Both Complete and Correct?
Multi-Objective Optimization of Touchscreen Keyboard

Xiaojun Bi **Tom Ouyang** **Shumin Zhai**
Google Inc.
Mountain View, CA, USA
xjunbi@gmail.com ouyang@google.com zhai@acm.org

ABSTRACT

Correcting erroneous input (i.e., correction) and completing a word based on partial input (i.e., completion) are two important "smart" capabilities of a modern intelligent touchscreen keyboard. However little is known whether these two capabilities are conflicting or compatible with each other in the keyboard parameter tuning. Applying computational optimization methods, this work explores the optimality issues related to them. The work demonstrates that it is possible to simultaneously optimize a keyboard algorithm for both correction and completion. The keyboard simultaneously optimized for both introduces no compromise to correction and only a slight compromise to completion when compared to the keyboards exclusively optimized for one objective. Our research also demonstrates the effectiveness of the proposed optimization method in keyboard algorithm design, which is based on the Pareto multi-objective optimization and the Metropolis algorithm. For the development and test datasets used in our experiments, computational optimization improved the correction accuracy rate by 8.3% and completion power by 17.7%.

Author Keywords

Smart touch screen keyboard; mobile; intelligent user interfaces; keyboard algorithm; optimization; correction; completion; text input

ACM Classification Keywords

H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

Text entry is one of the most basic, common and important tasks on touchscreen devices (e.g., smartphones and tablets). A survey reported by time.com [22] shows that the top three most common activities on smartphones are texting, emailing, and chatting on social networks: all of them involve intensive text input. According a study from Nielsen [20], a teenager on average sends over 3,000 messages per month, or more than six texts per waking hour.

Despite its popularity and importance, touchscreen text entry is inherently error-prone and challenging, because of the imprecision of finger touch and the small key size. To improve the efficiency of text entry, modern "smart" keyboards are becoming intelligent: they are able to correct user's erroneous input (i.e., correction) and complete a word based on partial input (i.e., completion). For example, the Google keyboard on Android corrects "thaml" to "thank", and completes the word "computer" after a user types "comput" (Figure 1).

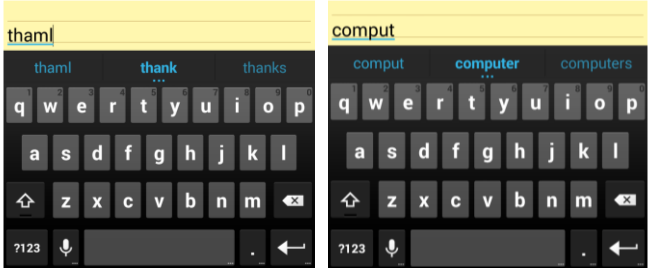


Figure 1. Examples of correction and completion on the Google keyboard. Left: "thaml" is corrected to "thank". Right: "comput" is completed to "computer". The corrected/completed words are displayed in the middle of the suggestion bar. A user enters the word by selecting it from the suggestion bar, or pressing the space bar.

Both correction and completion take advantage of the information regularities in a language, matching users' input signals against words from a lexicon. However, they cast the language regularities for different purposes. Correction uses the language regularities for correcting errors due to the imprecision of the finger touch or spelling errors such as inserting /omitting /substituting /transposing letters, while completion uses the language regularities for predicting unfinished letters based on partial (either correct or erroneous) input to offer keystroke savings.

Correction and completion are different, although related, aspects of modern text input systems. In some cases, these

CHI 2014, April 26–May 1, 2014, Toronto, Ontario, Canada.
ACM 978-1-4503-2473-1/14/04.
http://dx.doi.org/10.1145/2556288.2557414

two capabilities might conflict with each other. For example, given the input string “knowl”, a strong correction algorithm might treat it as a letter substitution error and suggest “known”, while a strong completion algorithm might be in favor of suggesting “knowledge” to save 4 more keystrokes. To effectively correct errors and also maximize potential keystroke savings, the keyboard algorithm needs to strike a balance between correction and completion.

