Honours Year Project Report

Analysis of SMS Efficiency

By

How Yi Jue

Department of Computer Science

School of Computing

National University of Singapore

2003/2004

Table of Contents

Title i
Abstract ii

1. Introduction 1
2. Background 3
    - 2.1 Multi-tap text entry method 3
    - 2.2 Predictive text entry method 3
3. Data Collection 5
    - 3.1 SMS Message Corpus Collection 5
    - 3.2 Videotaping the Input of SMS messages 7
4. Task Analysis 9
    - 4.1 Modeling User Actions 9
        - 4.1.1 Background-KLM 9
        - 4.1.2 A Different KLM for Mobile Phone Users 10
    - 4.2 Analysis of the Videotape to Obtain Timing Information 12
    - 4.3 Measuring the Efficiency of Text Entry Methods 15
        - 4.3.1 Multi-tap Text Entry 15
        - 4.3.2 Predictive Text Entry 17
        - 4.3.3 Measuring Efficiency 18
5. Methods and Results 20
    - 5.1 Baseline 20
        - 5.1.1 Multi-tap Text Entry Method 20
        - 5.1.2 Predictive Text Entry Method 20
    - 5.2 Reordering Letters for Multi-tap Text Entry Method 21
    - 5.3 Remapping Letters for Multi-tap Text Entry Method 23
    - 5.4 Huffman Codes as a Text Entry Method 24
        - 5.4.1 Background-Huffman Encoding 24
        - 5.4.2 Encoding Characters Using Huffman Encoding 26
    - 5.5 Remapping Letters for Predictive Text Entry Method 27
        - 5.5.1 Background-Genetic Algorithm 28
        - 5.5.2 Using Genetic Algorithm to Remap the Keypad 29
    - 5.6 Using Bigrams for Predictive Word Completion 33
        - 5.6.1 Background 33
        - 5.6.2 Determining When to do Predictive Word Completion 33
    - 5.7 Combining Predictive Word Completion with Remapped Keypad 35
    - 5.8 Summary of Results 35
6. DEMO 40
    - 6.1 Background-Trie 40
    - 6.2 Implementation 40
    - 6.3 Using the Prototype 41
7. Conclusions 42

References 44

Appendix A – Huffman Codes generated for all characters A-1

Honours Year Project Report

Analysis of SMS Efficiency

By

How Yi Jue

Department of Computer Science

School of Computing

National University of Singapore

2003/2004

Project No: H79020
Advisor: Dr Min-Yen Kan

Deliverables:
    Report: 1 Volume

Abstract

SMS messaging is becoming increasingly widespread among people. But the 12-key keypad that is found on many mobile phones today poses a problem for text entry. As three or four letters share the same key, some form of disambiguation is required to determine which letter is intended by the user. The multi-tap and the predictive text entry methods are the common text entry techniques used in present day mobile phones. To measure the efficiency of text entry, we do a task analysis to model the actions of mobile phone users and develop the keystrokes model and the time model. In addition, a corpus of SMS messages is collected and mobile phone users are video-taped to obtain timing information for data analysis. Then we propose a few ways to improve each of the text entry methods. Some of these ways include reordering and remapping the letters on the keypad and predictive word completion. Using the time model, the largest savings is 9.8% for multi-tap text entry using a remapped keypad and 26.8% for predictive text entry that has been improved by using a combination of a remapped keypad and predictive word completion.

Subject Descriptors:

H.5.2 User Interfaces

I.2.8 Problem Solving, Control Methods, and Search

Keywords:

mobile phone text entry, SMS message corpus, predictive word completion, genetic algorithm, KLM

1. Introduction

Short Message Service (SMS) allows people to send or receive text messages of up to 160 characters from mobile phones. SMS messaging has gained popularity among people, especially the younger generation, in recent years. The various uses of SMS extend from keeping in contact with one another to communicating vital business information to mobile executives, sales rep, field technicians and customers. The number of SMS messages sent has increased from approximately 4 billion in Jan 2000 to 24 billion in May 2002, Figure 1.1. Thus, it is essential to investigate and find ways to improve the efficiency of each of the text entry methods available.

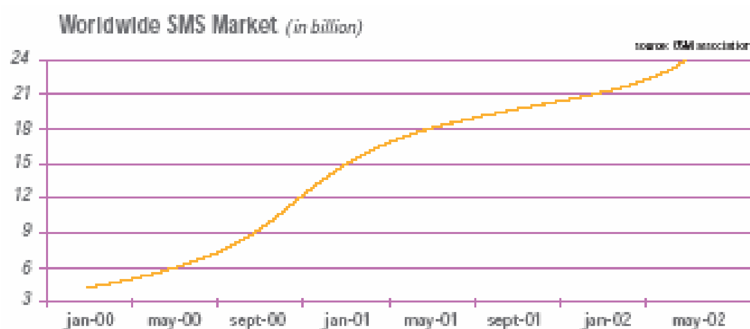

Figure 1.1 Number of SMS messages sent worldwide from European SMS Guide

As mobile phones are usually small in size, it is not feasible to use a full-sized keyboard on them. Although there are phones which are bigger in size that incorporate a full-sized keyboard, smaller phones have to make do with the 12-key keypad commonly found on mobile phones for text input. There are 26 letters in the English alphabet and these letters are normally spread over eight keys. Thus, three or four letters will have to share a same key. When a user presses any one of these keys, the system has to know which of these three or four letters the user actually wants. Therefore, some form of disambiguation technique is needed to decipher which is the letter intended by the user.

There are currently two main methods that are usually used on mobile phones for text entry. They are the multi-tap and the predictive text entry method which will be described in more detail in Section 2. Multi-tap disambiguation is inefficient as some frequently used letters require more taps than letters that are used infrequently. Although the predictive text entry method reduces the number of keystrokes as compared to multi-tap, it still suffers from inadequacies as many

commonly used words may share the same key sequence. Thus the need for disambiguation arises and users may need to select the word that is intended from a word list.

First, we collect a corpus of SMS messages so that analysis can be done on real messages that people have sent or received. Then, we perform a task analysis and developed a model to describe the interaction between humans and mobile phones. Subsequently, we describe how we obtain timing information for each of the operators in the model by videotaping users entering SMS messages and then analyzing the videotape. Next, we define how we are going to measure the efficiency of text entry methods using the keystrokes model and the time model. In the remaining sections, we propose methods to improve the efficiency of text messaging. For the multi-tap text entry method, we studied the possibility of reordering and remapping of the letters on the keypad according to the frequency of letters and the use of Huffman codes as a text entry method. For predictive text entry, we examine remapping the letters on the keypad, the use of bigrams to predict and possibly complete the next word that the user is going to type and also a combination of both improvement techniques. Then, a comparison of the results for present and proposed text entry methods is given. The next section gives a description of a prototype that has been developed to show how the predictive next word completion works with a remapped keypad.

2. Background

In this section, the two main types of text entry methods found in present day mobile phones are described. Both entry methods use the 12-keys mobile phone keypad with letters distributed alphabetically across keys 2-9, Figure 2.1. The placement of the SPACE character varies among different phones but typically, the 0-key or #-key is used. The 1-key is usually reserved for common punctuation marks or symbols.

Figure 2.1 Mobile phone keypad

2.1 Multi-tap text entry method

In this method, user taps the key that contains the letter repeatedly until the desired letter appears. The number of taps required depends on the position of the letter on the key. From Figure 2.1, we can see that the letters "A", "B" and "C" are located on the 2-key. So we tap the 2-key once to get 'a', twice to get 'b' and thrice to get 'c'.

This method is inefficient as frequently used letters may require more taps than words that are infrequently used. For example, the letter 'q' requires only 2 taps while 'r' and 's' which are more common letters require 3 and 4 taps respectively. Also, this method of text entry introduces the segmentation problem where the system needs to determine whether a sequence of key presses of the same key corresponds to one or more letters on the key. To resolve this problem, users have to wait for timeout or press a timeout kill key before entering the next letter.

2.2 Predictive text entry method

In this method, the user presses the key that corresponds to each letter of a word once. The system uses a dictionary of words to determine which of the words the key sequence matches. As many words may share the same key sequence, users often have to press a NEXT key to scroll among the alternatives. On many mobile phones, the NEXT key is typically the asterisk-key or the

directional-keys. Using the keypad in Figure 2.1, five sets of words that share the same key sequence is given in Table 2.1.

Key sequence	Words
2273	case, card, care, base, bare, bard, cape, acre
4663	good, home, gone, hood, hoof, hone, goof
2665	cool, cook, book
63	of, me
46	go, in

Table 2.1 Words sharing same key sequences

Likewise, the predictive method is imperfect in some ways. Some commonly used words in SMS messages may require a higher number of NEXT key presses than those used rarely. Also, it is impossible to achieve 1 keystroke per character because some words require disambiguation. For example, the word “home” which shares the same key sequence as “good” requires one NEXT key press but the less common word “gown” does not require any NEXT key presses. On top of that, user can only input words in the dictionary using the predictive method. If a user wants to input a word not in the dictionary, he will have to revert to using the multi-tap method or insert the word into the dictionary. Once a word is inserted into the dictionary, the user can enter the word as per normal using the predictive method.

Currently, T9 developed by Tegic Communications is the most widely used implementation of this method of text entry. In spite of this, T9 does not implement word completion which completes a word before the entire word is spelt out. Other implementations such as eZiText by Zi Corp and iTap by Motorola features word completion. The three implementations allow users to insert new words into the dictionary. All implementations are similar in the way that it is used but T9 is the main one that is studied in this paper.

3. Data Collection

In this section, we describe the methods that we used to collect a corpus of SMS messages and how we measure the amount of time a user takes to perform each different action to input a message.

3.1 SMS Message Corpus Collection

Due to the lack of a large public corpus of SMS messages for use, the first task was to collect SMS messages that people have sent or received. This is done so as to better reflect the language used in SMS and to increase the accuracy of analysis and design. The total number of SMS messages collected is 10117 and the corpus is publicly available at http://www.comp.nus.edu.sg/~rpnlpir/downloads/corpora/sms/.

The corpus of SMS messages were collected from different sources. In order for the corpus to have sufficient depth, messages were collected from several specific phone users on a regular basis. The age of these users fall between 18 and 22 and they contributed 6167 messages altogether, about 60% of the messages in the corpus.

Another 602 messages, almost 6% of the messages in the corpus are collected from the Yahoo SMS chat website at http://sg.mobile.yahoo.com/sms/chat/ychat_instructions.html which shows the live SMS chat transcript of certain SMS chat rooms. As such, we are unable to obtain the demographics of the users of the chat room.

