AN ADAPTIVE LEARNING SOLUTION TO THE KEYBOARD OPTIMIZATION PROBLEM*

B.J. Oommen, J.S. Valiveti and J. Zgierski

SCS-TR-162, OCTOBER 1989

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

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

AN ADAPTIVE LEARNING SOLUTION TO THE KEYBOARD OPTIMIZATION PROBLEM*

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

Let **A** be a finite alphabet, and **H** be a finite dictionary of words formed from the letters of this alphabet. In a conventional keyboard typically each key of the keyboard is assigned a unique symbol chosen from the alphabet. We consider the problem of assigning more than symbol of the alphabet **A** to the same key on the keyboard. Let $\mathbf{C}_i$ be a subset of **A**. Every time a character in $\mathbf{C}_i$ is to be represented, the keyboard represents it using the digit (i.e. the index) 'i'. In a keyboard of this form, since multiple symbols of the alphabet **A** have the same representation, the representations of all the (distinct) words in the dictionary **H** need not be unique. A useful measure of the effectiveness of a particular keyboard is the number of words of **H** which are mapped to the same encoded form (i.e. they collide). This metric will be termed the ambiguity associated with the keyboard. The problem we study is one of optimally assigning the symbols of the alphabet to the keys of a given keyboard with a view to minimize the resulting ambiguity.

The problem as stated above is proven to be NP-hard. This naturally leads us to seek fast and approximate solutions to the optimization problem on hand. After presenting the only reported solution to the problem, we report a fast learning automaton-based solution to this problem. Experimental results demonstrating the power of this solution have also been presented.

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

I. INTRODUCTION

All of text processing involves manipulating the symbols of an alphabet and in almost all cases this alphabet is finite. For example, the most restricted alphabet is the binary set {0,1}, and the alphabet encountered for English text is the set of 26 characters {a...z}. To distinguish between the word of a language, customarily, various punctuation marks have been defined, the most common one being the delimiter "space" (represented throughout this paper with the symbol '_'). Of course, other punctuation marks such as the comma, semicolon, etc. are not only needed to distinguish the boundaries of the words but also to impart a semantic implication to the text.

Most of text processing functions that are available have a complexity which increases with the size of the alphabet. For example to evaluate the distance between two words, the optimal algorithm requires quadratic time [1,27,33,34] if the size of the alphabet is infinite. Indeed, this is a lower bound on the complexity of the computation [1,27,34]. This will also be the lower bound on the complexity of the computation of other comparison metrics involving strings such as the longest common subsequence and the shortest common supersequence, etc. [1,9,10,17,18,27,34]. However, the complexity of this computation decreases as the size of the alphabet decreases and thus it is often advantageous to work with alphabets of small size [1,19,34].

Apart from considering the infinite alphabet case, there are many cases where decreasing the size of the alphabet is both advantageous and convenient. For example, consider the Touch-Tone telephone keypad. Associated with every digit in the set {2...9} is a set of characters listed as C_i below :

$C_1 = \{\}$; $C_2 = \{abc\}$; $C_3 = \{def\}$

$C_4 = \{ghi\}$; $C_5 = \{jkl\}$; $C_6 = \{mno\}$

$C_7 = \{pqrs\}$; $C_8 = \{tuv\}$; $C_9 = \{wxyz\}$

Let us assume that the digit '0', is used to represent the delimiter, '_'. Observe now that instead of transmitting English text using the English alphabet one can just as easily transmit the text using the symbols {2...9}. Thus, instead of transmitting the string 'box' we could just as easily have transmitted the digit string '259'.

There is one fundamental disadvantage to this approach, and that is that the representation of the words can now be ambiguous. Thus, using the above example, if the Touch-Tone keypad was used for transmission, observe that the string '259' could represent both the English words 'box' and 'boy'. In this paper we consider the problem of assigning the letters of the alphabet to the various digits so as to minimize the ambiguities.

Not only is such an ambiguity minimizing keyboard assignment technique advantageous -- it is also mandatory in many cases. People who are physically handicapped [15,16,20,21] and

constrained to use a limited keyboard have to be able to transmit all the characters on their constrained keyboard. Often, not only is the keyboard limited in size, but the handicapped user is also constrained in terms of his or her vertical and horizontal motion capabilities [15,16]. It would, of course, be tremendously powerful if the handicapped user could use a modified keyboard and be guaranteed to communicate unambiguously.

Various other applications of such a keyboard assignment technique come to mind. If the dictionary was the set of all the names in a telephone directory, and if the directory was represented in accordance with the keyboard assignment strategy, a telephone company could easily extend the dial-by-number facility currently provided to permit the user a dial-by-name option. Obviously, this would save the user the tedious task of remembering telephone numbers. The user could dial a person of his/her choice provided the keyboard assignment had minimum ambiguity.

Indeed, in a more general setting, this technique can be used to design keyboards for any language in which the application uses a single key symbol to represent a set of characters. A powerful consequence of such a mapping technique is the resulting increase in the channel utilization, as is explained below. Consider the case when the alphabet in question is the English alphabet augmented with the delimiter '_'. To transmit 27 distinct characters a minimum of five bits must be assigned for each character. However, if all the characters are assigned to seven digits and the delimiter, '_', is uniquely assigned, then a maximum of three bits is sufficient. The data compression is significant -- of the order of 40%. Obviously, in this case too, the string need not be uniquely transmitted. However, if the keys are assigned appropriately, ambiguities can be correspondingly minimized, and in the case of ambiguous transmissions, a simple (possibly, user friendly menu- driven) selection technique can be used by the transmitter to exactly communicate which of the ambiguous words have to be transmitted. Thus, if the string "259" represented the English strings "box" and "boy", by appropriately prompting the user, the transmission can be disambiguated by allowing "259*1" to represent "box" and "259*2" to represent "boy".

I.1 Problem Statement and Notation

We now formally present the problem under consideration.

A is a finite alphabet of cardinality A and **H**, the dictionary, is a subset of the words over $\mathbf{A}^*$. Let **K** be a subset of the integers, where, with no loss of generality, $\mathbf{K} = \{i \mid 1 \leq i \leq K\}$. With every element $i \in \mathbf{K}$ is associated the set $\mathbf{C}_i$ which is a subset of **A**, where the $\mathbf{C}_i$'s are mutually exclusive and collectively exhaustive. Whenever a string $X=x_1x_2...x_N \in \mathbf{H}$ is to be transmitted (represented), instead of transmitting X in its virgin form we transmit the string $Y=y_1y_2...y_N$, where $y_i \in \mathbf{K}$ and $x_i \in \mathbf{C}_{y_i}$.

The symbol Π will be used to represent the set (or partition) $\{C_i\}$. Thus $\Pi(X)$ will be the string Y, where if $X=x_1x_2...x_N \in \mathbf{H}$, $Y=y_1y_2...y_N$, with $y_i \in \mathbf{K}$ and $x_i \in C_{y_i}$. Also, for the sake of ease of expression we shall define $\mathbf{H}^{\Pi}$ to be the mapping of $\mathbf{H}$ due to Π. Thus,

$$\mathbf{H}^{\Pi} = \{ Y \mid \Pi(X)=Y; \quad X \in \mathbf{H} \} \tag{1}$$

Observe that the cardinalities of both $\mathbf{H}$ and $\mathbf{H}^{\Pi}$ need not be the same, and that $|\mathbf{H}^{\Pi}| \leq |\mathbf{H}|$.

Although every string Y now represents a set of strings in $\mathbf{A}^*$, it may or may not represent multiple strings in the original dictionary $\mathbf{H}$ primarily because $\mathbf{H}$ is but a subset of $\mathbf{A}^*$. To quantify the effectiveness of a particular partition Π being able to disambiguate the dictionary we first define $\delta(X_0,\mathbf{H},\Pi)$ to be the number of collisions which a particular string $X_0 \in \mathbf{H}$ possesses. Indeed, if X_0 is transformed to Y_0 and no other $X \in \mathbf{H}$ is mapped into Y_0, the number of collisions, $\delta(X_0,\mathbf{H},\Pi)$, for X_0 as per the partition Π is zero. However, if Z other strings in $\mathbf{H}$ are also mapped into Y_0 then the number of collisions, $\delta(X_0,\mathbf{H},\Pi)$, must be Z. Formally,

$$\delta(X_0,\mathbf{H},\Pi) = \left| \{ X \mid \Pi(X) = \Pi(X_0); X, X_0 \in \mathbf{H}; X \neq X_0 \} \right| \tag{2}$$

Since the number of collisions per word has been explicitly defined, we can now define $\Delta(\mathbf{H},\Pi)$ to be the total ambiguity caused by a particular keyboard assignment (partition) Π. Using the definition for $\mathbf{H}^{\Pi}$, $\Delta(\mathbf{H},\Pi)$ can be defined as :

$$\Delta(\mathbf{H},\Pi) = |\mathbf{H}| - |\mathbf{H}^{\Pi}| \tag{3}$$

This is indeed the total number of collisions in the entire dictionary due to the assignment Π. However, $\Delta(\mathbf{H},\Pi)$ can also be defined in terms of $\delta(X,\mathbf{H},\Pi)$ as below :

$$\Delta(\mathbf{H},\Pi) = \sum_{X \in \mathbf{H}} \frac{\delta(X,\mathbf{H},\Pi)}{\delta(X,\mathbf{H},\Pi)+1} \tag{3a}$$

The fact that (3) and (3a) both represent identical expressions is not so obvious but can be verified by observing that if each individual $\delta(X,\mathbf{H},\Pi)$ is included in the summation, the quantity would be accounted for $(\delta(X,\mathbf{H},\Pi)+1)$ times.

