Optimizing an Ambiguous Eyes-Free Keyboard

Dylan Gaines, Michigan Technological University, dcgaines@mtu.edu

Abstract

In this paper we explore ways to optimize an ambiguous keyboard that can be combined with an *n*-gram language model to create a familiar, yet efficient eyes-free text entry interface. We propose using genetic algorithms and the *n*-opt algorithm to generate candidate layouts that are optimized for tap clarity metrics. We will optimize layouts that have either four or six groupings of characters, and we will optimize both Qwerty-constrained and unconstrained layouts. We will then compare the performance of these layouts using simulated user data to narrow the field further for user testing.

I. Introduction

In today's society, text entry is a task that most people complete several times a day on a variety of devices. Smartphones, laptop computers, and smartwatches are all increasingly common. However, so are situations in which a user may not be able to visually see a keyboard. Perhaps their visual attention is needed elsewhere if they are multitasking, or perhaps the user has a visual impairment that prevents them from seeing the keyboard. While on a physical keyboard many people are able to touch type, this becomes much more difficult when the keyboard is virtual and the tactile separation between keys is removed.

While some solutions to this problem, such as Apple's VoiceOver, allow users to explore the keyboard with audio feedback before confirming their selection for each character [3], our past research has focused on using an ambiguous keyboard to enable eyes-free text entry [5]. Ambiguous keyboards are those that place multiple characters on the same key (e.g. standard telephone keypads that placed the letters 'ABC' on the number 2, the letters 'DEF' on the number 3, and so on). While some ambiguous keyboards require additional key presses to confirm the character within the grouping, others perform automatic disambiguation by determining the most likely word that corresponds to the sequence of selected keys. If the ambiguous "keys" are mapped to gestures that have little or no location dependency, the interface becomes much easier to use when visual feedback is not possible.

In this paper we will explore different ways to optimize an ambiguous keyboard for use in an eyes-free text entry interface. We will do so by attempting to reduce disambiguation collisions while maintaining a certain level of familiarity in the groupings to aid learnability of the interface. We will combine ideas from past work on this topic as well as introduce novel ideas to create multiple proposed groupings that are likely to have success in future user trials.

II. Background and Related Work

Eyes-Free Text Entry

As mentioned previously, there are several tried and tested methods for performing eyes-free text entry. One example is Perkinput [1], which converts multi-finger tap events into letters using Braille character encodings. The authors compared Perkinput to Apple's VoiceOver and found that users were significantly faster and more accurate using one-handed Perkinput. While a number of other methods use similar Braille encoding techniques, we chose to remove the prerequisite Braille knowledge to target situationally impaired users as well as the portion of visually impaired users that aren't familiar with Braille.

In past work, we proposed an ambiguous keyboard called Tap123 that was based strictly on a Qwerty keyboard [5]. Users would indicate the row of the keyboard by tapping with either one, two, or three fingers, and they would indicate the side of the keyboard by the side of the screen they tapped on (e.g. a one-finger tap on the left side of the screen corresponded to q, w, e, r, and t, while a two-finger tap on the right side corresponded to h, j, k, l, and apostrophe). After recognizing a right swipe gesture to designate the end of a word, the interface would perform disambiguation to determine the most likely word given the tap sequence and previously typed text. While users were able to achieve error and entry rates that were comparable to other methods, we noticed that several common words had identical tap sequences and would require users to scan through the list of disambiguated words if their intended word was not deemed most probable. Frequently, this event required users to go back and retype the word if they moved on before noticing or to slow their entry rate to pay closer attention to the resulting words, both of which reduced the performance of the interface. This observation led us to consider optimizing the character groupings to reduce disambiguation collisions instead of simply using groupings based on the Qwerty layout.