In addition to the aforementioned two sources, we also solicited SMS messages from a large pool of mobile phone users by sending out e-mails which contain a link directing them to a website. The e-mails were sent out to university undergraduates who belong to the age group of 18-25 years old. People in this age group are among the larger users of text messaging. At the website, users were asked to type in their hand phone number and a SMS message and then click “send” to upload the message to the server. Instructions were given in the e-mails to upload only conversational English messages that they have sent or received from their mobile phones. They were also asked to refrain from typing in any repetitive messages. To encourage people to enter SMS messages at the website, a reimbursement scheme, Table 3.1, is implemented. The reimbursement scheme is designed in such a way that returns diminish as more messages are uploaded. This is done so as to discourage a user from uploading too many messages.

25-49 messages	$5.00
50-74 messages	$8.00
>=75 messages	$10.00

Table 3.1 Reimbursement scheme

A total number of 3348 SMS messages were collected from the website after filtering out the obvious non-genuine messages. The number of people who have contributed to the website is 146. This large number of users contributing to the website shows that messages were collected from a wide range of users and this gives sufficient breadth to the corpus of messages collected. As website users were also asked to give messages they have received from other people, so in actual fact the messages may have come from a larger pool of people but we are unable to give figures for this. Out of those who have made contributions, 45 people entered more than 25 messages. These people were reimbursed by contacting them using the hand phone number they provided at the website.

Figure 3.1 shows the distribution of SMS messages collected in the corpus. Users P1 to P5 represent the specific phone users whom we collected SMS messages from. We were unable to identify the remaining phone users that we collected the messages from and so we are unable to show statistics of their contributions in the figure. Users W1 to W45 represent those users that have contributed more than 25 messages at the website. The statistics of the website users who have contributed less than 25 messages are not shown in the figure.

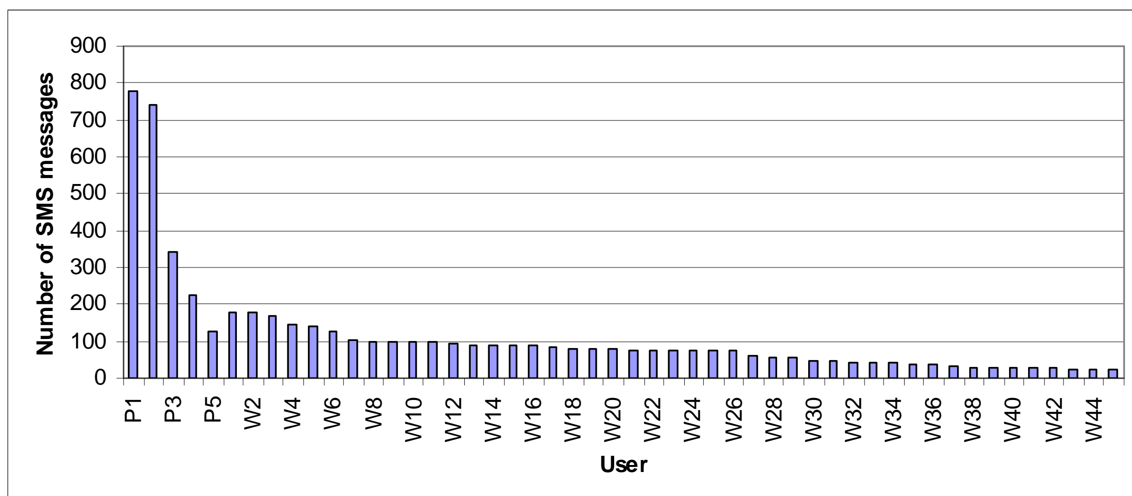

Figure 3.1 Distribution of SMS messages

3.2 Videotaping the Input of SMS Messages

In order to find out the time a user takes to perform each of the tasks specified in Table 4.1, nine users were videotaped entering SMS messages on their phones. Participants were asked to use their individual phones and not a common phone because this would slow down their performance as they may not be accustomed to another phone's interface. If a subject does not own a mobile phone, a phone using T9 predictive text input is provided for use.

The participants were recruited by responding to the e-mails sent out to them. Four undergraduates, four postgraduates and one professor in School of Computing participated in this study.

In the experiment, users are required to enter 5 sample SMS messages extracted from the corpus using the text entry method that they normally type in, either multi-tap or predictive. We chose to use messages from the corpus instead of inventing them so as to make our study closer to reality. Messages containing a mixture of letters, numbers, punctuation marks and symbols were selected so that users can be videotaped typing in each of these different categories of characters. Instructions were given to type the messages using one hand and to type the messages exactly as seen on the paper. A tutorial on how to enter input text on mobile phones was also given to participants who have no experience with SMS messaging. Subjects were also asked to complete a survey about their experience with SMS messaging. The 5 sample SMS messages are given in Figure 3.2.

1. U still wan me 2 reg e gown 4 u? But need ur add, IC n matric. Then e 3 measurement.
2. Sim lim square, shop around , maybe u chk out IT mart @ #03, in front of bullet lift, look 4 jack,
3. Matric pack, ccas & laptops
4. Hey you on your way? I cant go in yet. =(
5. Hey i'm going to be pretty late...

Figure 3.2 Sample messages used for videotaping

The first message was chosen because it has a mixture of letters in lower and upper cases, numbers and common punctuation marks. The second message was chosen as it contains letters, numbers and some of the uncommon symbols such as "@" and "#". As the first two messages were rather long, shorter messages were chosen for the remaining three messages so as not to overwork the participants. Message number three was picked as it has words such as "matric" and "ccas" that cannot be found in the predictive text entry dictionary. The fourth message was chosen because it contains an emoticon that people commonly use in messages. The fifth one was selected because it is simple to type as all the words can be found in the dictionary. All five messages contain words such as "need", "bullet", "pack", "on" and "pretty" that require timeout waits or timeout kills so measuring the time of these actions is not a problem.

4. Task Analysis

4.1 Modeling User Actions

In order to calculate the time required to input a message, we need to break the task down into smaller actions. These actions should be representative of the actions that users usually perform when entering a message. In this section, we describe how we adapt the keystroke-level model that is originally developed for computer users for mobile phone users.

4.1.1 Background-KLM

The Keystroke-Level Model (KLM) was proposed by Card, Moran and Newell (1983) to predict task execution times by modeling human and computer interaction in a formal manner. To predict the execution time of a task, a list of keystroke-level actions that are performed by the user is specified and the times required by each of the actions summed up. The KLM consists of several basic actions that can be performed by users called operators and each operator is assigned a fixed estimated time. The standard set of operators and their estimated times in KLM are: K for pressing a key or button on keyboard (0.12-1.2sec), P for pointing with mouse to a target on display (1.1sec), B for pressing or releasing of mouse button (0.1sec), BB for clicking the mouse button (0.2sec), H for homing hands to keyboard or mouse (0.4sec) and M for the metal act of routine thinking or perception (0.6sec).

In Dunlop and Crossan (2000), the KLM was used to predict multi-tap and predictive text entry times on mobile phones. They gave estimates of average time by developing equations that models the interaction using the operators K, H and M. In their equations, they made use of the estimated times given by the KLM for each operator. But the time stipulated by KLM was estimated for computer users using the keyboard. As the buttons on mobile phones are much smaller than those on keyboard and mobile phone users normally use one finger for input, the timings for pressing a button on mobile phone is unlikely to be similar to that of a keyboard. Therefore, we attempt to measure the timings of actions that user performs by videotaping users inputting SMS messages, Section 3.2, and then analyze the videotape to get the time for each operator. They also assumed that all words are entered in a single case, but in reality, users may type parts of a message in one case and certain words in another. Furthermore, they do not take into account the input of punctuation marks and symbols. Although their equations may be helpful in giving a rough estimate of the efficiency of text entry methods, the timings they obtained from the equations may not be accurate.

The KLM was originally developed to model the interaction between humans and computers. Therefore, the operators specified in KLM describe only the actions that users are likely to perform with computers. But in this project, we want to study the interaction between humans and mobile phones. So in Section 4.1.2, we develop a different KLM that models interactions between users and their mobile phones and addresses the inadequacies of the model developed by Dunlop et al (2000).

4.1.2 A Different KLM for Mobile Phone Users

Until now, no one has formulated a keystroke-level model that describes the actions that users carry out when doing text entry on mobile phones. Here, a different keystroke-level model for mobile phone users is created by adapting the KLM described in Section 4.1.1.

Table 4.1 gives a summary of the basic actions or operators for the multi-tap and predictive text entry methods. As we can see from the table, we make a differentiation between the first press of a key and the subsequent presses of a key. The first press of a key would logically take more time as users may have to look for the letter on the keypad or they may have to move their finger to the key. Subsequent key presses require less time as their finger is already on the key and no movement time or search time is needed. The KLM model does not make this differentiation because the keyboard user is unlikely to press a key multiple times. But such a situation occurs frequently for mobile phone users as a key is shared by more than one letter and the multi-tap text entry method requires users to tap a key repeatedly.

The Wait operator is likely to be used only for the multi-tap text entry method unless user is entering a word that is not in the dictionary and has to revert back to the multi-tap text entry method to input the word. The MPNextK, RPNextK and InsertWord operators are only used in the predictive text entry method. MPNextK and RPNextK describe the action of the user scrolling through the words one by one in the word list to find the intended word. InsertWord describes the auxiliary actions performed by user before he can insert a word and it does not include the actions needed to spell the word out using multi-tap.

Operator/Action	Description
MPAlphaK – Move and Press Alphabet/Symbol/Space Key	This operator describes the finger action of moving and pressing a key that corresponds to a letter in the alphabet, a symbol or the SPACE character.
RPAlphaK – Repeat Press Alphabet/Symbol/Space Key	This operator is for the repeat pressing of a key that corresponds to a letter in the alphabet, a symbol or the SPACE character when user's finger has already been placed on that key.
MPHAlphaK – Move and Press and Hold Alphabet/Symbol/Space Key	This operator is similar to MPAlphaK except that the user holds on to the key until the character appears on the screen.
RPHAlphaK – Repeat Press and Hold Alphabet/Symbol/Space Key	This operator is similar to RPAlphaK except that user holds on to the key until the character appears.
MPSymTabK – Move and Press Symbol Table Key	This operator describes all actions that need to be performed by the user in order to get to the symbol table. On some mobile phones, this is typically just one key press.
MPSelectK – Move and Press Select Key	This operator describes the action of moving and pressing a key that confirms the user's selection. Normally, this action is performed when selecting a symbol to insert from the symbol table.
MPDirK – Move and Press Directional Key	This operator describes the action of moving and pressing the directional key in order to move the selection cursor to navigate around in the symbol table.
RPDirK – Repeat Press Directional Key	This operator is for the repeat pressing of the directional key.
MPModeK – Move and Press Mode Key	This operator describes the action of moving and pressing the key that changes the case. During text entry, users can change between upper-case, lower-case and automatic-case. Automatic-case is where the first letter of each sentence will be automatically capitalized.
Wait/Timeout Kill	When 2 letters sharing the same key are entered consecutively, user need to wait for timeout or press the timeout kill key before entering the second letter.
MPNextK – Move and Press NEXT Key	This operator illustrates the action of moving and pressing the NEXT key.
RPNextK – Repeat and Press NEXT Key	This is for the repeat pressing of the NEXT key.
InsertWord – Inserting a word	This operator encompasses all actions that are performed by the user in order to insert a word into the dictionary.