A large amount of research [16, 8, 10] has been conducted to improve the qualities of touch screen keyboard algorithms. However, most of the previous research focused on correction. Completion has received little attention in the recent research literature. Furthermore, the relationship between these two capabilities remains unknown. We do not even know whether it is possible to design a keyboard algorithm performing well for both, given the potential conflict between them.

In this paper, we propose a technique based on the Metropolis optimization algorithm and Pareto multi-objective optimization to simultaneously improve the correction and completion capabilities of a keyboard algorithm, and investigated their relationship. We conceptualize the keyboard algorithm design as a multi-objective optimization problem, with the two optimization objectives 1) correction (measured in word score to be defined later) and 2) completion (measured in keystroke saving ratio).

Our investigation shows that correction and completion have only a minor conflict in the optimization space. The algorithm simultaneously optimized for both performs equally well with the algorithm exclusively optimized for correction in correcting input, and has only a minor loss in the completion capability (1.5% difference in keystroke saving ratio) compared with the keyboard exclusively optimized for completion.

Our research demonstrates that the proposed technique is effective in improving a keyboard’s correction and completion capabilities. One of the optimized keyboard algorithms (P_{W+S}) improves the performance by 8.3% in correction and 17.7% in completion over the keyboard before the optimization ($P_{Baseline}$).

RELATED WORK

Improving Correction

A sizeable amount of research has been conducted to improve the keyboard error correction ability. Goodman et al. [12] proposed combing a language model with a key press model to select the most probable key sequence other than the key sequence dictated by strict key boundaries. Kristensson and Zhai [16] invented an “elastic” keyboard, which viewed the hit points as a geometric pattern. The pattern was matched against patterns formed by the letter key center positions of legitimate words in a lexicon. The evaluation showed that the keyboard was capable of correcting words even if the user missed all the intended keys, as long as the user’s tapping pattern is close enough to the intended word. Findlater and Wobbrock [8] proposed improving ten-finger touchscreen typing through adaptation. By adapting the underlying key-press classification model, the keyboard improved the typing speed compared to a control condition with no personalization. Kane, Wobbrock, and Harniss et al. [14] developed *TrueKeys*, combining models of word frequency, keyboard layout and tying error patterns to correct typing mistakes. Goel, Findlater and Wobbrock [10] developed the WalkType keyboard which leveraged a mobile device’s built-in accelerometer to compensate for extraneous movement while walking. Goel, Jansen, and Mandel et al. [11] proposed reducing the text entry error rates by leveraging information about a user’s hand posture. Yin, Ouyang, Partridge, and Zhai [24] further proposed adapting the keyboard underlying spatial model to factors including input hand postures, individuals, and target key positions. With a hierarchical spatial backoff model, the adaptation was capable of reducing the language-model-independent error rate by 13.2%.

Improving Completion

In contrast to correction, improving the completion capability on a touchscreen keyboard has received little attention in the recent HCI research literature. Instead, some related work can be found in earlier research focused on users with motor impairments. Word prediction is a major component in computer-based augmentative and alternative communication (AAC) systems to help people with disabilities communicate in the areas of speech, writing, and computer applications. Keystroke savings provided by several commercial word prediction systems [13] have been measured in the range of 35- 47%. Koester and Levin [15] further developed a model to simulate users’ word entry time during the word prediction use.

Although word completion has received limited attention from academic research, it has been playing a major role in commercial products. The majority of present touchscreen keyboards such as Android, iPhone, Windows phone keyboards provide completion to various degrees.

Keyboard Optimization

Optimization has been widely adopted as an approach to improve the keyboard performance, though the previous work mostly focuses on optimizing the keyboard layout.

