BREAKING SUBSTITUTION CYPHERS USING STOCHASTIC AUTOMATA

B.J. Oommen and J.R. Zgierski

SCS-TR-182, OCTOBER 1990

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

BREAKING SUBSTITUTION CYPHERS USING STOCHASTIC AUTOMATA*

B. J. Oommen and **J. R. Zgierski**
School of Computer Science
Carleton University
Ottawa : ONT : K1S 5B6
CANADA

ABSTRACT

Let Λ be a finite plaintext alphabet, and V be a cypher alphabet with the same cardinality as Λ. In all one-to-one substitution cyphers there exists the property that each element in Λ maps onto exactly one element in V. This mapping of V onto Λ is represented by a function T^* which maps any $v \in V$ onto some $\lambda \in \Lambda$ (i.e., $T^*(v) = \lambda$). In this paper we consider the problem of learning the mapping T^* (or its inverse $(T^*)^{-1}$) by processing a sequence of cypher text. Traditional methods which break the cypher utilize the statistical information contained in the unigrams, and bigrams of the language from which the plaintext has been derived. A fast and more elegant method which uses trigrams of the language from which the plaintext has been derived is the one due to Peleg *et. al.* which utilizes the concept of relaxation [8,9]. In this paper we present a new learning automaton approach to break the cypher. A new finite state learning machine called the Cypher Learning Automaton (CLA) has been proposed which sequentially processes the cyphertext and a finite dictionary which is used as a model for the language from which the plaintext has been derived. This method is fast and the advantages of the scheme in terms of time and space requirements over the relaxation method [8,9] have been listed. The paper contains various simulation results comparing both the cypher breaking techniques.

* Partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada. The work of the second author was supported by an NSERC Summer Grant.

I. INTRODUCTION

The art of cryptography has very probably existed ever since writing was invented. The purpose of it was, of course, to hide a message by systematically transforming the original message into a form which is unintelligible to the reader unless it is first decyphered. In order to prevent unauthorized readers (or eavesdroppers) from decyphering the original message the strategy with which the transformed message is decyphered is maintained as a well-kept secret between the sender and the intended receiver. Hopefully, in this way the communication between the authentic sender and receiver cannot be intercepted by others for whom it is not meant. From the point of view of the eavesdropper, the heart of the problem is to break the cypher without a knowledge of the decyphering process, but this is usually not an easy task .

This paper concerns the breaking of substitution cyphers using Learning Automata (LA).

Throughout this paper we shall use the word plaintext to refer to the original message and the term cyphertext to refer to the transformed message. Also the verbs cypher, encrypt and encode will be used interchangeably. Similarly, the words decrypt, decypher and decode will be used interchangeably.

Perhaps the simplest cypher is the Caesar Cypher named after its inventor, the Roman Emperor Julius Caesar. In it, every letter of the plaintext is transformed to the letter three positions after it in the alphabet, and a wraparound rule is used for the last three letters of the alphabet. Thus, if the message 'veni vidi vici' was to be sent using the English alphabet, Caesar would transmit it as 'yhql ylgl ylfl'. Clearly, this message would be mysterious to a person unacquainted with the decyphering process.

Cyphers which use a direct one-to-one mapping are known as mono-alphabetic substitution cyphers. By mono-alphabetic we mean that only one alphabet is used in encyphering a message, and this is referred to as the cypher alphabet. In general, if Λ is a finite plaintext alphabet, and $\mathbf{V}$ is a cypher alphabet with the same cardinality as Λ, a substitution cypher may use any of the $|\Lambda|!$ permutations as its key. Clearly, finding the correct key to decypher such an encoded message is an extremely tedious task if one merely randomly tries all the different mappings. Even with a computer doing the job, this brute force method of decyphering the message is both infeasible and impractical, and this has prompted the study of using various other approaches.

Cryptographers who break cyphers frequently make use of the knowledge of the language from which the plaintext has been derived. Thus, for example, the cryptographer uses information about the relative frequencies of the various letters of the plaintext alphabet because, in most cases, some letters in an alphabet occur more frequently than others. Thus, for example, in the English language, the letters 'e' and 'z' can be easily distinguished in a substitution cypher if this information is utilized. This information is usually referred to as the unigram statistics of the

language from which the plaintext has been derived. Indeed, the unigram statistics are estimates of the probabilities of the occurrence of the individual symbols of the plaintext alphabet. More frequently, however, additional information is also used to aid in the decyphering process. Another piece of information that can be used to aid in the decyphering is the fact that certain letter combinations are more frequent (or nonexistent) in the plaintext language. Thus, for example, the decyphering process for the English language would utilize the information that whereas the suffix 'tion' is very common in English the suffix 'zyx' is non-existent. This type of information is typically referred to as the n-gram information of the language of the plaintext. Specifically, the trigram statistics are similar to the unigram statistics except that they represent the estimates of the probability of various three-lettered strings occurring in the plaintext language. Observe that the bigram statistics (estimates of the probability of various two-lettered strings in the plaintext language) can be derived from the trigram statistics.

The most elegant cypher breaking algorithm which utilizes the n-gram information is the one due to Peleg and Rosenfeld [9]. The algorithm utilizes both the unigram and trigram statistics of the language in a probabilistic relaxation scheme. Since the unigrams and trigrams are probabilities Peleg and Rosenfeld [9] use the comprehensive laws of probability to yield a crypt breaking mechanism with enhanced convergence properties. The specific algorithm used by Peleg and Rosenfeld [9] will be explained in greater detail in a subsequent section.

In this paper we shall propose an alternate method to break substitution cyphers. First of all we shall avoid the use of higher order n-grams of the plaintext language. Thus, instead of probabilistically representing the plaintext language we rather opt to model it using a finite dictionary containing representative words of the language. Various dictionaries of this sort have been compiled for the English language, the most well known being the one due to Dewey [1] comprising of about 1000 words which represent 80%-90% of the words most used in English. Utilizing the information contained in this finite dictionary we approach the crypt breaking problem from a completely different perspective. A new finite state learning machine called the Cypher Learning Automaton (CLA) has been proposed which sequentially processes the cyphertext and the plaintext finite dictionary and attempts to learn the key used by the cypher. The method proposed is fast and the advantages of the scheme in terms of time and space requirements over the relaxation method [9] have been presented in the paper. The paper also contains various simulation results comparing both the cypher breaking techniques.

I.1 Problem Statement and Notation

As was already mentioned earlier the problem at hand is that of successfully breaking a one-to-one substitution cypher. Let Λ be a finite plaintext alphabet, and $\mathbf{V}$ be a cypher alphabet. The cardinality of Λ is the same as that of $\mathbf{V}$. The key to the cypher is the mapping function T^*. Thus, during the cyphering process a given symbol $\lambda \in \Lambda$ is transmitted as a unique symbol $v \in \mathbf{V}$, where $(T^*)^{-1}(\lambda) = v$ (or equivalently, $T^*(v) = \lambda$). It is assumed that T^* (and thus its inverse) are known to the sender and the legal receiver, but that it is not known to the algorithm which attempts to break it.

Throughout this paper we assume that the algorithm has access to a "small" finite dictionary $\mathbf{H}$, representative of the language from which the plaintext has been derived. Clearly, $\mathbf{H}$ is a subset of Λ^*, where Λ^* is the set of strings over Λ. Typically, $\mathbf{H}$ is a dictionary that comprises of the most common words used in the plaintext language such as the one compiled by Dewey [1]. Let $\mathbf{H}^{-T^*}$ be the set of strings in $\mathbf{V}^*$ obtained by encyphering the individual words of $\mathbf{H}$ using the key T^*. We assume that the crypt breaking algorithm is presented with a large number of words from $\mathbf{V}^*$ each separated by an appropriate delimeter which is typically the 'Space' character, a piece of numeric data or a punctuation mark[+]. Clearly, since $\mathbf{H}$ is but a subset of Λ^*, it is quite likely that the crypt breaking algorithm will encounter words from $\mathbf{V}^*$ which are not elements of $\mathbf{H}^{-T^*}$.

In this paper we shall attempt to learn the mapping T^* (or equivalently its inverse $(T^*)^{-1}$). In the attempt to do this we assume that at time 'n' an estimate, T(n), of T^* is used to decypher a word or segment of the cypher text. Based on the result of the decyphering process this estimate is updated to yield the estimate T(n+1) for the next time instant. The aim is that as a result of the learning process T(n) converges to T^* as n tends to infinity.

Throughout this paper we shall assume that both Λ and $\mathbf{V}$ are the English alphabet. However, the techniques presented here are also valid for other general alphabets.

II. PREVIOUS SOLUTIONS TO CYPHER BREAKING

There exist various solutions for breaking one-to-one substitution cyphers [12]. Of all these, probably the most elegant solution is the relaxation method developed by Peleg *et. al.* [9]. Since our results have been quantitatively compared to those obtained by the latter algorithm we shall briefly describe this algorithm in this section.

[+] Note that a trigram (n-gram) distribution obtained which ignores these spaces and punctuation marks would be significantly different from the trigram (n-gram) distribution that we want. Observe that without spaces the message "This is an example" would give rise to many extra three letter sequences and thus alter the trigram distribution. These sequences would, of course, be: 'isi', 'sis', 'isa', 'san', 'ane', and 'nex'. Some of these sequences may never occur in the language from which the original plaintext is derived.

With every cypher letter $v \in V$, the relaxation algorithm associates a vector of probabilities $P_v=[P_{v1}, P_{v2}, \ldots P_{v|\Lambda|}]$. This vector constitutes the probabilities of the cypher letter v actually representing a letter $\lambda \in \Lambda$. Using the unigram and trigram information these probability vectors P_v are repeatedly updated using the principle of relaxation so as to yield finer approximations of the cypher mapping. Thus the authors of [9] have developed a means by which one can calculate the probability distribution vectors $P_v(n+1)$ at time (n+1) from the probability distribution vectors $P_v(n)$ defined at time 'n' for all $v \in V$. We define the following quantities* . Let:

(i) $P_a(\alpha)$ be the probability $\text{Prob}(T^*(a) = \alpha \mid P_a)$ that a cypher letter $a \in V$ represents the plaintext letter $\alpha \in \Lambda$, given that its probability distribution vector is P_a.