Table 4.1 Standard operators for mobile phone users

4.2 Analysis of the Videotape to Obtain Timing Information

In order to determine the time of each action specified in Table 4.1, videotapes of the subjects were analysed. The information on the videotapes was burnt onto two CDs and the CDs were watched using Windows Media player on a laptop computer. The timing for a portion of a message is clocked using a stopwatch. The same portion of a message is timed twice and the average time is taken. The number of each action carried out by subject to type the portion of message is computed. This set of actions is equated to the average time obtained.

To make this clearer we illustrate with an example. Suppose the time taken to type "Reach home" with predictive text entry is 3.5 seconds. The actions needed may be found in Table 4.4, entries one to three. Then we equate the sum of all these actions to the time recorded to get the linear equation: $8X + 2Y + 1Z = 3.5s$ where X, Y, Z is the time taken for the actions MPAlphaK, RPAlphaK and MPNextK respectively.

Using the technique described above we obtain a set of linear equations for each subject. Then the linear equations which share common variables are grouped together so that they can be solved for those variables. Because the linear equations are obtained from experimental data, the linear equations are imperfect and contain noise. Therefore we cannot just solve the linear equations directly to obtain solutions for the variables. Instead, we find the best fit values for the variables by minimizing the sum of squares of the difference between the time obtained from the experiment and the calculated time.

Although nine subjects were videotaped, the timing information of only seven subjects was analyzed. The two subjects that were not analyzed belong to the predictive expert user model where we already have three subjects. All subjects are classified into one of the four user models based on how often they use SMS messaging and the type of text entry method they normally use: multi-tap expert, multi-tap novice, predictive expert and predictive novice. The best fit values for each of the actions for each user model are listed in Table 4.2 and they are given in seconds. The two multi-tap subjects waited for timeout to occur instead of using the timeout kill key so we are only able to obtain the timings for the Wait operator. If there is more than one subject in a user model, we show the average time of the actions. Because we cannot find enough participants for some of the models, certain action timings in some of the models are missing. In such models, the missing action timings are obtained from other suitable user models and the source is indicated in brackets in the table. As such, the user models obtained are not real user models but they should

be good estimates of the real ones. In the analysis of the videotape, we do not differentiate between MPDirK and RPDirK and between MPHAlphaK and RPHAlphaK. Therefore the table shows these two sets of actions sharing the same value.

Number of Subjects: 1							
MPAlphaK	RPAlphaK	MPModeK (from Predicitve Expert)	MPSymTabK	MPDirK, RPDirK	MPSelectK	MPHAlphaK, RPHAlphaK	Wait
0.957256	0.274084	1.559445	1.57268	0.401065	2.612965	2.272333	1.351778

Table 4.2(a) Action timings for Multi-tap Expert (in seconds)

Number of Subjects: 1							
MPAlphaK	RPAlphaK	MPModeK	MPSymTabK	MPDirK, RPDirK	MPSelectK	MPHAlphaK, RPHAlphaK (from Predicitve Novice)	Wait
1.439554	0.198849	0.432547	0.971118	0.300674	1.344855	1.925554	2.160591

Table 4.2(b) Action timings for Multi-tap Novice (in seconds)

Number of Subjects: 3								
MPAlphaK	RPAlphaK	MPNextK, RPNextK	MPModeK	MPSymTabK	MPDirK, RPDirK	MPSelectK	MPHAlphaK, RPHAlphaK	InsertWord
0.704438	0.636117	0.912574	1.559445	0.845175	0.315154	3.167028	4.360455	3.586523

Table 4.2(c) Action timings for Predictive Expert (in seconds)

Number of Subjects: 2								
MPAlphaK	RPAlphaK	MPNextK, RPNextK	MPModeK	MPSymTabK	MPDirK, RPDirK	MPSelectK	MPHAlphaK, RPHAlphaK	InsertWord (from Predicitve Expert)
2.175226	1.21391	2.152877	0.646335	3.545888	0.855846	4.140946	1.925554	3.586523

Table 4.2(d) Action timings for Predictive Novice (in seconds)

4.3 Measuring the Efficiency of Text Entry Methods

In order to measure the efficiency of a text entry method, we need to calculate how many of each action or operator described in Table 4.1 is required for every message in the corpus. We explain how this is calculated for the multi-tap and predictive text entry in Sections 4.3.1 and 4.3.2 respectively. As there is more than one way of inputting certain characters or words, the way in which we calculate the actions here represents only one of the many ways. In our calculations, we also assume that there are no input mistakes. Finally in Section 4.3.3 we propose two different models to measure the efficiency of text entry methods.

4.3.1 Multi-tap Text Entry

In order to calculate the number of each of the actions performed by user, we need to examine the message character-by-character. To further explain how each of the actions is calculated, we introduce the notation: *num (act)*, to mean the number of times *act* is performed where *act* is an action specified in Table 4.1.

In multi-tap, a character is entered by repeatedly pressing a key. For each character entry, the first key press is counted as an MPAlphaK action and subsequent key presses are counted as an RPAlphaK action. For example, to enter the character "s" which is on the fourth position using the original keypad, Figure 4.1, *num (MPAlphaK) = 1* and *num (RPAlphaK) = 3*. If it happens that the previous character and the next character to be entered are located on the same key, then the first key press of the next character is considered as an RPAlphaK action. To illustrate this, consider the word "on" where both letters share the 6-key, *num (MPAlphaK) = 1* and *num (RPAlphaK) = 4*.

Figure 4.1 The original mobile phone keypad

The MPHAlphaK and the RPHAlphaK operators are typically used to describe the action performed by user while entering numbers. For example, to enter a string of numbers such as "1330", *num (MPHAlphaK) = 3* and *num (RPAHlphaK) = 1*.

The operators MPSymTabK, MPSelectK, MPDirK and RPDirK describe the user actions performed when entering a symbol in the symbol table. Basically, MPSymTabK refers to the action of getting to the symbol table, MPDirK and RPDirK refers to the actions of navigating around the symbol table and MPSelectK refers to the action of selecting a symbol. Different phones have different symbol tables but they normally have the common punctuation marks and symbols placed on the first two rows. The symbol table from Nokia 6510 is used in the calculation here and is given in Figure 4.2.

.	,	'	?	!	"	-	(	)
@	/	:	_	;	+	&	%	*
=	<	>	£	€	$	¥	¤	[
]	{	}	\	~	^	¡	¿	§
#	|							

Figure 4.2 Symbol Table

Here, we assume that we can traverse around the symbol table in four directions, up, down, left and right and that we will always use the shortest path. For instance, to select the symbol "&", we must shift 6 units to the right and then 1 unit down. Therefore, *num (MPSymTabK) = 1*, *num (MPDirK) = 2*, *num (RPDirK) = 5* and *num (MPSelectK) = 1*.

The action of the user changing from upper-case to lower-case or vice versa is described by the action MPModeK. For example, the word "haPPy" would require users to change to upper-case before typing the first "P" and then change to lower-case after typing the second "P", *num (MPModeK)=2*.

The operator Wait is counted when two consecutive characters from the same key are entered. For example, to input the word "cab", *num (Wait) = 2* because the letters "c", "a" and "b" are all on the 2-key, refer to Figure 4.1. The number of Timeout Kill is equivalent to the number of Wait, it just depends on which scheme users prefer to use.

To further illustrate how we compute the number of different actions for a message, we explain the computation using an artificial message, “Reach home @ ard 930". The actions required for each character in the message are listed in sequence in Table 4.3.

	Character	Actions		Character	Actions
1	R	1 MPAlphaK, 2 RPAlphaK	11	SPACE	1 MPAlphaK
2	e	1 MPAlphaK, 1 RPAlphaK	12	@	1 MPSymTabK,1 MPDirK, 1 MPSelectK
3	a	1 MPAlphaK	13	SPACE	1 MPAlphaK
4	c	1 Wait, 3 RPAlphaK	14	a	1 MPAlphaK
5	h	1 MPAlphaK, 1 RPAlphaK	15	r	1 MPAlphaK, 2 RPAlphaK
6	SPACE	1 MPAlphaK	16	d	1 MPAlphaK
7	h	1 MPAlphaK, 1 RPAlphaK	17	SPACE	1 MPAlphaK
8	o	1 MPAlphaK, 2 RPAlphaK	18	9	1 MPHAlphaK
9	m	1 Wait, 1 RPAlphaK	19	3	1 MPHAlphaK
10	e	1 MPAlphaK, 1 RPAlphaK	20	0	1 MPHAlphaK

Table 4.3 Actions for typing “Reach home @ ard 930" in multi-tap

In Table 4.3, the letter “c” in the fourth entry requires 1 Wait because the previous character “a” is located on the same key as that of “c”. Because “a” and “c” are located on the same key, users do not have to move his finger to the 2-key after typing “a” and therefore we do not have any MPAlphaK actions. Therefore, no MPAlphaK action is needed and users just have to repeatedly press the 3-key thrice. To input the symbol “@” using the symbol table in Figure 4.2, users have to change to the symbol table mode, represented by the operator MPSymTabK; then move and press the down directional-key, represented by MPDirK to select the symbol; and then press to select the symbol, represented by MPSelectK. The number characters in entries 18 to 20 are entered by moving to the key that represents the numeral, pressing and holding on to that key.

4.3.2 Predictive Text Entry

To calculate the number of each action performed by a user who is using predictive text entry to type a message, we need to break the message up into tokens of words. For each of the tokens, we can then calculate the number of different actions needed to enter the token. Here, we make use of the notation *num* which has been introduced in Section 4.3.1.

In predictive text entry, only one key press is needed per character. Each key press is counted as an MPAlphaK action. A key press is counted as an RPAlphaK action when the previous character entered shares the same key as the current character. Thus, to enter the word “need”, *num (MPAlpahK) = 2* and *num (RPAlphaK) = 2*.

The operators MPHAlphaK, RPHAlphak, MPSymTabK, MPSelectK, MPDirK, RPDirK and MPModeK are calculated in the same way as that in Section 4.3.1.

The operators MPNextK and RPNextK describe the action of scrolling through the word options by user. For example, the word “card” is the fourth word in the word list, to scroll to the word, we need to press the NEXT key three times, therefore *num (MPNextK) = 1* and *num (RPNextK) = 2*.

Likewise, we explain how we compute the different actions required to type a message using predictive text entry with the same artificial message in Section 4.3.1. Table 4.4 lists the actions that are necessary for each word.

