Implementation of Soft Keyboards for Indic Languages

Albert Brouillette

***Abstract*—The arrangement of letters on a keyboard determines the ease and efficiency of text input. On devices with limited space, the keyboard layout can have an even greater impact on effective data entry. Much research has been done proposing techniques for optimizing Roman-alphabet keyboards, including some for small devices. However, the large number of letters in other alphabet systems makes this problem more complex. Some alphabets can have as many as three times the number of characters as English. This paper investigates techniques for reducing the size of Indic keyboards while creating an optimized layout. To facilitate this, we propose the implementation of machine learning techniques such as the genetic algorithm in developing optimized soft keyboards for mobile phones.**

***Index Terms*—Soft keyboards, Indic languages, optimization, genetic algorithms, Android development.**

I. Introduction

As technology has progressed over the past several decades the world has gained the ability to process a large amount of information in a short time. With these developments comes the need for keyboards that allow users to input text more efficiently. This need for faster keyboards has been the focus of much research in recent years. While much progress has been made in the development of optimized keyboards for Roman-alphabet based languages, there has been little work done with languages based on other alphabets. As a result, many Indic languages have only rudimentary, unoptimized keyboards. For the most part, these keyboards have been inefficient and difficult to use.

The use of soft keyboards would allow users to input information without the actual existence of a physical keyboard. The concept of the soft keyboard is that data can be input through mouse clicks on an on-screen keyboard, or through a touch-screen device[1]. In addition, the virtual keyboards can be mapped to receive input from a standard physical keyboard. The development of efficient soft keyboards is becoming an increasingly attractive alternative for numerous applications.

Many of the current keyboards for Indic languages have been developed around some form of the QWERTY-based layout. While these keyboards can be functional, the greater number of characters in the Indic languages make them cumbersome and inefficient. Since soft keyboards do not have any physical limitations, they can easily be modified and programmed to reach a much more reasonable solution. These soft keyboards can then be adapted and customized for specific applications and devices.

At its root, the primary goal in any keyboard optimization is simply allowing a user to choose the desired characters as quickly as possible. In optimizing keyboards for Brahmic scripts the most important obstacle to overcome is the large number of characters in the languages. In an Indic language, there can be well over 60 individual characters which can be combined to form over 200 different ligatures. Many of these ligatures bear only slight resemblance to the original characters. Including every character and ligature would be highly impractical because of its large size and the difficulty in finding a specific letter. However, the opposite extreme, a small keyboard with only vowels and consonants, would be similarly unreasonable since it would require several characters to be chosen at a time. An optimal solution would logically involve a compromise between these extremes.

A. Brouillette is with the Department of Computer Science, University of Colorado, Colorado Springs, CO, 80918 USA e-mail: (abrouil2@uccs.edu).

II. Related Research

Most of the previous research in the area of keyboard optimization has focused on optimizing English soft keyboards. While there has not been extensive research specifically for optimizing Indic keyboards, the research into English keyboards is valuable in finding techniques for optimizing any keyboard.

A. Keyboard Optimization

The earliest soft keyboards focused on variations of alphabetic and QWERTY layouts. While these keyboards were effective for the technology at the time, many of the layouts were apparently arbitrary attempts at organizing the characters to conform to mechanical limitations. Since then, much progress has been made, starting with MacKenzie's development of one of the first character-frequency optimized layouts, the OPTI keyboard[1]. His approach was essentially an application of Fitts' law with a trial and error approach to hand placing the characters based on frequently occurring bigraphs. Those results were improved through the use of algorithms adapted from applications of physics such as Hook's Law. The use of the *Metropolis random walk algorithm* has further increased the efficiency of soft keyboards[2]. The use of this algorithm with machine testing has enabled the development of English keyboards with theoretical top typing speeds of up to 42 wpm. More recently, soft keyboards have reached a new level of efficiency through the development and use of genetic algorithms[3]. These layouts have so far produced the best results for optimal keyboards. Because of the effectiveness of this algorithm, it is thought that similar optimization techniques might be developed to further improve these keyboards. While not widely used, ant colony optimization has been implemented in developing keyboards for handicapped users[4]. In comparison, research into the development of soft keyboards for Indic languages is relatively primitive. Most examples seem to be designed simply for utility with no thought toward efficiency. Some work has been done in designing single layered keyboards similar to the English layouts[5]. However, there are some problems with this approach, due to the greater number of characters in these languages. In working to optimize keyboards for Brahmic scripts, the techniques developed for English keyboards are inadequate and need to be modified. Recently it was proposed that the development of layered keyboards would give a better solution to the difficulty of a large number of necessary characters[6][7][8]. At this point, this approach seems to give the best results. These keyboards use pop-up menus to allow users to access multiple characters from a single key. This technique could potentially increase users input speed by reducing the search time and distance between each character.

B. Adaptations for Cell Phones

Gong has done considerable research in optimizing English keyboards for cell phones. One of the primary techniques used for optimizing cell phone text input has been the idea of putting multiple letters on each key. Letters are selected either by pressing a key multiple times or by using a word-prediction program to allow the selection of possible words from a given combination of key-presses[9][10].

In our project, we have used a combination of these word-prediction techniques with the layered keyboard research in our efforts to reach an optimal Indic keyboard for mobile devices.