(ii) $Q_a(\alpha)$ be the probability $\text{Prob}(P_a \mid T^*(a) = \alpha)$ that the distribution vector for the cypher letter $a \in V$ is P_a, given that the true letter represented by $a \in V$ is $\alpha \in \Lambda$.

(iii) u_α be the be the unigram probability of α, $\text{Prob}(\alpha)$, where $\alpha \in \Lambda$. Observe that $\text{Prob}(\alpha) = \text{Prob}(T^*(a) = \alpha)$.

(iv) $t_{\alpha\beta\gamma}$ be the trigram probability that the sequence of three consecutive letters from the plaintext is $\alpha\beta\gamma$. Analogous to (iii) above, this trigram probability may be written as $\text{Prob}(\alpha\beta\gamma) = \text{Prob}(T^*(a) = \alpha, T^*(b) = \beta, T^*(c) = \gamma)$

Using the above definitions, through the application of Bayes rule an expression for $P_a(\alpha)$ may be obtained as :

$$P_a(\alpha) = \frac{Q_\alpha(a)\, u_\alpha}{\text{Prob}(P_a)}$$

Consider the joint distributions of

(i) three consecutive characters in any string from Λ^* being the substring $\alpha\beta\gamma$, and

(ii) (P_a, P_b, P_c).

This distribution, defined as $P_{abc}(\alpha\beta\gamma)$ can now be written down as :

$$P_{abc}(\alpha\beta\gamma) = \text{Prob}(T^*(a) = \alpha, T^*(b) = \beta, T^*(c) = \gamma \mid P_a, P_b, P_c).$$

It can be seen that this distribution has the form :

$$P_{abc}(\alpha\beta\gamma) = \frac{\text{Prob}(P_a, P_b, P_c \mid T^*(a) = \alpha, T^*(b) = \beta, T^*(c) = \gamma)\, t_{\alpha\beta\gamma}}{\text{Prob}(P_a, P_b, P_c)} \quad (1)$$

Under the assumption that P_a is dependent only on a, (i.e., P_a is independent of P_b, for all $b \neq a$) the authors of [9] have shown that this distribution has the form :

---

* Throughout this paper we have deviated from the notation used in [9] primarily because we believe that this cuurent notation is more consistent than that which involves the extensive use of subscripts.

$$P_{abc}(\alpha) = \frac{P_a(\alpha) \sum_{\beta \in \Lambda} \sum_{\gamma \in \Lambda} P_b(\beta) P_c(\gamma) r_{\alpha\beta\gamma}}{\sum_{\lambda \in \Lambda} P_a(\lambda) \sum_{\beta \in \Lambda} \sum_{\gamma \in \Lambda} P_b(\beta) P_c(\gamma) r_{\alpha\beta\gamma}} \tag{2}$$

where

$$r_{\alpha\beta\gamma} = \frac{t_{\alpha\beta\gamma}}{u_\alpha u_\beta u_\gamma}$$

The complete step-by-step proof of (2) is included in [9].

The next phase of computations in the relaxation algorithm involves the recursive (or rather, the iterative) expression for $P_a^{(n+1)}$. The final expressions for the recursive form of $P_a^{(n+1)}$ are derived in [8,9], but for the sake of brevity we shall merely state the final results in this paper. Two recursive expressions for the quantity $P_a^{(n+1)}$ have been derived. The first of these is called the Average Updating Method (AUM). It involves the averages of all the estimates of the various quantities $P_a^{(n+1)}$ obtained by processing the information contained in the vectors associated with *its* neighbouring characters. This means that in the AUM, only those P_{abc}'s are used in averaging for which the cypher letter sequence abc occurs in the cypher text. Thus,

$$P_a^{(n+1)}(\alpha) = \text{Average} \{ P_{abc}^{(n+1)}(\alpha) \} \tag{3}$$

where $P_{abc}^{(n+1)}(\alpha)$ is defined by expression (2) above, in which all quantities on the right hand side are values at time 'n'.

In an attempt to simplify (2) we define :

$$Q_{abc}^{(n)}(\alpha) = \sum_{\beta \in \Lambda} \sum_{\gamma \in \Lambda} P_b^{(n)}(\beta) P_c^{(n)}(\gamma) r_{\alpha\beta\gamma}$$

whence the recursive form form for $P_a^{(n+1)}$ can be shown to be :

$$P_{abc}^{(n+1)}(\alpha) = \frac{P_a^{(n)}(\alpha) Q_{abc}^{(n)}(\alpha)}{\sum_{\beta \in \Lambda} P_a^{(n)}(\beta) Q_{abc}^{(n)}(\beta)}$$

The second formula for computing $P_a^{(n+1)}$ involves an alternate expression derived using the Product Updating Method (PUM). The explicit form of the PUM was developed in [3,9] and it considers the updating as being done pair by pair. This formula is given by (4) :

$$P_a^{(n+1)}(\alpha) = \frac{P_a^{(n)}(\alpha) \prod_b \prod_c Q_{abc}^{(n)}(\alpha)}{\sum_{\beta \in \Lambda} P_a^{(n)}(\beta) \prod_b \prod_c Q_{abc}^{(n)}(\beta)} \quad (4)$$

In (4) above, the products are again only over those b's and c's where abc is a three letter sequence in the cypher.

In order to obtain initial values for the P_a vectors (i.e., for the probability of cypher letter v being the true letter λ), an initial estimate was derived (See [9]) to be of the form given by (5) below :

$$P_v^{(0)}(\lambda) = \frac{[u_\lambda]^k [1 - u_\lambda]^{n-k}}{\sum_{\alpha \in \Lambda} [u_\alpha]^k [1 - u_\alpha]^{n-k}} \quad (5)$$

where u_λ is the unigram of λ, n is the total length of the cypher ignoring the delimiters, and k is the number of times that a particular cypher letter appears in the cypher text.

The relaxation algorithm essentially involves computing the trigram statistics of the cypher text. The algorithm then repeatedly updates the P_a vectors until the vectors "converge" to a desired degree of accuracy. Every iteration of the algorithm has a time complexity of the order $O(|V|^5)$ if the size of the cypher alphabet and the true alphabet is $|V|$. The memory requirements are obviously dominated by the storage of the "expensive" trigrams, and thus the space complexity is of the order $O(|V|^3)$. For the sake of completeness the algorithm is formally given below.

The relaxation algorithm is extremely elegant because it not only utilizes the information contained in the unigrams of the plaintext language. Indeed, as can be seen, it attempts to "squeeze" out the information resident in the trigrams of the language too. Additionally, except for the complex formulae involved, it is also quite simple to implement. However, the method has some inherent drawbacks. First and foremost, the time complexity of the algorithm shows that the scheme is computationally very expensive. Apart from estimating the trigrams, each iteration of the algorithm involves enormous real time computations. Furthermore, the space required is also very large. For the English alphabet, it requires at least 26^3 real number memory locations.

From the point of view of implementation, we have observed that relaxation algorithm also possesses another drawback. This observation is on the basis of the numerous experiments that we have conducted. When the actual trigram probabilities are estimated from an finite ("small") stream of plaintext, many of the trigrams are non-existent and hence their corresponding estimated probabilities are zero. This in turn simplifies many of the computations, and thus, although the

trigrams are inaccurate the resulting computations that are done can be done to a fine degree of accuracy. However, if the stream of plaintext is large ("infinite"), the corresponding estimated probabilities converge to their maximum likelihood values. In this case, the trigram array is no longer sparse. Although this is good from a statistical point of view, it is far from desirable from a computational point of view. In this scenario, many of the computation errors involved in the recursive computations of the set $\{P_v\}$ augment one another with the result that the accuracy of the probability vectors degenerates as the algorithm proceeds. The alternative, of course, is to resort to double precision arithmetic which, although it guarantees the accuracy of the computations, further adds to the sluggishness of the convergence process in terms of the computing time utilized.

ALGORITHM RELAXATION

Input : The Unigrams and Trigrams of the plaintext language, and **C**, the cyphertext.

Output : A table of final values for the vectors P_v for every $v \in V$. For each cypher letter the algorithm chooses the most probable true letter as its decypher.

Notation:
(i) n is the total length of the **C**, cypher text.
(ii) u is the plaintext unigram distribution, and $r_{\alpha\beta\chi}$ is the quantity defined in (2).
(iii) K is the frequency distribution of all cypher letters, and,
(iv) B_{abc} is a Boolean variable recording whether the string abc occurs in **C**.

Assumptions:
(i) The procedure *GetInitialVectors* exists which uses parameters u, K, and n and returns the initial estimates for P_v for every v.
(ii) The procedure *PrintPresentVectors* exists. It processes the vectors P_v, and outputs for each vector the component and the index with the largest value.
(iii) One of the procedures *AverageUpdatingMethod* or *ProductUpdatingMethod* is available. Both of these take $r_{\alpha\beta\gamma}$, and B_{abc} as input parameters and return updated estimates for the vectors P_v as per (3) and (4) respectively.

Method

```
GetInitialVectors (u, K, n)                          /* equation (5) */
Repeat
   PrintPresentVectors
   Update the Vectors {Pv} Using
      Either
         AverageUpdatingMethod (rαβγ , Babc)         /* equation (3) */
      Or
         ProductUpdatingMethod (rαβγ , Babc)         /* equation (4) */
ForEver.
```

END ALGORITHM RELAXATION

III. BREAKING SUBSTITUTION CYPHERS USING AUTOMATA

Our solution to the problem of breaking substitution cyphers involves the use of a stochastic learning automaton. The intention is to attempt to overcome the drawbacks listed in the previous section, and in this endeavour we shall design a finite state machine which processes the cypher text sequentially and does not perform any floating point or real computations.