	Word	Actions
1	Reach	4 MPAlphaK, 1 RPAlphaK
2	SPACE	1 MPAlphaK
3	home	3 MPAlphaK, 1 RPAlphaK, 1 MPNextK
4	SPACE	1 MPAlphaK
5	@	1 MPSymTabK,1 MPDirK, 1 MPSelectK
6	SPACE	1 MPAlphaK
7	ard	1 InsertWord, 3 MPAlphaK, 2 RPAlphaK
8	SPACE	1 MPAlphaK
9	9	1 MPHAlphaK
10	3	1 MPHAlphaK
11	0	1 MPHAlphaK

Table 4.4 Actions for typing “Reach home @ ard 930" in predictive

As we can see from Table 4.4, the word “home” requires only three MPAlphaK and one RPAlphaK because “o” and “m” are located on the same key, so we only need to move to the 6-key once. Likewise, the word “Reach” requires only four MPAlphaK and one RPAlphaK because the letters “a” and “c” share the same key. The word “ard” cannot be found in the dictionary and will have to be inserted. Therefore, user will have to perform one InsertWord action and then spell the word out using multi-tap.

4.3.3 Measuring Efficiency

We use two types of models to measure the efficiency of each of the different text entry methods. They are the keystrokes model and the time model. Both models use the SMS corpus collected for evaluation of efficiency.

Keystrokes Model

In the keystrokes model, a keystroke is equivalent to one action performed by user. Thus, we measure efficiency by calculating the average number of keystrokes per message:

$$averageKeystrokes = \frac{\sum_{m \in Corpus} totalKeystrokes(m)}{totalNumMessages}$$

where *totalKeystrokes(m)* is the total number of keystrokes or actions needed to type a message, *m*, and *totalNumMessages* is the total number of messages in the corpus.

Although the average number of keystrokes is a good approximation of the efficiency of a text entry method, it has its inadequacies. This is because it assumes that every action takes the same amount of time. All actions irregardless of its type are counted as one keystroke and no distinctions are made between each of the different actions. However, this is not the case because some of these actions require more time and some require much lesser time. To be more precise, we need the time model.

Time Model

In the time model, each action has a specified amount of time. Therefore, the total time required to type a message, m, is calculated by:

$$totalTime(m) = \sum_{a \in Actions} num(a) \times time(a)$$

where *num(a)* is the number of times action *a* is performed, *time(a)* is the time needed to perform action *a* and *Actions* is the set containing all the actions required to type *m*. The timings of each action is given in Table 4.3.We can then measure efficiency using the SMS corpus by calculating the average time per message:

$$averageTime = \frac{\sum_{m \in Corpus} totalTime(m)}{totalNumMessages}$$

where *totalNumMessages* is the total number of messages in the corpus.

5. Methods and Results

We measure the efficiency of the multi-tap and predictive text entry methods and present the results in Section 5.1. In the remaining sections, we present a couple of methods used to improve the text entry methods and the results of the improvement. All the evaluations of text entry methods are done based on the methodology described in Section 4.3. We assume that users always input numbers by pressing and holding the number key. In addition, we also assume that users will always input symbols that are mapped to the 1-key by pressing the 1-key and remaining symbols not mapped to 1-key will be entered using symbol table.

5.1 Baseline

Here, we evaluate the two text entry methods without any improvements using the keystrokes model and time model. But only the result of the keystrokes model is cited here. Evaluations done here is a basis for comparisons with the improved text entry methods suggested in the following sections.

5.1.1 Multi-tap Text Entry Method

In our computation, we make the assumption that the initial mode when typing a message is the upper-case mode and that mode changes can be made throughout the message to suit the cases of the letters. For example, if the first letter of a message is in upper-case then no change of mode is required, but if the first letter is in lower-case then we have to change to the lower-case mode. Using the keystrokes model, the average number of keystrokes required for the messages in the corpus is 118.925.

5.1.2 Predictive Text Entry Method

In order to compute the number of actions required for each message, we need a dictionary of words. We compile this dictionary from all the words that can be found in the SMS corpus. We do this by tokenizing a string of message into words using spaces, punctuation marks, symbols and numbers as delimiters. Common punctuation marks and symbols that can be obtained from the 1-key are also treated as words. In addition, combinations of the apostrophe mark and letters such as ‘m, ‘ll, ‘t, ‘s, ‘ve, ‘re and ‘d are also considered as words in the dictionary. In order to count the number of MPNextK and RPNextK actions required, we need to know the number of NEXT key presses for each word in the corpus. That is, we need to find out the position of each word in the word list in T9. As T9 is proprietary, we cannot obtain a copy of the dictionary that it uses. Thus, the position of each word in the word list is obtained by inputting each word manually

into a T9-enabled phone and then counting how many times the NEXT key is pressed before the word appears. A value of -1 is assigned to words that cannot be found in the T9 dictionary. Therefore the dictionary compiled here does not include all the words in the T9 dictionary. It only contains those words that we have seen before in the corpus.

In our computation, we ignore any actions carried out by user to realize that a word is not in the T9 dictionary. This is because it is difficult to quantify the actions that need to be done as we do not have the full T9 dictionary. We also assume that users input words not in the dictionary by typing the word in multi-tap and then inserting the word into the dictionary. Two calculations for predictive text entry are done, one with auto-capitalization supported and the other without. Auto-capitalization is a function where the first letter of a sentence is automatically capitalized. In the case where auto-capitalization is not supported, the initial mode is the upper-case mode.

Using the keystrokes model, the average number of keystrokes is 74.004 when auto-capitalization is supported and 76.207 when auto-capitalization is not supported. Comparing the results of predictive and multi-tap text entry, we see that predictive text entry performs about 36% better than multi-tap.

5.2 Reordering Letters for Multi-tap Text Entry Method

In English, the frequency distribution of letters in the alphabet is not balanced. Certain letters such as vowels appear more frequently while letters such as ‘q’ occur rarely. Figure 5.1 shows the frequency distribution of all English letters obtained from WebBase (UC Berkeley, n.d.) which computes the frequency of characters from a large number of web pages. On the other hand, Figure 5.2 shows the frequency distribution of all English letters in the corpus.

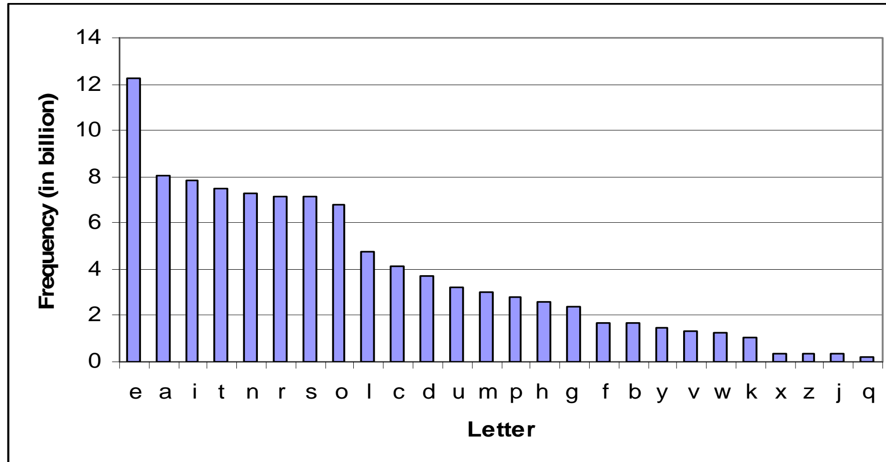

Figure 5.1 Letter frequency table from WebBase statistics

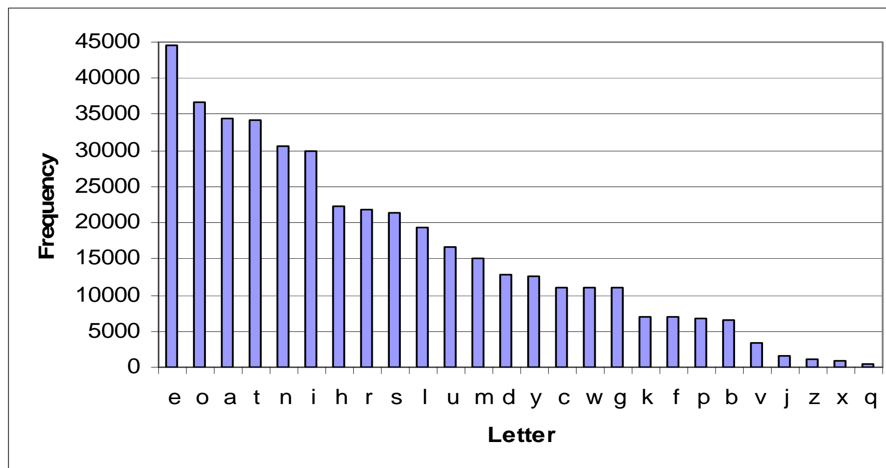

Figure 5.2 Letter frequency table from SMS corpus

Comparing Figure 5.1 and Figure 5.2, we see that the letter frequency table for the corpus is rather close to that obtained from the website. Both graphs show that vowels and the letters “t”, “n” and “r” appear frequently while letters such as “j”, “q”, “z” and “x” seldom occurs in English words. An interesting point to note is that although the letter “h” occurs less often in English texts, it appears quite frequently in SMS messages. The letter “h” is the seventh most common letter in the corpus and this may be due to the fact that many people usually start their messages with “hi” or “hey”. Since Figure 5.2 is a more accurate portrayal of the language used in SMS messages, we use the letter frequency table in Figure 5.2 for our analysis. This also explains why we need to collect a corpus of SMS messages for our analysis.

As we can see from Figure 5.2, the letter "o" has a much higher frequency than that of "m" (approximately 2.45 times higher). But using the original keypad in Figure 2.1, where the letters "o" and "m" are on the same key, the more common letter "o" requires three key presses while the letter "m" requires only one key press. Thus an alphabetic arrangement of letters on the keypad is very inefficient and we attempt to reorder the letters on each key to improve efficiency. The letters are reordered according to the occurrence of the letters in the SMS corpus. As a result, letters that appear more often in the corpus require fewer keystrokes. Figure 5.3 shows the layout of the keypad that has been reordered. In this new layout, the letter "o" requires just one key press and the less frequent letter "m" requires three key presses.

1	2 ACB	3 EDF
4 IHG	5 LKJ	6 ONM
7 RSPQ	8 TUV	9 YWZX

Figure 5.3 Reordered keypad layout

Using the new reordered keypad in Figure 5.3 for computation, the average number of keystrokes obtained is 95.336. The results here show an improvement of 19.8% over the results of the multi-tap baseline.

An advantage of reordering over remapping the letters is that users can adapt quickly to the new layout of the keypad as the letters are still located on the same keys but with a different position.