The problem studied in this paper is one of computing the keyboard assignment, Π^* which minimizes $\Delta(\mathbf{H},\Pi)$. This problem is referred to as the Keyboard Optimization (OptKeyboard) problem and is formally stated below.

Problem OptKeyboard:

Input: Alphabet **A**, Dictionary **H**, and an integer K

Output: Π^* referred to as the optimal solution, i.e. $\Pi^* = \{C_i^* \mid 1 \le i \le K\}$ which is defined as follows. Let $\Pi = \{C_i \mid 1 \le i \le K\}$, where:

(i) $\bigcup_{i=1}^{K} C_i = A$

(ii) $C_i \cap C_j = \phi$, for $i \neq j$.

Then

$$\Pi^* = \text{Arg}_{\Pi}[\min \{ \Delta(H,\Pi) \}].$$

The decision version of the above optimization problem, will be called ApproxKeyboard and is formally stated below:

Problem ApproxKeyboard:

Input: Alphabet **A**, Dictionary **H**, and an integer K, and an integer MaxCollisions

Output: An approximate partition $\Pi^+ = \{C_i^+ \mid 1 \le i \le K\}$ where:

(i) $\bigcup_{i=1}^{K} C_i^+ = A$

(ii) $C_i^+ \cap C_j^+ = \phi$, for $i \neq j$.

In this paper we shall prove the result that ApproxKeyboard is NP-Complete if the dictionary **H** is known *a priori*. We conjecture that the problem is provably exponentially hard when **H** is unknown but only the values of $\Delta(H,\Pi)$ are available to the program for all possible permutations, Π. As a consequence of the above result, the original problem OptKeyboard can then be easily seen to be an NP-hard problem. The previous known solutions to the problem will be discussed, and a fast learning automaton solution to the problem will be proposed. Experimental results demonstrating the power of the automaton will also be presented.

I.2 Dictionary Representation Mechanisms

Central to the Keyboard Optimization problem is the technique by which the dictionary **H** is represented. Indeed, the way by which **H** will be modelled depends on the application, and the set of possible models spans a whole spectrum of techniques including representing it as a finite dictionary [1,7,9-12,17,30], representing **H** as being fully defined by bigrams and trigrams [4,24,28-30], or alternatively by representing **H** using binary positional n-grams [7,30]. In this paper we shall present a disambiguation strategy for the finite dictionary model. The primary reasons why we have chosen to represent the dictionary as a finite set of word is explained in this subsection.

In most applications encountered in real life the dictionary used is finite. This is especially true in the case of telephone directories, and the vocabulary used by hospitalized handicapped individuals. It is also true if the device is used to communicate with a machine with a finite command set. Indeed, even in the case of written English text, various studies have been made which indicate that large proportions of the words used in English form a very small subset of the possible English words. In fact Dewey [3] (See [12] for results of string correction experiments done using these words) has compiled such a collection and claimed that this collection consisting of 1023 words comprise a very large proportion of written English text. Thus, in both string processing and string recognition it is not uncommon to represent the dictionary as a finite set of words, and using this model, string correction is often achieved by comparing the received (noisy) word with every single word in the dictionary using a similarity metric [7,11,12,19,28,33,34].

The advantages of using a finite dictionary in text recognition applications are many. First of all, the accuracy of the recognition is very high [7,11,12,19]. Secondly, a noisy string is never recognized as a word which is not in the language, and thus, the question of 'meaningless' decisions is nonexistent. Finally, the time complexity of the computation involved in the text recognition process is typically quadratic per word and is linear in the size of the dictionary. The quadratic complexity per word can be often decreased if the dictionary is modelled using a trie [11], and if the alphabet size is decreased [1,28,34].

When the dictionary becomes prohibitively large, it is represented using a statistical model [4,24,28-30,32]. The most elementary model is the one in which only the unigram (single character) probabilities of the dictionary are required. A word in the dictionary is then modelled as a sequence of characters, where each character is independently drawn from a distribution referred to as the unigram distribution, and which is exactly the probabilities of the letter occurring in the language. A generalization of this is the one in which the bigrams of the language model the dictionary. A word in the dictionary is modelled as a sequence of symbols where two subsequent symbols $x_i x_{i+1}$ occur with the probability with which they occur in the language. The storage required for the bigrams is now quadratic with the alphabet size, and is independent of the size of the dictionary. However, if all the unigram and bigram probabilities have non-zero values, a string of length N can be one of $O(|A|^N)$ possible strings, and the computation can be excessive if the search tree is not appropriately pruned.

One of the biggest disadvantages of modelling the dictionary using n-grams or positional n-grams [4,24,28-30,32] is the fact that the word which is obtained after the string processing (correction) operation need not be a valid word in the language under consideration. Thus, all too often the word that has been obtained is "rejected", and this is especially the case when only bigrams are used to model the dictionary. The percentage of meaningless decisions decreases if the dictionary is modelled using n-grams where $n > 2$, but the memory required becomes excessive --

it is of $O(|A|^n)$ real numbers. Observe that representing a finite dictionary as a finite dictionary itself (i.e., as a set of words and not using a statistical model) is tantamount to specifying all the n-grams of the dictionary, and that for all values of 'n'. Notice too that if the latter is done, the problem of having meaningless words is nonexistent.

II. COMPLEXITY OF KEYBOARD OPTIMIZATION

In the previous section, we have formally presented OptKeyboard as a classical optimization problem, with a well defined objective function. In this section, we present the reduction which establishes the corresponding decision problem (i.e. ApproxKeyboard) to be NP-complete. For the sake of concreteness, we establish the result for the case when K=2. The proof that ApproxKeyboard is NP-complete for all values of K can be easily generalized from the proof for the case when K=2.

The problem we have on hand, can be related to the family of clustering problems studied by Brucker [2] and the MAX CUT problem [5], which are known to be NP-complete. In [2], Brucker establishes a correspondence between the MAX-CUT problem and the 2-Clustering problem, in which the set of nodes is to be split into two clusters, according to a given criterion function. A derivative of the former (i.e. the MAX-CUT problem) in which all the edges of the graph involved have only unit edge weights was shown to be NP-complete in [6], and thus following straightforward techniques analogous to Brucker's, it is easy to see that the "Simplified 2-Clustering Problem" (SIMP2C) is also NP-complete. Since our proof involves SIMP2C, we formally state the result which we shall use later.

Lemma 0:

Given a graph G=(V,E) with only unit edge weights (i.e. w(e) = 1 for each $e \in E$), and an integer M, the problem of partitioning the set of vertices V into two disjoint sets V_1 and V_2 such that $\psi(V_1, V_2) \leq M$, is NP-complete, where:

$$\psi(V_1, V_2) = \Sigma^2_{1=1} \Sigma_{a,b \in V_1, <a,b> \in E} w(<a,b>) \qquad (4) \qquad \bullet\bullet\bullet$$

We now state and prove a very important measure of complexity of problem ApproxKeyboard.

Theorem I.

ApproxKeyboard is NP-Complete.

Proof :

In order to prove the problem to be NP-complete, we have to establish the following [5]:

(i) ApproxKeyboard is in the class NP,

(ii) Some known NP-complete problem reduces to ApproxKeyboard or to a restricted instance of it.

To prove the first contention, i.e. that our problem belongs to the class NP is rather straightforward. The "non-deterministic" part of our algorithm need only guess the correct partition; the verification merely consists of the following steps:

(i) Encoding each word in the dictionary **H**

(ii) Counting the number of collisions (possibly after sorting the encoded dictionary $\mathbf{H}^{\Pi}$).

(iii) Verifying that the number of collisions is less than or equal to MaxCollisions.

Obviously steps (i) to (iii) in the verification process can be accomplished in polynomial time.

To prove the second contention of the Theorem, we shall use the SIMP2C problem and show that an instance of it reduces to the ApproxKeyboard problem. The reduction between SIMP2C and ApproxKeyboard is formally presented in Algorithm Reduce below. Observe that in the algorithm we have used the notation that if $a_p \in \mathbf{A}$, a_p^M is the string consisting of M consecutive occurrences of a_p.

ALGORITHM REDUCE

Input: An instance of SIMP2C which is a graph G = (V,E), where V = $\{a_1, a_2, \ldots a_R\}$, and the edge weights are unity for the edges in E and the integer M.

Output: A set of characters representing the alphabet **A**, and the dictionary **H**, and the integer MaxCollisions

Notation: For the sake of simplicity we have used the notation that if $a_p \in \mathbf{A}$, a_p^M is the string consisting of M consecutive occurrences of a_p.

