbers. The reason to represent Res(T') as an array of numbers is that it is more space efficient because the number of pattern occurrences is usually much smaller than the file size. The reason to represent Matches(B) as an array of numbers is simply because the length of block B is not fixed.

Chapter 5

Comparison and Experimental Results

In this chapter, we compare the performance of our generalized algorithm with the one that performs decompression followed by pattern searching with the KMP algorithm [8]. The algorithms were implemented in C++ and the experiments were carried out on a PC with an AMD Athlon XP 1800+ CPU operated at 1.15GHz with 224MB of RAM running Microsoft Windows XP operating system. In the first test case, we use dosx.exe (an executable file in Windows) as our text which contains no pattern at all. The uncompressed size of dosx.exe is 53856 bytes. In the second test case, we insert various numbers of patterns with m = 4 in dosx.exe at randomly selected positions. The experimental results of test cases 1 and 2 are shown in Figures 5-1 and 5-2, respectively. As one can see, in comparison with the decompress-then-search algorithm, our proposed generalized algorithm has significantly better performance as expected. The performance of the NR scheme is very close to that of our generalized algorithm and thus is not shown in the figures. However, we can compare the space requirements of our generalized algorithm and the NR scheme.

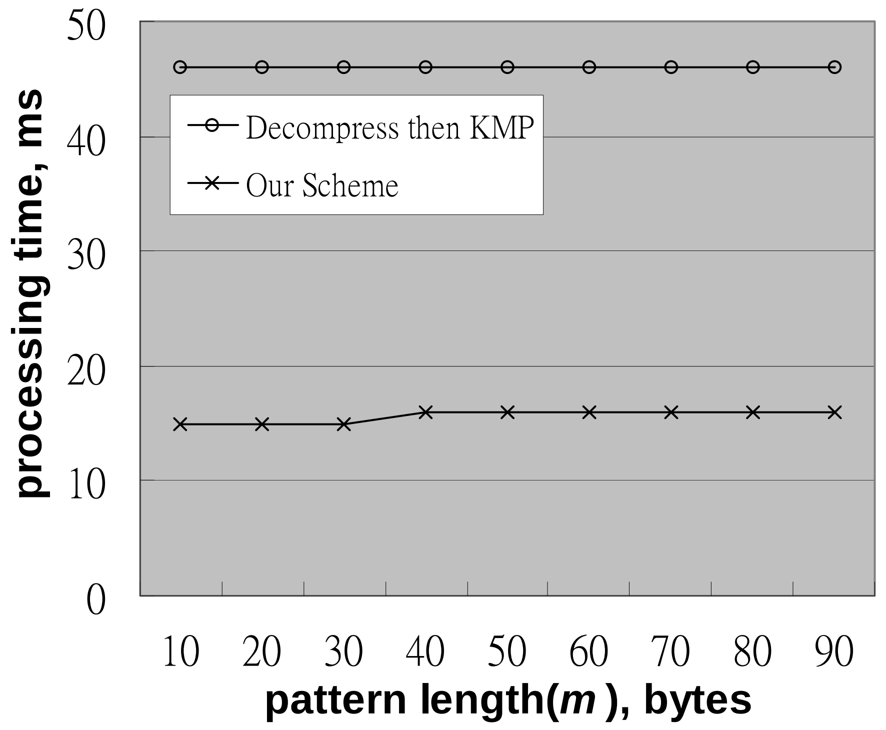

Figure 5-1. Performance comparison for test case 1

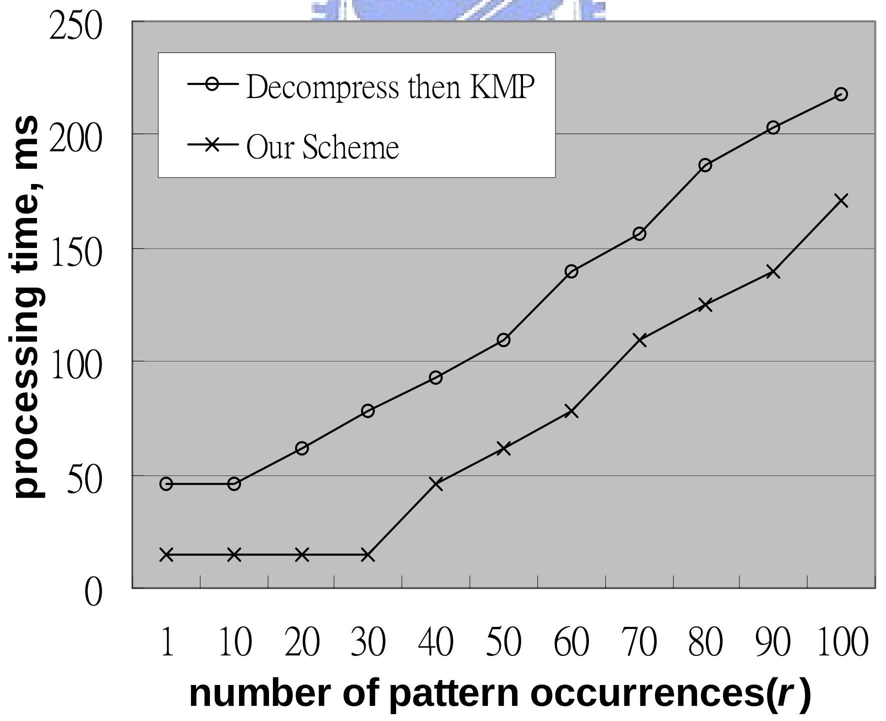

Figure 5-2. Performance comparison for test case 2

In our generalized algorithm, the number of bitmaps, including prefix bitmaps, suffix bitmaps, and the bitmaps associated with the nodes on the compacted suffix trie ST_P is $O(m)$. The LZW trie T_S takes space $O(t)$, where t is the number of nodes on T_S. The prefix number, the suffix number, and the internal ranges stored in every node of T_S are replaced by three pointers, each of size $O(\log_2 m)$ bits, which point to the appropriate bitmaps. Therefore, the space complexity of our generalized algorithm