5.3 Remapping Letters for Multi-tap Text Entry Method

Likewise, remapping letters is done based on the fact that the occurrence of letters in English is biased. Letters on the keypad are remapped by assigning fewer keystrokes, that is, the smaller positions on keys to letters of higher frequencies. As a result, letters with higher frequencies require fewer key presses. We can see from Figure 5.2 that the frequency of the letter "o" is roughly 23 times higher than that of letter "j". However, the letter "o' requires two more key presses than letter "j" if we use the original keypad in Figure 2.1. Figure 5.4 shows the layout of the keypad remapped according to the frequencies of letters in the corpus. In the new layout, the more common letter "o" requires only one key press and "j" requires three key presses.

1	2 ESG	3 OLK
4 AUF	5 TMP	6 NDB
7 IYVX	8 HCJ	9 RWZQ

Figure 5.4 Remapped keypad layout

We evaluate the efficiency of multi-tap using the remapped keypad in Figure 5.4. The average number of keystrokes is 91.673. From the results, we see that the remapped keypad on average helps the user save 3.663 keystrokes more than the reordered keypad.

A disadvantage of this remapped keypad is that users may require some time to adapt as the letters may be located on different keys. However, this layout is more optimal as compared to that in Figure 5.3.

5.4 Huffman Codes as a Text Entry Method

In this section, we describe how we use Huffman Encoding to assign unique key sequences to each letter. And then use the Huffman codes generated to input characters. But before we do that, an introduction to Huffman Encoding is given in Section 5.4.1.

5.4.1 Background-Huffman Encoding

Normally, we would use a fixed length of bits to encode each character. For example in the ASCII code, seven bits are used to encode each character. Therefore codes such as the ASCII are known as fixed-length codes. But since certain letters appear more frequently, it would be more efficient to encode these letters using fewer bits and then encode other letters which do not occur as frequently using more bits. Codes which use a different number of bits for different symbols are termed variable-length codes. One of the difficulties in using variable-length codes is not knowing when the codes for a character ends. A solution is to design a prefix code where no complete code for any character is the prefix of a code for another character.

As such, a data compression technique that assigns variable-length prefix codes to characters was developed by David Huffman. In Huffman encoding, we begin with a set of nodes containing the characters and their respective frequencies. Then we choose two nodes with the lowest frequencies and join them with a new node that has a frequency equal to the sum of both frequencies. Then we replace the two nodes from the original set by this new node. This process

is repeated until there is only one node left in the set. The only node left is the root of the Huffman tree. Thus more frequent characters are near the root and are encoded with fewer bits while the rare characters are far from the root and are encoded with more bits.

We demonstrate the workings of Huffman encoding with an example. Suppose we want to encode the four letters "A", "B", "C" and "D" with frequencies 12, 3, 9 and 6 respectively. First, we combine the nodes {B} and {D} as they have the lowest frequencies. Next we combine the composite node {B, D} and {C}. Finally, we combine the two remaining nodes, {A} and the composite node {B, C, D} in the set. The Huffman tree obtained is given in Figure 5.5. From Figure 5.5, we see that the most frequent letter "A" is encoded with only one bit and the rarer letters "B" and "D" are encoded with three bits. The left branch from a node is assigned the 0 bit and the right branch is assigned the 1 bit. The code for each character can be read off the branches that follow from the root to the leaf node of the character. Thus the code for "D" would be 101.

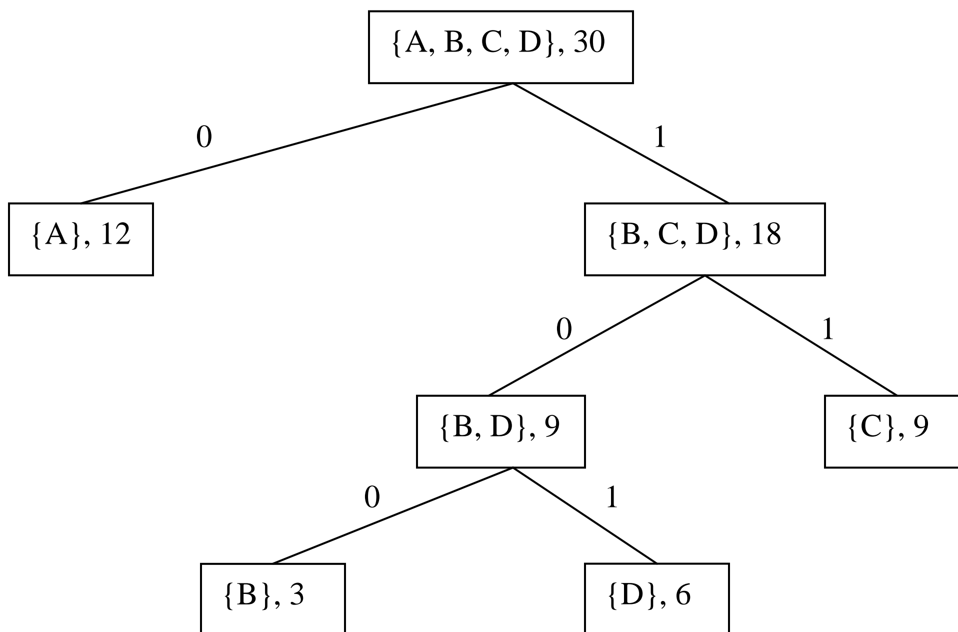

Figure 5.5 Example of Huffman tree

To decode a bit sequence, we traverse the Huffman tree from the root starting with the first bit of the sequence. Each bit in the sequence will determine if we take the left or the right branch. Once we reach a leaf node, we have successfully decoded a character. The bit sequence 100011 is decoded into the string "BAC".

5.4.2 Encoding Characters Using Huffman Encoding

Here, we attempt to use Huffman encoding to generate codes for all characters that are found in the SMS corpus. Since there are 12 keys on the keypad, we make use of all of them to encode the characters. Therefore the Huffman tree generated here is a 12-ary tree instead of a binary tree like the one in Figure 5.5.

To ensure that the 12-ary Huffman tree generated is an optimal one, we cannot simply follow the algorithm in Section 5.4.1 and straightaway combine the 12 character nodes that have the lowest frequency. We have to do some computation to make sure that we have will always have 12 nodes to combine in the later stages of the encoding process. To know how many characters we have to combine in the first step so as to encode *k* characters using an *n*-ary tree, we have to first calculate *k* modulo $n-1$, $k\%(n-1)$. If the result is zero then we have to combine $n-1$ character nodes and if the result is one then n nodes are combined. Otherwise, we combine the number of nodes indicated by the result. For example, we want to encode 26 letters using a 12-ary tree. In the first step of Huffman encoding, instead of combining 12 character nodes with a new node, we only combine four. After replacing the four nodes with the newly created node, we have 23 nodes in total. Then we can proceed with the encoding process normally by combining 12 nodes at each of the remaining stages. Note that starting with 23 nodes, we will always have 12 nodes to combine at each stage.

First, we have to find out the frequency of each of the 117 distinct characters in the corpus. As we are using all the 12 keys for encoding we do not have a spare key to specify a letter in upper-case or lower-case. Thus the upper-case and lower-case of a letter is considered as two separate characters. Then we sort the characters according to the frequencies and start generating the 12-ary Huffman tree for the characters. Since $117\%(12-1)$ is 7, in the first step we only combine the seven character nodes with the lowest frequencies. A portion of the Huffman tree produced is given in Figure 5.6 and the complete set of Huffman codes for all 117 characters can be found in Appendix A. We can see from the figure that commonly used characters such as "r", "a", ".", SPACE and "s" are placed closer to the root and only require one key press. Whereas characters like "q" and "z" are placed further away from the root and require three key presses. For example, to spell out the letter "f", we have to input the key sequence "9#". Therefore, users input a character by specifying the code sequence that represents that character. This scheme of text input is rather similar to that of multi-tap text entry. Although different forms of key sequences are

used, both methods allow users to input a character unambiguously by specifying the corresponding key sequence for the character.

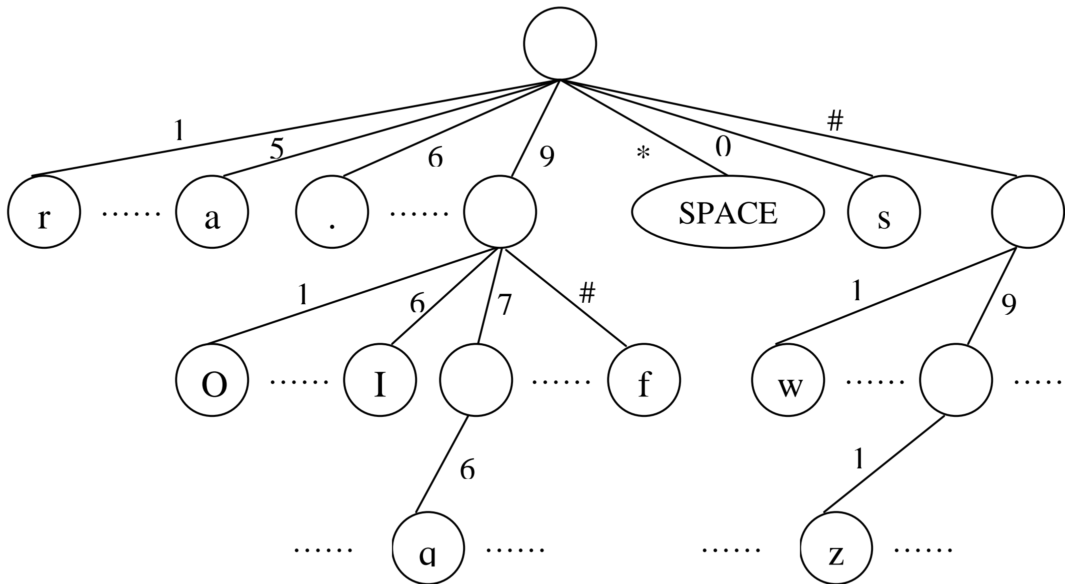

Figure 5.6 Huffman tree for characters in corpus

By now, one may realize that such an input scheme may impose an excessive cognitive load on the user. To use this input scheme, users will have to remember the codes for each character. For example, it may be difficult for users to remember that the key sequence “976” actually stands for “q”. However, the study of human cognition is outside the scope of this report. Nevertheless, Huffman encoding is an optimal coding technique that will generate codes that specify each character unambiguously.

Counting each key press as a keystroke, the average number of keystrokes is 80.910 which is an improvement of 32.0% over the normal multi-tap method. However, the time model is not applicable here as we do not have the timings that users take to input a code sequence when using this input scheme.

5.5 Remapping Letters for Predictive Text Entry Method

In the predictive text entry method, disambiguation is done by matching the key sequence entered by the user to words in the dictionary. As such, multiple words may share a common key sequence and these words are said to collide. When words collide, user has to press the NEXT key to choose the intended word. From Table 4.3(c) and (d), we see that NEXT key presses are

expensive as they require more time than some other actions and is one of the actions that are performed frequently. Users take a longer time to press the NEXT key because cognitive processes are involved when he is scrolling through the words in the list. So in order to reduce the number of NEXT key presses, letters on the keypad are remapped so as to minimize word collisions.