Method:

```
A := V      /*Step1: Choose alphabet A*/
q := 0
For each edge e = <a,b> ∈ E do
    q := q + 1
    Word1 := a^q b^q
    Word2 := b^q a^q
    H := H ∪ { Word1, Word2}
Endfor   /* Step2 : Form dictionary H */
MaxCollisions := M
```

END ALGORITHM REDUCE

An example which illustrates the operation of Algorithm Reduce is included in the Appendix.

We now analyze the complexity of Algorithm Reduce. Step 1 of the transformation takes time O(n), where n = $|V|$. Note that in Step 2, the length of the words increase linearly with every

edge $e \in E$ which is examined and hence the cost is $O(1+2+...+|E|)$ or $O(|E|^2)$. Hence the total reduction can be achieved at a total cost of $O(n+|E|^2)$, which is polynomial in the input size.

It now remains to be proved that the sequence of steps mentioned above, indeed constitutes a transformation, i.e. that this sequence preserves the one-to-one correspondence between the solutions of the two problems. We observe the following two properties of the Dictionary **H** created by Algorithm Reduce.

(i) A pair of letters (α, β) appear in **exactly two words** in the entire dictionary. These words have the form $\alpha^m\beta^m$ and $\beta^m\alpha^m$. We shall refer to these words as Word(α, β) and Word(β,α) respectively.

(ii) There are exactly two words of the same length in the entire dictionary.

As a consequence of the above properties, the word pairs of the form $\alpha^m\beta^m$ and $\beta^m\alpha^m$ can contribute to exactly one collision, and that, only when the letters α and β are placed on the same key. These words contribute no collisions otherwise. To be more explicit:

$$\delta(\alpha^m\beta^m, \mathbf{H}, \Pi) = \begin{cases} 1 \text{ if } \alpha, \beta \in \mathbf{C}_i \text{ for some } i \le K \\ 0 \text{ otherwise} \end{cases}$$

Consider the expression:

$$\frac{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)}{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)+1} + \frac{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)}{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)+1}. \tag{5}$$

First of all, observe that the expression is meaningless if the words $\alpha^m\beta^m$ and $\beta^m\alpha^m$ are not in H. In other words, the expression is meaningless if the instance of SIMP2C did not possess the edge $<\alpha, \beta>$. In this case when (5) is meaningful, $w(<\alpha, \beta>) = 1$, for the underlying graph. However, with regard to ApproxKeyboard, both the terms in (5) are either 0 or $\frac{1}{2}$ simultaneously. They are zero if α, β are on different keys. Thus,

$$\frac{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)}{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)+1} + \frac{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)}{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)+1} = \begin{cases} 1 \text{ if } \alpha, \beta \in \mathbf{C}_i \text{ for some } i \le K. \\ 0 \text{ otherwise }. \end{cases}$$

Or equivalently,

$$\frac{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)}{\delta(\alpha^m\beta^m, \mathbf{H}, \Pi)+1} + \frac{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)}{\delta(\beta^m\alpha^m, \mathbf{H}, \Pi)+1} = \begin{cases} w(<\alpha, \beta>) \text{ if } \alpha, \beta \in \mathbf{C}_i \text{ for some } i \le K. \\ 0 \text{ otherwise }. \end{cases} \tag{6}$$

It now remains to be proved that the objective functions for the 2-Clustering problem and the reduced ApproxKeyboard problems are identical. The total number of collisions in the dictionary, $\Delta(\mathbf{H}, \Pi)$ can be written down using (3a) as:

$$
\begin{aligned}
\Delta(\mathbf{H},\Pi) &= \sum_{X \in H} \frac{\delta(X,\mathbf{H},\Pi)}{\delta(X,\mathbf{H},\Pi)+1} \\
&= \sum_{<\alpha,\beta> \in E} \frac{\delta(\text{Word}(\alpha,\beta),\mathbf{H},\Pi)}{\delta(\text{Word}(\alpha,\beta),\mathbf{H},\Pi)+1} + \frac{\delta(\text{Word}(\beta,\alpha),\mathbf{H},\Pi)}{\delta(\text{Word}(\beta,\alpha),\mathbf{H},\Pi)+1} \\
&= \sum_{\substack{\alpha,\beta \text{ in same cluster} \\ <\alpha,\beta> \in E}} w(<\alpha,\beta>) \\
&= \sum_{Q=1}^{2} \sum_{\substack{\alpha,\beta \text{ in same cluster } Q \\ <\alpha,\beta> \in E}} w(<\alpha,\beta>)
\end{aligned}
\tag{7}
$$

The reader will notice that (7) is exactly the objective function for the underlying 2-Clustering problem, which was specified by (4). Hence a solution to the ApproxKeyboard problem leads to a solution of the original instance of the 2-Clustering problem as well and the theorem is proved.

It is clear that our reduction is completely general and does not depend on the value of K. Thus the ApproxKeyboard is NP-complete for all values of K. •••

It must be noted that the above proof only established the NP-completenesss of ApproxKeyboard with the implicit assumption that the dictionary **H** was known to the algorithm which is solving the problem. If we work with the assumption that the dictionary **H** was not known to the algorithm, but rather that the algorithm was presented with a function or a table which returned the number of collisions for a particular partition Π, the problem would become much more difficult. We conjecture that when the problem is posed in this form, the computational complexity is provably exponential.

The reader will no doubt realize that the original optimization problem OptKeyboard is computationally at least as hard as its decision version ApproxKeyboard. Owing to the fact that the optimality of a partition reported as the solution to an instance of OptKeyboard **cannot be verified** in polynomial amount of time, the following result follows.

Corollary I.1

OptKeyboard is NP-hard.

III. PREVIOUS SOLUTIONS TO KEYBOARD OPTIMIZATION

Although the Keyboard Optimization Problem is NP-hard, in this paper we are interested in finding fast (approximate) solutions to it. The solutions which are presented here converge to a solution which is, hopefully, close enough to the optimal one. In this section we shall present the

only solution reported in the literature and in the subsequent sections we shall present a new learning automaton solution which is superior to the former.

The only reported solution to the keyboard optimization problem is an evolutionary method [15,16,21] based on a technique described by De Jong [14]. The latter is an adaptive algorithm based on genetic reproduction. It is represented using the standard feedback loop of classical control theory. This adaptive system is formally represented by De Jong as:

(i) A set **E** of environments in which the complex process must function.

(ii) A set **I** of all allowable control inputs which are available to an adaptive strategy for modifying the behavior of the complex process.

(iii) A process performance measure u: $\mathbf{E} \times \mathbf{I} \rightarrow \mathbf{R}$, where **R** represents the set of reals

(iv) A feedback function f: $\mathbf{E} \times \mathbf{I} \rightarrow \mathbf{R}^n$ providing the adaptive strategy with information about the process being controlled.

(v) A set **S** of adaptive strategies each of which attempts to use the knowledge obtained by the feedback interaction to improve the performance of the process.

De Jong's technique assumes that the adaptive system is operating in discrete time and that it has no *a priori* information about the set of environments. Using these assumptions De Jong presented an algorithm known as the reproductive plan, which belongs to an entire variety of genetic-based adaptive strategies. In the reproductive plan each point or each element $c \in \mathbf{I}$ is considered to be an individual organism. This organism is further represented uniquely by a string generated from some alphabet, where this string is the equivalent of the DNA of a real organism, and with the additional constraint that specific positions on the string take on different values exactly as genes take on different alleles. The performance measure is now defined as the fitness of the organism in a particular environment, and a more fit organism relates directly to a better control input.

The reproduction plan simply starts of by randomly creating a number of organisms which constitute the first generation. Each organism is assigned a fitness according to the performance measure and is allowed to reproduce. Various reproductive techniques are permitted and these include the mutative and inversive techniques. Independent of the reproductive technique used a superior (more fit) organism always possesses a higher probability of generating an offspring than an inferior (less fit) ones. The number of offsprings which an organism can have may be defined by its fitness factor divided by the average fitness of the population as follows :

$$O(c) = \frac{u(c,e)}{\sum_{c \in I} u(c,e) \div N} \quad \text{where} \quad N \text{ is the population size} \tag{8}$$

In general, an average organism will have one offspring, a more fit organism will have more offspring and a less fit organism might be barren. It should be noted that O(c) is a real number, and thus, if in a particular reproductive mechanism O(c) = 1.23, this could imply that the organism would certainly have one offspring, and furthermore that the second offspring will be created with a probability of 0.23. After all the organisms in a particular generation have reproduced, the offspring themselves form the next generation and the process continues, while the old generation dies off. As time progresses the population should become more fit on the average since the better organisms will take over a larger portion of the population. The latter statement, of course, assumes that the children of a more fit organism will on the average be more fit than the children of a less fit individual. Although this assumption is hard to prove, it is intrinsically tied in with the genetic DNA representation and is thus probably justified.