Learning automata have been used in the literature to model biological learning systems and also to learn the optimal action which an environment offers. The learning is achieved by actually interacting with the environment and processing its responses to the actions that are chosen. Such automata have various applications such as parameter optimization, statistical decision making and telephone routing [4-6]. Since the literature on learning automata is extensive, we refer the reader to a review paper [6] and two excellent books on the field by Lakshmivarahan [4] and Narendra and Thathachar [5] for a review of the families and applications of learning automata.

The learning process of an automaton can be described as follows: The automaton is offered a set of actions by the environment with which it interacts, and it is constrained to choose one of these actions. When an action is chosen, the automaton is either rewarded or penalized by the environment with a certain probability. A learning automaton is one which learns the optimal action, which is the action which has the minimum penalty probability. Hopefully, the automaton will eventually choose this action more frequently than other actions.

Stochastic learning automata can be classified into two main classes: (a) Fixed Structure Stochastic Automata and (b) automata whose structure evolves with time. Some examples of the former type are the Tsetlin, Krinsky and Krylov automata [5,6,10]. Although the latter automata are called variable structure stochastic automata, (because their transition and output matrices are time varying) they are merely defined in terms of action probability updating rules [4-6]. Throughout this paper we will only be considering Fixed Structure Stochastic Automata. We are currently investigating the use of variable structure automata to solve the cypher breaking problem.

III.1 Fundamentals of Learning Automata

A FSSA is a quintuple $(\alpha, \Phi, \beta, F, G)$ where :

(i) $\alpha = \{\alpha_1, ..., \alpha_R\}$ is the set of actions that it must choose from.

(ii) Φ is its set of states.

(iii) $\beta = \{0, 1\}$ is its set of inputs. The input '1' represents a penalty.

(iv) F is a map from $\Phi \times \beta$ to Φ. It defines the transition of the state of the automaton on receiving an input. F may be stochastic.

(v) G is a map from Φ to α, and determines the action taken by the automaton if it is in a given state. With no loss of generality, G is deterministic [5,6].

The selected action serves as the input to the environment which gives out a response $\beta(n)$ at time 'n'. $\beta(n)$ is an element of $\beta = \{0,1\}$ and is the response of the environment which is fed back to the automaton. The environment penalizes the automaton with the probability c_i, where,

$$c_i = \Pr [\, \beta(n) = 1 \mid \alpha(n) = \alpha_i \,] \qquad (i = 1 \text{ to } R).$$

Thus the environment characteristics are specified by the set of penalty probabilities $\{c_i\}$ (where i = 1 to R). On the basis of the response $\beta(n)$ the state of the automaton $\phi(n)$ is updated and a new action is chosen at the time instant (n+1).

The $\{c_i\}$ are unknown initially and it is desired that as a result of interaction with the environment the automaton arrives at the action which presents it with the minimum penalty response in an expected sense. Note that if c_L is the minimum penalty probability, and if

$$P_i(n) = \Pr [\, \alpha(n) = \alpha_i \,],$$

the solution which sets $P_L(n) = 1, \quad P_i(n) = 0 \quad \text{for } i \neq L$

achieves this result. Automata are designed with this solution in view.

With no *a priori* information, the automaton chooses the actions with equal probability. The expected penalty is thus initially M_o, the mean of the penalty probabilities.

With regard to learning criteria, an automaton is said to learn Expediently if, as time tends towards infinity, the expected penalty is less than M_o. We denote the expected penalty at time 'n' as E[M(n)]. The automaton is said to learn Absolutely Expediently if for all n, $E[M(n+1)] < E[M(n)]$. We say that the automaton is Almost Absolutely Expedient if the latter inequality not strict, i.e., if for all n, $E[M(n+1)] \leq E[M(n)]$. The automaton is said to be optimal if E[M(n)] asymptotically equals the minimum penalty. It is ε-optimal if, in the limit, E[M(n)] can be made as close to this minimum penalty as desired by the user. This is usually achieved by a suitable choice of some parameter of the automaton, for example, the number of states of the automaton.

Since the learning automaton which we introduce uses some properties of the Tsetlin automaton, we shall briefly describe the latter .

III.2 The Tsetlin Automaton

The reward characteristics of the automaton solution which we shall present is akin to that of a well known automaton due to Tsetlin [6,10]. The K-action Tsetlin automaton, $L_{KN,K}$, has KN states $\{\phi_1, \phi_2, \ldots, \phi_{KN}\}$ and chooses one of the K actions $\{\alpha_1, \alpha_2, \ldots, \alpha_K\}$. N is called the "Depth" of the automaton. Explicitly, $L_{KN,K}$ is defined as follows :

$$L_{KN,K} = \left(\{\phi_1, \phi_2, \ldots, \phi_{KN}\},\ \{\alpha_1, \alpha_2, \ldots, \alpha_K\},\ \{0, 1\},\ F(\cdot\,,\cdot),\ G(\cdot) \right).$$

The $G(\cdot)$ map is deterministic and is defined as :

$$G(\phi_i) = \alpha_j \text{ if } (j-1)N + 1 \leq i \leq jN \qquad (9)$$

Observe that since this means that the automaton chooses α_1 if it is in any of the first N states, it chooses α_2 if it is in any of the states from ϕ_{N+1} to ϕ_{2N}, etc., the states of the automaton are automatically partitioned into groups, each group representing an action.

The transition map, $F(\cdot, \cdot)$ is **deterministic** and is described as below :

(1) If $\beta = 0$ (it gets a favorable response), the F map requires that the automaton go towards the most internal state corresponding to that action -- $\phi_{(j-1)N+1}$ for action α_j one step at a time. Explicitly,

$$\begin{aligned} F(\phi_i, 0) &= \phi_{i-1} && \text{if } (j-1)N + 1 < i \leq jN \\ &= \phi_i && \text{if } i = (j-1)N + 1 \end{aligned} \tag{10}$$

(2) If $\beta = 1$ (it gets an unfavorable response), the automaton moves towards the boundary state -- ϕ_{jN} if α_j is the action chosen -- one step at a time. If it is in the boundary state for the action, it moves to the boundary state for the next action. Thus, for $1 \leq j \leq k$.

$$\begin{aligned} F(\phi_i, 1) &= \phi_{i+1} && \text{if } (j-1)N + 1 \leq i < jN \\ &= \phi_{((j+1)N) \bmod KN} && \text{if } i = jN, j \neq K-1 \\ &= \phi_{KN} && \text{if } i = (K-1)N \end{aligned} \tag{11}$$

The theoretical properties of $L_{KN,K}$ are found in [10] where it is proved that the automaton is ϵ-optimal in random environments in which the minimum penalty probability is less than 0.5.

III.3 The Cypher Learning Automaton

The learning automaton solution presented in this paper is called the Cypher Learning Automaton (CLA). It is based on an automaton which was designed to solve the object equi-partitioning problem [7,11] which involved partitioning a set of objects into various subsets of equal size such that the objects that are accessed more frequently together are placed in the same partition. The Equi-partitioning problem is NP-Complete primarily because the number of partitions involved grows exponentially with the number of objects to be partitioned. In [7] Oommen and Ma designed a learning scheme which processed the queries one at a time and moved the objects (or rather, symbols representing the objects) into partitions. The symbols learned their optimal places as the stream of queries was processed.

First of all, we assume that rather than model the plaintext language as an infinite subset of Λ^*, the plaintext language is specified using a "reasonable" finite subset of Λ^*. We refer to this dictionary as **H**. Various such subsets can be compiled for the English language, the most well-known and acclaimed subset for English being the 1023 most common words compiled by Dewey [1]. Statistically, these words comprise a large proportion of English literature. We also assume

that the algorithm has a finite stream of cyphertext **C** which is a subset of $\mathbf{V}^*$ encyphered using the actual underlying cypher T^*.

We now describe the structure and operation of the Cypher Learning Automaton (CLA). We define the CLA as a 6-tuple as below :

$$\left(\Lambda\ ,\ \{\phi_1, \phi_2, ..., \phi_{KN}\},\ \{\alpha_1, \alpha_2, ..., \alpha_K\},\ \{0, 1\},\ Q(\cdot\ ,\ \cdot),\ G(\cdot)\right), \text{ where,}$$

(i) Λ is the set of characters to be distributed among different states of the CLA. Indeed, Λ is the plaintext alphabet.

(ii) $\{\phi_1, \phi_2, ..., \phi_{KN}\}$ is the set of states.

(iii) $\{\alpha_1, \alpha_2, ..., \alpha_K\}$ is the set of K actions, each representing a specific element of **V**, the cypher alphabet.

(iv) {0,1} are the inputs to the automaton. As in the traditional case, '0' represents rewarding the automaton and '1' represents penalizing it.

(v) Q, the transition function is quite involved and will be explained in detail presently.

(vi) The function G is identical to the one described in (9). Thus, for each action α_k, there is a set of states $\{\phi_{(k-1)N+1}, ..., \phi_{kN}\}$, where N is the depth of memory and $1 \le k \le K$. With no loss of generality, we assume $\phi_{(k-1)N+1}$ to be the most internal state of action α_k and ϕ_{kN} to be the boundary state.

As opposed to the automata customarily studied in the literature [4-6,10], in the case of the CLA we have **all** the characters of Λ moving around in the states of the machine. If the character $a_i \in \Lambda$ is in action α_k, it signifies that the current solution chosen by the automaton is that the cyphertext character corresponding to α_k is decyphered as the plaintext character $a_i \in \Lambda$. Observe that if the states occupied by the characters of Λ are given, the actions chosen by these elements can be trivially obtained using (9). This specifies the current solution to the cypher chosen by the CLA.