The number of ways to remap the 26 letters on a keypad is:
$^{26}C_3 \times {}^{22}C_3 \times {}^{20}C_3 \times {}^{17}C_3 \times {}^{14}C_3 \times {}^{11}C_4 \times {}^{7}C_3 \times {}^{4}C_4 = 1.5 \times 10^{19}$
where $^{x}C_{y}$ means the number of ways to choose y from x.
As there are approximately 1.5×10^{19} different keypad layouts that are possible, it is impossible to search the entire space for the optimal keypad layout. Thus, the genetic algorithm is used to search for the keypad layout that is close to optimum.

5.5.1 Background-Genetic Algorithm

The genetic algorithm is based on the evolutionary process that happen to organisms in Nature. Offspring produced from organisms are not identical but they share some similarities with their parents. Organisms that naturally fit their environments will live to reproduce and those less fit will die off without leaving any offspring. Therefore each new generation has organisms that are genetically similar to the fit members of the previous generation.

Genetic algorithm uses this mechanism to “evolve” an individual that is considered to be the fittest, as measured by a fitness function. The algorithm starts with a population of one or more individuals and selected individuals reproduce to yield a new population for the next generation. The same selection and reproduction processes take place for the next generation population pool. The processes repeats itself until the best individual is found.

An example of the genetic algorithm is given in Figure 5.7. An initial population of 4 individuals is present in (a). In (b), they are evaluated by a predetermined fitness function and their scores are as shown. Individuals with higher fitness are more likely to be selected for reproduction. In (c), selected individuals are paired up randomly for reproduction. The crossover points are randomly chosen and the offspring after crossover has taken place is shown in (d). Lastly, mutation occurred to two of the offspring in (e). The offspring produced makes up the population used in the next generation.

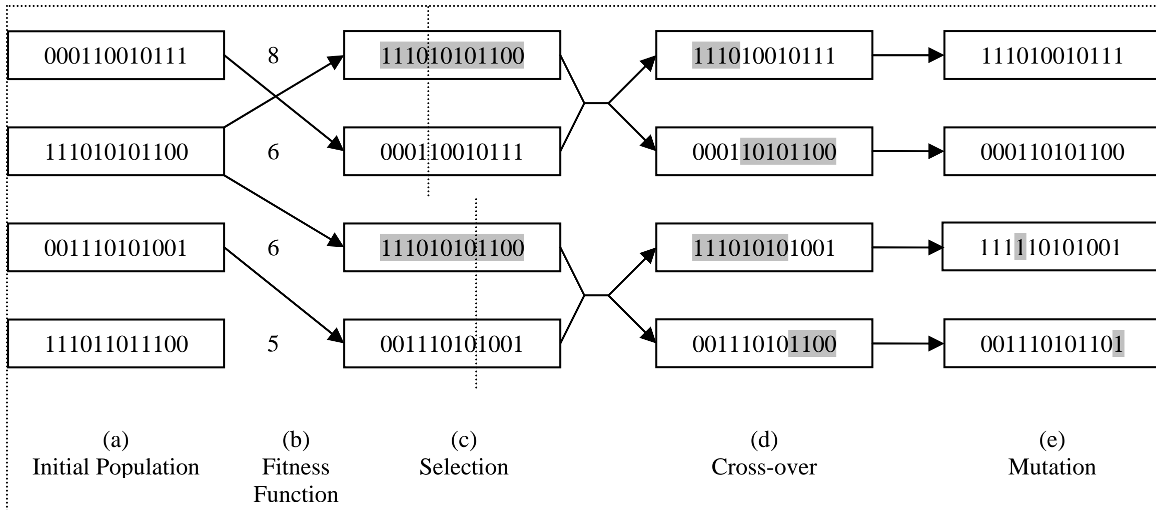

Figure 5.7 An example of the genetic algorithm

5.5.2 Using Genetic Algorithm to Remap the Keypad

In the first cycle, using the original alphabetic arrangement of the keypad as the parent, 100 offspring are generated. 80% of the offspring are generated by swapping the positions of 2 randomly picked letters on the keypad, SWAP2. The remaining 20% of the offspring are generated by randomly swapping the positions of 4 letters picked at random, SWAP4. We use two swap functions so as to obtain wider variations of offspring in a shorter time since SWAP4 can exchange the positions of more letters at one time. This implementation of the genetic algorithms uses asexual reproduction as all offspring are produced by only one parent and no crossover among keypads is carried out. The fitness of each all the offspring is then evaluated and 10 offspring with the best fitness are added into the population pool. Figure 5.8 gives an illustration of how letters on the keypad are swapped using SWAP2 and SWAP4.

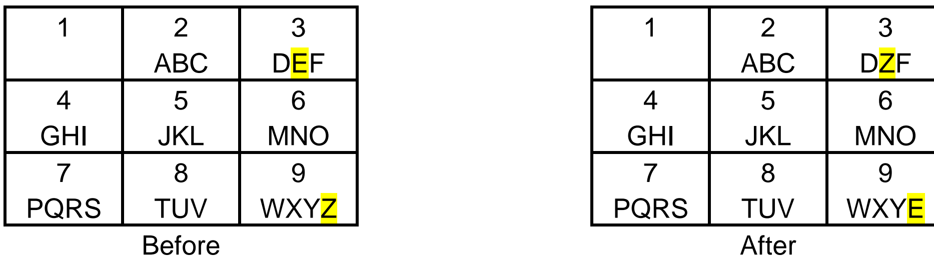

Figure 5.8(a) Keypads before and after SWAP2

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ

Before

1	2 AOC	3 DEF
4 GHI	5 JKL	6 MNT
7 PBRS	8 QUV	9 WXYZ

After

Figure 5.8(b) Keypads before and after SWAP4

In Figure 5.8(a), the positions of two randomly chosen letters, “Z” and “E” are swapped. From Figure 5.8(b), we can see that the positions of the letters “B”, “Q”, “O” and “T” have changed after carrying out the function SWAP4.

In each of the remaining cycles, every parent in the population pool will be used to generate 8 offspring using SWAP2 and 2 offspring using SWAP4. This gives a total of 100 offspring generated at each iteration. All the offspring generated by the different parents will be evaluated together with the parents by the fitness function and the 10 keypads with the best fitness will replace the keypads in the population pool. A cycle then starts again with the new parents in the population pool. This continues for the number of iterations specified. Therefore the population pool will always contain the 10 best keypads that will be used for reproduction.

The fitness of a keypad is measured by calculating the total number of NEXT key presses that is required to type in all the messages in the corpus using the keypad. To calculate the NEXT key presses needed for a word, we need to first compute words that share the same key sequence according to the placement of letters on the keypad. Words that have the same key sequence are then sorted by frequency of the word in the corpus. As a result, words that occur frequently should require fewer NEXT key presses. The total number of NEXT key presses for a particular keypad is then calculated by the following formula:

$$\sum_{w \in corpus} freq(w) \times numNext(w)$$

where w is a word in corpus and freq(w) is the frequency of w in the corpus and numNext(w) is the number of NEXT key presses required for w. Therefore the lower the fitness of the keypad, the better the keypad is.

The implementation of the genetic algorithm described above is run for 200 iterations. Figure 5.9 shows a plot of the fitness level of the best keypad in the pool of population at every iteration.

After the 131st iteration, the fitness level of the best keypad remains unchanged. In the remaining 69 iterations we further produce and evaluate approximately 6900 keypads and none of these keypads can perform better than the best keypad generated in the 131st iteration. Thus indicating that the best keypad produced in the 131st iteration is likely to be very close to optimum.

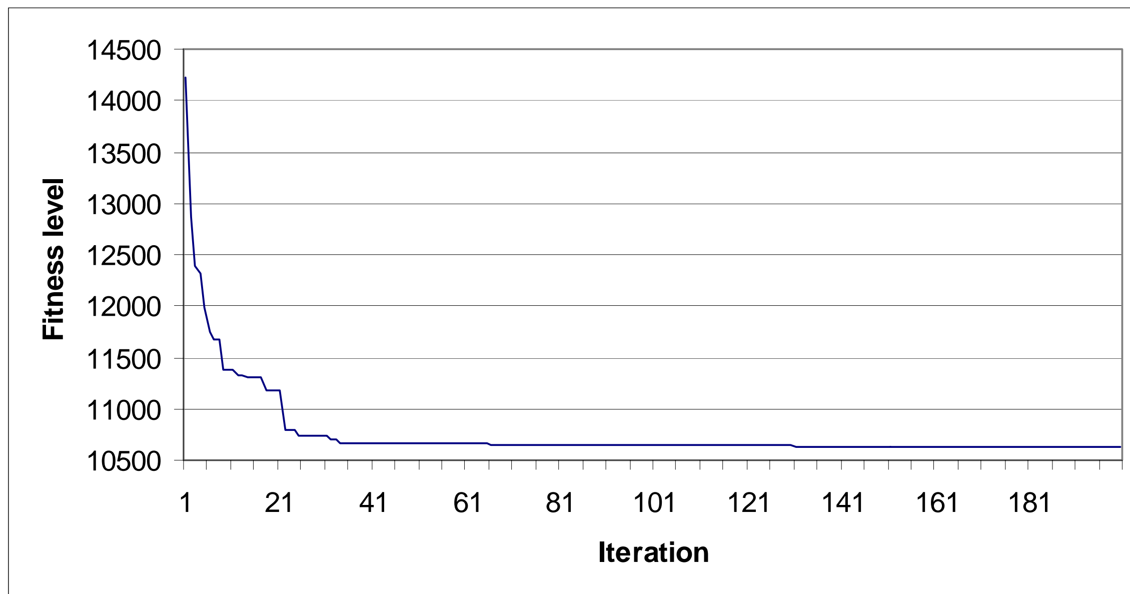

Figure 5.9 Plot of fitness level versus the iteration

Next, we want to calculate the percentage overlap of the populations from two neighboring iterations. To do this, we introduce the notion of equivalent keypads. Two keypads are considered to be equivalent when the combinations of letters on the keys are the same and irregardless of where on the keypad each letter combination is placed. The 2 keypads in Figure 5.10 are equivalent although in the second keypad, the letter sets on 3-key and 7-key have been exchanged and the positions of the letters on the 7-key has been changed from 'dqf' to 'dfq'.

1	2 WBC	3 DQF
4 JHI	5 GKL	6 MNO
7 PERS	8 TUV	9 AXYZ

1	2 WBC	3 PERS
4 JHK	5 GKL	6 MNO
7 DFQ	8 TUV	9 AXYZ

Figure 5.10 Two equivalent keypads

The graph in Figure 5.11 shows the percentage of the population that has survived from the previous iteration to the current iteration. This is done by computing the percentage of equivalent keypads in the population pool of consecutive iterations at various intervals. The graph indicates that from iteration 19 to 20 the population did not change. One of the reasons may be that none of the offspring generated at the 20th iteration are better than their parents. This is probably the local maxima where the algorithm seems to have found the best keypad. The graph levels off for quite long after the 39th iteration signifying that any future iterations are unlikely to make changes to the population.