One of the simpler forms of reproduction involves crossover. For every organism in the population a mate is chosen randomly (or otherwise). The DNA strings of the two individuals are now cut and spliced together at some mutual point. For example, if the length of the DNA is L, a random point p , $1 \leq p < L$ is picked on the string. The child's DNA string will be formed using the first parent's DNA string positions from 1 through p spliced with the other parent's DNA string from positions p+1 through L. Observe, however, that a crossover operation by itself is not able to create individuals which are radically different from their parents. For example, if every organism in the population has the element α in the first position, a child produced as a result of crossover will also always have the same element in its first position. In order to overcome this limitation a second factor is introduced into the reproductive process, namely, mutation. A mutation operation is able to independently alter one or more of the gene values, and this is typically achieved randomly. The probability of a mutation occurring is usually small but this operation nevertheless guarantees that the set of organisms generated will eventually span the whole space.

The adaptation of this approach to the keyboard disambiguation problem presented earlier is quite simple. The set of environments **E** will be represented by a set of dictionaries, where one specific environment is a particular dictionary **H**. The set **I** now represents a set of all possible keyboards. The process performance function u is identically equal to the function $\Delta(\mathbf{H},\Pi)$. Finally, the set **S** of adaptive strategies contains only one adaptive strategy - that of reproduction involving the crossover and mutation operations. More explicitly, the DNA of the organism will be represented by a string of length $|\mathbf{A}|$, where **A** is the finite alphabet. In each position the index of the key on which the character currently appears will be used to represent the organism's (keyboard's) DNA string, and thus if 'a' appears on key 3 it is meant to represent that 'a' $\in \mathbf{C}_3$.

Levine and Minneman used this fundamental principle to minimize keyboard ambiguities. However, as opposed to our measure, $\Delta(\mathbf{H},\Pi)$, which evaluated the ambiguity of a particular keyboard assignment, their solution used an evaluation function based on a bigram metric. The

particular version of the algorithm described above is not monotonic in its optimization of organisms. An optimized modification of this technique which exhibits monotonic properties has been described elsewhere [27,36] and has been used as a benchmark against which our learning automaton solution can be compared. Although in [15,16,21] the authors refer to various results obtained for the disambiguation of text, we have failed in our attempts to duplicate their results. The discrepancy was probably due to the differences between their bigram statistics and ours.
We are currently working on a new evolutionary method [27,36] which is distinct from the ones described in [15,16,21].

IV. AUTOMATA SOLUTIONS TO KEYBOARD OPTIMIZATION

Our solution to the problem involves the use of stochastic learning automata. 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 [13,22,23]. Since the literature on learning automata is extensive, we refer the reader to a paper by Narendra and Thathachar [23] and two excellent books on the field by Lakshmivarahan [13] and Narendra and Thathachar [22] 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 evolve with time. Some examples of the former type are the Tsetlin, Krinsky and Krylov automata [23,31]. Although the latter automata are called variable structure stochastic automata, because their transition and output matrices are time varying, in practice, they are merely defined in terms of action probability updating rules [13,23]. Although throughout this paper we will only be considering Fixed Structure Stochastic Automata, automata with a variable structure are generally much faster in their convergence. We are currently investigating the use of automata of the latter class to solve the Keyboard Optimization Problem.

IV.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 [23].

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 $\{c_i\}$ (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

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

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.

In the case under consideration, if the set of actions is the set of all possible partitions of **A**, the penalty function associated with the actions is a deterministic function. For a particular partition Π, this function is indeed the associated number of collisions $\Delta(\mathbf{H},\Pi)$.

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.

IV.2 The Krinsky Automaton

The reward characteristics of the automaton solution which we shall present is akin to that of a well known automaton due to Krinsky [31]. The K-action Krinsky automaton, $Z_{KN,K}$, has KN states $\{\phi_1, \phi_2, ..., \phi_{KN}\}$ and chooses one of the K actions $\{\alpha_1, \alpha_2, ..., \alpha_K\}$. N is called the "Depth" of the automaton. Explicitly, $Z_{KN,K}$ is defined as follows :

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

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

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

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 $Z(\cdot\,,\cdot)$ map is **deterministic** and is described as below :

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

$$Z(\phi_i, 0) = \phi_{(j-1)N+1} \qquad \text{if} \quad (j-1)N + 1 \le i \le jN \tag{10}$$

(2) If $\beta = 1$ (it gets an unfavorable response), the automaton moves towards the boundary state -- ϕ_{jN} if a_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 all $1 \le j \le k$.

$$\begin{aligned} Z(\phi_i, 1) &= \phi_{i+1} \quad \text{if} \quad (j-1)N + 1 \le i < jN \\ &= \phi_{((j+1)N) \bmod KN} \quad \text{if} \quad i = jN,\ j \ne K-1 \\ &= \phi_{KN} \quad \text{if} \quad i = (K-1)N. \end{aligned} \tag{11}$$

We now state the following theorem which was proved two decades ago [31].

Theorem II

The $Z_{KN,K}$ automaton is ε-optimal in **all** random environments. •••

IV.3 The Keyboard Learning Automaton

Closely related to the problem which we are studying is the Equi-partitioning problem which can be described as follows. Let $\{A_1, ..., A_W\}$ be a set of W objects. In the Equi-partitioning problem we intend to partition this set into R classes $\{P_1, ..., P_R\}$ of equal size. Further, we intend to partition them in such a way that the objects that are accessed (used) more frequently together lie in the same class. Note that to render the problem non-trivial, we assume that these joint access probabilities are unknown. Applications of the object partitioning problem are found in information retrieval, attribute partitioning, record clustering and in distributed file allocation.

Various adaptive solutions to the Equi-partitioning problem have suggested in the literature. The best known one in the early 70's was due to Hammer *et. al.* [8] which was essentially a hill-climbing solution to the underlying optimization problem. Yu *et. al.*[35] proposed a superior algorithm to solve this problem called the Basic Adaptive Method. The best known solution of the Equi-partitioning problem uses learning automata [25]. In [25], a new automaton called the Object Migrating Automaton was designed by Oommen *et. al.* which converged an order of magnitude faster than the Basic Adaptive Method. The reader who is aware of the work done in equi-partitioning [25] will find the following solution to the Keyboard Optimization Problem to be an interesting generalization of the Object Migrating Automaton.

The learning automaton solution presented in this paper is called the Keyboard Learning Automaton (KLA). We define the Keyboard Learning Automaton (KLA) as a 6-tuple as below :

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

(i) **A** is the finite 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 certain class into which the elements of **A** must fall.

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

(v) Q, the transition function is quite involved and will be explained in detailed 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 $1 \leq k \leq K$.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, in this case, we have **all** the characters of **A** moving around in the states of the machine. If the character $a_i \in A$ is in action α_k, it signifies that this character will be in C_k, the set numbered k. Observe that if the states

occupied by the characters are given, the actions chosen by the characters of **A** can be trivially obtained using (9), and these specify the partition currently chosen by the automaton.

Let $\xi_a(n)$ be the index of the state occupied by $a \in \mathbf{A}$ at the n^{th} time instant. Also let $\Xi(n)=\{\xi_a(n) \mid a \in \mathbf{A}\}$. Based on $\Xi(n)$, let $\Pi(n)$ be the current partition which the automaton decides. Observe that, as mentioned above, this partition is trivially obtained by repeatedly using (9) to the various elements of $\Xi(n)$ to determine the action chosen by each character $a \in \mathbf{A}$. Using this notation we shall now describe the transition map of the KLA. However, in what follows, in the interest of brevity, we shall omit the reference to the time instant 'n'.

Initially, the KLA begins its learning process by starting from an arbitrary partition $\Pi(0)$, and this initial partition may be obtained randomly or otherwise.Thereafter the following process continues until the user is satisfied with the partition.

Based on the current partition, Π, the mapping of **H** due to Π, $\mathbf{H}^{\Pi}$, is computed.The words in **H** are now processed in a sequential manner. On processing a word $X \in \mathbf{H}$ the mapped dictionary, $\mathbf{H}^{\Pi}$, is searched and the number of collisions $\delta(X_O,\mathbf{H},\Pi)$ is computed using (2). If the number of collisions is zero, obviously X can be encoded using Π uniquely and unambiguously. In this case the partition Π is rewarded. Otherwise, the representation is ambiguous and the partitioning Π is penalized. The operations involved with rewarding and penalizing the partitioning Π are described below.

The Reward operation is quite simple. Every letter in the processed word, X, is moved to the internal state of its corresponding action as per the transition map of the Krinsky automaton (see Figure Ia).

As opposed to this, the Penalty operation of penalizing a partition Π is more intricate and it involves penalizing a set of characters called the set of colliding characters. This set is essentially the set of characters in X which are "responsible" for the collisions. To explicitly define this set, let $X=x_1x_2...x_N$ and $W=w_1w_2...w_N$ be two words in **H** such that $\Pi(X) = \Pi(W)$. Then **T**, the set of colliding characters in X is defined as follows :

$$\mathbf{T} = \bigcup_{W \in \mathbf{H}, \Pi(W)=\Pi(X)} \{x_i \mid x_i \neq w_i\}$$

For example, if X="ace", and it collides with the strings W and V, where W="age" and V="ago", **T**, the set of colliding characters in X is {'c','e'}. The elements of **T** will be moved as a consequence of the penalty operation.

