Fast Full Text Search Using Tree Structured [TS] File

Takashi SATO
Dept. Arts and Sciences (Computer Science)
Osaka Kyoiku University
4-698-1 Asahigaoka, Kashiwara, 582 Japan
e-mail: sato@cs.osaka-kyoiku.ac.jp

Abstract. The author proposes a new data structure (TS-file) in order to make a fast search for an arbitrary string in a large full text stored in secondary storage. The TS-file stores the location of every string of length $L$ (the level) in the text. Using this, we can efficiently search for, not only strings of length $L$ but also those shorter than or longer than $L$. From an analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length $l_k$ is shorter than or equal to $L$, and $2(L - l_k + 1)$ otherwise. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS-file, a compressed TS-file is introduced in order to lower storage costs for practical use without losing search speed. The experimental results on compression using UNIX online manuals and network news show that the space overhead of the TS-file against the text searched is from 17% (when $L = 3$) to 212% (when $L = 12$) which is small enough for practical use.

Key words: data storage and indexing, gram based index, full text search, no false drop, TS-file

1 Introduction

The capability to search for strings which are not specified in advance is required more and more recently in the various ways of processing online data such as documents, articles, books, manuals, news, dictionaries and so on. When a text becomes huge, methods which search the full text directly[1-4] are not practical. So auxiliary data structures are used in order to speed up the search[5-8]. A signature file[9,10] is a typical data structure for such purposes and it is widely used in practical applications[11-13]. However, when we consider the recent status of secondary storage which is rapidly increasing in space per drive and decreasing in cost per bit, faster and more flexible string searches are needed more than those which require less space.

In this paper, we propose a new data structure called a TS-file (Tree Structured file) and a set of algorithms using this in order to make arbitrary string searches especially fast. In a previous paper, using a compressed data file we proposed an algorithm which is efficient when the length of the search string is rather long[14].
The method in this paper is most suitable when the length of the search string is rather short. The basic ideas of the TS-file is to store the location of every string of length \( L \) in the text. Using a TS-file, not only strings of length \( L \) but also those shorter than or longer than \( L \) can be searched efficiently.

A retrieval system using transposed files based on single characters, pairs of adjacent characters and longer strings of adjacent characters has been reported for searches of Japanese text\(^{[15]}\). Since the size of the Japanese character set is large, multiple data structures are provided for these combinations of character classes. This system is analogous to our method for each \( L = 1, 2, \ldots \), however, our method prepares only one data structure and it has a unique \( L \) value.

Since one can find arbitrary strings using the TS-file alone, the proposed method is more accurate than the one using signature files or PAT tree\(^{[7]}\) by which one can only know the possibility of existence. The proposed data structure is not made from a word by word index stored as an inverted index\(^{[16]}\). In other words, it does not depend on any specific language styles, for example, in which words are separated by blanks and so on. So, the proposed method is applicable to a wide variety of pattern matches which includes bit strings and genetic information.

From the analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length \( l_k \) is shorter than or equal to \( L \), and \( 2(L - l_k + 1) \) otherwise. This is far less than in the case of signature file search. The proposed algorithm is one of the fastest for arbitrary string searches. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS-file, we introduce a compressed TS-file by (1) making the data structure a tree to remove unused slots (null pointers), and (2) storing differences between adjacent elements if possible in order to lower storage costs for practical use without losing search speed.

Experiments using UNIX online manuals (up to 6.1Mbyte) and network news (up to 500Mbyte) as source text show that the searches are very fast and their time is less than 200msec in most cases and several hundred milli seconds even when keys have many matches or their length is much longer than the level. The overhead of storing the TS-file compared to text size is 30% when \( L = 4 \) and 66% when \( L = 6 \) for UNIX online manuals and 47% when \( L = 4 \) and 212% when \( L = 12 \) for network news. These overheads are small enough for practical use.

2 Definitions

The alphabet is denoted by \( \Sigma \). \( \sigma (=|\Sigma|) \) denotes the size of the alphabet. The TS-file stores the location of every string of a given length (called a gram) in the text. This length is called the level, \( L \). A string sought is a key, \( k \), whose length is \( l_k \). A key is constituted of characters \( c_i \). So a key is denoted by \( k = c_1c_2 \cdots c_{l_k} \ (c_i \in \Sigma) \). The length of the text searched is \( n \) characters. The text is assumed to be large compared with the size of main memory and is stored on secondary storage. Data are transferred to and from the main memory in blocks of size \( B \) words. The units of memory are word, half word and character. The number of characters per word is \( w \). In typical cases, 1 word = 4 byte and 1 character = 1 byte so that \( w = 4 \).