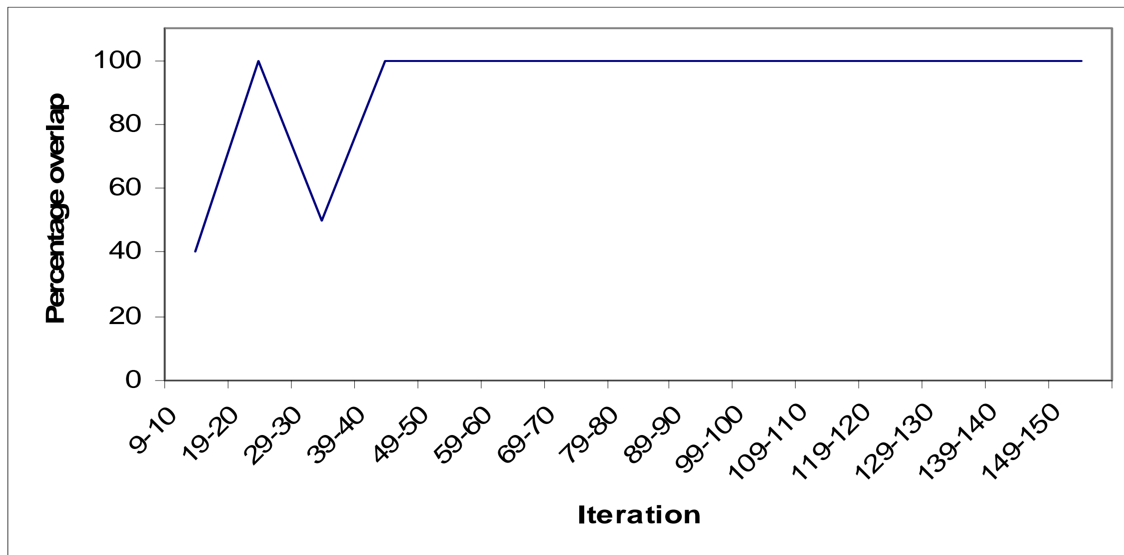

Figure 5.11 Percentage overlap of the population at various iterations

The layout of the fittest keypad that is generated by the genetic algorithm is given in Figure 5.12. We evaluate the remapped keypad in using predictive text entry with auto-capitalization and find that the average number of keystrokes is 62.357. This is an improvement of 15.7% over the baseline for predictive text entry method.

1	2 EGP	3 CDR
4 HIVZ	5 LNQX	6 KMO
7 BSY	8 JTU	9 AFW

Figure 5.12 Remapped keypad layout

5.6 Using Bigrams for Predictive Word Completion

To further improve the predictive text entry method, predictive word completion is proposed. Predictive word completion attempts to complete the current word based on the previous word. In order to do this, we have to analyze the corpus and source out the words that can be completed with some degree of certainty.

5.6.1 Background

In this section, we introduce the noisy channel model and n-gram model that will be used to determine which words to predict in Section 5.6.2.

Noisy channel model/Bayesian classification

The intuition of the noisy channel is to treat the surface form (noisy word) as an instance of the lexical form (correct word) which has been passed through a noisy communication channel. This channel introduces "noise" and makes it hard to recognize the correct word. The task now is then to recover the correct word from the noisy word. In Bayesian classification, we are given some observation and our job is to determine which set of classes it belongs to. Thus, the estimation of the correct word given an observation *O* and a vocabulary *V* is:

$$correctWord = \arg\max_{w \in V} P(w \mid O)$$

N-gram model

An N-gram model is a model for word prediction, it uses the previous N-1 words to predict the next one. Thus, for each of the words in the corpus, we need to find:

$$P(w_n \mid w_{n-N+1}^{n-1}) .$$

In a bigram model, where N = 2, we use the previous word to predict the following word. To find the probability of a word given the previous word we use the following:

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}, w_n)}{C(w_{n-1})}$$

where $C(x,y)$ is the count of the bigram (x,y) in the corpus.

5.6.2 Determining When to do Predictive Word Completion

We can visualize the problem of doing predictive word completion using the noisy channel model. The current word that user is inputting has some noise in it because the user has so far entered

only a partial key sequence. Thus we try to guess the full word that user wants based on the partial key sequence entered and the previous word.

Firstly, the frequency of all the bigrams that exist in the corpus is computed. The first word in the bigram is the given word and the second word in the bigram is the word to be predicted. Secondly, the probability of the current word given the previous word and various lengths of input for the current word is calculated using the formula given below:

$$P(currWord \mid prevWord \,\&\, keyseq) = \frac{C(prevWord, currWord)}{\sum_{w \in words(keyseq)} C(prevWord, w)} \qquad (5.1)$$

where *prevWord* is the previous word that has been entered by the user, *currWord* is the current word that user is inputting, *keyseq* is the partial key sequence that has been entered by user so far, *words(k)* is a function that returns the set of words which have key sequences starting with k and $C(x,y)$ is the count of the bigram (x,y) in the corpus.

Condition 1

Given a previous word, p, and a key sequence, k, a word, w, is predicted only when:

(i) $P(w \mid p \,\&\, k)$ which is calculated using Equation 5.1 is greater than the threshold of 0.5, and

(ii) $C(p,w)$ must be greater than 2 .

Condition 2

Given a previous word, p, a set of words, W, which shares the same partial key sequence, k, and for all $w \in W$ Condition 1 holds, then the word that is predicted is given by:

$$wordpredicted = \arg\max_{w \in W} P(w \mid p \,\&\, k)$$

If there exists more than one $w \in W$ such that $P(w \mid p \,\&\, k)$ is maximized, then the w with the longest word length is predicted.

Condition 3

If a given key sequence, k, codes for at least one complete word then no word completion is done but all complete words are suggested according to their probability calculated by Equation 5.1.

Predictive word completion for a word will only be kick-started after the first key that corresponds to the first letter to the word has been entered. This is done so as to make the

prediction more accurate, because one key press can narrow down the number of words that can follow a particular word to words that begin with the three or four letters as specified by the key press.

The average number of keystrokes obtained is 63.546 if we use the original mapping of letters on the keypad in Figure 2.1 to do predictive word completion and with auto-capitalization enabled. Therefore predictive word completion has saved 14.1% keystrokes as compared to using predictive text entry only.

5.7 Combining Predictive Word Completion with Remapped Keypad

Both predictive word completion and the remapped keypad in Figure 5.12 improve the efficiency of predictive text entry. So if we can combine these two techniques together, we can further improve the efficiency of predictive text entry.

First, we have to re-compute the probability function given in Equation 5.1 for each bigram as the mapping of letters has changed. After doing this, we will then know which word and when to predict. Having this information at hand, we can then proceed with doing predictive word completion using the remapped keypad.

Using the combination of both techniques and with auto-capitalization supported, the number of keystrokes calculated is 57.878. The result shows an improvement of about 8% compared to just using predictive word completion or remapped keypad alone.

5.8 Summary of Results

In this section, we give a summary of all the results obtained from Section 5.1 to 5.7 in Table 5.1 for multi-tap text entry and Table 5.2 for predictive text entry. In all of these tables, we state the average and standard deviation of each of the different text entry techniques. Also given in each of the tables is the percentage improvement of the text entry method over the baseline. In addition, for the time model we have a set of timings for the expert user model and the novice user model. The results given in these tables do not take into account the increase in cognitive load as a result of using these improvement techniques. We are aware of this limitation but it is outside the scope of this report.

From Table 5.1(a), we see that reordering and remapping letters on the keypad improves the efficiency of multi-tap by 19.8% and 22.9% respectively. Although remapping letters performs better than reordering letters, the improvement is only by 3.8%. Because of the similarities of conventional multi-tap and text entry using Huffman codes, we also make a comparison between these two methods in this table. The results show that text entry using Huffman codes makes the most improvement at 32.0%. However, this optimistic result may be because we optimize the number of keystrokes for punctuation marks and symbols as well by generating Huffman codes for them. On the other hand, for remapping and reordering letters we do not do any optimizations for punctuation marks and symbols.

Table 5.1(b) compares the results using the time model. Using the expert user model, the timings for reordering and remapping letters have improved by 8.1% and 9.8% correspondingly. Remapping letters has only made a slight improvement of 1.8% over the reordering of letters. The time model shows that the expert user would on average use 79.373 seconds to type a message and the novice user would use 100.659 seconds.

Comparing Table 5.1(a) and Table 5.1(b), we see that the percentage improvement of each text entry method is greater for the keystrokes model than for the time model. This is probably because the keystrokes saved correspond to those that consume a shorter amount of time. This makes sense as the keystrokes that we are trying to save by remapping and reordering the letters are the repeated taps on the keys and these keystrokes require less time.

Table 5.2(a) shows the results of comparing the different improvements made to the predictive text entry method. In the table, two sets of data are given, one with auto-capitalization and the other without. For the baseline, auto-capitalization saves users 2.9% of keystrokes. Remapping letters, predictive word completion and a combination of both have shown a significantly reduced the number of keystrokes by 15.7%, 14.1% and 21.8% respectively. The greatest improvement shown is by using both a remapped keypad and predictive word completion.

Table 5.2(b) and Table 5.2(c) show the results of the evaluations of the various improvement techniques applied to predictive text entry using the time model. The time model for the expert user reduces the average time required to type a message by 21.1% for the remapped keypad and 18.5% for predictive word completion in Table 5.2(b). A combination of both techniques further reduces the average time, with a 28.6% improvement over the baseline. Using the combination of

both techniques, an expert user would on average use 30.925 seconds to type a message and a novice user would require 86.468.

As we can see from the tables in Table 5.2, the remapped keypad generally performs better than predictive word completion. This may be because we adopt a more conservative strategy for predictive word completion as we only suggest the most probable word and we start predicting only after the first key sequence has been entered. From the tables, we can also see that the gains from these two improvement techniques overlap as the resulting gain from the combination of both is less than the total gains from both improvement techniques. This is probably because both improvement techniques reduce the same type of actions and those are the NEXT key presses.