Once the set **T** is identified, the individual characters in it are each moved towards the boundary state from one state to its adjacent one until at least one character is at the boundary state. One of the colliding characters in the boundary state is randomly chosen and moved to the boundary state of another randomly chosen action with an attempt to ameliorate the present

partitioning $\Pi(n)$ (see Figure Ib). The total number of collisions for the current partitioning is evaluated, and if the new partitioning causes less collisions or an equal number of collisions than the original partitioning, this becomes the current partitioning, $\Pi(n+1)$.

To catalyze the convergence it is sometimes advantageous to use *a priori* information and specify additional restrictions on the set of possible partitions. For example, one such restriction is to constrain every key to have no more than four characters and no less than two characters. In such a case, in the event of a penalizing operation, when the colliding character moves to a new action the other characters associated with the new action may have to be moved if this constraint is violated. Thus, if the user imposes such a restriction and this restriction is violated on moving the colliding character, a random character associated with the new action is swapped so as to move to the boundary state of the original action of the colliding character (see Figure II).

The Keyboard Learning Algorithm (KLA) which has been informally described above is formally presented below in ALGORITHM KLA 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 **A**.

ALGORITHM KLA

Notation :

The KLA has N states per action, and ξ_a is the index of the state associated with the character $a \in A$. Thus ξ_a satisfies $1 \leq \xi_a \leq KN$, and furthermore if $a \in A$ 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 $\Xi = \{\xi_a | a \in A\}$ the partition chosen by the KLA is computed trivially using (9). We assume that the KLA has access to a function ComputePartition which has as its input the set Ξ and yields for its output the partition Π. We assume that the number of characters per key is unrestricted by the user.

Input :

(i) An initial partition Π. The characters are initially placed on the boundary states of the actions of the KLA. Thus if $a \in A$ is assigned to the key k, then ξ_a is assigned the value kN.

(ii) A finite dictionary **H**.

(iii) A stream of words from **H**. The words are processed one at a time.

Output :

A final partition Π^* obtained as a result of the learning process.

Assumptions :

(i) The algorithm has access to functions FindNumberOfMatches and TotalCollisions. Given a word X and a set of character assignments, Ξ, FindNumberOfMatches yields $\delta(X,H,\Pi)$. TotalCollisions computes $\Delta(H,\Pi)$ for Ξ.

(ii) The algorithm has access to a function CollidingCharacters, which has as its parameters a given string X, the dictionary **H**, and the current states occupied by the characters, Ξ. The function first computes H^{Π} and subsequently evaluates the set of words $\{X_j\}$ for which $\Pi(X_j) = \Pi(X)$. The set of characters in X which cause the collisions with the various strings in $\{X_j\}$ are then evaluated.

```
MainAlgorithm
Begin
        GetInitialAssignment (Ξ)      /* Ξ is the current state assignments */
        Π := ComputePartition (Ξ)  /* Π is the current partition obtained using    (9)*/
        Repeat
            ReadWord (X)
            Matches := FindNumberOfMatches (X, H, Π)
            If (Matches = 0) Then Ignore X          /* X is not an element of H */
            Else
                If (Matches = 1) Then               /* X is uniquely mapped by Π */
                    Reward ( X, Ξ )
                Else
                    Punish ( X, H, Ξ )
                EndIf
            EndIf
            Π := ComputePartition (Ξ)
        Until Satisfied
        Π* := Π
End /* MainAlgorithm */.

Procedure Reward (X, Ξ)
        For j := 1 to |X| Do   /* process all characters  in X */
            If ( ξ_{x_j} mod N ) ≠ 1 Then
                ξ_{x_j} := (( ξ_{x_j}-1) div N) * N + 1          /* Move ξ_{x_j} to most internal state */
        EndFor
End Procedure  /* Reward */

Procedure Punish (X, H, Ξ)
        PreviousΔ := TotalCollisions (H, Ξ )
        T := CollidingCharacters (X, H, Ξ )
        Repeat
            For all a ∈ T Do  /* Do for all colliding characters in X */
                If ( ξ_a mod N ) ≠ 0 Then          /* a is not in boundary state */
                        ξ_a = ξ_a + 1              /* Move a towards boundary state */
            EndFor              /* Till at least one element is at the  boundary */
        Until some elements of T  are in their boundary states.
        b := Random element in T in boundary state.
        Repeat                                     /* b in boundary state is to be moved */
            k := Random(1, K)
            ξ_b :=  kN              /* Move b to boundary state of α_k */
        Until TotalCollisions (H, Ξ) ≤ PreviousΔ.
End Procedure . /* Punish */
END ALGORITHM KLA
```

The learning property of Algorithm KLA now follows.

Theorem III.

Algorithm KLA is both Expedient and Almost Absolutely Expedient.

Proof :

The proof of this result is very straightforward and actually obvious especially because the evaluation function $\Delta(\mathbf{H},\Pi)$ is deterministic. Observe that given the current partitioning $\Pi(n)$, the KLA assigns $\Pi(n+1)$ to have the value $\Pi(n)$ unless it has found a superior partitioning on moving a colliding character. Thus the total number of collisions caused by the partitions chosen by the KLA is strictly non-increasing and the result follows. •••

Remarks

(i) In the above description we have assumed that the user has no additional constraint on the number of characters per key. As mentioned earlier, often, to catalyze the convergence of the automaton it is advantageous to specify additional restrictions on the set of possible partitions. If the user requires that such a restriction is necessary, in the Procedure Punish, whenever the constraint is not satisfied the character in α_k closest to the boundary is moved to the boundary state of the action from which the colliding character b is moved (see Figure II). This is analogous to the operation used to moving objects in the Object Migrating Automaton for the Equi-partitioning problem [25].

(ii) Observe that the result proved above, that the automaton is Almost Absolutely Expedient, is weaker than we would have liked had we been able to design a machine which is Absolutely Expedient. However, considering the fact the problem is inherently NP-Complete, this solution, although not as strong as we would like, is both fast and accurate. Indeed, on the basis of our experimental results we conjecture that the machine is also ε-optimal.

An example will help clarify the operation of Algorithm KLA.

Example I

Let **A** be the English alphabet and the dictionary **H** consist of eight words and be:

H= {ace, ale, ago, each, eels, ape, big, dig}.

Also let $\Pi(n)$ be the partition :

$\Pi(n)$ = ((aeo), (cgl), (bd), (fgr), (km), (inpz), (tuy), (hsv), (jwx)).

Then, the mapping of the various words in **H** is the set

{121, 121, 121, 1128, 1128, 161, 362, 362}.

Notice that if we extract the duplicate entries in the latter set, we get H^{Π} to be :

H^{Π} = {121, 1128, 161, 362}.

Thus, $\Delta(H,\Pi)$ has the value 4.

On processing X=ace it is observed that there are two collisions, and that the set of colliding characters are {'c', 'e'}. Let us assume that 'c' is closer to the boundary state than 'e'. Since no restrictions are placed on the number of character per key, the character 'c' is now moved to a random action. If the random action picked is 6, the character 'c' will now be moved action α_6. Effectively, the character 'c' simply moves from key 2 to key 6 to form a temporary partition Π', where,

Π' = ((aeo), (gl), (bd), (fgr), (km), (cinpz), (tuy), (hsv), (jwx)).

Observe that $H^{\Pi'}$ is the set :

$H^{\Pi'}$= {161, 121, 1168, 1128, 161, 362}.

Since $\Delta(H,\Pi')$ is 2, the partition $\Pi(n+1)$ is assigned the value Π'.

If the user had placed an additional restriction that the number of characters per key had to be between 2 and 4 inclusive, on moving 'c' to α_6 a random character is chosen from the characters choosing α_6 - {'i', 'n', 'p', 'z'}. If 'p' was this random character, 'p' is moved to α_2, the original action of 'c' to yield the partitioning Π (n+1) to be :

$\Pi(n+1)$ = ((aeo), (glp), (bd), (fgr), (km), (cinz), (tuy), (hsv), (jwx)). •••

V. EXPERIMENTAL RESULTS

The Evolutionary method described in Section II and the KLA were compared experimentally primarily to compare their rates of convergence and to evaluate their relative efficiencies. In all the cases the alphabet used consisted of the English alphabet. The dictionary used consisted of about 1100 words compromising primarily of the 1023 most common English words [3] and various words encountered in computer science.

Various simulations were conducted to evaluate the performance of the algorithms under a variety of constraints. In each simulation ten parallel experiments were conducted so as that an accurate ensemble average of the results could be obtained. The size of the set K was also a parameter to the programs, but in all the results we report below we assume that K is 9. Also in each case the original keyboard was chosen randomly.

The experiments were primarily of two types. In the first type of experiments the keyboard was to be learnt without any additional user's constraints. Unlike these, in the second it was assumed that every key on keyboard was constrained to have between two and four characters (inclusive). The results of the experiments are explained below, and a catalogue of these results is given in Table I.