III. Optimizing Keyboards

A simplistic approach for keyboard optimization would be to simply use an exhaustive search and evaluate every possible keyboard layout. However, for any given number of characters n, there are $n!$ possible combinations of the characters. For languages of 60+ characters, this number quickly reaches 10^{100}, making this approach impossible.

So far the best results for keyboard optimization have been reached through the use of machine learning techniques. Our approach to optimization in this project involves the use of the Genetic Algorithm to generate optimized keyboard layouts.

A. Specific Design Decisions

The creation of these keyboards required some arbitrary decisions to be made from the start. These decisions are explained here.

Based on the results of earlier research it was decided that the space-bar should be placed below the keyboard layout. The distance from a given character to the space-bar is calculated as the distance to the center of the space-bar. This compensates for the fact that users will not necessarily choose the shortest distance every time. Although, theoretically, multiple optimized space-keys would give better results, the results of experiments done by Zhai et al. showed that, given the choice of four space-bars, users chose the optimum space bar only 38-47% of the time. As an extra benefit, an arbitrarily placed space-bar prevents inaccuracies in calculations due to "free-warping", a common error in keyboard evaluation where the stylus enters the space-bar in one location and leaves in an unrelated random location[2]. Finally an easily accessible space-bar improves the users ease of learning a layout.

Another decision was to use a rectangular 'grid' keyboard layout with each character occupying a square key. While some other designs such as the Metropolis Keyboard have used a hexagonal, honeycomb design, this rectangular layout is the most familiar to users[2]. Additionally, this layout and simplifies the comparison of our test results with results from previous research which uses this layout.

B. Implementing the Genetic Algorithm

The genetic algorithm is a heuristic designed to imitate the concept of natural selection. This algorithm is used to optimize a function for which there might not actually be an optimal solution. The general idea behind the genetic algorithm is to create a random population of potential solutions. These solutions are combined and mutated with the objective of keeping the 'good' parts of each solution and combining them toward a theoretical 'optimum'.

There are three essential parts of this algorithm. First, there is a population of potential solutions. These possible solutions then need to be evaluated using a fitness function. Finally there needs to be a method of reproduction that will change the population of solutions over time. In any given generation, the individual solutions are evaluated by the fitness function and assigned a score. This score is based on its distance from the theoretical optimum solution. A new population is created including a percentage of the best solutions and adding some new solutions made by combining and mutating the best individuals. This process is repeated until an acceptable maximum is reached.

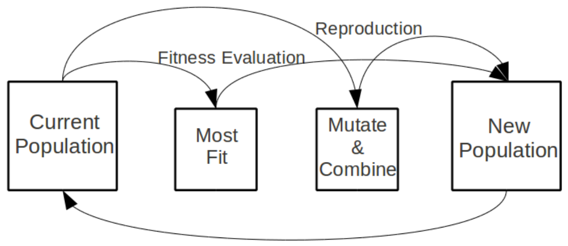


Fig. 1. Illustration of the evolutionary cycle of a population of chromosomes in a genetic algorithm.

In our implementation of the genetic algorithm, each individual, known as a chromosome, represents a different keyboard layout. Each of these chromosomes holds a number of genes equal to the number of characters in the given alphabet. Each character in the alphabet is assigned an integer. Each of the genes then contains one of these integers.

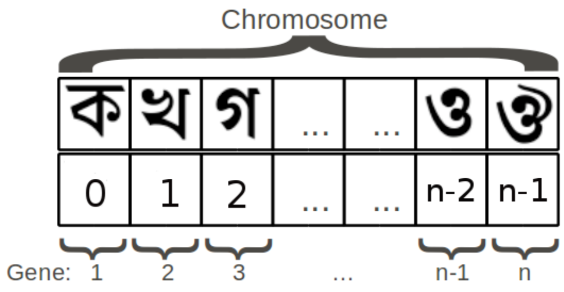
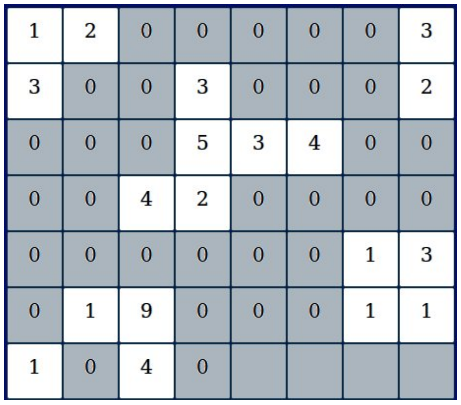


Fig. 2. Illustration of a chromosome for the Assamese keyboard. The genes are stored in a one dimensional list of integers while the genetic algorithm is running. Once finished, it is converted to an $x \times y$ rectangular keyboard layout with the first x genes representing the first row of keys, the second group of x genes represents the second row, etc.

The chromosomes are scored with the highest score being given to the layout with lowest mean time per character. This way the fastest layouts are kept for the next generation. Some of the chromosomes are then randomly selected to be combined or mutated. These new chromosomes are included with the fastest layouts in the next generation.