Keyboard Optimization

There has been a considerable amount of past research done on the topic of optimizing keyboard layouts, with several papers introducing pieces of information that are directly relevant to this particular application. The first of these papers, written by Lesher et al. [7], shows that fully enumerating and evaluating every possible set of groupings is computationally infeasible, but proposes an algorithm that can be used to efficiently find locally optimized groupings in ambiguous keyboards. The first step in their algorithm is to compute a confusability matrix for some corpus of English text. They do this by stepping through the corpus one character at a time and keeping track of how frequently a different character is predicted as more likely than the actual next character in the text. The authors define the mutual confusability of two characters α and β as

$$M(\alpha, \beta) = C(\alpha, \beta) + C(\beta, \alpha),$$

where $C(\alpha, \beta)$ is the number of times α was mistaken for β, and $C(\beta, \alpha)$ is the reverse. They go on to further explain that for any number of characters placed in a single ambiguous group, the total mutual confusability is the sum of the mutual confusabilities between each pair of characters.

Once the confusability matrix is computed, Lesher et al. [7] run their "*n*-opt" algorithm. The algorithm starts with a valid arrangement of characters into groupings, and then checks every possible tuple of n characters to see if a particular shuffle of those characters will result in a better overall arrangement according to whatever metric is being optimized (in this case, the confusability of the groupings). If any swaps are made during the course of a single pass, the pass repeats once it has finished. The algorithm continues until a pass completes with no swaps being made. The *n*-opt algorithm requires factorially increasing computations for higher values of *n*; the highest pass completed in the paper was five-opt. Since the *n*-opt algorithm only finds local optimums, the authors start with many initial arrangements and run the two-opt algorithm before running five-opt on the best result.

An alternative approach to keyboard optimization is the use of genetic algorithms proposed by Gong and Tarasewich [6]. In this paper, the authors compared an ambiguous layout that was constrained to be in strict alphabetical order with a layout that was unconstrained and freely optimized. While they were able to fully enumerate the possible constrained layouts, they used genetic algorithms to optimize the unconstrained layout. They first generated a random population of layouts from which to start. Each successive generation of layouts was the product of reproduction, crossover, and mutations from the prior generation. The authors report that this genetic algorithm was quite successful at locating optimal or near optimal layouts. The authors also found that the constrained layout aided users' ability to learn the interface since it was more familiar.

Other authors found similar results when comparing freely optimized keyboards to ones with familiarity constraints. For example, Bi et al. [2] performed a study with a Qwerty keyboard, a "Quasi-Qwerty" keyboard, and a freely optimized keyboard. The Quasi-Qwerty and freely optimized keyboards were designed to minimize the travel distance for a user's finger, thereby increasing the entry rate. However, the Quasi-Qwerty layout had the constraint that letters could not move more than one row and one column from their initial Qwerty position. The authors found that while the Quasi-Qwerty layout had an improved movement efficiency from the standard Qwerty layout, it was not as efficient as the freely optimized layout. However, during user trials, the authors found that users took the longest to locate the initial letter of a word on the freely optimized keyboard, followed by the Quasi-Qwerty layout, and finally the standard Qwerty layout. The authors concluded that the Quasi-Qwerty layout was effective at

obtaining an increased movement efficiency while not sacrificing too much time in the initial visual search.

Multi-Parameter Optimization

Instead of simply enforcing a concrete Qwerty restriction, Dunlop and Levine [4] chose to optimize their keyboard layout on three different parameters: finger travel distance, tap ambiguity, and familiarity. They minimized finger travel distance by calculating the product of distance between letter combinations with their frequency in the language corpus and summing these products for a given layout. While this metric is relevant to many optimization problems and is common in related work, it does not apply to an ambiguous keyboard that is not location-dependent. Dunlop and Levine also calculated a metric on tap ambiguity to reduce the number of commonly interchanged letters that were adjacent in the layout. They first created a table of these commonly interchanged letters, which they referred to as bad bigrams, or "badgrams". After counting the frequency of badgrams in same-length words in the text corpus, they converted each to a probability by dividing by the total number of badgram occurrences. The authors defined their tap clarity metric for a given keyboard layout by summing the badgram probabilities for each pair of adjacent letters and subtracting this sum from one. We can adapt this metric for ambiguous keyboards to reduce the number of badgrams that belong to the same grouping as opposed to adjacent keys. The final metric Dunlop and Levine used in their optimization was familiarity. They calculated the similarity of a given layout to the Qwerty layout by summing the squared distances between each key's position and its Qwerty position and then normalizing the results to the range between 0 and 1. This allowed for potentially high-scoring layouts that had most letters near their Qwerty positions with a few exceptions that would have been restricted by the Quasi-Qwerty constraints.