Table I reports the average number of collisions in the dictionary as a function of the number of keyboards that are examined for various algorithms. For the case of the KLA in which the

number of characters per key was unrestricted, the average initial number of collisions was 65.5. This figure fell to 28 after examining but 30 keyboards. Before 600 keyboards were examined the average number of collisions fell to 5.5. Indeed, the method examined 160 keyboards to reduce the number of collisions by 90%. The power of the scheme is remarkable considering the fact that the problem is inherently NP-Complete and the number of distinct keyboards in the solution space is combinatorially explosive.

To compare our scheme with the evolutionary method described in Section II, in Table I we have also reported the average number of collisions in the dictionary as a function of the number of keyboards that are examined by the optimized version of Levine and Minneman's solution [15,16,21]. In this case too the average initial number of collisions was approximately 65.5. An exact value for this figure cannot be obtained in the case of evolutionary algorithms because the latter require an initial starting population of keyboards as opposed to a single initial keyboard required by the KLA. Thus the average initial value which we have reported is exactly the same as we have in the case of the KLA. A starting population of 15 keyboards were used to trigger the evolutionary process. The average number of collisions fell to 40 after examining 30 keyboards, and this diminished only to a terminal value of 23 when the experiment was terminated after 600 keyboards were examined. The method examined 390 keyboards to reduce the number of collisions by 90% and even at this juncture the average number of collisions was as high as 27.5.

Apart from reporting the initial and the final values of $\Delta(\mathbf{H},\Pi)$ we have also plotted the average variation of $\Delta(\mathbf{H},\Pi)$ as a function of the number of keyboards examined. This plot is shown in Figure III. The advantage of the KLA over the method due to Levine and Minneman is obvious.

It was observed that the convergence of the KLA could be hastened by constraining every key to have between two and four characters. The effect of imposing such a constraint on the convergence is also reported in Table I.The advantage gained by using this *a priori* information and constraining the number of keys to be between two and four is also obvious.

Finally, to present a "worst case" scenario, the KLA was assigned the task of learning the optimal keyboard after starting from the worst possible initial keyboard. In this case, all the characters in the alphabet were initially assigned to the same key. Thus, initially, all the words of the same length collided with each other as they all possesed the same representation. From this configuration the KLA processed the words sequentially. The results we have obtained are truly remarkable. The initial number of collisions was 1067. The average number of collisions fell to the amazing low value of 21.8 after examining 60 keyboards. The average final number of collisions was 5.7 when the experiments were terminated, i.e., after examining 600 keyboards. The plot of the number of collisions as a function of the number of keyboards examined is given in Figure IV. Also in Figure Va we have given a sample initial worst case keyboard in which all the keys are

assigned to the digit 5. In Figure Vb the final assignment of the characters which yields 9 collisions is shown. The power of the scheme is obvious.

Algorithm	Mean Initial $\Delta(H, \Pi)$	Mean Final $\Delta(H, \Pi)$	MTC
EVOL-M	65.5	23.0	390
KLA-U	65.5	5.5	160
KLA-R	58.9	4.9	150

Table I: Comparison of the three algorithms used for keyboard optimization. The results were obtained as an average of 10 experiments with random initial keyboards.

Notation:

H : 1081 words including the 1023 most common English words [3].

EVOL-M: Optimized version of Levine and Minneman's scheme [15,16,21]. See the text for the details of how the initial value was obtained.

KLA-U: Keyboard Learning Automaton with N = 10. In this case the number of characters per key is unrestricted.

KLA-R: Keyboard Learning Automaton with N = 10. In this case the number of characters per key is between 2 and 4 inclusive.

The results that we have presented above are typical. Hundreds of other experiments have been conducted using subsets of the above dictionary and by varying the size of the set of keys, **K**. In the interest of brevity these experimental results have been omitted here. They are found elsewhere [36].

V. CONCLUSIONS

In this paper we have studied the problem of optimizing a keyboard for a particular finite dictionary, **H**, defined as a subset of the words over a finite alphabet, **A**. The letters of **A** are assigned to elements of a set **K**, which is the set of integers between unity and K. More explicitly, associated with every element $i \in \mathbf{K}$ is a set $\mathbf{C}_i$ such that the union of all $\mathbf{C}_i$'s is **A** and all pairwise intersections of $\mathbf{C}_i$'s are null. The aim of the optimization problem is to compute the set $\{\mathbf{C}_i\}$ so that if every character in **A** is replaced by the index of the set in which it belongs, the transformed version of **H** has the minimum number of collisions.Thus we aim to compute the sets $\mathbf{C}_i$ so as to minimize the ambiguity in such a keyboard assignment process. In this paper we have proved the result that the problem is inherently NP-hard. Subsequently, the only known solution to the problem has been described and a new learning automaton solution to the problem has been proposed. Experimental results demonstrating the power of the automaton have been presented.

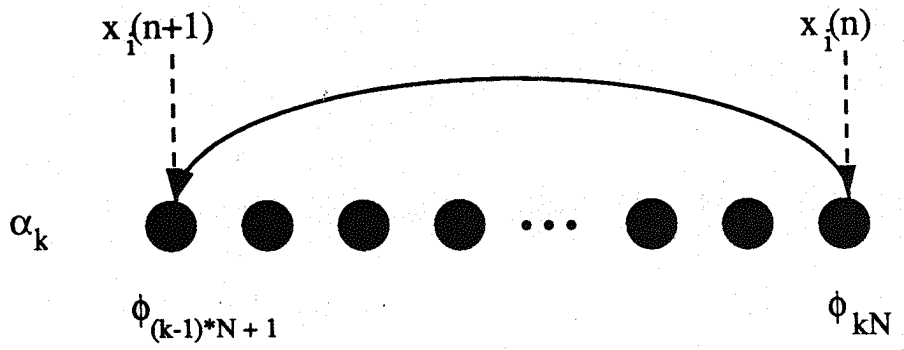
ACKNOWLEDGEMENT

The authors wish to thank Prof. M. D. Atkinson for initial discussions on the subject of NP-completeness.