As early as 1986, Getstow et al [9] used a simple greedy algorithm to optimize a keyboard for motor-impaired users. MacKenzie and Zhang [19] manually optimized their “OPTI” layout. Zhai, Hunter and Smith [25, 26] introduced the Metropolis keyboard and later ATOMIK using the Metropolis randomized optimization technique. All these were layouts optimized to improve the speed of stylus/one finger text entry for a single language. Bi, Smith and Zhai [5] optimized the touch screen keyboard layout for five languages: English, Spanish, French, German, and Chinese at the same time. The optimized layout has minor performance drop compared to the layout exclusively optimized for one language. Bi, Smith and Zhai [4] also explored optimizing the layout while maintaining a certain level of familiarity with Qwerty layout. Their work led to the Quasi-Qwerty layout, which minimized the finger/stylus travel distance and each key was at most one-key away from its original position on the Qwerty layout. Dunlop and Levine [6] later introduced the tap interpretation clarity as the third measure, conducting multidimensional Pareto optimization for speed, familiarity and improved spell checking. Their research was the first in the keyboard literature to apply Pareto optimization and led to a triple optimized keyboard layout. Oulasvirta, Reichel and Li et al. [21] extended the layout optimization to a split keyboard for two-thumb typing. The optimized KALQ layout minimizes the thumb travel distance and maximizes the alternation between thumbs.

THE KEYBOARD ALGORITHM

Text entry on touch keyboards can be viewed as an encoding and decoding processes: the user encodes the intend word w into a series of touch points $(s_{1,}s_2, s_3 \dots s_n)$, and the keyboard algorithm acts as the decoder, retrieving w from the spatial signals.

At an abstract level, the decoder selects a word w_i from the keyboard's lexicon and calculates its probability of being the intended word based on the information from the following three sources:

1) The proximities of letters in w_i to the touch points $(s_{1,}s_2, s_3 \dots s_n)$

2) The prior probability of w_i from a language model.

3) The possibilities of spelling errors (e.g., inserting/omitting/transposing/substituting letters).

Pruning out the improbable ones, a list of the most probable words is then ranked and suggested to the user.

While many modern keyboard algorithms may work similarly at this abstract level, the actual implementations of industrial modern keyboards may be rather complex and vary across products. Since most of the commercial keyboard algorithms are not published, it is impossible to develop a general model representing all of them. Fortunately, we could conduct the present research on open source keyboards.

The Baseline Keyboard

We developed a keyboard, referred as $P_{Baseline}$ hereafter, based on the latest version (Android 4.3_r3) of the Android Open Source Project (AOSP) Keyboard [1] which is open-sourced and broadly deployed in Android mobile devices.

$P_{Baseline}$ shared the similar algorithm as the AOSP keyboard which can be read and analyzed by any researcher or developer. In brief, it worked as follows. The lexicon composed of around 170,000 words was stored in a trie data structure. As the algorithm received spatial signals $(s_{1,}s_2, s_3 \dots s_n)$, it traversed the trie and calculated the probabilities for nodes storing words, based on the aforementioned three sources.

A critical part of the algorithm is to calculate the probabilities of word candidates by weighting information from three sources. Same as the AOSP keyboard, the weights of different sources are controlled by 21 parameters illustrated and explained in Table 1. These parameters are pertinent to the performance of $P_{Baseline}$. Altering them adjust the relative weights of information from different sources, leading to different correction and completion capabilities. For example, reducing the cost of un-typed letters (i.e., lowering COST_LOOKAHEAD) would make the keyboard in favor of suggesting long words, but it might be detrimental to the correction ability. $P_{Baseline}$ is implemented based on the AOSP keyboard, and the parameters in $P_{Baseline}$ shared the same values as those in the AOSP keyboard, which serves as the baseline condition in the current research.

MEASURING CORRECTION AND COMPLETION

Typical laboratory-based studies demand intensive labor and time to evaluate keyboards and may still lack the necessary sensitivity to reliably detect important differences amid large variations in text, tasks, habits, and other individual differences. Bi, Azenkot, Partridge and Zhai [3] proposed using *remulation* – replaying previously recorded data from the same group from the same experiment in real time simulation – as an automatic approach to evaluate keyboard algorithms. We adopt a similar approach in the present research: we evaluated a keyboard algorithm against previously recorded user data to measure its performance. Unlike the *Octopus* tool [3] which evaluated keyboards on devices, we simulated the keyboard algorithm and ran tests on a workstation. Octopus evaluates keyboards in a "black box" fashion without access to their algorithms and source code. The off-device approach in this work is limited to the keyboards whose algorithm and source code are accessible, but is much faster than Octopus.