is $O(m+t)$. For the NR scheme, the space complexity is $O(t+r)$, where r is the number of pattern occurrences in text S. Each of the $O(t)$ descriptions contains five elements, L, O, S, P and M. Hence, there are $O(t)$ bitmaps of S and $O(t)$ bitmaps of P, each of the bitmaps has size m bits. Clearly, the space requirement of these two sets of bitmaps increases proportional to the size of the LZW trie. Another significant difference between the NR scheme and our generalized scheme is that the number of pattern occurrences r affects the space requirement of the NR scheme, but not ours. This effect will be studied later. Since the element O is not necessary for every node, we assume that it is omitted and instead a global counter *COUNT* is adopted in comparison.

We ignore the space requirement of the NR scheme caused by r, that is, we intentionally let $r = 0$, in the third test case. The text used in test case 3 is dfrgntfs.exe (an executable file in Windows) whose uncompressed size is 104960 bytes. Comparison of the space requirements of our generalized scheme and the NR scheme for test case 3 is shown in Figure 5-3. It can be seen from the curves that our generalized scheme requires less storage than the NR scheme does if the pattern length is longer than 25.

In test cases 4 and 5 we use case4.txt (a randomly generated text file with 3000 patterns inserted) of uncompressed size 296126 bytes and case5.txt (another randomly generated text file with 18000 patterns inserted) of uncompressed size 1705714 bytes as the texts, respectively. Figures 5-4 and 5-5 show, respectively, the space requirements under these two test cases for different pattern lengths. The NR scheme requires significantly more memory space than our generalized scheme, especially for long patterns. In Figure 5-6, we show the space requirements for pattern length m = 25 with various numbers of patterns inserted in dfrgntfs.exe. The uncompressed size of the modified dfrgntfs.exe is 249860 bytes. As one can see, the space requirement of the NR scheme increases as r increases while the space requirement of our generalized scheme is insensitive to the value of r.