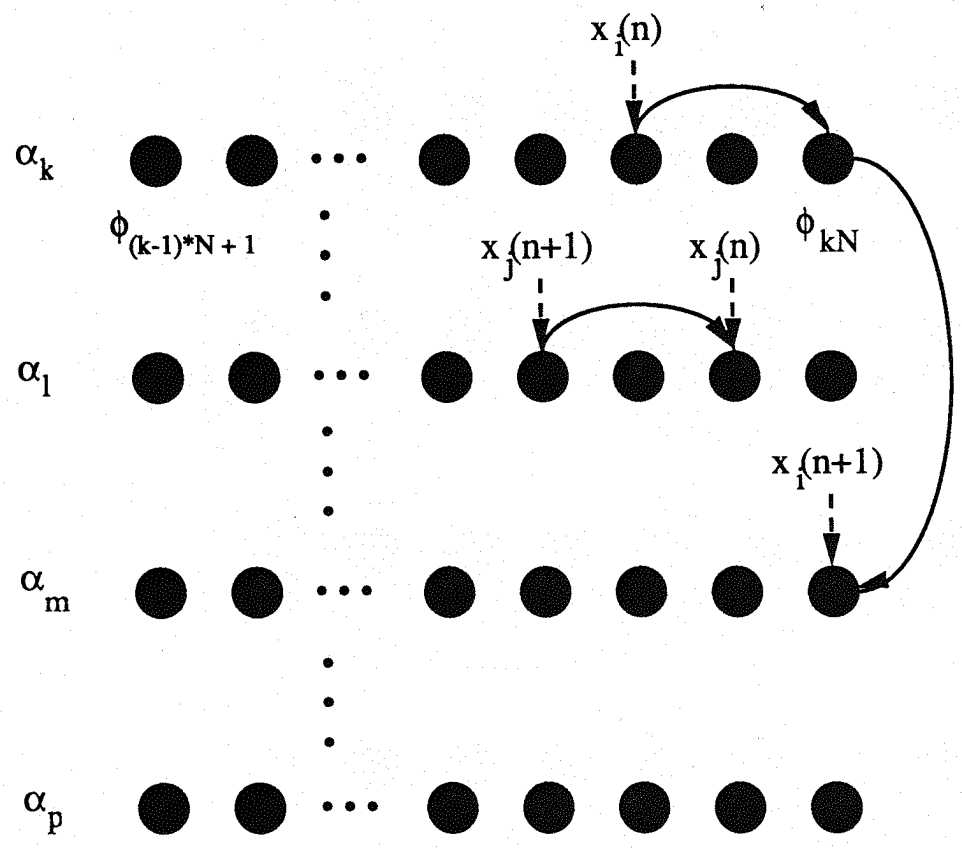
REFERENCES

1. Aho, A. V., Hirschberg D.S., Ullmann J. R., "Bounds on the Complexity of the Longest Common Sub-Sequence Problem", *J-ACM*, , 1976, pp.1-12.
2. Brucker, P, "On the Complexity Of Clustering Problems", in R.Henn, B.Korte and W. Olettii (eds.), *Optimierung und Operations Research*, Lecture Notes in Economics and Mathematical Systems, Berlin, 1978.
3. Dewey, G., *Relative Frequency of English Speech Sounds*, Cambridge, MA, Harvard Univ. Press, 1923.
4. Forney, G. D., "The Viterbi Algorithm", *Proceedings of the IEEE*, , 1973, Vol. 61.
5. Garey, M.R. and Johnsion, D.S., "*Computers and Intractability: A Guide to the Theory of NP-Completeness*", W.H. Freeman and Co., 1979.
6. Garey, M.R.,Johnson, D.S. and L. Stockmeyer, "Some Simplified NP-Complete Graph Problems", Theoretical Computer Science, Vol.1, pp. 237-267.
7. Hall, P.A.V., and Dowling G.R., "Approximate String Matching", *Computing Surveys*, 1980, pp. 381-402.
8. Hammar, M., and Chan, A., "Index Selection in a Self-adaptive Database Management System, *Proc. of the ACM SIGMOD Conference*, 1976, pp.1-8.
9. Hirschberg, D.S., "A Linear Space Algorithm for Computing Maximal Common Subsequences", *C-ACM*, , 1975, pp. 341-343.
10. Hunt, J.W., and Szymanski, T.G., "A Fast Algorithm for Computing Longest Common Subsequences", *C-ACM*, , 1977, pp. 350-353.
11. Kashyap, R.L., and Oommen, B.J., "An Effective Algorithm for String Correction using Generalized Edit Distances -I. Description of the Algorithm and its Optimality", *Information Sciences*, , 1981, No. 2, pp. 123-142.
12. Kashyap, R.L., and Oommen, B.J., "Probabilistic Correction of Strings", *Proc. of the IEEE Int. Conf. on Pat. Recog. and Image Processing*, 1982 pp. 28-33.
13. Lakshmivarahan, S., *Learning Algorithms Theory and Applications*, Springer-Verlag, New York, 1981.

14. De Jong, K., "Adaptive System Design: A Genetic Approach", *IEEE Transactions on Systems, Man, and Cybernetics*, 1980, pp. 566-574.
15. Levine, S. H., "An Adaptive Approach to Optimal Keyboard Design for Nonvocal Communication", *Proc. of the International Conference of Cybernetics and Society*, 1985, pp. 334-337.
16. Levine, S. H., Minneman, S.L., Getschow, C.O., and Goodenough-Trepagnier, C., "Computer Disambiguation of Multi-Character Text Entry : An Adaptive Design Approach", *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*, 1986, pp. 298-301.
17. Lu, S.Y. and Fu, K.S., "A Sentence-to-Sentence Clustering Procedure for Pattern Analysis", *Proc. of the IEEE COMPSAC Conference*, , 1977, pp. 492-498.
18. Maier, D., "The Complexity of Some Problems on Subsequences and Supersequences," *J-ACM*, , 1978, pp. 322-336.
19. Masek, W.J. and Paterson, M. S., "A Faster Algorithm Computing String Edit Distances", *Journal of Comp. and Sys. Sciences*, , 1980, pp. 18-31.
20. Minneman, S. L., "A Simplified Touch-Tone Telecommunication Aid for Deaf and Hearing Impaired Individuals", *Proc. of the 8h. Annual RESNA Conference*, Memphis, 1985, pp.209-211.
21. Minneman, S. L., "Keyboard Optimization Technique to Improve Output Rate of Disabled Individuals", *Proc. of the 9th. Annual RESNA Conference*, Minneapolis, 1986, pp.402-404.
22. Narendra, K.S., and Thathachar, M.A.L., *Learning Automata*, Prentice-Hall, 1989.
23. Narendra, K.S., and Thathachar, M.A.L., "Learning Automata -- A Survey", *IEEE Trans. on Syst. Man and Cybernetics*, 1974, pp.323-334
24. Neuhoff, D. L., "The Viterbi Algorithm as an Aid in Text Recognition", *IEEE Trans. Information Theory*, 1975, pp. 222-226.
25. Oommen, B. J., and Ma, D. C. Y., "Deterministic Learning Automata Solutions to the Equi-Partitioning Problem", *IEEE Transactions on Computers*, 1988, pp.2-14.
26. Oommen, B.J., and Ma, D.C.Y., "Fast Object Partitioning Using Stochastic Learning Automata". *Proc. of the 1987 International Conference on Research and Development in Information Retrieval*, New Orleans, 1987, pp.111-122.
27. Oommen, B.J., and Zgierski, J., "Keyboard Optimization for Minimizing Ambiguities Using Evolutionary Algorithms". In Preparation.
28. Sankoff, D., and Kruskal, J.B., *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison Wesley, 1983.
29. Shinghal, R., and Toussaint, G.T., "Experiments in Text Recognition with the Modified Viterbi Algorithm", *IEEE Trans on Pat. Anal. and Mach. Intel.*, 1979, pp. 184-192.
30. Srihari, S., *Computer Text Recognition and Error Correction*, IEEE Computer Society Press, 1984.
31. Tsetlin, M.L., *Automaton Theory and the Modelling of Biological Systems*, New York and London, Academic, 1973.
32. Viterbi, A. J., "Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm", *IEEE Trans. on Information Theory*, 1967, pp.260-26.
33. Wagner, R. A., and Fisher, M.J., "The String to String Correction Problem", *J-ACM*, , 1976, pp. 168-173.
34. Wong, C. K. and Chandra, A.K., "Bounds for the String Editing Problem, *J-ACM*, , 1976, pp. 13-16.
35. Yu, C.T., Siu, M.K., Lam, K., and Tai, F., "Adaptice Clustering Schemes : General Framework", *Proc. of the IEEE COMPSAC Conference*, 1981, pp.81-89.
36. Zgierski, J., *Application of Learning Automata to Various Optimization Problems*, Honours Project, School of Computer Science, Carleton University, Ottawa.

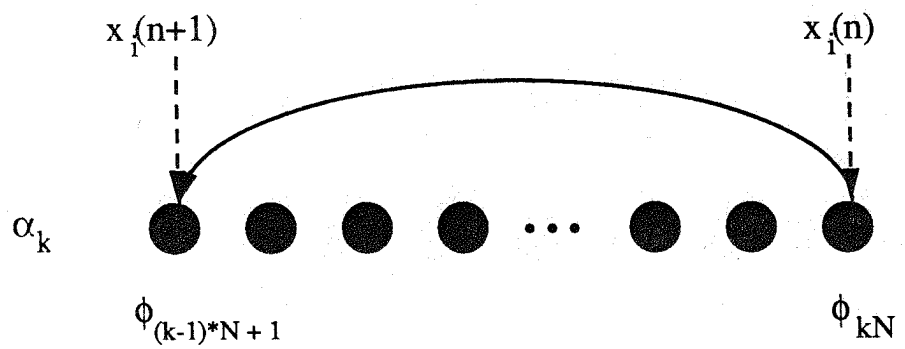


(a) Every character x_i of X is rewarded. x_i is in α_k.

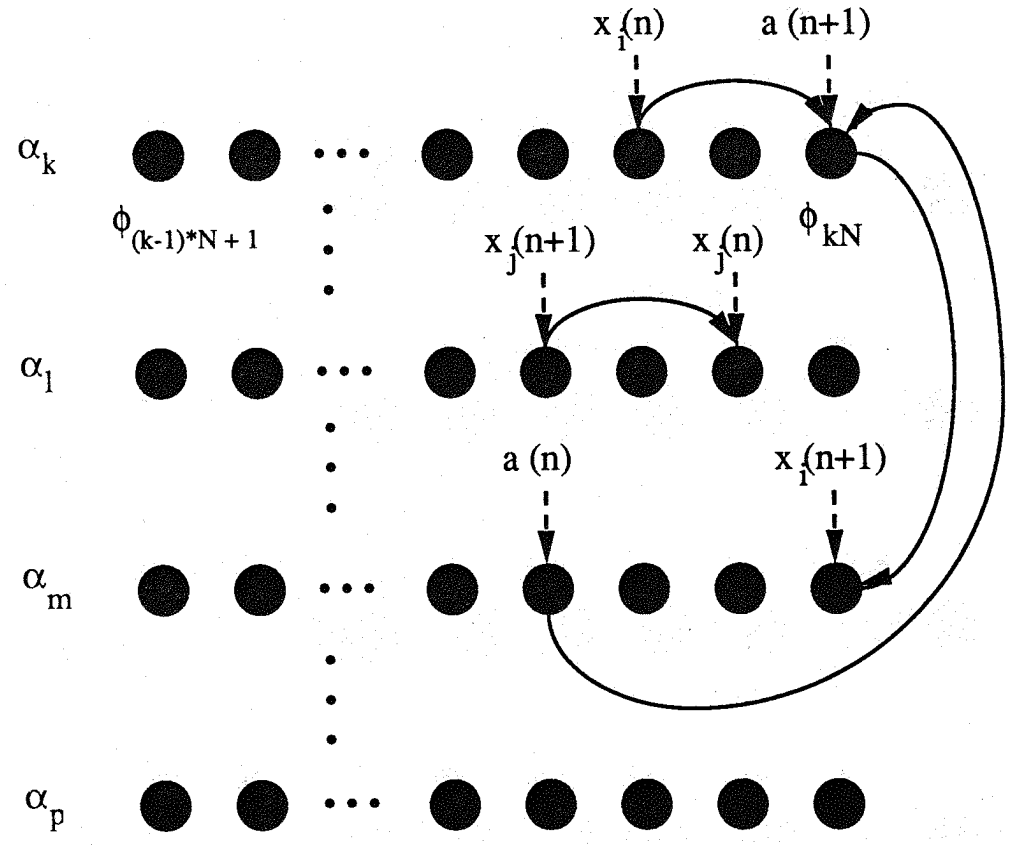


(b) Colliding characters x_i and x_j of X are penalized. x_i is in α_k, x_j in α_l. x_i moves to a random action α_m and to its boundary state if the number of collisions decreases.

