WSEAS TRANS. on INFORMATION SCIENCE & APPLICATIONS, Issue 1, Vol. 1, July 2004 ISSN: 1790-0832

Computing the Hamming Distance of a Regular Language in Quadratic Time*

L. KARI[1], S. KONSTANTINIDIS[2,+], S. PERRON[3], G. WOZNIAK[1], J. XU[2]

[1]Department of Computer Science
University of Western Ontario
London, Ontario, N6A 5B7
CANADA
lila@csd.uwo.ca, wozniak@csd.uwo.ca

[2]Department of Mathematics and Computing Science
Saint Mary's University
Halifax, Nova Scotia, B3H 3C3
CANADA
s.konstantinidis@smu.ca, j_xu@cs.stmarys.ca

[3]Department of Computer Science
University of Toronto
10 King's College Road, Toronto, Ontario, M5S 3G4
CANADA
steven_perron@hotmail.com

*Work supported by the Natural Sciences and Engineering Research Council of Canada
+Corresponding author

Abstract: - Given an arbitrary (nondeterministic) finite state automaton A we consider the problem of computing the Hamming distance of the language $L(A)$ that is accepted by the automaton. This quantity is simply the minimum Hamming distance between any pair of distinct words from the language $L(A)$. We show an algorithm that solves the problem in time quadratic with respect to the size of the given automaton. To our knowledge, this problem has not been addressed before.

Key-Words: - automata, Hamming distance, regular language, shortest-path algorithm.

1. Introduction

The problem of measuring the difference between words (strings) and languages (sets of words) is important in various applications of information processing such as error control in data communications, bio-informatics, and spelling correction. Well-known measures of the difference between two words are the edit (or Levenshtein) distance and the Hamming distance – this is used in the theory of error correcting codes [6], [1]. Typical problems pertaining to differences between strings and languages are (i) computing the distance (also known as self-distance) of a given language, (ii) computing the edit-distance between two words, and (iii) correct a given word to a word of a given language using a minimum cost string of edit operations.

The first problem can be used to find the maximum number of errors that a given code can detect. To our knowledge, it has not been addressed for the case of regular languages. The second problem can be solved using a dynamic

programming algorithm - see [8]. The third problem is solved in [9] for regular languages and unrestricted edit operations, and more generally and efficiently in [3].

In this work, we address the first problem for the well-known case of Hamming distance. In the next section we provide the basic notation about automata. In Section 3, we introduce the sigma automaton of a given automaton that, loosely speaking, represents all pairs of strings that are different and belong to the language accepted by the given automaton. The size of the sigma automaton is quadratic with respect to the size of the given automaton. Section 4 shows a quadratic time algorithm for computing the Hamming distance of a given regular language. The algorithm operates in linear time on the sigma automaton corresponding to the given automaton. Finally, Section 5 contains a few concluding remarks.

2. Basic Notions and Notation

An *alphabet* is a finite nonempty set of symbols. In the sequel we shall use a fixed alphabet Σ. A *word* or *string* (over Σ) is a finite sequence $a_1 \cdots a_n$ such that each a_i is in Σ. The set of all words over Σ is denoted by Σ^*. A *language* is any set of words, that is, any subset of Σ^*. The *Hamming distance* $H(u, v)$ between two words u and v of the same length is the number of corresponding positions in which the words u and v differ. If the words u and v are of different lengths then we agree that $H(u, v) = \infty$. For example, $H(aabbcc, abbbac) = 2$ and $H(ab, aba) = \infty$. The Hamming distance $H(L)$ of a language L is the smallest quantity $H(u, v)$ over all pairs (u, v) of words in L with $u \neq v$.

A (nondeterministic finite) automaton, an *NFA* for short, is a quintuple $A = (\Sigma, Q, s, F, T)$ such that Q is a finite nonempty set, the set of states, s is the start state, F is the set of final states, and T is the set of transitions. Each transition in T is of the form $q_1 x q_2$, where q_1 and q_2 are states and x is an alphabet symbol in Σ - we assume that the sets Q and Σ are disjoint. A *computation* of A is an expression of the form $q_0 x_1 q_1 \cdots x_n q_n$ such that each $q_{i-1} x_i q_i$ is a transition in T. We say that such a computation is *accepting* if q_0 is the start state and q_n is a final state. In this case, $x_1 \cdots x_n$ is called the accepted word. We use the notation $L(A)$ for the *language accepted by* A, that is, the set of all accepted words of A. The *size* $|A|$ of the automaton A is the quantity $|Q|+|T|$. An NFA is said to be *trim* if every state is reachable from the start state and can reach a final state. Note that in every trim NFA we have that $|Q| \leq |T| + 1$ and, therefore, the size of A is $O(|T|)$. For further information on automata and languages we refer the reader to [7] - see also [4] for information on the size of automata in constructions involving these objects.