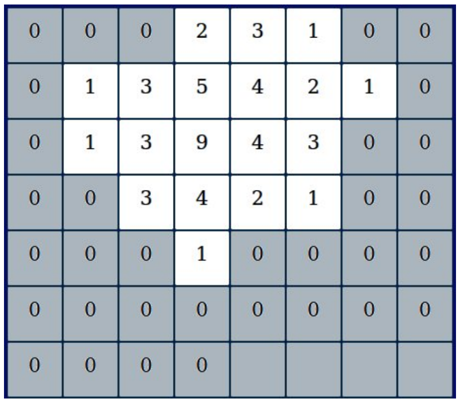
In testing the effectiveness of this algorithm, we used a diagram to track the positions of the most frequently occurring characters on the keyboard. In the first generation, the high frequency characters were spread randomly across the layout.



1	2	0	0	0	0	0	3
3	0	0	3	0	0	0	2
0	0	0	5	3	4	0	0
0	0	4	2	0	0	0	0
0	0	0	0	0	0	1	3
0	1	9	0	0	0	1	1
1	0	4	0				

Fig. 3. A diagram representing the layout of the most frequently used characters in the first generation of the genetic algorithm. The characters are represented by the numbers from 0-9 with 0 being the lowest frequency and 9 being the highest.

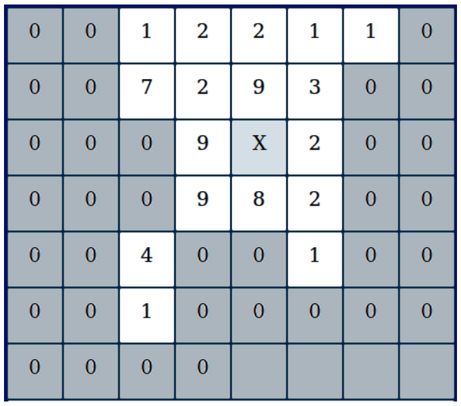
However, after 400 generations, it became apparent that the most frequent characters were being clustered together near the center of the keyboard.



0	0	0	2	3	1	0	0
0	1	3	5	4	2	1	0
0	1	3	9	4	3	0	0
0	0	3	4	2	1	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0				

Fig. 4. A diagram representing the layout of the most frequently used characters after 400 generations of the genetic algorithm. The characters are represented by the numbers from 0-9 with 0 being the lowest frequency and 9 being the highest.

Logically, it can be expected that most of the frequently occurring digraphs consist of combinations of the most frequent characters. A diagram tracking the position of the most frequent character in relation to its most common digraphs shows the digraphs getting closer together as the algorithm progresses.



0	0	1	2	2	1	1	0
0	0	7	2	9	3	0	0
0	0	0	9	X	2	0	0
0	0	0	9	8	2	0	0
0	0	4	0	0	1	0	0
0	0	1	0	0	0	0	0
0	0	0	0				

Fig. 5. A diagram representing the layout of the most frequently occurring character in relation to its most common digraphs after 400 generations of the genetic algorithm. The character is represented by an X its digraphs are represented by the numbers from 0-9 where 0 is the least common and 9 is the most common.

These results are consistent with the theoretical optimal positioning of the frequently used characters and their digraphs close together. Based on test results by earlier research, it is expected that this pattern will continually be improved with testing at a greater number of generations.

C. Parameter Changes

There are several parameters in the genetic algorithm that can be adjusted to achieve optimal results for a given application. These can include changes to the population size as well modifications to the mutation rates and crossover functions.

In determining the best parameters for the genetic algorithm, we performed several tests with the Bengali language, varying the population size. The result of these tests showed that simply creating larger populations will not always give the best results. The best keyboards were generated with a more moderate population size.

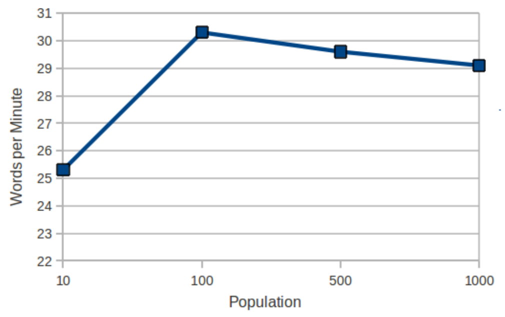


Fig. 6. This graph shows the variation in typing speed with changes in population. The typing speed was calculated after 100 stable generations.

Further testing showed that looking for a greater number of stable generations gave the best indication of an optimal keyboard. Stable generations are defined as generations with a consistent highest score. Populations with a large number of stable generations consistently generated the best keyboards.

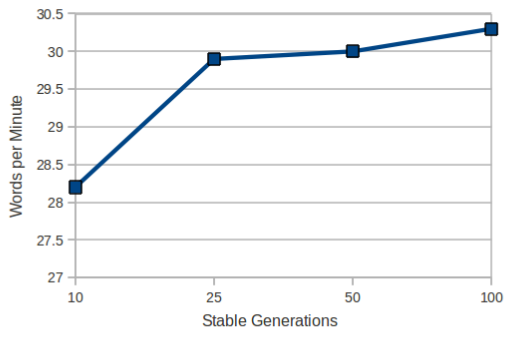


Fig. 7. This graph shows the change in typing speed after different numbers of stable generations. The typing speed was calculated after populations of 100 chromosomes.

IV. Evaluation of Generated Keyboards

As keyboards are developed, they need to be evaluated in order to compare them and determine how much improvement has been made. By definition, an efficient keyboard is one that allows users to input their text as quickly as possible. Although human testing is necessary to determine the actual effectiveness and learn-ability of a keyboard, the first step is to compute theoretical upper and lower limits of typing speed. The upper-bound is commonly calculated by using Fitts' Law. This number gives us an estimate of the typing speed for an experienced user with minimal search time on each character. The lower-bound can also be estimated using a combination of Fitt's Law and the Hick-Hyman Law. The result of this calculation estimates the decision time for new users.