Let $\xi_a(n)$ be the index of the state occupied by $a \in \Lambda$ at the n^{th} time instant. Based on $\{\xi_a(n)\}$, let T(n) be the current solution for the cypher chosen by the automaton. Observe that, as mentioned above, this function is trivially obtained by repeatedly using (9) to the various elements of Λ to determine the cypher letter to which $a \in \Lambda$ encyphers to. Using this notation we shall now describe the transition map of the CLA. However, in what follows, in the interest of brevity, we shall omit the reference to the time instant 'n'.

Initially, the CLA begins its learning process by starting from an arbitrary solution T(0) such that the mapping from **V** to Λ is one-to-one and onto. It is thus a feasible solution to the problem. This initial solution (mapping) is obtained by computing the unigram statistics of **C** and comparing it with the corresponding statistics of the underlying language. If the latter is not readily available, a crude approximate of this can be obtained by computing the unigram statistics of **H**.

Subsequent to assigning T(0) the following process continues until the user is satisfied with the accuracy of the solution.

To quantify the goodness of a particular solution T, we define a function *TotalMatches*. This function has as its input the cypher text **C** and decyphers it word by word using the current solution T. The function returns the total number of decyphered words found in **H**. Since **H** is assumed to be a representative subset of the plaintext language, it is not unreasonable to assume that a solution which yields a higher value for *TotalMatches* is superior to one which yields a lower value.

On processing a string to be decyphered $X \in \mathbf{V}^*$ the string X is decyphered based on the current solution, T. More formally, if the cypher word is $X = x_1x_2...x_N$ then the decyphered plaintext string is $Y = y_1y_2...y_N$ where $y_i = T(x_i)$. The algorithm searches the dictionary **H** to see if it contains the string Y. The presence of Y in **H** serves to reward or penalize the automaton, and hence this module essentially serves as the Environment which the automaton interacts with. If $Y \in \mathbf{H}$, a decyphering of X is assumed to have succeeded, and the current solution T is rewarded. Otherwise, it is assumed that the cypher word has "most probably" not been decyphered correctly and the current solution T is penalized. The operations involved with rewarding and penalizing the function T are described below.

The Reward operation is quite simple because it involves rewarding the choice taken by every single letter of the decyphered word. In this case, every element y_i, where $y_i = T(x_i)$ is moved towards the internal state of its corresponding action as per the transition map of the Tsetlin automaton (see Figure I).

As opposed to this, the Penalty operation which penalizes the automaton for its current solution T is more intricate. In this case, the individual characters of Y are each moved towards the boundary state from one state to its adjacent one until at least one character is at the boundary state. The set of characters in the boundary states are clearly those which are most "uncertain" about the correctness of their current choices. One of these characters in the boundary state is randomly chosen and moved to the boundary state of another randomly chosen action. Since we are dealing with a one-to-one substitution cypher, the character which previously chose this latter action has now to be migrated. The CLA opts to move it to the boundary state of the *former* action, and thus in short the two characters swap actions and terminate in boundary states of their new choices. This operation is described pictorially in Figures II.1 and II.2. Observe that the aim of this migration is to attempt to ameliorate the present mapping function T (n).

Once this migration process has been executed, the efficiency of T(n) and that of the new potential solution obtained are compared by invoking the function *TotalMatches*. If the new potential solution is superior or is as good as T(n) this becomes the current solution, T(n+1).

Otherwise the character being migrated is swapped with another random character until this condition is met (See II.1 and II.2). Figure III highlights the the difference between T^* and T(n) and shows how a new T(n+1) which is superior to T(n) can be obtained.

It should be noted, in passing, that the cypher text need not merely consist of encyphered elements in **H**. Indeed, in the form explained here, the cypher text may contain some encyphered words which are not in **H**, or for that matter which are not even in the language from which the plaintext is derived. In such cases we assume that such words occur less frequently than those which are in **H**, and thus we rely on the learning properties of a stochastic automaton when operating in a random environment to lead the CLA to learn the correct solution. Indeed, in these cases, the CLA does the best job it can with the information it can use and our experimental evidence shows that the scheme converges rapidly to the true solution T^*.

The Cypher Learning Algorithm (CLA) which has been informally described above is formally presented below in ALGORITHM CLA. This has been done for the sake of completeness and clarity. Also, for the sake of clarity, we shall merely manipulate the indices of the states occupied by the various characters in Λ.

By a simple analysis it is easy to see that if the dictionary is sorted, the average time required for a single iteration of the "Repeat" loop in the *Main Algorithm* is O(log(|**H**|).|**C**|.w), where w is the average word length in **H**, since the dominating factor is the evaluation of *TotalMatches* invoked in *Punish*. However since there are a total of $|\Lambda|$ actions, the process of migrating a character to a random destination action can, in the worst case scenario, have to try all of the other actions. Thus the worst case complexity for the algorithm will be O(|Λ|.log(|**H**|).|**C**|.w). Note that apart from maintaining the dictionary and **C**, the memory requirements of the scheme essentially involves maintaining the state locations of the various elements of Λ.

One of the major advantages of this scheme over the former relaxation algorithm is the great improvement in space and memory usage. Also, the method does not depend on an elaborate definition of the plaintext language described by either its n-grams or even the positional binary n-grams [2]. However, the greatest advantage of the scheme is that the computations involved in this case are merely integer updates. Indeed, the algorithm does not involve any floating-point or real computations. Also, since the efficiency of the solution measured in terms of the TotalMatches is always nondecreasing, it is clear that the scheme is Almost Absolutely Expedient. Based on the experimental evidence we have we also conjecture that it is ε-optimal.

ALGORITHM CLA

Input : An initial solution T(0). This assignment is based on the unigram statistics of the language and the cypher as explained earlier. The characters are initially placed on the boundary states of the actions of the CLA. Thus if $a \in \Lambda$ is assigned to the action α_k, then ξ_a is assigned the value kN. The algorithm is also given the finite dictionary **H** and a cypher text stream **C** repeatedly read by CLA using a function ReadWord.

Output : A final mapping function T^{∞} obtained as a result of the learning process of the CLA.

Notation : The CLA has N states per action, and ξ_a is the index of the state associated with the character $a \in \Lambda$. Thus ξ_a satisfies $1 \leq \xi_a \leq KN$, and furthermore if $a \in \Lambda$ is assigned to action α_k, $(k-1)N+1 \leq \xi_a \leq kN$. Note that all states of the form kN are boundary states. The j^{th} character of a given word X is represented by x_j, where $1 \leq j \leq |X|$. Given the set $\Xi = \{\xi_a | a \in \Lambda\}$ the current solution T(n) chosen by the CLA is computed trivially using (9).

Assumptions : We assume that the algorithm has access to the following functions :

(i) A Boolean function *MatchPresent* which takes a word Y decyphered using the current solution T and searches to see if it is contained in **H**.

(ii) The function *Decypher* which takes a string X and the set $\Xi = \{\xi_a | a \in \Lambda\}$ and decyphers it accordingly to the current solution T. It returns the decyphered string.

(iii) The function *TotalMatches* which decyphers **C** using T and returns the total number of decyphered words that are in **H**.

(iv) The function *ComputeFunction* which has as its input the set Ξ and yields for its output the current solution T.

```
MainAlgorithm Begin
    GetInitialAssignment (Ξ)              /* Ξ is the current state assignments */
    Repeat
        ReadWord (X)
        Y = Decypher (X, Ξ)               /* Decypher the cypher word */
        If MatchPresent (Y, H) Then
            Reward (Y, Ξ)
        Else
            Punish (Y, C, H, Ξ)
        EndIf
    Until Satisfied
    T^∞ := ComputeFunction (Ξ)            /* T* is the final function obtained using (9) */
End MainAlgorithm
```

```
Procedure Reward (Y, Ξ)
        For j := 1 to |Y| Do                                  /* process all characters in Y */
            If ( ξ_yj mod N ) ≠ 1 Then
                ξ_yj := ξ_yj - 1                  /* Move ξ_yj towards internal state by one state*/
        EndFor
End Procedure Reward

Procedure Punish (Y, C, H, Ξ)
        PreviousMatches := TotalMatches (C, H, Ξ )
        Repeat
            For all a ∈ Y Do                                  /* Do for all characters in Y. */
                If ( ξ_a mod N ) ≠ 0 Then                     /* a is not in boundary state. */
                    ξ_a = ξ_a + 1                             /* Move a towards boundary state */
            EndFor                  /* until at least one element is at the boundary state. */
        Until some elements of Y are in their boundary states.
        b := Random element in Y in boundary state.
        Repeat                                        /* b in boundary state is to be moved. */
            c := RandomElement (Λ)                            /* Choose random character, c */
            If (b ≠ c) Then
                ξ_b := (((ξ_b - 1) div N) + 1) • N     /* Move b to boundary state of its action. */
                ξ_c := (((ξ_c - 1) div N) + 1) • N     /* Move c to boundary state of its action. */
                Swap (ξ_b, ξ_c)                          /* Swap states of two abstract objects. */
            EndIf
        Until TotalMatches (C, H, Ξ) ≥ PreviousMatches.
End Procedure Punish

END ALGORITHM CLA
```

An example will help clarify the operation of Algorithm CLA.

III.4 An Illustrative Example of the CLA

Let Λ be the English alphabet and the dictionary **H** be:

H= {cat, bat, bet, it, bite, road, of}.

Also let T* be the mapping function shown in Figure I, and let the cypher text **C** be:

C = {gbf, dbf, dif, of, dofi}.

We also assume the present mapping function to be T(n) as shown in Figure I. This leads to a decyphering of **C** as {caf, baf, brf, **of**, bofr} with matches shown in bold. Thus for this particular T(n) we have TotalMatches(**C**, **H**, Ξ) = 1.