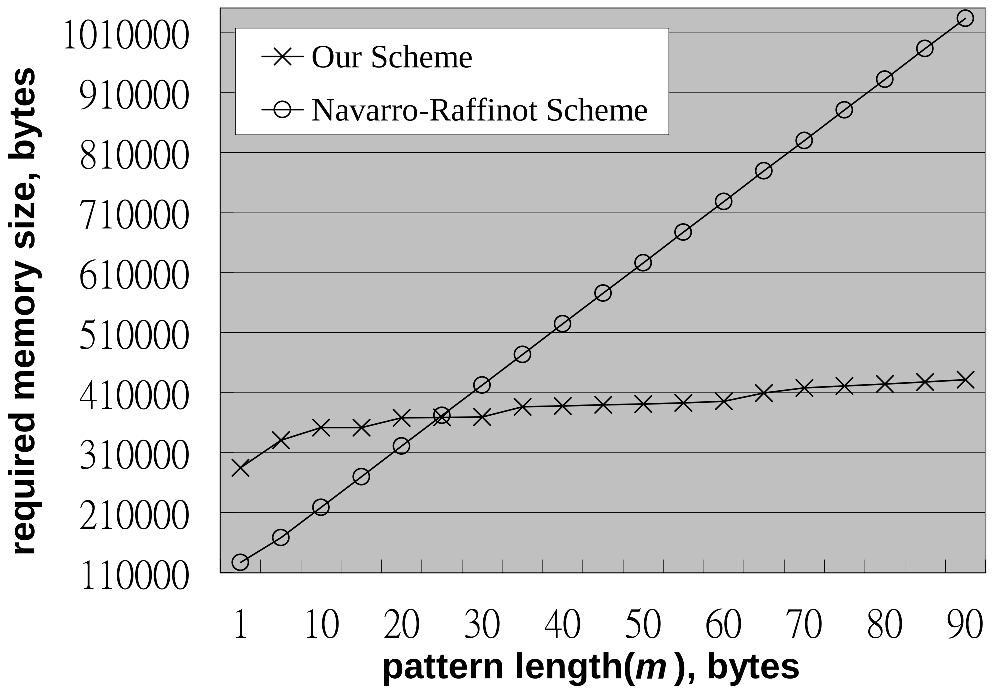

Figure 5-3. Comparison of space requirements for test case 3

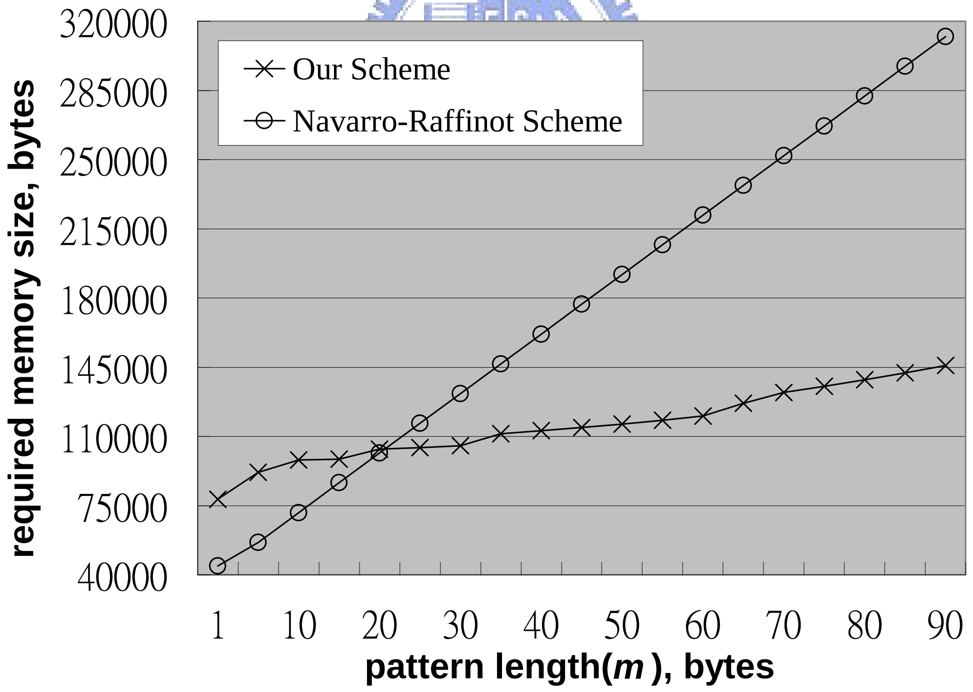

Figure 5-4. Comparison of space requirements for test case 4

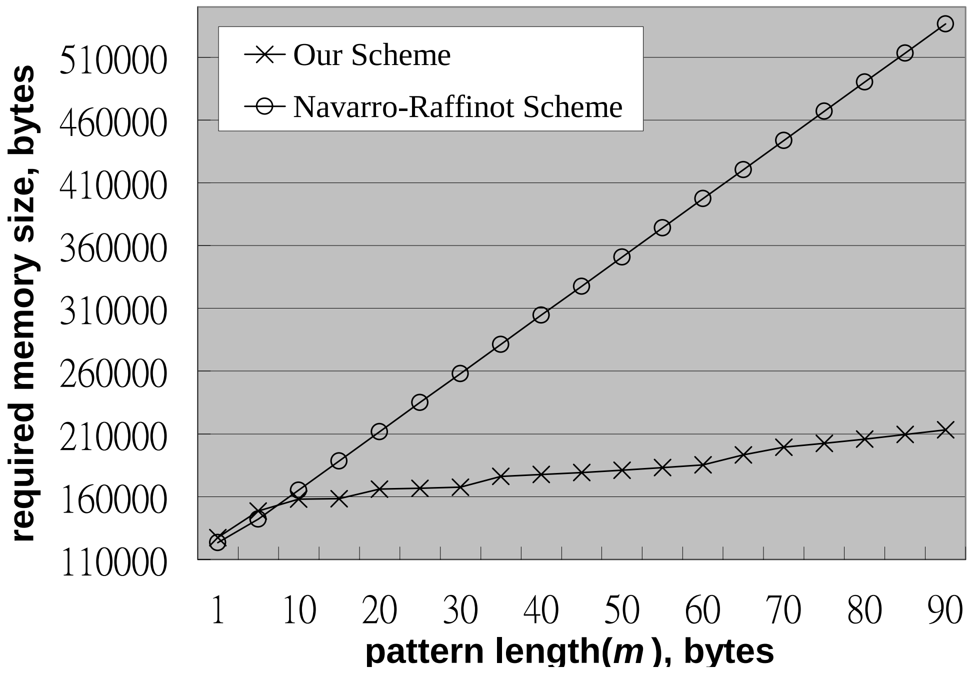

Figure 5-5. Comparison of space requirements for test case 5

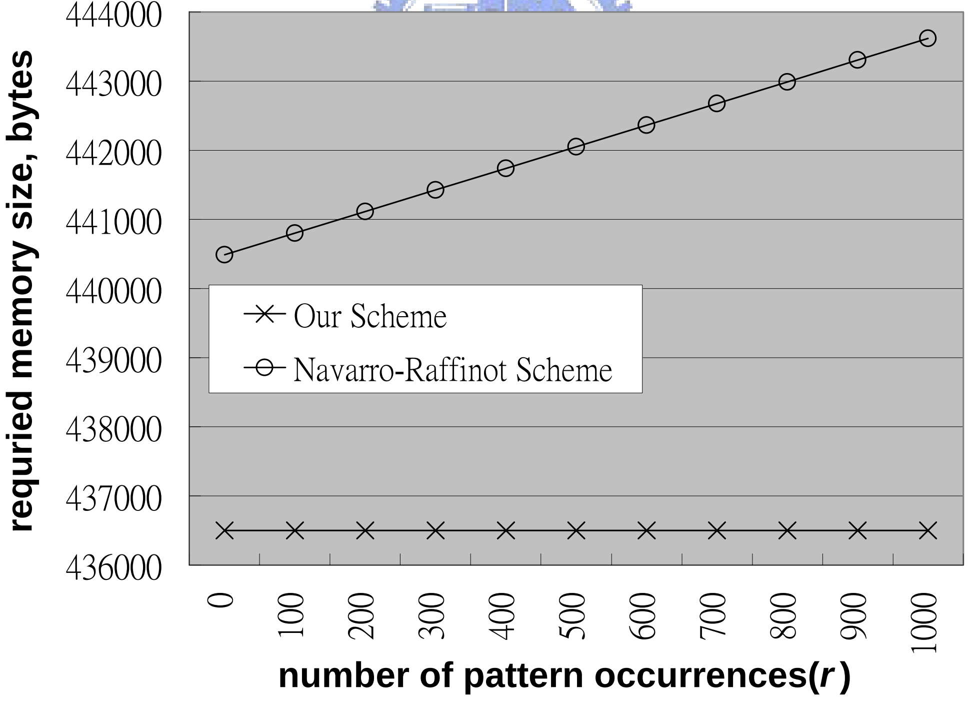

Figure 5-6. Comparison of space requirements for different number of pattern occurrences (*m* is fixed to be 25)

Chapter 6

Conclusion

We have presented in this thesis an efficient bitmap-based realization of the Amir-Benson-Farach algorithm for pattern search in LZW compressed sequences. The realization is then generalized to detect all pattern occurrences and report the absolute match positions. It is shown with experimental results that our proposed realization performs pattern search much faster than the decompress-then-search scheme. Moreover, compared with the Navarro-Raffinot scheme, another algorithm which can be realized with bitmaps, our proposed generalized