A. Fitts' Law

Fitts' Law models human movement and can be used to predict the time required to move to a given target point. For our application, we use this as a technique for determining the average time in seconds the enter a character. This number can then be used to calculate a theoretical upper-bound in words per minute of input for the keyboards. This is the most widely used method for evaluating English keyboards. An adaption of Fitts' Law has been made in order to find the average time required to move between two characters, i and j, for a given alphabet of n characters. This is done by looking at the distance between the characters, D_{ij}, as well as the frequency of occurrence for that particular digraph, P_{ij}. After assigning a key width W_j, and an index of performance IP, the equation for Fitts' Law becomes:

$$\bar{t} = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{P_{ij}}{IP} \left[\log_2 \left(\frac{D_{ij}}{W_j} + 1 \right) \right]. \qquad (1)$$

We will use an IP of 4.9, as determined by earlier research, in order to maintain a consistent comparison of our progress[1].

In its most basic state, Fitts' Law will only work for single layered keyboards. However it has since been adapted to return reasonable results for other layouts as well, including multi-layered and menu based keyboards[11].

B. Hick-Hyman Law

The Hick-Hyman Law is used to predict the time required for a human to make a decision based on a given number of choices. In our application, this number is used to estimate the typing speed for a novice user who would need to search the keyboard to determine their next key-press. This number gives us a more realistic estimate to compare to human testing results.

In using the Hick-Hyman Law to evaluate our keyboards, we are given n as the number of possible choices (in our case the number of keys) and p_i being the probability of a given key being selected. The number b is a constant that is determined through experimentation. This reaction time can then be described by the equation:

$$T = b \sum_{i=1}^{n} p_i \log_2 (\frac{1}{p_i} + 1). \qquad (2)$$

The Hick-Hyman Law can be adapted to estimate the decision time for both the base layer of the keyboard and the hierarchical menus.

C. WPM calculation

In calculating an average number of words per minute, the corpus of words is processed to determine the average number of characters per word. From our corpus of the Assamese language this number turns out to be approximately 6 characters per word. Given the calculated mean time per character, the calculation for average words per minute is simply: $wpm = \frac{60}{6t}$

V. Testing on Other Indic Languages

In this section, we discuss our the results we have obtained while developing soft keyboards for several other Indian languages. The languages we work with are Bengali, Hindi, Gujarati, Punjabi, Oriya, Kannada and Telugu. Of all the languages we have investigated for this paper, the scripts used by Assamese, Bengali, Hindi, Gujarati, Punjabi and Oriya belong to the Northern branch of Brahmic scripts. Assamese and Bengali use two variants of the Eastern Nagari script. Hindi uses Devanagari script. Punjabi uses the Gurmukhi script although it can be written using the Shamukhi script as well. Gujarati and Oriya have their own individualized scripts. Two of the languages, Kannada and Telugu use scripts that belong to the Southern branch of Brahmi scripts. Each of these languages have its own script.

For each language, we develop alphabetically sorted keyboards, a flat GA-based soft keyboard and a layered GA-based soft keyboard using the techniques we used for Assamese.

A. Bengali

For Bengali, each alphabetic layout tested for WPM results listed the vowels before the consonants in alphabetical order. The row ordering of three alphabetic layouts were tested: one with the diacritics listed after the vowels and consonants, one with the diacritics listed in between the vowels and consonants, and one with the diacritics listed before the vowels and diacritics. Row ordering means that each of the characters was listed alphabetically left to right from the top row to the bottom row. The best arrangement was row ordered and listed the diacritics after the vowels and consonants, which yielded an expected input speed of 22.19 WPM.

Diacritic Arrangement	Words per minute	Time per char
First-row diacritic	20.4	0.490
Center-row diacritic	21.0	0.476
End-row diacritic	22.2	0.451

Fig. 8. Variations in input speed for alphabetic layouts with different arrangements of diacritics.

The best results of our genetically designed Bengali flat keyboard yielded a theoretical input speed of 30.35 WPM as predicted by Fitts Law. This was an 8 x 8 square keyboard constructed with the following genetic algorithm parameters: a population of 100 and 100 stable generations.

As discussed earlier in the paper, greater improvements to input speeds can be achieved through the use of layered keyboards. Thus for Bengali, we kept the consonants and vowels on a base layer and a diacritic menu as a second layer that could come up whenever a user clicked on a consonant as done earlier for Assamese. The best genetically designed layered keyboard for Bengali had a base layer of dimensions 7x6 and was genetically constructed from a population of 500 and 15 stable generations. It yielded an expected input speed of 36.13 WPM. Adding a vowel menu in addition to the diacritic menu for the layered keyboard made very little difference for the layered keyboard. For Bengali, taking the vowels out of the base layer of consonants and putting them in their own separate menu decreased the expected speed by only 0.04 WPM.