With these three metrics, Dunlop and Levine [4] used Pareto front optimization to find candidate layouts. They defined a small set of initial layouts that were then taken through 2000 iterations of changes. At each iteration, a certain number of keys were swapped and the metrics recalculated. The algorithm kept track of the set of "Pareto optimal" layouts, which were those that for all other layouts in the set, there was no layout that was better on all three metrics. After the final set, or Pareto front, was formed, the authors selected the keyboard nearest the 45° line (the line with all metrics equal), which they determined was the best overall layout given the metrics.

Qin et al. [8] also used a Pareto front to perform optimization, but they did so in two dimensions and with an ambiguous keyboard. They defined their clarity metric for a given word as the frequency of that word in the corpus divided by the total frequency of identical tapping sequences given an ambiguous layout. They then defined the clarity of a layout as the sum over all words of the product of word frequency and word clarity. The second metric used in this work

was a typing speed metric based on the relative location of frequent character combinations. As with the previous paper, this is not relevant to interfaces that remove location dependency. Instead of choosing the layout closest to the 45° line in the Pareto front as done by Dunlop and Levine [4], Qin et al. opted to select the layout that had the highest average of normalized metrics. Another difference between these works was that instead of optimizing on a third metric that indicated similarity to Qwerty, Qin et al. enforced a strict adherence to the Qwerty ordering of characters and split each row into three ambiguous groups. This created a total of nine groups, which is referred to as a T9 ambiguous keyboard.

III. Proposed Methodology

Given this prior research, we propose the following methodology for optimizing an ambiguous keyboard for eyes-free text entry.

Corpus Analysis

We will first perform analysis on the corpus of text written on mobile devices released by Vertanen and Kristensson [9]. We will combine the ideas of Lesher et al. [7] and Dunlop and Levine [4] to generate a table of badgrams given past context of what was typed. We will iterate through each phrase in the corpus one word at a time. Since we intend to perform word-level disambiguation as opposed to character-level as Lesher et al. did, we will use an *n*-gram language model to generate the most likely word given the previous *n* words. We will use software based on the VelociTap decoder proposed by Vertanen et al. [10] to perform these predictions. Since in practice the disambiguation algorithm will know the number of characters based on the number of keystrokes or input actions, we can restrict our search to that word length. For each word that is deemed more probable than the true word, we will track any badgrams (single letter differences from the true word) in a table and compute their probabilities as done by Dunlop and Levine [4]. As Lesher et al. did, we will compute the mutual confusability of two characters α and β as the sum of the frequencies of badgram $\alpha\beta$ and badgram $\beta\alpha$.

Performance Metrics

We will evaluate our keyboard designs primarily with the metric of tap clarity, which will be the best indicator of the performance of an experienced user entering text with an ambiguous and location-independent interface such as Tap123 [5] for a given layout. This is because higher values of tap clarity will result in fewer mistakes during disambiguation. While other studies have measured entry speed by computing the distance between frequently sequential characters, this metric is not relevant to a location-independent entry method since groups of characters are mapped to specific gestures as opposed to locations on the screen.

