Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

String analysis by sliding positioning strategy

Manuel Baena-García [a,b,*], José M. Carmona-Cejudo [b], Rafael Morales-Bueno [b]

[a] Dpto. Informática, Clínica Rincón Bejar, 29740, Torre del Mar, Málaga, Spain
[b] Dpto. Lenguajes y Ciencias de la Computación, Universidad de Málaga, 29071, Málaga, Spain

A R T I C L E   I N F O

Article history:
Received 16 July 2012
Received in revised form 9 December 2012
Accepted 14 March 2013
Available online 19 March 2013

Keywords:
String mining
Frequent factors
Short tandem repeats
Trie data structures

A B S T R A C T

Discovering frequent factors from long strings is an important problem in many applications, such as biosequence mining. In classical approaches, the algorithms process a vast database of small strings. However, in this paper we analyze a small database of long strings. The main difference resides in the high number of patterns to analyze. To tackle the problem, we have developed a new algorithm for discovering frequent factors in long strings. We present an Apriori-like solution which exploits the fact that any super-pattern of a non-frequent pattern cannot be frequent. The SANSPOS algorithm does a multiple-pass, candidate generation and test approach. Multiple length patterns can be generated in a pass. This algorithm uses a new data structure to arrange nodes in a trie. A Positioning Matrix is defined as a new positioning strategy. By using Positioning Matrices, we can apply advanced prune heuristics in a trie with a minimal computational cost. The Positioning Matrices let us process strings including Short Tandem Repeats and calculate different interestingness measures efficiently. Furthermore, in our algorithm we apply parallelism to transverse different sections of the input strings concurrently, speeding up the resulting running time. The algorithm has been successfully used in natural language and biological sequence contexts.

1. Introduction

At this time of exponential growth of biological information such as nucleotide or protein databases, processing a vast quantity of sequences efficiently is of the utmost importance. Namely, data sequences have to be represented as long strings of symbols in a small alphabet. We need to understand hidden information in these strings. Then, we wish to discover knowledge from this information. One way to extract knowledge from data is to discover frequent or, in general, interesting patterns in strings [1].

The problem of pattern discovery in strings can be classified as a special case of the problem of pattern discovery in databases. Basic solutions are based on employing the Apriori property proposed in association mining [2] to generate the candidate patterns to be analyzed in a pass over the data, by using only the patterns in the previous pass. However, there is a fundamental difference between pattern discovery in strings and pattern discovery in databases. In the first, we have a vast database of small transactions over a big set of items. In the latter, on the contrary, we have a big string (the transaction) over a small set of symbols (the items). Due to this difference, the algorithms of pattern discovery in databases are not valid for pattern discovery in strings. In fact, the difficulty lies in the high number of patterns to analyze.

* Corresponding author at: Dpto. Informática, Clínica Rincón Bejar, 29740, Torre del Mar, Málaga, Spain.
E-mail addresses: manuelbaena@clinicasrincon.com (M. Baena-García), jmcarmona@lcc.uma.es (J.M. Carmona-Cejudo), rmorales@uma.es (R. Morales-Bueno).

0022-0000/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcss.2013.03.004

In particular, the problem of finding repeating substrings has been studied extensively in the field of string algorithms. However, our objective is not to find all patterns, but to discover the most interesting ones in the set of all possible patterns according to a given interestingness measure [3], without taking their positions into account.

In this paper, we present an Apriori-like solution called SANSPOS (String ANalysis by Sliding POsitioning Strategy), which exploits the fact that any super-pattern of a non-frequent pattern cannot be frequent. The method does a multiple-pass, candidate generation and test approach. The first scan finds all of the frequent items which form the set of single-symbol frequent patterns. Each subsequent pass starts with the set of patterns found in the previous pass. These previous patterns are used to generate new longer patterns. We show that the presented SANSPOS algorithm can be parallelized, which reduces the running time. This is done by transversing different substrings concurrently. The SANSPOS algorithm uses a variation of a trie structure called SP-Trie (Sliding Positioning trie), which is also defined. This trie represents all frequent factors of a given string. SP-Trie has a node organization and routing strategy based on a compressed representation of nodes. We define Positioning Matrices to locate a pattern in the trie. A sliding positioning strategy is used to insert new patterns into the trie. The proposed algorithm has been successfully used to compute interestingness measures of string factors in order to generate datasets used to validate a methodology for interestingness measure mining [4].