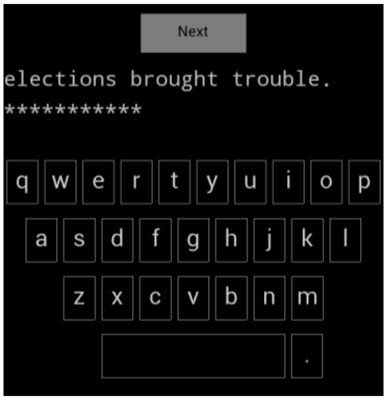
Datasets

We first ran studies to collect data for optimization and testing. The data collection studies were similar to those in Azenkot and Zhai [2], and Bi et al. [3]. To avoid data collection's dependencies and limitations on keyboard algorithms, a Wizard of Oz keyboard was used in the study (Figure 2), which provided users with only asterisks as feedback when they were entering text.

Each participant entered the same set of 50 phrases randomly chosen from the MacKenzie and Soukoreff's phrase set [10]. All touch events were logged. Participants were asked to enter text "as naturally and as fast as possible." The first 10 phrases for each user were considered a warm-up and excluded in the dataset. A Galaxy Nexus phone was used throughout the study, and

Parameter	Current Value	Description
DISTANCE_WEIGHT_LENGTH	0.132	Weight for the length of a word
PROXIMITY_COST	0.086	Weights for the distances between touch points and the corresponding letters in a word
FIRST_PROXIMITY_COST	0.104	
OMISSION_COST	0.458	Weights for omission errors
OMISSION_COST_SAME_CHAR	0.491	
OMISSION_COST_FIRST_CHAR	0.582	
INSERTION_COST	0.730	Weights for insertion errors
INSERTION_COST_SAME_CHAR	0.586	
INSERTION_COST_FIRST_CHAR	0.623	
TRANSPOSITION_COST	0.516	Weights for transposition errors
SPACE_SUBSTITUTION_COST	0.319	Weights for space substitution errors
ADDITIONAL_PROXIMITY_COST	0.380	Weights for the distances between touch points and the corresponding letters in a word
SUBSTITUTION_COST	0.403	Weights for substitution errors
COST_NEW_WORD	0.042	Weights for starting a new word
COST_SECOND_OR_LATER_WORD_FIRST_CHAR_UPPERCASE	0.25	Weights for capitalized words
DISTANCE_WEIGHT_LANGUAGE	1.123	Weight for language model
COST_FIRST_LOOKAHEAD	0.545	Weight for un-typed letters in a word
COST_LOOKAHEAD	0.073	
HAS_PROXIMITY_TERMINAL_COST	0.105	Weights for terminal nodes
HAS_EDIT_CORRECTION_TERMINAL_COST	0.038	
HAS_MULTI_WORD_TERMINAL_COST	0.444	

Table 1. Parameters for calculating the probability of a word candidate in the P_{base} and AOSP keyboard algorithms. They control the weight of information from different sources. These values were from "scoring_params.cpp" in the AOSP source tree [1] (Android 4.3 on Sep. 16, 2013).


elections brought trouble.

****

Figure 2. The WOZ keyboard for data collection, which was designed to capture fundamental text entry behaviors.

the keyboard was equal to the stock Android keyboard in dimensions.

We conducted two studies. The first one was to collect data for the algorithm parameter optimization (referred as the development dataset), and the second one was for testing (referred as the test dataset). In the first study, we recruited 40 participants. The average age was 32 (the youngest was 18 and the oldest was 59). Five were left-handed. They were allowed to freely choose text entry hand postures. Twenty-four users entered text with two thumbs, 4 with one thumb, and 12 with the index finger. It included 1,597 phrases in total.