On processing X = 'dif' the cypher word is decyphered using T(n) into 'brf' and each letter of the decyphered word is punished as shown in Figures II.1 and II.2. Since in our example 'f'

was the first to reach a boundary state it is swapped with another random character, say - 'u'. The whole process leads to a new function T(n) shown in Figure I. The decyphering of **C** due to T(n+1) is now {**cat**, **bat**, brt, ot, botr} and TotalMatches(**C**, **H**, Ξ) = 2.

IV. EXPERIMENTAL RESULTS

To experimentally evaluate the results given in this paper, the relaxation algorithms and the CLA were tested in a variety of settings. The dictionary used in the case of the CLA was a set which primarily contained a subset of the 1023 most common English words compiled by Dewey [1]. It also contained some words currently used in computer literature. In the case of the relaxation algorithms, the trigram statistics were extracted from this dictionary before the cypher breaking algorithm was invoked. The unigram statisctics used were ones obtained for the English language from a large English text.

To demonstrate the power of the CLA we have shown in Figure IV a plot of the average percentage of words in $\mathbf{C} = \mathbf{H}^{-T^*}$ correctly decyphered as a function of the number of cypher keys that were examined. The number of experiments performed was 10. The initial key was obtained, as explained earlier, by ordering the characters of the alphabet based on the unigram statistics of the language and the cyphertext. Notice that initially, on the average, none of the words in **C** were correctly decyphered. The percentage rose to 78.0 as the average number of cypher keys examined increased to 600. In the worst case, all the words in **C** were correctly decyphered before 800 keys were examined. To quantify the power of the CLA in terms of the key decyphering capability, we have shown in Figure V a plot of the average percentage of characters in the key correctly decyphered in the experiments as a function of the number of cypher keys examined. Initially, on the average, six of the 26 characters of the key were decyphered. The percentage rose to 87.7 as the average number of cypher keys examined increased to 600, and to 95.0 as the average number of cypher keys examined increased to 700. Again, in the worst case, all the characters of the key were correctly decyphered before 800 different keys were examined.

To demonstrate the power of the CLA for the case when **C** is a superset of $\mathbf{H}^{-T^*}$(i.e., the plaintext from which the cyphertext is derived contains words not in **H**) we have performed similar experiments in which **H** consisted of 810 words and **C** consisted of 900 words with the additional constraint that $\mathbf{C} \supseteq \mathbf{H}^{-T^*}$. Notice that in this case, on the average ten percent of the words encountered are not derived from **H**. In Figure VI a plot of the average percentage of words in $\mathbf{C} \cap \mathbf{H}^{-T^*}$ correctly decyphered as a function of the number of cypher keys that were examined is given. The average was taken over the ten experiments performed. As before, the initial key was obtained based on the unigram statistics. Initially, on the average, none of the words in $\mathbf{C} \cap \mathbf{H}^{-T^*}$ were correctly decyphered. The percentage rose to 49.2 as the average number of cypher keys

examined increased to 600. In the worst case, all the words in **C** were correctly decyphered before 1500 keys were examined. A plot of the average percentage of characters in the key correctly decyphered in the experiments as a function of the number of cypher keys examined is shown in Figure VII. Again, initially six of the 26 characters of the key were correctly decyphered. The percentage rose to 69.6 as the average number of cypher keys examined increased to 600, and to 80.8 as the average number of cypher keys examined increased to 800. Again, in the worst case, all the characters of the key were correctly decyphered before 1500 different keys were examined. This convergence is remarkable in both these experiments considering that the key could be any one of the 26! possible permutations of the English alphabet. The power of the CLA is obvious.

Also, as a means of comparing the two vastly different cypher breaking algorithms extensive simulations of both were run on a SUN system. The comparison was done based on *actual real time* computing speed, and the results are given in Table I. Whereas in the case of the Relaxation method cypher texts between 300-400 words in length were used, in the case of the CLA various cypher and dictionary texts between 810-900 words were used.

From Table I we observe that the CLA outperforms the relaxation algorithm in terms of real time speed sometimes by a factor which is almost **six**. Furthermore, apart from the superiority of the CLA observed from the point of view of the real time used, the greatest asset of the scheme is the fact that it utilizes only a linear memory storage space and it requires no floating point computations. Thus, whereas the relaxation method is unable to handle long cyphers (more than 1000 words) due to underflows caused by the cumulation of the computing errors in each iteration of the scheme, the CLA requires less iterations as the size of the dictionary is increased.

Relaxation - Average Updating Method	Relaxation - Product Updating Method	CLA
673.00	232.82	115.10

Table I: Average real time duration in seconds for the three algorithms to successfully break a cypher. In each case all the cypher letters were correctly decyphered. Also, in each case the average was obtained by performing ten identical simulations with varying cyphertext.

V. CONCLUSIONS

Let Λ be a finite plaintext alphabet, and $\mathbf{V}$ be a cypher alphabet with the same cardinality as Λ. In all one-to-one substitution cyphers there exists the property that each element in Λ maps onto exactly one element in $\mathbf{V}$. This mapping of $\mathbf{V}$ onto Λ is represented by a function T^* which maps any $v \in \mathbf{V}$ onto some $\lambda \in \Lambda$ (i.e., $T^*(v) = \lambda$). In this paper we have considered the problem of learning the mapping T^* (or its inverse $(T^*)^{-1}$) by processing a sequence of cypher text. Unlike traditional methods which break the cypher utilize the statistical information of the plaintext language, in this paper we have shown how learning automata can be used to solve the problem. Indeed, a new finite state learning machine called the Cypher Learning Automaton (CLA) has been proposed which sequentially processes the cyphertext and a finite dictionary which is used as a model for the language from which the plaintext has been derived. This method is fast and the advantages of the scheme in terms of time and space requirements over the relaxation method [8,9] have been listed. The question of using variable structure stochastic automata to solve the same problem is currently being studied. Also the problem of breaking more general substitution cyphers such as DES using automata remains an extremely interesting avenue for further research.

REFERENCES

[1] Dewey, G., Relative Frequency of English Speech Sounds, Cambridge, MA, Harvard Univ. Press, 1923.

[2] Hall, P. A. V., and Dowling G. R., "Approximate String Matching", Computing Surveys, 1980, pp. 381-402.

[3] Kirby, R. A., "A Product Rule Relaxation Method" in Computer Graphics and Image Processing, vol. 13, no. 2, June 1980, pp. 158-189.

[4] Lakshmivarahan, S., Learning Algorithms Theory and Applications, Springer-Verlag, New York, 1981.

[5] Narendra, K. S., and Thathachar, M.A.L., "Learning Automata -- A Survey", IEEE Trans. on Syst. Man and Cybernetics, 1974, pp.323-334.

[6] Narendra, K. S. and Thathachar, M. A. L., *Learning Automata: An Introduction.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.

[7] Oommen, B. J., and Ma, D. C. Y., "Deterministic Learning Automata Solutions to the Equi-partitioning Problem" in *IEEE Transactions on Computers*, vol. 37, no. 1, January 1988, pp. 2-13.

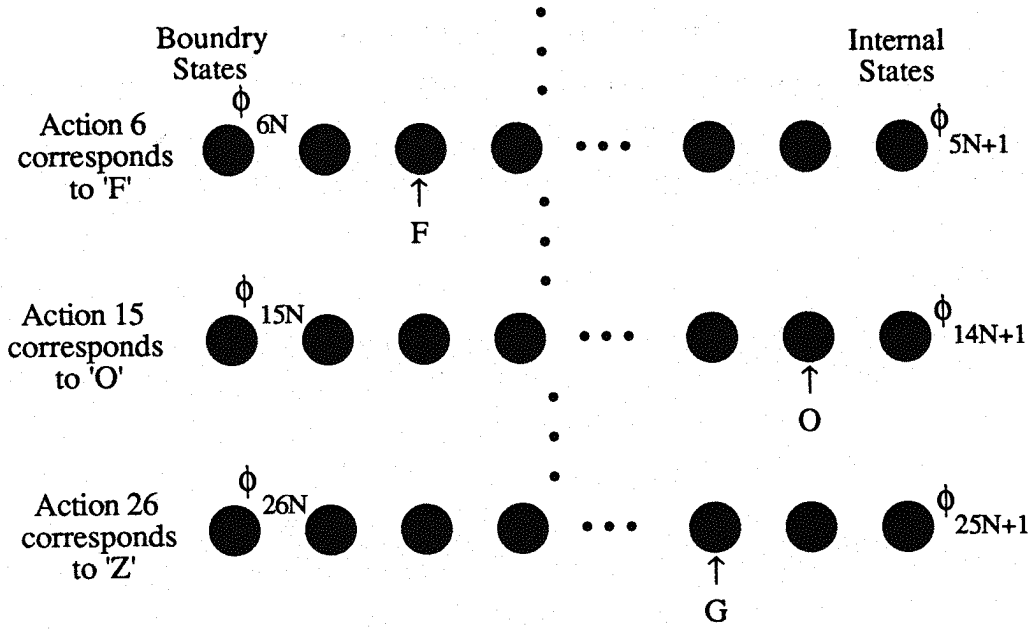
[8] Peleg, S., "A New Probabilistic Relaxation Scheme" in *IEEE Conference on Pattern Recognition and Image Processing*, Chicago, August 1979, pp. 337-343.

[9] Peleg, S. and Rosenfeld, A., "Breaking Substitution Ciphers Using a Relaxation Algorithm" in *Communications of the ACM*, vol. 22, no. 11, November 1979, pp. 598-605.