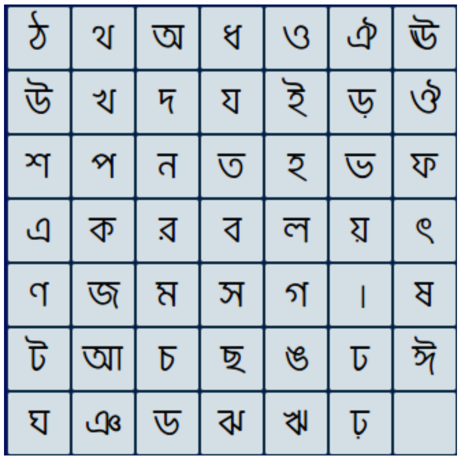


Fig. 9. The best layered Bengali keyboard. This keyboard was designed using the genetic algorithm with a population of 500 and 15 stable generations.

B. Hindi

For Hindi, the same six alphabetic layout arrangements as in Bengali were tested with Hindi characters. The layout with the best expected input speed, which was 22.04 WPM, listed the diacritics before the vowels and consonants and was column ordered.

The best theoretical input speed generated from the genetically designed Hindi flat keyboards was 29.01 WPM as predicted by Fitts Law. This was also an 8 x 8 square keyboard constructed with a population of 100 and 100 stable generations. The same diacritic menu was made for the Hindi layered keyboard using Hindi characters. The best genetically designed layered keyboard for Hindi had a base layer of dimensions 7 x 5 characters and was genetically constructed from a population of 500 and 25 stable generations. It yielded an expected input speed of 37.03 WPM. Both of these layered keyboards offered an average 6.9 WPM improvement over the expected WPM scores of the flat keyboard layouts and an average 14.47 WPM improvement over the WPM scores of the alphabetic layouts. For Hindi, having a vowel menu decreased the expected input speed by only 0.7 WPM.

Future research may include testing the layered keyboard layouts with and without a vowel menu on actual users to determine whether a separate vowel menu can significantly improve or worsen the efficiency of a layered keyboard for the Bengali and Hindi languages.

Keyboard Type / Language	Flat Alphabetic (WPM/CPM)	Layered Alphabetic (WPM/CPM)	Flat GA-Designed (WPM/CPM)	Layered GA-Designed (WPM/CPM)
Assamese	25.1 / 0.399	33.9 / 0.295	34.2 / 0.292	40.2 / 0.249
Bengali	22.7 / 0.440	26.6 / 0.377	30.3 / 0.330	36.1 / 0.277
Hindi	25.8 / 0.465	34.5 / 0.348	34.4 / 0.349	43.7 / 0.275
Gujarati	22.2 / 0.449	24.5 / 0.407	29.8 / 0.335	31.7 / 0.315
Punjabi	26.5 / 0.453	29.9 / 0.402	34.9 / 0.343	39.9 / 0.300
Oriya	16.7 / 0.450	19.4 / 0.386	23.5 / 0.319	26.7 / 0.281
Kannada	14.7 / 0.455	17.1 / 0.386	20.2 / 0.330	22.8 / 0.292
Telugu	15.8 / 0.474	19.1 / 0.393	22.7 / 0.331	25.6 / 0.294

Fig. 10. Expected Upper Bound of Input Speeds in WPM/CPM for Various Languages. The numbers were computed using Fitt's Law only.

Keyboard Type / Language	Flat Alphabetic (WPM/CPM)	Layered Alphabetic (WPM/CPM)	Flat GA-Designed (WPM/CPM)	Layered GA-Designed (WPM/CPM)
Assamese	9.70 / 1.031	13.3 / 0.754	10.8 / 0.924	14.1 / 0.708
Bengali	9.42 / 1.061	12.5 / 0.800	10.5 / 0.956	14.3 / 0.700
Hindi	10.8 / 1.111	15.4 / 0.782	12.1 / 0.992	17.5 / 0.685
Gujarati	9.24 / 1.082	11.9 / 0.838	10.1 / 0.986	13.2 / 0.757
Punjabi	11.0 / 1.092	14.0 / 0.857	12.0 / 0.997	15.7 / 0.764
Oriya	6.99 / 1.073	9.24 / 0.811	7.73 / 0.970	9.44 / 0.795
Kannada	6.06 / 1.101	7.75 / 0.860	6.85 / 0.974	8.44 / 0.790
Telugu	6.87 / 1.092	8.84 / 0.848	7.69 / 0.976	9.89 / 0.758

Fig. 11. Expected Lower Bound of Input Speeds in WPM/CPM for Various Languages. The numbers were computed using Fitt's Law and Hick-Hyman's Law.

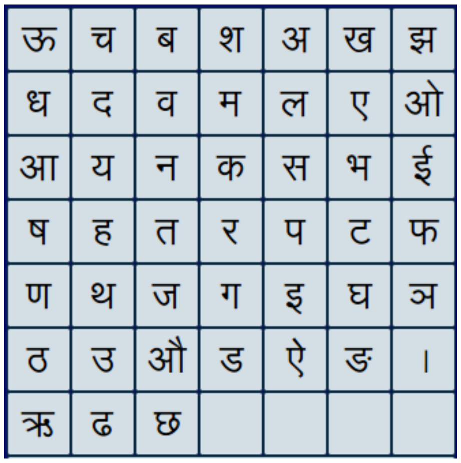


Fig. 12. The best layered Hindi keyboard. This keyboard was designed using the genetic algorithm with a population of 500 and 25 stable generations.