We propose two separate methods of calculating tap clarity. The first is a slight modification of Dunlop and Levine's definition [4] to suit an ambiguous keyboard. We will define this badgram clarity as:

$$Clarity_{badgram} = 1 - \sum_{\forall i,j \in \alpha} p_{ij} ,$$

where p_{ij} is the previously calculated badgram probability if *i* and *j* are letters in the same group, or 0 otherwise. This also conforms with Lesher et al.'s [7] definition of mutual confusability as the sum of the pairwise mutual confusabilities for each letter in a group. The main benefit of this method of calculation is that it is relatively computationally simple. It only takes a series of pairwise table lookups, bounded by the number of characters in the alphabet. However, it may oversimplify the calculation by only taking into account instances of single-letter differences between words. For example, Dunlop and Levine give "for/fir" as an example of easily confusable words with a Qwerty layout, creating an *IO* badgram. However, this would not take into account combinations such as "for/due" that had the same sequence of gestures in the layout defined in Tap123 [5].

The second method of calculating tap clarity is based on the method used by Qin et al. [8] for their ambiguous layout. We will modify their equation to use *n*-gram probabilities instead of frequencies for each word. When initially iterating through each word in the corpus, we will calculate and store the *n*-gram probability of each word of the same length as the true word. We can then iterate through these stored word/probability pairs for a given layout *L* and calculate the word clarity:

$$C(w) = P(w \mid Sequence) = \frac{P(w)}{\sum_{a \in A} P(a)} ,$$

where *w* is the true word, *A* is the set of words with the same gesture sequence as *w* given *L*, and *P(x)* denotes the *n*-gram probability of word *x*. Since the context and therefore the *n*-gram probabilities may change between occurrences of each true word in the corpus, we cannot simply multiply the clarity of a word by its frequency as Qin et al. did, but we must instead treat each occurrence as a separate *w* when we compute the sequence clarity of layout *L* as the simple sum of word clarities:

$$Clarity_{sequence}(L) = \frac{1}{M} \sum_{j=1}^{M} C(w_j) ,$$

where *M* is the number of words in the corpus, including repeated words. Multiplying the sum by the factor $\frac{1}{M}$ transforms the possible values of this metric to the range [0, 1], since word clarity is maximized at 1 and minimized at 0. While this sequence clarity metric is a better representation of potential conflicts than the badgram clarity, it is significantly more computationally expensive, since it requires an iteration through the entire corpus to evaluate every layout instead of the simple table lookups (the training set contains nearly 13 million

words). We will attempt to calculate the sequence clarity for each layout, but if it proves to be too expensive we will fall back to the badgram clarity.

Keyboard Design

Instead of optimizing the layouts on another metric that demonstrates their similarity to the Qwerty layout, we propose optimizing a layout with similar constraints to the Quasi-Qwerty [2] layout as well as a layout that is unconstrained. A few of the related works we discussed [2, 6] showed that users had a much more difficult time learning unconstrained layouts. However, we conjecture that a location-independent method may not have the same benefits from a similarity to Qwerty. The character groupings used in Tap123 [5] are shown in Figure 1a. Our Qwerty-constrained layout will allow a character to move one group in any direction from this baseline position. For example, the letter 'E' could move down to the group with 'A', or to the right to the group with 'Y', but not down *and* right to the group with 'H'. In addition to this six-grouping (T6) layout, we will optimize a T4 layout. While only having four groups of characters may make disambiguation more difficult, it would allow a user to enter text with a single hand, as opposed to Tap123's two-handed technique. A T4 approach would also enable us to remove location dependency completely, with the user simply tapping with between one and four fingers. We will also optimize both constrained and unconstrained T4 layouts, with the same constraint as the T6 layout and the Qwerty-based groupings shown in Figure 1b.

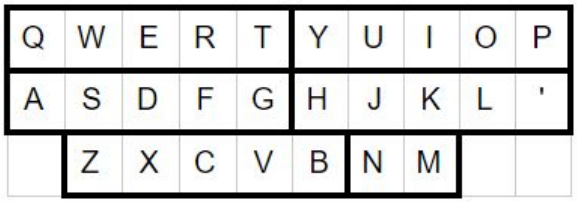

Figure 1a. Qwerty-based T6 groupings used in Tap123.