algorithm requires less storage when the pattern is longer than 25 bytes. The difference could be huge if the number of pattern occurrences in the text is large. An interesting further research topic which is currently under investigation is to apply the idea of our design to pattern search with other compression techniques.

Bibliography

[1] Welch, T.A.: 'A technique for High-Performance Data Compression', June 1984, IEEE Computer, 17(6): 8-19

[2] Amir, A., Benson, G., and Farach, M.: 'Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files', April 1996, Journal of Computer and System Sciences, vol. 52, pp. 299-307

[3] Tao, T., and Mukherjee, A.: 'Pattern Matching in LZW Compressed Files', August 2005, IEEE Transactions on Computers, vol. 54, no. 8, pp. 929-938

[4] Ho, M.H., and Yen, H.C.: 'A Dictionary-based Compressed Pattern Matching Algorithm', IEEE Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, August 2002, pp. 873-878

[5] Navarro G., and Raffinot, M.: 'A General Practical Approach to Pattern Matching over Ziv-Lempel Compressed Text', Proc. 10th Ann. Symp. on Combinatorial Pattern Matching, Springer-Verlag, London, UK, 1999, Lecture Notes in Computer Science, vol. 1645, pp. 14-36

[6] Kida, T., Takeda, M., Shinohara, A., and Arikawa, S.: 'Shift-And Approach to Pattern Matching in LZW Compressed Text', Proc. 10th Ann. Symp. on

Combinatorial Pattern Matching, Springer-Verlag, London, UK, 1999, Lecture Notes in Computer Science, vol. 1645, pp. 1–13

[7] Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., and Arikawa, S.: ‘Multiple Pattern Matching in LZW Compressed Text’, 2000, J. Discrete Algorithms, vol. 1, no. 1, pp. 133-158

[8] Knuth, D.E., Morris, J.H., and Pratt, V.R.: **‘**Fast Pattern Matching in String**s’**, 1977, SIAM Journal on Computing, 6, (2), pp.323-350

簡歷

一、　個人資料

姓名	黃迺倫
性別	女
生日	1981/11/29
E-mail	nellen.cm93g@nctu.edu.tw
實驗室	交大電信所 網路技術實驗室 (工程四館 823 室)

二、　學歷

國小	桃園縣新明國民小學
國中	桃園縣私立復旦高級中學國中部
高中	桃園縣私立復旦高級中學高中部
大學	國立交通大學電信工程學系
研究所	國立交通大學電信工程學系 碩士班 系統組
碩士畢業去向	國立交通大學電信工程學系 博士班 系統組