C. Other languages

We tested our algorithm with five other languages in order to detect some trends in their development and draw conclusions about the variations in input speed. Each language was tested with four different keyboard arrangements. Our first step was to evaluate keyboards with an unoptimized, alphabetic arrangement as a basis for comparison. These keyboards were developed with both flat and layered designs. We then used the genetic algorithm to develop optimized flat and layered keyboards.

1) Gujarati: For the Gujarati language, our corpus had an average of approximately 6 characters per word which we used to calculate words per minute from the average time per character. The alphabetic layouts yielded an upper-bound of 0.449 seconds per character or 22.2 WPM for the flat keyboard and 0.407 seconds per character or 24.5 WPM with the layered keyboard. After optimization using the Genetic Algorithm, the results improved to 0.335 seconds per character or 29.8 WPM for the flat keyboard and 0.315 seconds per character or 31.7 WPM with the layered keyboard.

2) Punjabi: For the Punjabi language, our corpus had an average of approximately 5 characters per word which we used to calculate words per minute from the average time per character. The alphabetic layouts yielded an upper-bound of 0.453 seconds per character or 26.5 WPM for the flat keyboard and 0.395 seconds per character or 30.4 WPM with the layered keyboard. After optimization using the Genetic Algorithm, the results improved to 0.343 seconds per character or 34.9 WPM for the flat keyboard and 0.300 seconds per character or 39.9 WPM with the layered keyboard.

3) Oriya: For the Oriya language, our corpus had an average of approximately 8 characters per word which we used to calculate words per minute from the average time per character. The alphabetic layouts yielded an upper-bound of 0.450 seconds per character or 16.7 WPM for the flat keyboard and 0.386 seconds per character or 19.4 WPM with the layered keyboard. After optimization using the Genetic Algorithm, the results improved to 0.319 seconds per character or 23.5 WPM for the flat keyboard and 0.281 seconds per character or 26.7 WPM with the layered keyboard.

4) Kannada: For the Kannada language, our corpus had an average of approximately 9 characters per word which we used to calculate words per minute from the average time per character. The alphabetic layouts yielded an upper-bound of 0.455 seconds per character or 14.7 WPM for the flat keyboard and 0.386 seconds per character or 17.1 WPM with the layered keyboard. After optimization using the Genetic Algorithm, the results improved to 0.330 seconds per character or 20.2 WPM for the flat keyboard and 0.292 seconds per character or 22.8 WPM with the layered keyboard.

5) Telugu: For the Telugu language, our corpus had an average of approximately 8 characters per word which we used to calculate words per minute from the average time per character. The alphabetic layouts yielded an upper-bound of 0.474 seconds per character or 15.8 WPM for the flat keyboard and 0.393 seconds per character or 19.1 WPM with the layered keyboard. After optimization using the Genetic Algorithm, the results improved to 0.331 seconds per character or 22.7 WPM for the flat keyboard and 0.294 seconds per character or 25.6 WPM with the layered keyboard.

D. *Summary*

In analyzing these test results, we were able to draw a few basic conclusions regarding the optimization process.

Looking at the numbers, we notice that the Oriya language shows the greatest improvement after optimization, giving the lowest time per character, while the improvement for the Gujarati language was somewhat less significant. One explanation for these results considers the relative frequencies of characters in the two languages. The most frequently occurring character in the Oriya language has a relative frequency of approximately 8.6%. In Gujarati, the most frequent character has a relative frequency of 5.9%. Considering Figure 4 we can see that the optimized keyboard has the most frequent characters clustered together near the center. Our conclusion is that languages with a small number of high frequency characters have a greater potential to be optimized. It can be surmised that languages with characters that have nearly equal frequencies require the user to travel a greater average distance between each character.

Additionally, it can be seen that Gujarati has a smaller improvement between its layered and flat layouts when compared to the other languages. One reason for this could be the higher frequency of the diacritics in this language. The Kannada language, which showed the highest improvement in its layered layout, also has the lowest frequency of diacritics. Essentially, the smaller number of diacritics means less use of the menu making the layering more effective.

Other than these minor variations, the results from the optimization tests of these languages are all very similar in their progress. Graphing the results of the tests all of the languages show very similar improvement over an increasing number of generations.

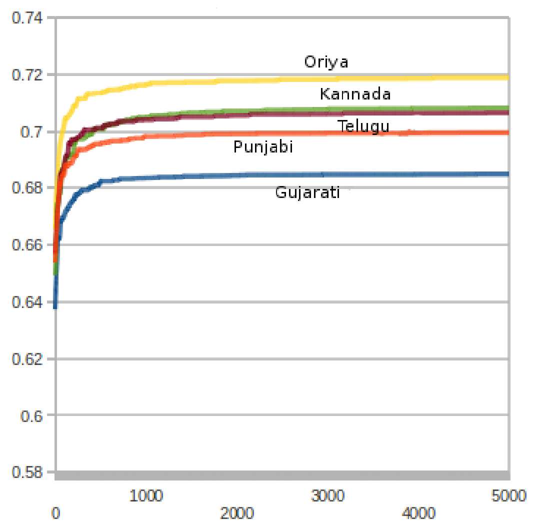


Fig. 13. A graph of the improvement in input speed for the 5 languages over 5000 generations.

VI. Android Keyboard Design

As we worked to develop Android soft keyboards from our optimized layouts we had several challenges to overcome first. Our first step was to find a suitable format to install the keyboards on the Android phone. Once we had actually tested the keyboards in their basic form, we did several experiments in an attempt to modify the designs for more practical use on the actual device.