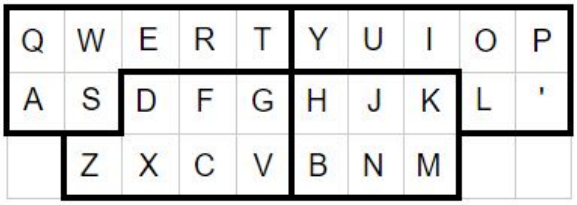

Figure 1b. Qwerty-based T4 groupings we plan to use in this study.

To actually optimize the keyboards, we will try two different approaches. The first approach will be based on the *n*-opt algorithm developed by Lesher et al. [7], but we will modify it to suit an ambiguous keyboard. In an ambiguous keyboard, all keys need not have the same number of characters. Therefore, when running an *n*-opt pass, we will still select every combination of *n* characters, but we will need to explore additional ways to shuffle them. For example, in the original paper if we selected α and β, the two options would be α in position one and β in position two, or the reverse. With an ambiguous layout, however, we also need to consider α and β both in position one, or both in position two. This will increase the computational complexity of a single comparison within a pass, especially for higher values of *n,* but a pass will require fewer comparisons overall since shuffling two characters on the same key will not result in any gains. We will use the same approach that Lesher et al. did in which we will

run the two-opt algorithm on a large number of initial layouts and then run a five-opt algorithm on the best result.

The second approach we will use to optimize our layouts will be the genetic algorithm (GA) approach used by Gong and Tarasewich [6]. While the authors mentioned top solutions in the population "mating" to produce the next generation as well as the possibility of mutations, they do not specify the details of their algorithm. In each generation of our genetic algorithm we will implement reproduction as follows. The highest scoring layout on the clarity metric will stay the same; we will refer to this layout as *L*. Each other layout will randomly choose some character α. It will then choose a character β that is in the same group as α in the layout *L*. It will swap it with another random character γ that is in the same group as α in this layout but not in the layout *L*. For example, let the layout in Figure 2a represent a hypothetical top layout *L*, and the layout in Figure 2b represent the layout we are modifying from this generation to the next. The algorithm could select 'D' as α, and character 'E' as β, since they are in the same group in *L* but different groups in the current layout. It could then select character 'F' as γ, since it is the in the same group as 'D' in the current layout but a different group in *L*. The algorithm would then swap 'E' with 'F' in the current layout to produce a member of the next generation, shown in Figure 2c. Once the mating process is complete, each layout in the new generation will have a small chance to mutate, swapping a random two characters. The actual value of this chance will be tuned using development data and observing how the clarity of the population progresses from one generation to the next. The clarity of each layout will be recalculated at the start of each generation to determine the new top-scoring layout *L*.

O	D	E	C	T	Y	U	I	Q	P
Z	S	W	J	G	H	F	K	L	'
	A	X	R	V	B	N	M		

Figure 2a. A hypothetical top-scoring layout *L* in a genetic algorithm generation.

Q	W	E	Y	T	R	U	K	O	P
A	S	D	F	B	H	J	I	L	'
	Z	X	N	V	G	C	M		

Figure 2b. A hypothetical "mating" layout in a genetic algorithm generation.

Q	W	**F**	Y	T	R	U	K	O	P
A	S	D	**E**	B	H	J	I	L	'
	Z	X	N	V	G	C	M		

Figure 2c. One potential result of mating the layout in Figure 2b with the one in Figure 2a. 'F' and 'E' have swapped positions.

Performing both the *n*-opt and genetic algorithms for constrained and unconstrained T4 and T6 layouts will result in a total of eight candidate layouts that will advance to the next stage of evaluation.