	Average	Standard Deviation	% improvement
Baseline	118.925	81.915	0
Reordering letters	95.336	65.343	19.8
Remapping letters	91.673	62.801	22.9
Huffman codes	80.91	56.701	32.0

Table 5.1(a) Results for multi-tap text entry using the keystrokes model

	Expert			Novice		
	Average (sec)	Standard Deviation	% improvement	Average (sec)	Standard Deviation	% improvement
Baseline	79.373	54.032	0	100.659	69.017	0
Reordering letters	72.908	49.528	8.1	95.968	65.726	4.7
Remapping letters	71.578	48.583	9.8	94.643	64.749	6.0

Table 5.1(b) Results for multi-tap text entry using the time model

	With Auto-capitalization			Without Auto-capitalization		
	Average	Standard Deviation	% improvement	Average	Standard Deviation	% improvement
Baseline	74.004	51.176	0	76.207	52.34	0
Remapping letters	62.357	43.276	15.7	64.566	44.443	15.3
Predictive Word Completion	63.546	46.542	14.1	65.762	42.671	13.7
Combination	57.878	42.451	21.8	60.096	42.629	21.1

Table 5.2(a) Results for predictive text entry using the keystrokes model

	Expert			Novice		
	Average	Standard Deviation	% improvement	Average	Standard Deviation	% improvement
Baseline	59.791	42.143	0	149.563	103.131	0
Remapping letters	47.175	33.355	21.1	128.774	89.748	13.9
Predictive Word Completion	48.743	36.568	18.5	129.378	94.383	13.5
Combination	42.684	30.925	28.6	120.245	86.468	19.6

Table 5.2(b) Results for predictive text entry with auto-capitalization using the time model

	Expert			Novice		
	Average	Standard Deviation	% improvement	Average	Standard Deviation	% improvement
Baseline	63.235	43.9	0	151.099	103.904	0
Remapping letters	50.633	35.094	19.9	130.363	90.502	13.7
Predictive Word Completion	52.208	38.295	17.4	130.929	95.139	13.3
Combination	46.152	32.759	27.0	121.789	87.221	19.4

Table 5.2(c) Results for predictive text entry without auto-capitalization using the time model

6. DEMO

A prototype that demonstrates predictive word completion with the remapped keypad in Figure 5.12 is implemented. The prototype is written in Java and comes with a Graphical User Interface (GUI). A portion of the keyboard is assigned the 12 keys that are commonly found on mobile phones. In the implementation, the remapped keypad that has been optimized by the genetic algorithm is used.

6.1 Background-Trie

The origin of the name “trie” is from the middle section of the word “retrieval” and the origin of the name suggests that it is primarily used for information retrieval. The trie data structure is an N-ary tree for storing strings in which there is one node for every common prefix, where N is determined by the number of distinct characters that we can have in the strings. Each node in the trie corresponds to either a digit or a character and the leaf node is used to store strings or records depending on the information retrieval task. By traversing the tree based on some input, information at the leaf node at the end of the traversal can be retrieved. For example to store a dictionary of English words, we would use a 26-ary tree where each node can have 26 children, one for each of the letters in the English alphabet.

6.2 Implementation

The primary data structures used in the prototype are 11-ary trees that have a structure similar to that of a trie. All the nodes in the trees can have at most 11 children. They are used to store words in the dictionary and words for predictive word completion. Each node in the tree represents a key on the keypad. Thus, a node corresponds to all the characters that are mapped to that key. Each path from the root to every node represents a unique key sequence. At each node, all the possible words that share a common path from the root are stored. In addition, incomplete words are also stored at the respective nodes along the path. Incomplete words are the prefixes of a complete word. For example, the word “there” has five incomplete words: “t”, “th”, “the”, “ther” and “there”. To find words or incomplete words that share a common key sequence we just need to traverse the tree from the root using the key sequence and the words can be found at the node where traversal ends.

To do predictive word completion, we need to have more than one tree, one for each previous word. This is because the words that can appear given a key sequence may be different for each

previous word. Thus, for every previous word, we need to have a tree, *predictiveTree*, that stores all the words that are likely to follow after it. As these trees only store words that are predicted, another tree, *normalTree*, is constructed to store all the words in the dictionary. In the *normalTree*, complete words are stored according to their probability at each node. The probability of a word is determined by the frequency of the word in the corpus. We chose to implement a common *normalTree* instead of putting all of the words in each of the *predictiveTree*s so as to save memory space.

With each key press from the user, the *predictiveTree* that corresponds to the previous word and the *normalTree* is simultaneously traversed down one level according to the key that is pressed. The words on the nodes of both trees are combined into a word list. In the word list, words obtained from the *predicitiveTree* are placed before the words from the *normalTree*. The word list is displayed on the screen and users can scroll through the list to make a selection.

6.3 Using the Prototype

The prototype reads input from the QWERTY keyboard and the assignment of the mobile phones keys on the keyboard is given in Figure 6.1.

Keyboard	Phone	Keyboard	Phone	Keyboard	Phone
1	1	2	2	3	3
		EGP		CDR	
Q	4	W	5	E	6
HIVZ		LNQX		KMO	
A	7	S	8	D	9
BSY		JTU		AFW	
Z	*	X	0	C	#

Table 6.1 Assignment of mobile phone keys on keyboard

The usage of the prototype is similar to that of T9. Users need only press the key that corresponds to the letter once. Pressing the #-key selects the highlighted word from the word list and appends a space. To scroll to the next word in the word list, press the asterisk-key. If for a particular sequence of key presses, there is a probable word that can be completed, the complete word will appear at the top of the word list. User can select the word by pressing the #-key.

7. Conclusions

In this thesis, we have presented a number of different ways to improve present text entry techniques. These include reordering and remapping letters, using Huffman codes and predictive word completion.

For multi-tap text entry, although using Huffman codes shows the largest improvement, it is the least feasible among the three improvement techniques. This is because it may be cognitively impossible for users to remember the codes for each character. But if users are able to do that, using this input scheme will save them 32% of keystrokes. The next best improvement technique would be remapping letters, which saves users 9.8% of time as compared to normal multi-tap. But using a remapped keypad would require user to perform more cognition than using the reordered keypad. Since remapping letters performs only 1.8% better than reordering letters using the expert time model, it may not be useful to use the remapped keypad.

For predictive text entry, the largest improvement obtained is for using a combination of both remapped keypad and predictive word completion. Expert users on average would be able to save almost one-third of their time. Using the expert time model, the remapped keypad alone gives an improvement of about 21% while predictive word completion alone saves users 19% of their time. This is significant as the normal predictive text entry is already quite efficient so if we can improve it further, we can save more time and keystrokes for users. A very simple prototype, DEMO, is developed to show how the combination of remapped keypad and predictive word completion works.

We evaluate the efficiency of text entry methods using the keystrokes model and time model. The discrepancy in results obtained for the time model and the keystrokes model is the reason why we have both the keystrokes model and time model. While the keystrokes model may be a good indicator for efficiency, the time model gives a much more accurate measurement of efficiency. These two models are developed based on the KLM that we have constructed for mobile phone users. Similar to the KLM for computer users, the KLM for mobile phone users describes the different types of actions that users carry out when doing mobile text entry. In the keystrokes model, we count each action as one keystroke whereas in the time model, each of the actions is weighted by the action time.

In order to collect statistics that can help design text entry methods that are more efficient and also to do analysis, we collected a corpus of 10117 of SMS messages which is publicly available. We also collected timing information for each of the user actions in the KLM for mobile phone users by videotaping phone users typing in sample messages on mobile phones.

As the SMS message corpus was collected mainly from mobile phone users in Singapore, and the improvements made are based on statistics obtained from the corpus, the improvements proposed may only be applicable to Singapore mobile phone users.

Further research needs to be carried out to refine the multi-tap expert and novice user models and include more subjects as these user models are derived from only one subject. As the present keystrokes model and time model assumes that users do not make any mistakes during input, the models may be extended in future to also include mistakes made by users and also the actions taken by users to correct mistakes. We may also need to study the plausibility of using a remapped keypad to find out if the rise in cognitive load as a result of using the remapped keypad offsets the positive results from using it. A usability study of the DEMO prototype may also be carried out in future research.

References

Davis, J.R. (1991). Let your fingers do the spelling: Implicit Disambiguation of words spelled with the telephone keypad. The Media Laboratory, Massachusetts Institute of Technology.

Dunlop, M.D. and Crossan, A. (2000). Predictive Text Entry methods for mobile phones. Personal Technologies, pp. 134-143.

Hasselgren, J., Montnemery, E. and Svensson, M. (2002). HMS: A Predictive Text Entry Method Using Bigrams. Lund Institute of Technology, Sweden.

Huffman Encoding Trees. (n.d.). Retrieved March 20, 2004, from http://mitpress.mit.edu/sicp/full-text/sicp/book/node41.html

Jurafsky, D. and Martin, J.H. (2002). Speech and Language Processing. Prentice Hall, 2002.

Kieras, D. (2001). Using the Keystroke-Level Model to Estimate Execution Times. University of Michigan.

Netsize (2003). European SMS guide, February 2003.

Russell, S.J. and Norvig, P. (1995). Artificial Intelligence A Modern Approach. Prentice Hall, Upper Saddle River, New Jersey 07458, 1995.

Silfverberg, M., Mackenzie, I.S. and Korhonen, P. (2000). Predicting text entry speeds on mobile phones. Proceedings of the ACM Conference on Human Factors in Computing Systems – CHI 2000, New York:ACM, 2000, pp. 9-16.

UC Berkeley and Stanford Digital Library Projects. (n.d.). WebBase. Retrieved March 19, 2004, from http://elib.cs.berkeley.edu/docfreq/

Appendix A – Huffman Codes generated for all characters

Character	Code Sequence	Character	Code Sequence	Character	Code Sequence
a	5	N	#94	¤	#9#6*7
b	99	O	91	[	#9#6*02
c	#3	P	977	]	#9#6*03
d	#6	Q	9799	{	#9#6*04
e	8	R	#21	}	#9#6*05
f	9#	S	#9*	\	#9#6*4
g	#4	T	90	~	#9#62
h	##	U	#92	^	#9#69
i	2	V	#9#7	¡	#9#6*06
j	#93	W	#99	¿	#9#6*1
k	#0	X	#9#3	§	#9#6*2
l	#*	Y	#2#	#	#9#60
m	#7	Z	979*	|	#9#6*3
n	3	SPACE *		0	#25
o	7	.	6	1	#28
p	9*	,	95	2	#96
q	976	'	92	3	97#
r	1	?	98	4	#27
s	0	!	#97	5	97*
t	4	"	9791	6	978
u	#8	-	975	7	970
v	94	(	979#	8	#9##
w	#1	)	#9#8	9	#9#9
x	#26	@	9795	ä	#9#67
y	#5	/	9796	å	#9#6*9
z	#91	:	971	à	#9#61
A	#98	_	#9#68	è	9798
B	#24	;	9794	é	9793
C	#29	+	9797	ì	#9#63
D	#2*	&	#9#5	ñ	#9#1
E	#95	%	#9#66	ö	#9#6#
F	973	*	9792	ø	#9#6*#
G	#20	=	#9#0	ò	#9#64
H	93	<	#9#65	ù	9790
I	96	>	#9#4	ü	#23
J	972	£	#9#6*6	É	#9#6**
K	974	€	#9#6*00	Ö	#9#6*5
L	#22	$	#9#2	Ü	#9#*
M	#90	¥	#9#6*01	Ñ	#9#6*8