A. *General Keyboard Framework*

There has been considerable research done in the development of frameworks for soft keyboards on Android phones and there have been many keyboard applications created. The tool we are using for this project is the AnySoftKeyboard app[1]. This is a free application that is easily available to users. In addition, it supports the development of plug-ins for other languages. Using this tool allows us to quickly test our keyboard layouts and make them available on-line for human testing.

Using this tool as our structure, we have created an AnySoftKeyboard plug-in for the Assamese language. This keyboard was designed using the genetic algorithm test code.

[1] $https://market.android.com$

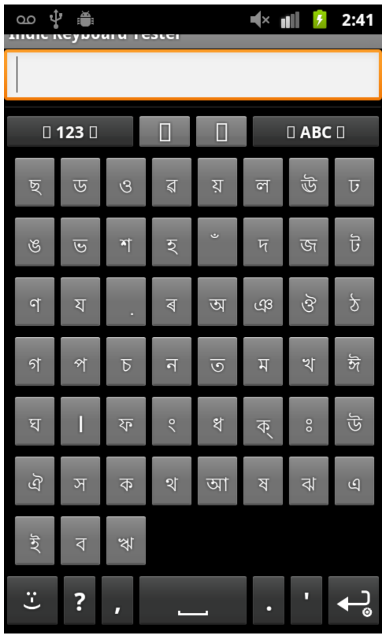


Fig. 14. This Assamese keyboard was implemented as an AnySoftKeyboard plug-in. It was designed by the genetic algorithm after 50 generations. The diacritics are placed in a pop-up menu.

The keyboard was designed to be used with a diacritic layer. This diacritic layer is implemented as a menu that pops up when a consonant key is held down.

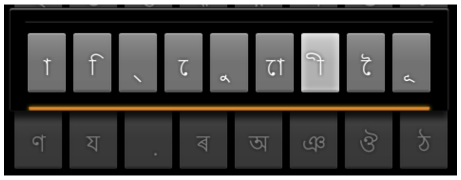


Fig. 15. The diacritic pop-up menu. The menu appears when a consonant key is held down.

An additional feature supported by AnySoftKeyboard is predictive test entry. We implemented this by running a program to process the corpus and create a list of words and their frequencies. This data was then used to create the binary dictionary used by the AnySoftKeyboard program. The word predictions appear as a menu at the top of the screen after at least two characters have been entered.

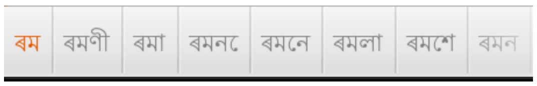


Fig. 16. The word prediction menu. This menu appears after at least two characters have been entered.

B. Experiments With Keyboard Dimensions

Our initial keyboard designs were functional on the Android device. However, the dimensions of the keyboard ended up obstructing parts of the text entry. In an effort to make the keyboard more practically usable, we tried several different variations in the dimensions of the keyboard. The best layout we found consisted of longer, more narrow format.

Our original design consisted of a nearly square layout of 8×8 or 8×7 keys. We ran the genetic algorithm and tested a layered keyboard with dimensions of 5×10 keys. The results of these tests showed only a minimal reduction in typing speed, while the ease of using the keyboard was greatly improved. Running a test over 500 generations, we got results of 36.9 WPM for the rectangular keyboard compared to the 40.2 WPM with the square layout.

The main side-effect of this layout is the key-size in this format. In order to position the keys in this layout we had to reduce their size. This reduces the accuracy for most users and results in more errors.

C. Text Input From Two Points

After running tests with rectangular keyboards, we experimented with modifying the genetic algorithm program to optimize for text input with two fingers.

The basic layout we chose consisted of a long rectangular shape with the assumption that each finger or input point occupying an essentially square section of the keyboard. The logic behind the implementation of the genetic algorithm relies on the assumption that each input point is only used in its square.

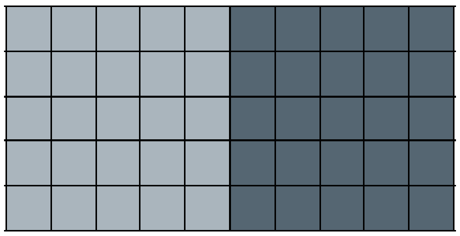


Fig. 17. Basic shape of keyboard for two finger input. Each finger is responsible for pressing the keys in one of the squares.

Because of the complexity of this problem, at this point we have only been able to roughly estimate a lower bound for text entry speed. There are essentially two cases to be considered in the calculation of the fitness function: First the case of digraphs that consist of two characters in the same square. And secondly, the case of digraphs where the characters occur in different squares.

To take care of the first case, the fitness function uses Fitt's Law to calculate an average time per character for each input point in each square. In the second case, it is assumed that each character requires a decision time calculated using the Hick-Hyman Law as well as the time for the input point to move to the desired key.

Using this algorithm, we were able to develop a two input keyboard for the Assamese that we estimate to have a lower bound of text input around 22.6 WPM.