3. The sigma NFA

In this section we introduce the NFA A^σ of a given NFA A. Recall - see for instance [10] - that for any two NFAs A_1 and A_2 one can use the standard product construction to define the trim NFA $A_1 \cap A_2$ of size $O(|A_1||A_2|)$ accepting the language $L(A_1) \cap L(A_2)$. The construction of the NFA A^σ can be viewed as a generalization of the standard product construction. We note that an interesting product construction between two copies of the same automaton is defined in [2] for the purpose of deciding the property of unique decodability for regular languages. Although the topic of [2] is not relevant to the present work, we wish to acknowledge that our product construction was inspired in part by the product construction in [2].

Let E be the set consisting of all expressions x/y, where x and y are in Σ, that is, alphabet symbols. We consider E to be an alphabet and, therefore, we can define strings and languages over E as usual. We call the strings over E as *e-strings* and we write $(x_1/y_1) \cdots (x_n/y_n)$ for an arbitrary e-string.

Construction 1. Given an NFA A, the NFA A^σ is defined as follows.

1. Firstly, define the NFA A_1 that is constructed as follows. The states of A_1 are all the pairs of the form (p, q), where p and q are states of A. The start state of A_1 is the pair (s, s), where s is the start state of A, and the set of final states of A_1

consists of all pairs (p, q) such that p and q are final states of A. The transitions of A_1 are of the form $(p_1, q_1)(x/y)(p_2, q_2)$ such that $p_1 x p_2$ and $q_1 y q_2$ are any two transitions of A.

2. Secondly, define the NFA A_2 that is constructed as follows. Let B be the NFA that has two states s and g, with s being the start state and g being the only final state, and transitions

$$s(x/x)s,\ s(x/y)g,\ g(x/x)g,\ \text{and}\ g(x/y)g$$

for all alphabet symbols $x, y \in \Sigma$ with $x \neq y$. It is evident that B accepts all e-strings in E^* containing at least one element x/y with $x \neq y$. Then, A_2 is defined to be the NFA $A_1 \cap B$.

3. Finally, the NFA A^σ is the trim part of the NFA A_2.

Theorem 1: Given any NFA A, the NFA A^σ is of size $O(|A|^2)$ and accepts the language of all the e-strings $(x_1/y_1)\cdots(x_n/y_n)$ such that the words $x_1\cdots x_n$ and $y_1\cdots y_n$ are different and belong to $L(A)$.

Proof: It is sufficient to show that the NFA A_1 of Construction 1 accepts all e-strings $(x_1/y_1)\cdots(x_n/y_n)$ such that both of the words $x_1\cdots x_n$ and $y_1\cdots y_n$ are in $L(A)$. First let $h = (x_1/y_1)\cdots(x_n/y_n)$ be an e-string accepted by some computation

$$(p_0, q_0)(x_1/y_1)(p_1, q_1)\cdots(x_n/y_n)(p_n, q_n)$$

of A^σ. By Construction 1, h is in E^* and the expressions $p_0x_1p_1\cdots x_np_n$ and $q_0y_1q_1\cdots y_nq_n$ are accepting computations of A. Hence, both of the words $x_1\cdots x_n$ and $y_1\cdots y_n$ belong to $L(A)$.

Conversely, suppose $h = (x_1/y_1)\cdots(x_n/y_n)$ is an e-string in E^* such that both of the words $x_1\cdots x_n$ and $y_1\cdots y_n$ belong to $L(A)$. Then one can define two accepting computations of A of the form

$$p_0x_1p_1\cdots x_np_n \quad \text{and} \quad q_0y_1q_1\cdots y_nq_n.$$

This implies that

$$(p_0, q_0)(x_1/y_1)(p_1, q_1)\cdots(x_n/y_n)(p_n, q_n)$$

is an accepting computation of A^σ and, therefore, h must be in $L(A^\sigma)$. □

4. Computing the distance

In this section we obtain the following result.

Theorem 2: The following problem is computable in quadratic time.

Input: An NFA A.

Output: The Hamming distance of $L(A)$.

