Fast Implementations of Automata Computations

Anne Bergeron and Sylvie Hamel

LACIM, Université du Québec à Montréal,
C.P. 8888 Succursale Centre-Ville, Montréal, Québec,
Canada, H3C 3P8,
anne@lacim.uqam.ca

Abstract. In [6], G. Myers describes a bit-vector algorithm to compute the edit distance between strings. The algorithm converts an input sequence to an output sequence in a parallel way, using bit operations readily available in processors.

In this paper, we generalize the technique, and characterize a class of automata for which there exists equivalent parallel, or vector, algorithms. As an application, we extend Myers result to arbitrary weighted edit distances, which are currently used to explore the vast data-bases generated by genetic sequencing.

1 Introduction

Finite automaton are powerful devices for computing on sequences of characters. Among the finest examples, very elegant linear algorithms have been developed for the string matching problem [1]. Automata are also widely used in fields such as metric lexical analysis [3] or bio-computing, where approximate string matching is at the core of most algorithms that deal with genetic sequences [4]. In these fields, the huge amount of data to be processed – sometimes billions of characters – calls for algorithms that are better than linear.

One way to accelerate the computations is to exploit the parallelism of vector operations, especially bit-vector operations. For example, in [2] and [5], bit-vectors are used to code the set of states of a non-deterministic automaton. In this paper, as in [6], we want to accelerate computations done with deterministic automata, and we use vectors to represent sequences of events or sequences of states.

Given a deterministic finite automaton, and an input sequence $x_1 \ldots x_m$, we are interested in the output sequence $y_1 \ldots y_m$ of visited states. Since executing one transition is usually considered to be a constant time operation, the output sequence can be obtained in $O(m)$ time.

In order to improve the efficiency of such algorithms, we have to find quicker ways to obtain $y_1 \ldots y_m$ from $x_1 \ldots x_m$. The best possible algorithm would do it in constant time:
Clearly, this can seldom be done when \( m \) is unbounded. The next best alternative would be to obtain \( y_1 \ldots y_m \) with a bounded number of vector operations on \( x_1 \ldots x_m \):

\[
x_1 x_2 \ldots x_m \\
\downarrow \quad \downarrow \quad \downarrow \\
y_1 y_2 \ldots y_m
\]

Indeed, vector operations can be implemented in parallel, in dedicated circuits, or using high-speed bit-wise operations available in processors. The drawback is that vector operations are applied component by component, meaning that the only computations that one could hope to solve with pure vector operations are those where the value of \( y_i \) depends only on the value of \( x_i \), and its close neighbors.

On the bright side, some bit operations widely available in processors do have a memory of past events. Using these, it is possible to parallelize complex computations done with automata.

### 2 The Basics of Vector Algorithms

As a simple example, consider the following automaton. On input sequence \( babba \), it will generate the output \( s_1 s_0 s_1 s_2 s_0 \) — we omit the leading initial state.

Let’s associate to a string \( x_1 \ldots x_m \), the characteristic vector of a letter \( l \), denoted by the bold letter \( \mathbf{l} \), and defined by:

\[
l_i = \begin{cases} 
1 & \text{if } x_i = l \\
0 & \text{otherwise}
\end{cases}
\]

Characteristic vectors are sequences of bits, and we will operate on them with the standard bit operations: bit-wise logical operators, left and right shifts, binary addition, etc. We can obtain, for example, the characteristic vector of a set \( S \) of letters by computing the disjunction of the characteristic vectors of the letters in \( S \).