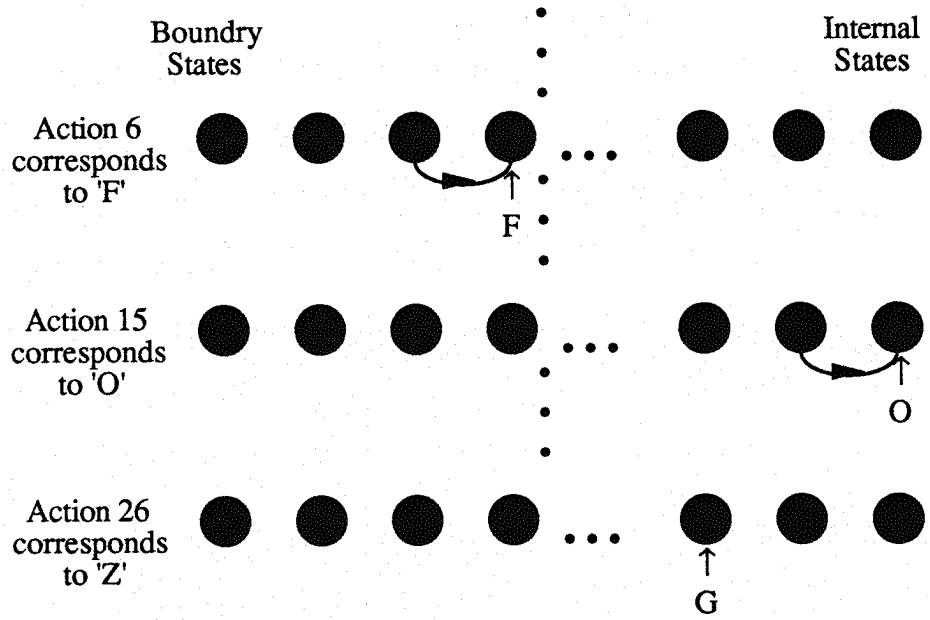
[10] Tsetlin, M.L., *Automaton Theory and the Modelling of Biological Systems*, New York and London, Academic, 1973.

[11] Yu, C.T., Siu, M.K., Lam, K., and Tai, F., "Adaptive Clustering Schemes : General Framework", *Proc. of the IEEE COMPSAC Conference*, 1981, pp.81-89.

[12] Denning, D.E.R, *Cryptography and Data Security,* Addison Wesley, Reading, 1983.

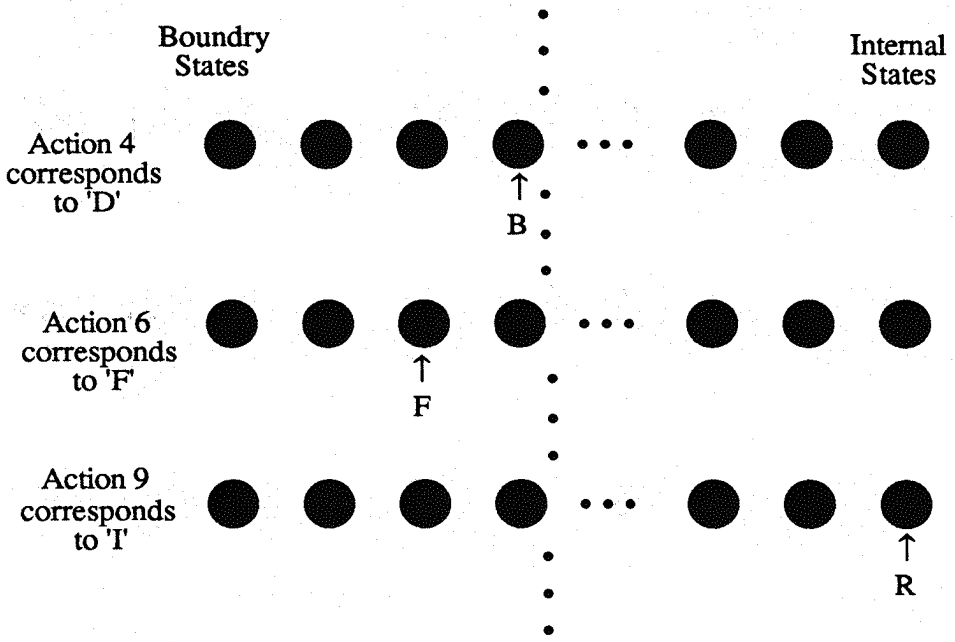


(a) The states of 'F', 'O', and 'G' before 'OF' is processed.

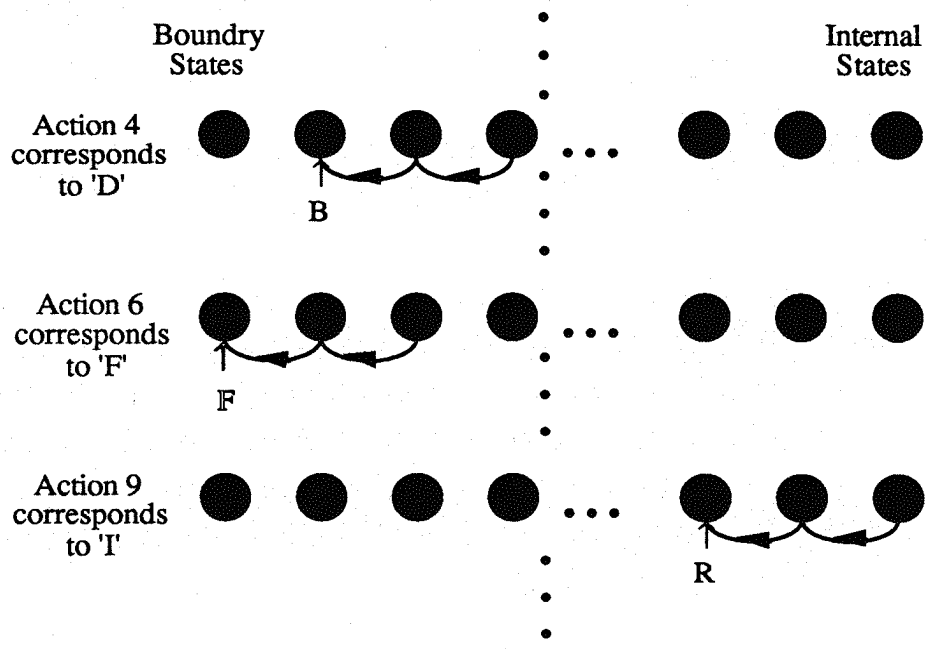


(b) The states of 'F', 'O', and 'G' after 'OF' is processed.

Figure I: Diagram of the CLA before and after the input cypher word 'of' is procesed. As explained in Section III.3 the CLA is rewarded. The state numbers in Ib are the same as those in Ia.



(a) Original states before processing.



(b) States of CLA after characters were moved towards bounadry states.

Figure II.1: The states of the CLA before and after the input word 'DIF' is decyphered. 'DIF' is actually the English word 'BET' but is decyphered incorrectly into 'BRF'. Since no matches are found the CLA is penalized as explained in Section III.3. This diagram describes the first stage of the penalizing process. The state numbering is identical to that of Figure I.

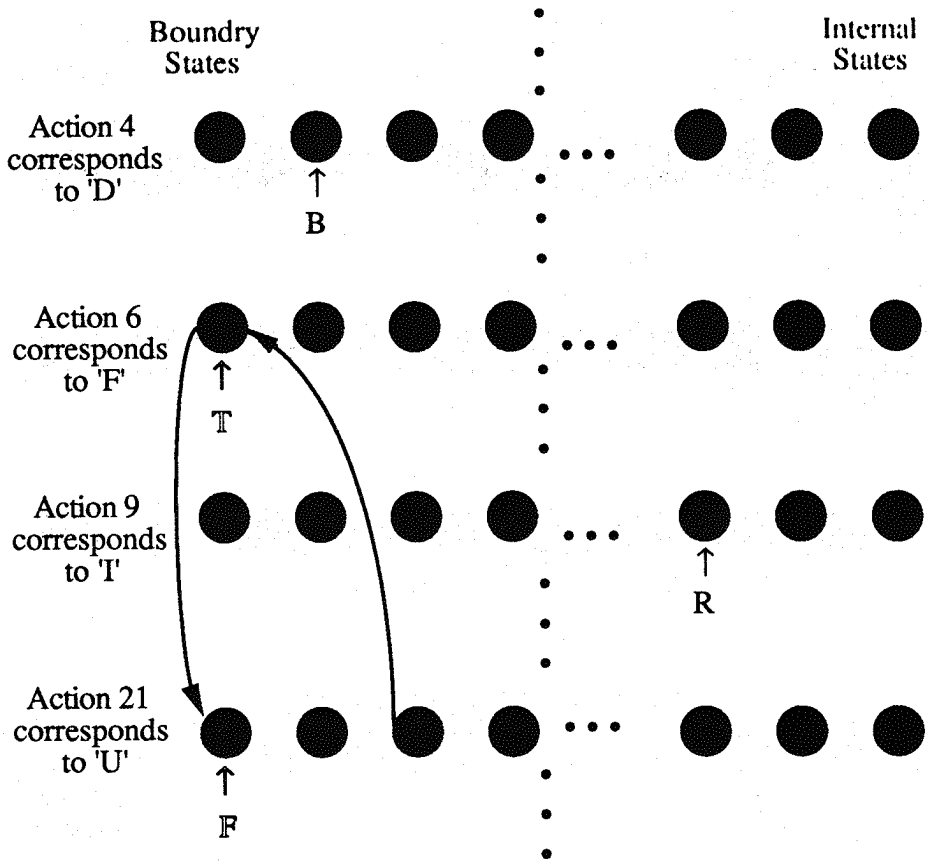


Figure II.2 : The states of the CLA after completion of the last stage of punishment for the scenario described in Figure II.1. This stage involves swapping of the character 'F' (since 'F' was the first character to reach a boundary state) with another character on any random action - Action 21 in this example. Each of the two characters end up in the boundary state of the other. The solution based on this configuration is the one given as T(n+1) in Figure I. The state numbering is identical to that of Figure I.