In the second study, we recruited 24 participants, with ages between 20 and 30. All were right handed. They all entered text with index fingers. This data set included 960 phrases in total.

The keyboard algorithm's correction and completion capabilities were evaluated using the collected data. Current touchscreen keyboards perform word-level correction: they correct or complete a word after a word terminator such as a space or punctuation is entered. Therefore, the collected data was segmented into words in evaluation. After the segmentation, the development dataset included 7,106 words, and the test dataset had 6,323 words.

Correction

Correction is measured in *word score*, which reflects the percentage of correctly recognized words out of the total test words. The word score (W) is defined as:

$$W = \frac{\text{the number of correctly recognized words}}{\text{the number of words in test}} \times 100\% \qquad \text{Eq. 1}$$

Completion

Completion is measured in keystroke saving ratio (S). Given a data set with n test words, S is defined as:

$$S = \left(1 - \frac{\sum_{i=0}^{n} S_{min}(w_i)}{\sum_{i=0}^{n} S_{max}(w_i)}\right) \times 100\% \qquad \text{Eq.2}$$

$S_{max}(w_i)$ is the maximum number of keystrokes for entering a word w_i. In the worst scenario where the keyboard fails to complete w_i, the user needs to enter all the letters of w_i and presses the space bar to enter it. Therefore,

$$S_{max}(w_i) = \text{length of } w_i + 1 \qquad \text{Eq. 3}$$

$S_{min}(w_i)$ is the minimum number of keystrokes needed to enter w_i. Assuming that users fully take advantage of the completion capability, w_i will be picked as soon as it is suggested on the suggestion bar. Therefore, $S_{min}(w_i)$ is the least number of keystrokes needed to bring w_i on the suggestion bar plus one more keystroke for selection. The number of the slots on the suggestion bar may vary across keyboards. The keyboard used in this work provides three suggestion slots, the same as the AOSP keyboard.

Note that the measure defined above is the maximum savings offered to a user by the keyboard algorithm. If and how often the user takes them depends on the UI design, the user's preference and bias in motor, visual and cognitive effort trade-offs which are separate research topics.

OPTIMIZING KEYBOARD ALGORITHM FOR CORRECTION AND COMPLETION

In this section, we introduce a method for optimizing a keyboard algorithm for correction and completion, and apply it to $P_{baseline}$. In brief, we conceptualize the keyboard algorithm design as a multi-objective optimization problem with two objectives 1) correction and 2) completion. For $P_{baseline}$ particularly, it is equivalent to optimizing the 21 parameters in Table 1 for these two objectives.

To solve a multi-objective optimization problem, a simple approach is to convert the multiple objectives into one objective function, where each objective is a component in a weighted sum. However, the challenge of this approach is choosing an appropriate weight for each objective. Also, this method returns only one solution, which might not be the most desirable solution.

Instead of returning a single solution, we performed a Pareto optimization [7], which returns a set of Pareto optimal solutions. A solution is called Pareto optimal or non-dominated, if neither of the objective functions can be improved in value without degrading the other. A solution that is not Pareto optimal is dominated: there is a Pareto optimal solution which is better than it in at least one of the criteria and no worse in the other. Analyzing the Pareto optimal solutions reveals the trade-off between multiple objectives. Additionally, the Pareto set provided a range of optimal solutions, allowing keyboard designers and developers to pick the desirable one according to their preferences.

We adopted the Metropolis optimization algorithm [25] to search for the Pareto set. The process was composed of multiple sub processes. Each sub process started from a random set of the 21 parameters in the range [0, 5], the estimated range of the parameter values based on the analysis of the algorithm. The sub process then optimized the 21 parameters for the objective function (Eq. 4) based on the collected development dataset:

$$Z = \alpha W + (1 - \alpha) S \qquad \text{Eq. 4}$$

which is a sum of word score (W) and keystroke saving ratio (S) with a weight α $(0 \leq \alpha \leq 1)$. α was randomized at the beginning of each sub process and remained unchanged during the sub process. α changed across sub processes. Our purpose was to ensure that the Pareto set covered a broad range of Pareto optimal solutions.