The NFA A^σ can be viewed as a labelled directed graph and can be modified conceptually by replacing the label x/y of every edge $p(x/y)q$ with the label 0 if $x = y$, or 1 if $x \neq y$. It should be evident now that the Hamming distance of the language $L(A)$ is equal to the smallest weight of a path from the start state to a final state in the directed weighted graph A^σ. It is well-known that this value can be computed in time $O(n \log n)$, where n is the size of the graph, using Dijkstra's algorithm – see [5], for instance. Note that the factor $\log n$ in the time complexity is due to the fact that A might contain cycles. However, using the fact that each weight in the graph A^σ is either 0 or 1, we show next that the shortest accepting path can be computed in time linear with respect to the size $O(|A|^2)$ of A^σ, even when this graph contains cycles. This would also establish the validity of Theorem 2.

Consider a directed graph G all the nodes of which are reachable from a start node s, and all the weights on the edges of the graph are 0 or 1. We can view G as two graphs G_0 and G_1 such that G_0 results by removing from G the edges of weight 1 and G_1 results by removing from G the edges of weight 0. The main idea of the algorithm is as follows: Let $Q_0^{(0)}$ be equal to s. Define $Q_1^{(0)}$ to be $Q_0^{(0)}$ union the the set of all new nodes (i.e., nodes that have not been visited before) that are reachable from $Q_0^{(0)}$ via the graph G_0. The nodes in $Q_1^{(0)}$ are exactly those of distance 0 from s. Let $Q_0^{(1)}$ be the set

of new nodes that are reachable from $Q_1^{(0)}$ using exactly one edge of G_1. Each node in $Q_0^{(1)}$ is of distance 1 from s. This process is repeated by defining $Q_1^{(i)}$ from $Q_0^{(i)}$, and $Q_0^{(i+1)}$ from $Q_1^{(i)}$, until all nodes in G have been visited. It is evident that the nodes in $Q_1^{(i)}$ are exactly those of distance i from s.

We turn the above idea to an algorithm by using two queues Q0 and Q1, a counter length that plays the role of i, an array Seen to keep track of whether a node has been visited, and an array Distance to store the distance of each node from the start node.

Algorithm.

```
Define two empty queues Q0 and Q1
Initialize all entries of the boolean
   array Seen to false
Initialize all entries of the integer
   array Distance to 0
Q0.insert(startNode);
Seen[startNode] = true;
length = 0;

while (Q0 is not empty)

   while (Q0 is not empty)
      a = Q0.front()
      for each edge (a,b) in G0 with
      not Seen[b]
         Q0.insert(b), Seen[b] = true;
         Distance[b] = length
      end for
      Q0.delete(), Q1.insert(a)
   end while

   length = length + 1

   while (Q1 is not empty)
      a = Q1.front()
      for each edge (a,b) in G1 with
      not Seen[b]
         Q0.insert(b), Seen[b] = true;
         Distance[b] = length
      end for
      Q1.delete()
   end while

end while
```

As each node of the graph G can be examined no more than two times, the above algorithm runs in time proportional to the size of the graph.

5. Concluding Remarks

For the problem of computing the Hamming distance of a given regular language, we were able to give a fast algorithm by constructing the sigma automaton and taking advantage of the fact that one can compute in linear time shortest paths in a directed graph when the weights involved are zero and one. On the other hand the problem of computing the edit-distance of a given regular language in polynomial time remains open.

References:

[1] M. A. Armand: Efficient decoding of Reed-Solomon codes over z_q based on remainder polynomials. *WSEAS Transactions on Communications*, 1 (2002), 116–121.

[2] T. Head, A. Weber: Deciding code related properties by means of finite transducers. In *Sequences II, Methods in Communication, Security, and Computer Science*. 260–272, 1993.

[3] L. Kari, S. Konstantinidis, S. Perron, G. Wozniak, J. Xu. Finite-state error/edit-systems and difference-measures for languages and words, 2003. Technical Report 2003-01, Department of Mathematics and Computing Science, Saint Mary's University, Canada. Available at www.smu.ca/academic/science/compsci/.

[4] S. Konstantinidis: Some remarks on regular factorizations. *WSEAS Transactions on Communications*, 1 (2002), 167–172.

[5] U. Manber: *Introduction to Algorithms, A Creative Approach*. Addison-Wesley Publishing Company, 1989.

[6] S. Roman: *Coding and Information Theory*. New York, 1992.

[7] G. Rozenberg, A. S. (eds): *Handbook of Formal Languages, Vol. I*. Springer-Verlag, 1997.

[8] D. Sankoff, J. K. (eds): *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. CSLI Publications, 1999.

[9] R. A. Wagner: Order-n correction for regular languages. *Communications of the ACM*, 17 (1974), 265–268.

[10] S. Yu: Regular languages. 41–110. In [7].