V: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

↓ ↓

Λ: K A D B M T C R E J W Z L S I O G V Y Q X U P N F H

(a) $\overset{*}{T}$ - the true key of the cypher, $V \rightarrow \Lambda$.

V: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

↓ ↓

Λ: Z A L B E F C U R J K P S D O Y N M X H T V I W Q G

(b) T (n) - the solution of the CLA.

V: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

↓ ↓

Λ: Z A L B E 𝕋 C U R J K P S D O Y N M X H 𝔽 V I W Q G

(c) T (n+1) - the solution right after punishement has occurred at time (n+1).

Figure III: Three different solutions represented by the CLA. Both **V** and Λ are the English alphabet. T^* is the actual key. The function T (n+1) was obtained from T (n) when the penalized as per the strategy explained in Section III.3 and diagramatically outlined in Figures II.1 and Figure II.2. The outlined English characters are the ones that were swapped during punishment.

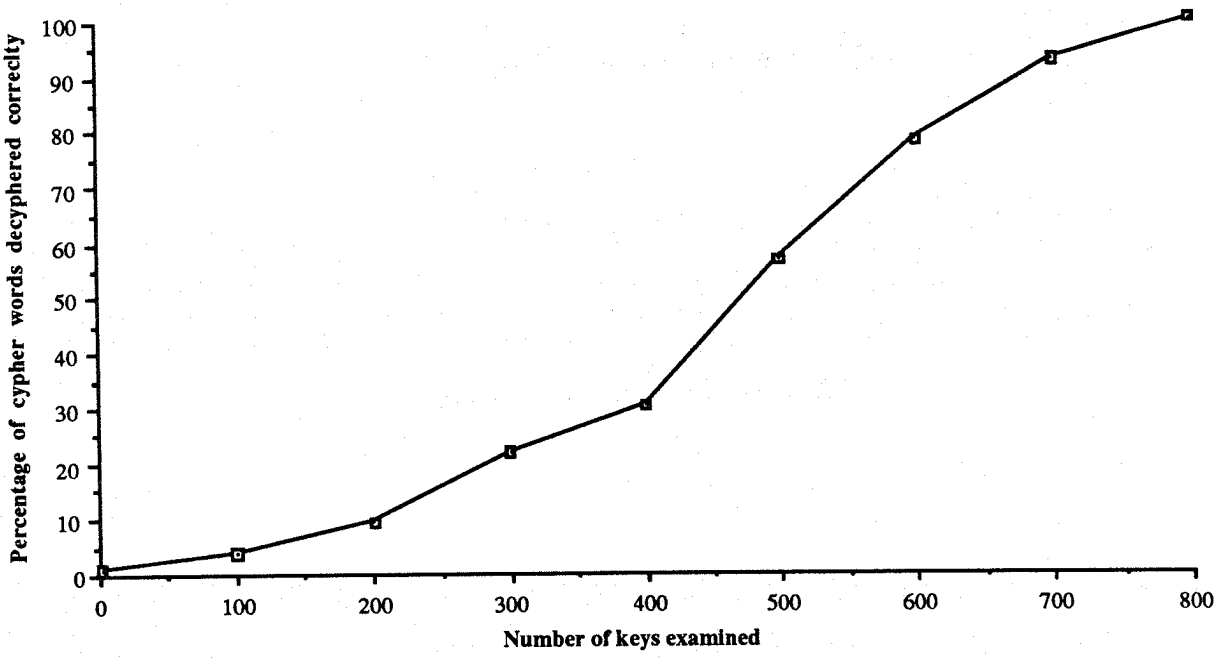


Figure IV: A plot of the average percentage of words in $\mathbf{C} = \mathbf{H}^{-\mathbf{T}^*}$ correctly decyphered as a function of the number of cypher keys examined. The number of experiments performed was 10.

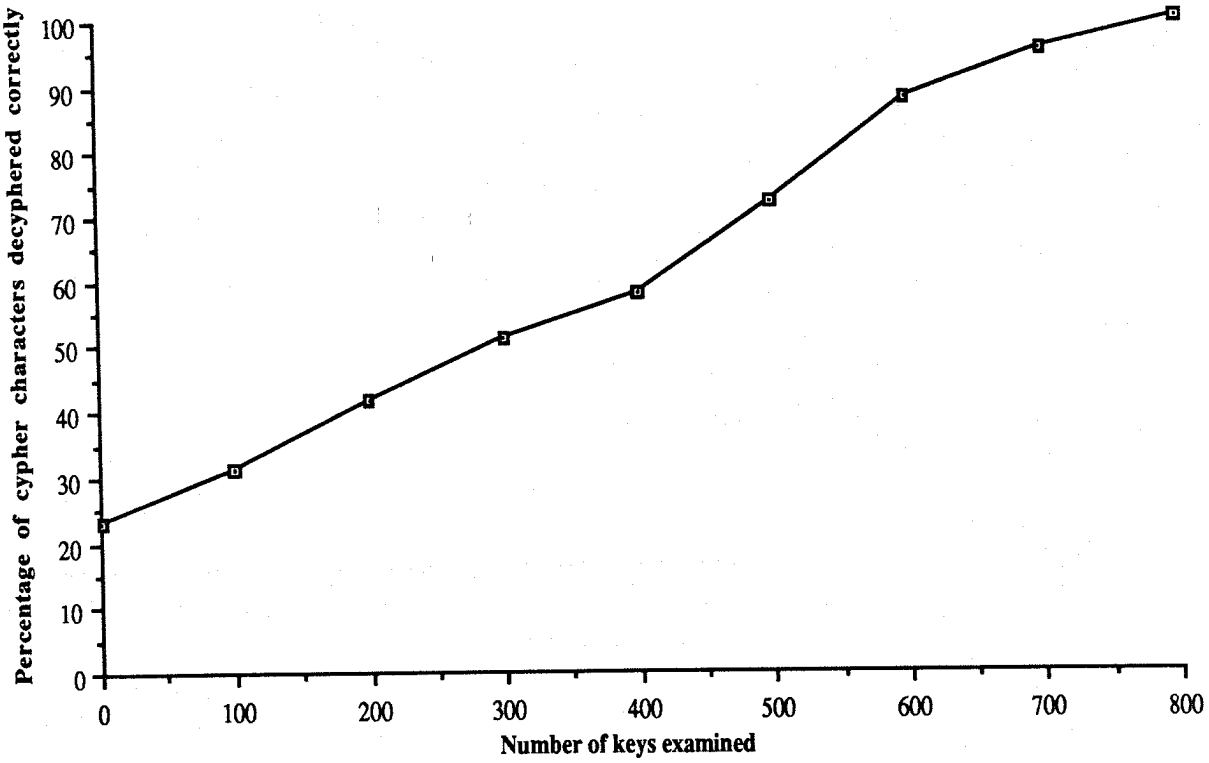


Figure V: A plot of the average percentage of letters in the key correctly decyphered as a function of the number of cypher keys examined. In this case, $C = H^{-T^*}$. The number of experiments performed was 10.

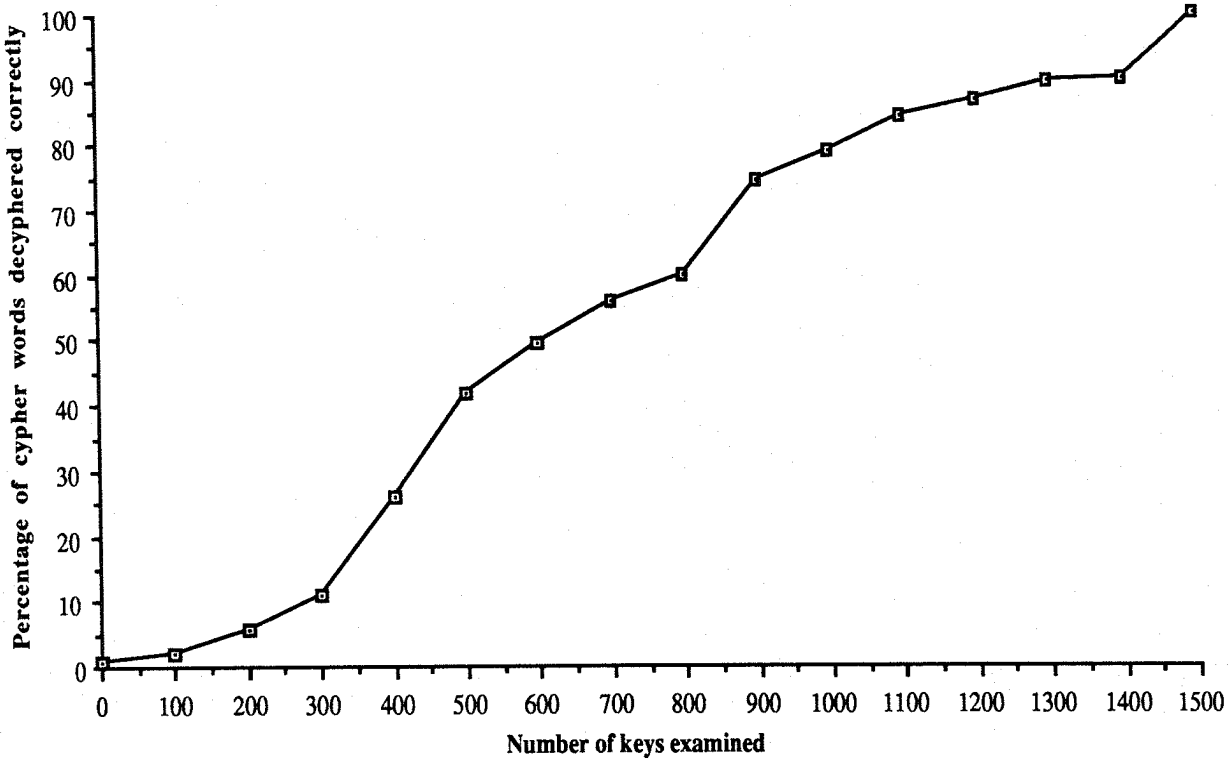


Figure VI : A plot of the average percentage of words in $\mathbf{C} \supseteq \mathbf{H}^{-T^*}$ correctly decyphered as a function of the number of cypher keys examined. The number of experiments performed was 10.