Figure I: The transition map of the KLA. A word $X = x_1x_2...x_N$ is processed. In (a) the transition on being rewarded is depicted. In (b) the transition on being penalized is shown. The number of characters per key is unrestricted.



(a) Every character x_i of X is rewarded. x_i is in α_k.



(b) Colliding characters x_i and x_j of X are penalized. x_i is in α_k, x_j in α_l. x_i moves to a random action α_m and to its boundary state if the number of collisions decreases. a is a random character moved to α_k to satisfy the user's restriction.

Figure II: The transition map of the KLA. A word $X = x_1x_2...x_N$ is processed. In (a) the transition on being rewarded is depicted. In (b) the transition on being penalized is shown. The number of characters per key is restricted.

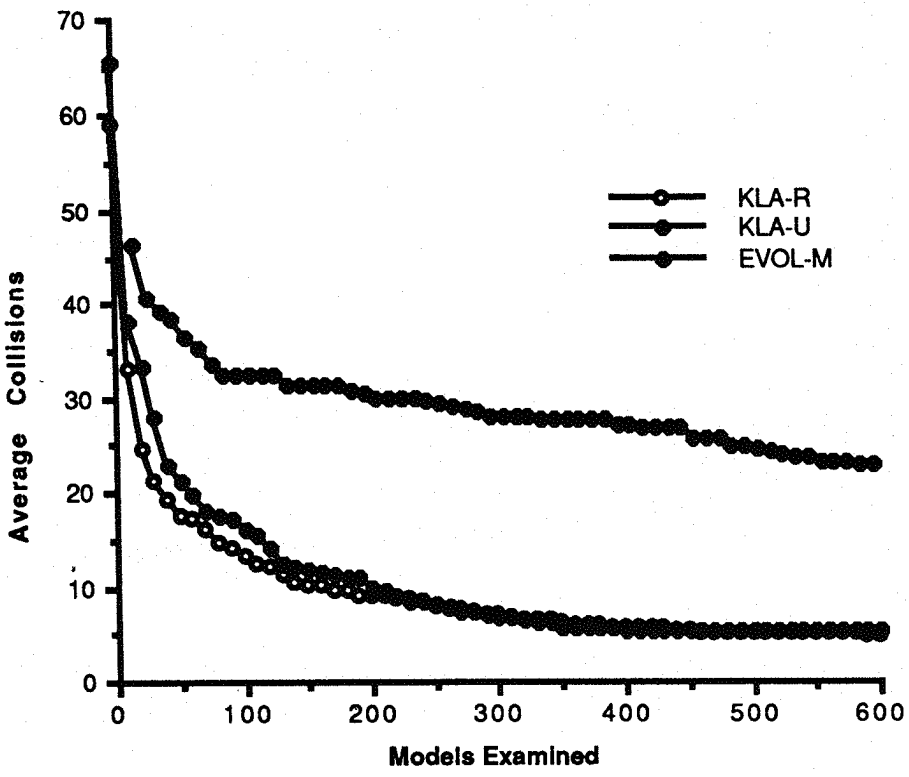


Figure III: Rate of convergence of the three algorithms used for keyboard optimization. The results were obtained from the average of 10 experiments with random initial keyboards. The notation is the same as in Table I.

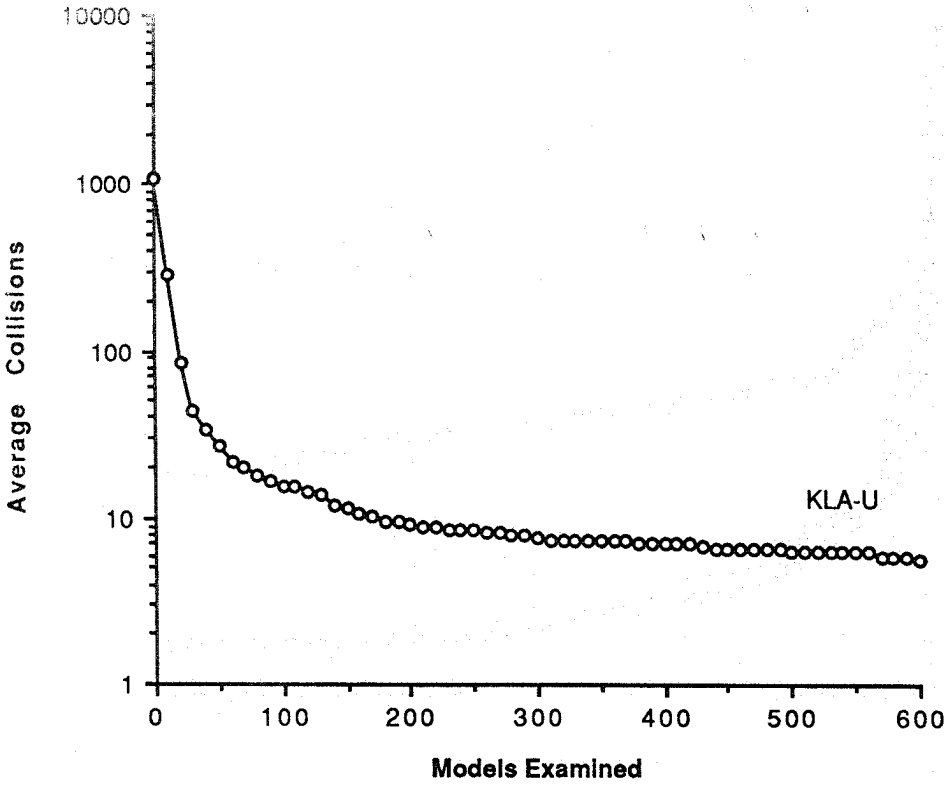


Figure IV: Worst case scenario featuring the rate of convergence of the KLA algorithm for keyboard optimization with the initial keyboard having all the characters on one key.

1	2	3
4	5 ABCDEFGHIJ KLMNOPQRS TUVWXYZ	6
7	8	9

(a) Initial configuration of the Keyboard: All the letters of the alphabet are assigned to the same key 5.

1 HM	2 KT	3 GPUY
4 IS	5 JLOQVXZ	6 ABR
7 CN	8 EW	9 DF

(b) The "Final" Keyboard obtained after 600 models were examined: This keyboard yields 9 collisions.

Figure V: An illustration of the initial and final Keyboards obtained by Algorithm KLA, in a particular run when the initial keyboard is the worst case scenario and all the keys are initially assigned to the same digit.

APPENDIX

We present an example of the reduction performed by Algorithm Reduce. The input graph consists of 6 nodes and 8 edges as depicted in Figure VI. When Algorithm Reduce is applied to the graph shown in Figure VI, the outputs from Algorithm Reduce, which serve as input to ApproxKeyboard are:

A = {a,b,c,d,e,f}

H = {ab, ba,
a^2c^2, c^2a^2,
a^3d^3, d^3a^3,
b^4d^4, d^4b^4,
b^5f^5, f^5b^5,
c^6f^6, f^6c^6,
d^7e^7, e^7d^7,
f^8e^8, e^8f^8}

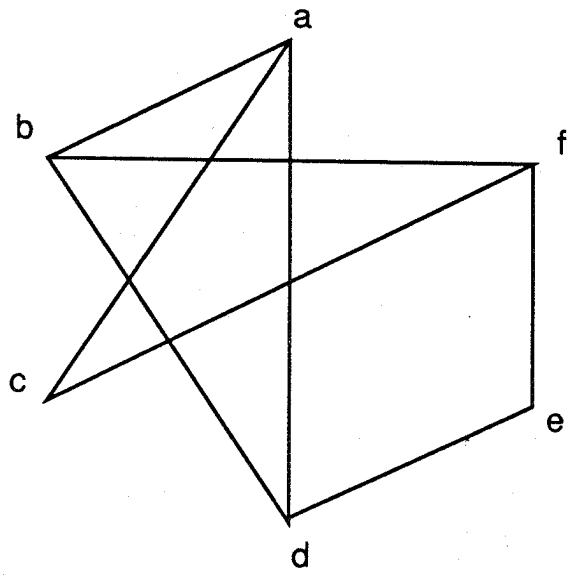


Figure VI: Input Graph G=(V,E)