Keyboard testing

Once the optimization algorithms have chosen the eight candidate layouts they will each be evaluated using simulated user input. We will use the test portion of the data set released by Vertanen and Kristensson [9] and the modified decoder previously used to determine the *n*-gram probabilities to disambiguate text entered by a simulated user. Since the purpose of this optimization is to increase text entry rate and reduce error rate by reducing disambiguation conflicts and not to explore error correction techniques, the simulated user will not make errors. For each phrase in the test data set, we will pass the proper sequences of groups into the decoder one word at a time, allowing it to use both the sequences and context to disambiguate. We will evaluate the performance of each candidate layout on its average word error rate without correction.

Once the candidate layouts have been further narrowed down, we will conduct a between subjects longitudinal user study in which users enter text with one of the layouts. As in the study of Tap123 [5], the text will grow increasingly difficult as users learn the entry method. We will investigate any potential familiarity benefits in a location-independent entry method by comparing the best constrained layout to the best unconstrained layout. We will also compare users' ability to learn and use a T4 layout and a T6 layout. Instead of the bimanual entry performed with Tap123, users will enter text with a single hand to emulate the common occurrence of holding the device with the other hand.

IV. Conclusions

This paper explored multiple potential methods for optimizing a location-independent, eyes-free, ambiguous text entry method. We discussed using variations on the *n*-opt algorithm in addition to genetic algorithms to generate competitive T4 and T6 ambiguous layouts, both unconstrained and constrained to be similar to Qwerty-based layouts. We will evaluate these layouts first by metrics of *n*-gram tap clarity, then with simulated user data, and finally with a user trial. Through this optimization process we will explore the tradeoffs that exist between computation time and quality of the layouts, as well as between accuracy and ease-of-use in practice for the final layouts.

Bibliography

[1] Shiri Azenkot, Jacob O. Wobbrock, Sanjana Prasain, and Richard E. Ladner. 2012. Input Finger Detection for Nonvisual Touch Screen Text Entry in Perkinput. In *Proceedings of Graphics Interface 2012.* 121-129.

[2] Xiaojun Bi, Barton A. Smith, and Shumin Zhai. 2010. Quasi-qwerty Soft Keyboard Optimization. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 283-286.

[3] Matthew Bonner, Jeremy Brudvik, Gregory Abowd, and W. Keith Edwards. 2010. No-Look Notes: Accessible Eyes-Free Multi-Touch Text Entry. In *Proceedings of IEEE Pervasive Computing,* 409-426.

[4] Mark D. Dunlop and John Levine. 2012. Multidimensional Pareto Optimization of Touchscreen Keyboards for Speed, Familiarity and Improved Spell Checking. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 2669-2678.

[5] Dylan Gaines. 2018. Exploring an Ambiguous Technique for Eyes-Free Mobile Text Entry. In *ASSETS '18: Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. 471-473.

[6] Jun Gong and Peter Tarasewich. 2005. Alphabetically Constrained Keypad Designs for Text Entry on Mobile Devices. In *CHI '05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 211-220.

[7] Gregory W. Lesher, Bryan J. Moulton, and D. Jeffery Higginbotham. 1998. Optimal Character Arrangements for Ambiguous Keyboards. In *IEEE Transactions on Rehabilitation Engineering*. 415-423.

[8] Ryan Qin, Suwen Zhu, Yu-Hao Lin, Yu-Jung Ko, and Xiaojun Bi. 2018. Optimal-T9: An Optimized T9-like Keyboard for Small Touchscreen Devices. In *ISS '18: Proceedings of the 2018 ACM International Conference on Interactive Surfaces and Spaces*. 137-146.

[9] Keith Vertanen and Per Ola Kristensson. 2021. Mining, Analyzing, and Modeling Text Written on Mobile Devices. *Natural Language Engineering*. 1-33.

[10] Keith Vertanen, Haythem Memmi, Justin Emge, Shyam Reyal, and Per Ola Kristensson. 2015. VelociTap: Investigating Fast Mobile Text Entry using Sentence-based Decoding of

Touchscreen Keyboard Input. In *CHI '15: Proceedings of the ACM Conference on Human Factors in Computing Systems*. 659-668.