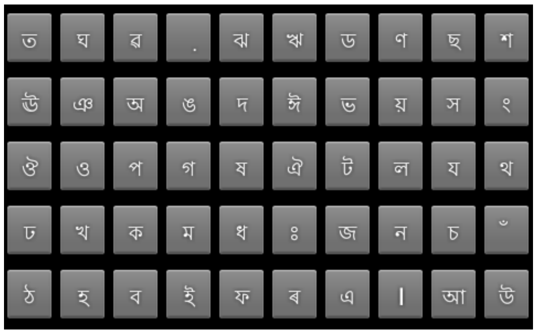
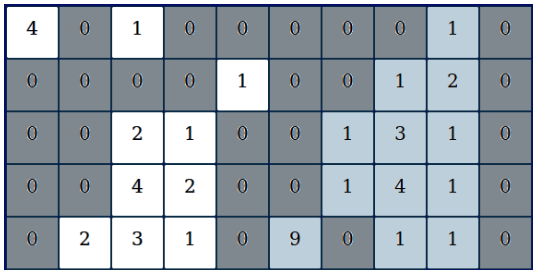


Fig. 18. Android keyboard designed for two finger input. This keyboard has a lower bound of text input estimated to be 22.6 WPM.

Looking at the locations of the most frequent characters we can see two clusters being form in the location of the two input points. There seems to be a nearly equal number of high frequency characters on each side.



4	0	1	0	0	0	0	0	1	0
0	0	0	0	1	0	0	1	2	0
0	0	2	1	0	0	1	3	1	0
0	0	4	2	0	0	1	4	1	0
0	2	3	1	0	9	0	1	1	0

Fig. 19. A diagram representing the layout of the most frequently used characters after optimization by the genetic algorithm. The characters are represented by the numbers from 0-9 with 0 being the lowest frequency and 9 being the highest. The frequently used characters appear to be forming two clusters around the two input points.

These results are consistent with a comment made in a paper by Zhai et al.[12]. In this paper, they mention that the QWERTY keyboard is most effective when used for two-handed input because of the frequency of alternation between the two hands. The algorithm that we implemented gives a higher score to keyboards that more frequently alternate hands.

D. Future Work

The optimized keyboards that we generated for the Android phone performed considerably better than the alphabetic alternatives. However, even after adding the diacritic layer, the number of keys is simply to large for the size of the Android phone. An area for future research would be to experiment with putting multiple characters on each key. It would be desirable to be able to optimize keyboards given a physically constrained number of keys. This might be the best option for creating effective cell phone keyboards for languages with large numbers of characters.

Another area for continued research would be to investigate the development of keyboards optimized for the specific needs of other devices such as the iPad and the iPhone.

VII. Conclusion

The versatility of soft keyboards makes them an ideal research tool in seeking optimal layouts for Indic languages. The programs developed to predict the best keyboard layouts can be easily reused to generate optimal keyboard layouts for other languages. Analysis of the keyboards generated by implementing the genetic algorithm was done by using Fitts' Law and the Hick-Hyman Law to estimate input speed. Based on these results, we were able to demonstrate how the efficiency of a keyboard is improved when the keys are arranged based on character and digraph frequencies.

The results of our tests with the various Indian languages show the keyboards developed by the genetic algorithm to be comparable in efficiency for all of the languages tested. It can be assumed that these techniques will be easily adapted to create optimized keyboards for many other languages that have large numbers of characters.

Acknowledgment

The research reported in this document has been funded partially by NSF grants CNS-0958576 and CNS-0851783.

References

[1] S. MacKenzie and S. X. Zhang, "The design and evaluation of a high-performance soft keyboard," *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pp. 25–31, 1999.
[2] S. Zhai, M. Hunter, and B. A. Smith, "The metropolis keyboard - an exploration of quantitative techniques for virtual keyboard design," *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pp. 119–128, 2000.
[3] M. Raynal and N. Vigouroux, "Genetic algorithm to generate optimized soft keyboard," *CHI'05 extended abstracts on Human factors in computing systems*, pp. 1729–1732, 2005.
[4] S. Colas, N. Monmarché, P. Gaucher, and M. Slimane, "Artificial ants for the optimization of virtual keyboard arrangement for disabled people," pp. 87–99, 2007.
[5] V. Varma and V. Sowmya, "Design and evaluation of soft keyboards for telugu," *ICON 2008: 6th International Conference on Natural Language Processing*, 2008.
[6] A. Rathod and A. Joshi, "A Dynamic Text Input scheme for phonetic scripts like Devanagari," *Proceedings of Development by Design (DYD)*, 2002.
[7] S. Shanbhag, D. Rao, and R. K. Joshi, "An intelligent multi-layered input scheme for phonetic scripts," *Proceedings of the 2nd international symposium on Smart graphics*, pp. 35–38, 2002. [Online]. Available: http://doi.acm.org/10.1145/569005.569011
[8] L. Hinkle, M. Lezcano, and J. Kalita., "Designing soft keyboards for brahmic scripts," *ICON 2010: International Conference on Natural Language Processing*, pp. 191–200, 2010.
[9] J. Gong, "Improved text entry for mobile devices : alternate keypad designs and novel predictive disambiguation methods," *Northeastern University Boston, MA, USA*, 2007.
[10] M. Selander and E. Svensson, "Predictive text input for indic scripts," *Citeseer*, 2009.
[11] S. Matsui and S. Yamada, "Genetic algorithm can optimize hierarchical menus," pp. 1385–1388, 2008.
[12] S. Zhai, P. Kristensson, and B. Smith, "In search of effective text input interfaces for off the desktop computing," *Interacting with Computers*, vol. 17, no. 3, pp. 229–250, 2005.