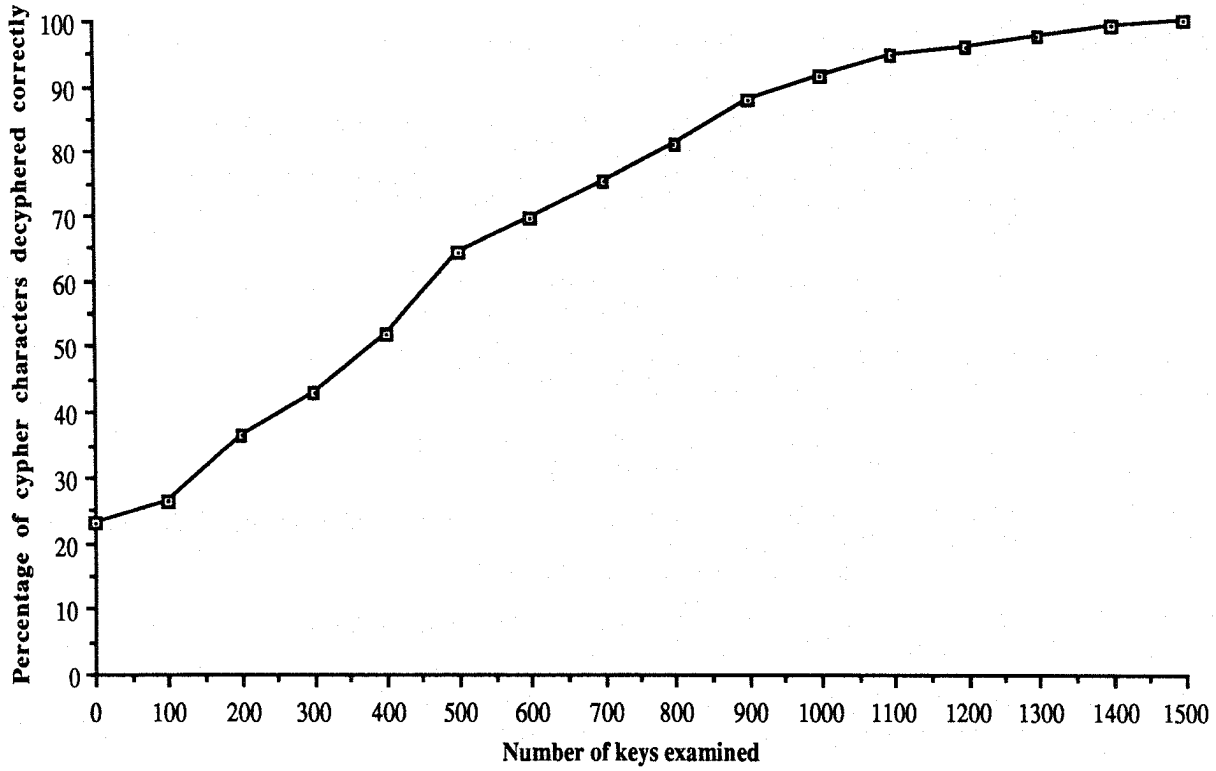


Figure VII: A plot of the average percentage of letters in the key correctly decyphered as a function of the number of cypher keys examined. In this case, $\mathbf{C} \supseteq \mathbf{H}^{-T^*}$. The number of experiments performed was 10.

School of Computer Science, Carleton University
Bibliography of Technical Reports

SCS-TR-140 **Computing the Configuration Space of a Robot on a Mesh-of-Processors**
F. Dehne, A.-L. Hassenklover and J.-R. Sack, June 1988.

SCS-TR-141 **Graphically Defining Simulation Models of Concurrent Systems**
H. Glenn Brauen and John Neilson, September 1988

SCS-TR-142 **An Algorithm for Distributed Mutual Exclusion on Arbitrary Networks**
H. Glenn Brauen and John E. Neilson, September 1988

SCS-TR-143 to 146 are unavailable.

SCS-TR-147 **On Transparently Modifying Users' Query Distributions**
B.J. Oommen and D.T.H. Ng, November 1988

SCS-TR-148 **An O(N Log N) Algorithm for Computing a Link Center in a Simple Polygon**
H.N. Djidjev, A. Lingas and J.-R. Sack, July 1988
Available in STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Lecture Notes in Computer Science, Springer-Verlag No. 349

SCS-TR-149 **Smallscript: A User Programmable Framework Based on Smalltalk and Postscript**
Kevin Haaland and Dave Thomas, November 1988

SCS-TR-150 **A General Design Methodology for Dictionary Machines**
Frank Dehne and Nicola Santoro, February 1989

SCS-TR-151 **On Doubly Linked List ReOrganizing Heuristics**
D.T.H. Ng and B. John Oommen, February 1989

SCS-TR-152 **Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry**
Frank Dehne and Andrew Rau-Chaplin, March 1989

SCS-TR-153 **The Use of Chi-Squared Statistics in Determining Dependence Trees**
R.S. Valiveti and B.J. Oommen, March 1989

SCS-TR-154 **Ideal List Organization for Stationary Environments**
B. John Oommen and David T.H. Ng, March 1989

SCS-TR-155 **Hot-Spot Contention in Binary Hypercube Networks**
Sivarama P. Dandamudi and Derek L. Eager, April 89

SCS-TR-156 **Some Issues in Hierarchical Interconnection Network Design**
Sivarama P. Dandamudi and Derek L. Eager, April 1989

SCS-TR-157 **Discretized Pursuit Linear Reward-Inaction Automata**
B.J. Oommen and Joseph K. Lanctot, April 1989

SCS-TR-158 (revised) **Parallel Fractional Cascading on a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, May 1989 (Revised April 1990)

SCS-TR-159 **Epsilon-Optimal Stubborn Learning Mechanisms**
J.P.R. Christensen and B.J. Oommen, June 1989

SCS-TR-160 **Disassembling Two-Dimensional Composite Parts Via Translations**
Doron Nussbaum and Jörg-R. Sack, June 1989

SCS-TR-161 (revised) **Recognizing Sources of Random Strings**
R.S. Valiveti and B.J. Oommen, January 1990
Revised version of SCS-TR-161 "On the Data Analysis of Random Permutations and its Application to Source Recognition", published June 1989

SCS-TR-162 **An Adaptive Learning Solution to the Keyboard Optimization Problem**
B.J. Oommen, R.S. Valiveti and J. Zgierski, October 1989

SCS-TR-163 **Finding a Central Link Segment of a Simple Polygon in O(N Log N) Time**
L.G. Alexandrov, H.N. Djidjev, J.-R. Sack, October 1989

SCS-TR-164 **A Survey of Algorithms for Handling Permutation Groups**
M.D. Atkinson, January 1990

SCS-TR-165 **Key Exchange Using Chebychev Polynomials**
M.D. Atkinson and Vincenzo Acciaro, January 1990

SCS-TR-166 **Efficient Concurrency Control Protocols for B-tree Indexes**
Ekow J. Otoo, January 1990

SCS-TR-167 **A Hierarchical Stochastic Automaton Solution to the Object Partitioning Problem**
B.J. Oommen, January 1990

SCS-TR-168 **Adaptive List Organizing for Non-stationary Query Distributions. Part I: The Move-to-Front Rule**
R.S. Valiveti and B.J. Oommen, January 1990

SCS-TR-169 **Trade-Offs in Non-Reversing Diameter**
Hans L. Bodlaender, Gerard Tel and Nicola Santoro, February 1990

SCS-TR-170 **A Massively Parallel Knowledge-Base Server using a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990

SCS-TR-171 **Parallel Processing of Quad Trees on the Hypercube (and PRAM)**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990

SCS-TR-172 **A Note on the Load Balancing Problem for Coarse Grained Hypercube Dictionary Machines**
Frank Dehne and Michel Gastaldo, May 1990

SCS-TR-173 **Self-Organizing Doubly-Linked Lists**
R.S. Valiveti and B.J. Oommen, May 1990

SCS-TR-174 **A Presortedness Metric for Ensembles of Data Sequences**
R.S. Valiveti and B.J. Oommen, May 1990

SCS-TR-175 **Separation of Graphs of Bounded Genus**
Ljudmil G. Aleksandrov and Hristo N. Djidjev, May 1990

SCS-TR-176 **Edge Separators of Planar and Outerplanar Graphs with Applications**
Krzystof Diks, Hristo N. Djidjev, Ondrej Sykora and Imrich Vrto, May 1990

SCS-TR-177 **Representing Partial Orders by Polygons and Circles in the Plane**
Jeffrey B. Sidney and Stuart J. Sidney, July 1990

SCS-TR-178 **Determining Stochastic Dependence for Normally Distributed Vectors Using the Chi-squared Metric**
R.S. Valivetei and B.J. Oommen, July 1990

SCS-TR-179 **Parallel Algorithms for Determining K-width- Connectivity in Binary Images**
Frank Dehne and Susanne E. Hambrusch, September 1990

SCS-TR-180 **A Workbench for Computational Geometry (WOCG)**
P. Epstein, A. Knight, J. May, T. Nguyen, and J.-R. Sack, September 1990

SCS-TR-181 **Adaptive Linear List Reorganization under a Generalized Query System**
R.S. Valiveti, B.J. Oommen and J.R. Zgierski, October 1990

SCS-TR-182 **Breaking Substitution Cyphers using Stochastic Automata**
B.J. Oommen and J.R. Zgierski, October 1990