An important advantage of the SANSPOS algorithm is its capacity to process strings with short tandem repeats (STRs) efficiently. STRs are patterns of two or more symbols that are repeated a high number of times at adjacent positions. They represent an important source of complexity of algorithms for discovering frequent factors: we have previously shown [5] that the space complexity of discovering frequent factors in a string is quadratic in general, but can in certain cases be reduced to lineal by detecting STRs. The SANSPOS algorithm uses Positioning Matrices to detect and ignore this kind of patterns, which is an advantage from the computational complexity point of view over other algorithms.

The rest of this paper is organized as follows. In Section 2 we review and discuss previous work related to our proposal. In Section 3 we provide some basic notations and definitions that will be used later on. Section 4 is devoted to our proposed SP-Trie structure and its main properties, while Section 5 explains the SANSPOS algorithm for constructing SP-Tries. Then, the SANSPOS algorithm is empirically evaluated in Section 6. Finally, in Section 7 we provide our conclusions.

2. Related work

Works such as the ones by Fischer et al. [6,7] and Kügel and Ohlebusch [8], propose algorithmic solutions for discovering frequent factors in strings. But their approaches are based on a different definition of the problem: the authors consider a pattern as frequent if it appears in a minimal number of strings, no matter what the number of repetitions in each string is. Another algorithms for this problem are SMILE [9], Weeder [10] and Verbumculus [11].

Our approach, on the other hand, is based on the number of repetitions in a single, long string. A popular tool for performing pattern discovery in this sense are suffix trees [12,13]. A suffix tree is a data structure that represents the suffixes of a given string. It allows us to quickly access the factors of a string. The use of suffix trees for locating frequent factors was first proposed in the work of Sagot [14]. Arguably, the most influential algorithm for constructing suffix trees from strings is Ukkonen's algorithm [13]. Ukkonen's algorithm uses linear time for constructing suffix trees. His algorithm adds successive characters from the string until completing the tree, which means that it is an online algorithm. Parallel construction of suffix trees has been recently explored in the work of Mansour et al. [15]. Their proposed ERa algorithm optimizes dynamically the use of memory and amortizes the I/O cost. Their algorithm has a version for shared-memory and shared-nothing systems.

A suffix trees-based algorithm for frequent pattern discovery was proposed by Sætrom [16], who uses a genetic programming approach together with an ad-hoc high performance hardware for string searching able to predict rules that describe relations between consecutive patterns in the string. De Rooij [17,18] also uses Ukkonen algorithm to construct suffix tress, adding some contextual information to the tree nodes. De Rooij uses his proposed approach for data compression.

There is a problem with suffix tree usage for discovering interesting patterns: only the frequent part of the suffix tree will be used. There are unnecessary nodes in the structure. A different data structure for finding frequent patterns, which we use in our proposal, is the trie. A trie, introduced in the work of Fredkin [19], is an ordered tree structure where each edge represents a symbol. Each node is associated with a given pattern. In order to obtain the pattern corresponding to the node n, the symbols in the path from the root to n are concatenated. All the descendants of the same node n have the same prefix. The creation of a trie data structure involves choosing an appropriate node organization and routing strategy. Bodon, in his survey of frequent itemset mining [20], defines two ways of trie implementation: *compact representation* and *non-compact representation*. The compact representation uses an array of pointers to node structures. As the string to analyze is very long, the memory requirement of the compact representation is huge. This space requirement makes the structure unfeasible and inefficient. The non-compact representation uses a linked list or binary search tree to link the sons of a node. However, this solution saves space but with a higher search cost.

Vilo defines SPEXS [21], a new algorithm to avoid the problems associated with suffix trees. This algorithm generates a pattern trie with information summaries about the occurrences of each pattern. Nodes of frequent patterns are expanded incrementally to incorporate new patterns to the trie. This prune method guarantees that only frequent patterns are constructed and evaluated. In SPEXS, the node organization of the generated trie is not important. It is due to the fact that the algorithm only needs one access per node in the trie in order to calculate frequent factors in a string. SPEXS does not add every word to the tree but instead builds it on all possible words in breadth-first order. At every step, it checks if the