In each iteration of a sub process, the 21 parameters moved in a random direction with a fixed step length (0.01). We then evaluated the keyboard algorithm with the new parameters against the development dataset according to the objective function (Eq. 4). Whether the new set of parameters and search direction were kept was determined by the Metropolis function:

$$W(O \rightarrow N) = \begin{cases} e^{\frac{\Delta Z}{kT}} \; if \; \Delta Z < 0 \\ \\ 1 \quad if \Delta Z \geq 0 \end{cases} \qquad \text{Eq. 5}$$

$W(O \rightarrow N)$ was the probability of changing from the parameter set O (old) to the parameter set N (new); $\Delta Z = Z_{new} - Z_{old}$, where Z_{new}, and Z_{old} were values of objective functions (Eq. 4) for the new and old set of parameters respectively; k was a coefficient; T was "temperature", which can be interactively adjusted.

This optimization algorithm used a simulated annealing method. The search did not always move toward a higher value of objective function. It occasionally allowed moves with negative objective function value changes to be able to climb out of a local minimum.

After each iteration, the new solution was compared with the solutions in the Pareto set. If the new solution dominated at least one solution in the Pareto set, it would be added to the set and the solutions dominated by the new solution would be discarded.

After 1000 iterations, the sub process restarted with another random set of parameters and a random α for the objective function Eq. 4. We ensured that there was at least one sub process for each of the following three weights: $\alpha = 0$, 0.5, and 1. The optimization led to a Pareto set of 101 Pareto optimal solutions after 100 sub processes with 100,000 iterations in total, which constituted a Pareto frontier illustrated in Figure 3.

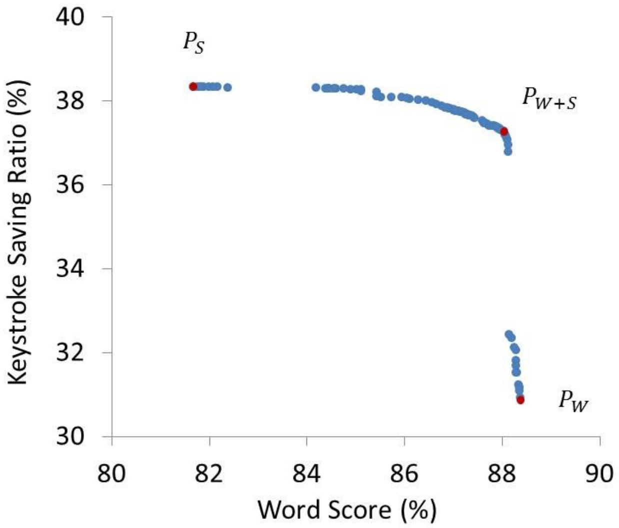


Figure 3. The Pareto frontier of the multi-objective optimization. The three red dots were solutions with the most word score (P_W), the most keystroke savings (P_S), and the highest $\mathbf{0.5}W + \mathbf{0.5}S$ (P_{W+S}) respectively.

The Pareto frontier shows that the Pareto optimal solutions distribute in a small region, with keystroke saving ratio ranging from 30% to 38% and word score ranging from 81% to 88%. As shown in Figure 3, the Pareto frontier forms a short, convex, L-shaped curve, indicating that correction and completion have little conflict with each other and the algorithm can be simultaneously optimized for both with minor loss to each. Among the 101 Pareto optimal solutions, we are particularly interested in three solutions, illustrated in red dots in Figure 3:

1) The solution with the highest word score (W), denoted by P_w. It is the solution exclusively optimized for correction.
2) The solution with the highest keystroke saving ratio (S), denoted by P_S. It is the solution exclusively optimized for completion.
3) The solution with highest $0.5W + 0.5S$, denoted by P_{W+S}. It is the solution optimized for both correction and completion, with 50% weight for each objective.

P_w and P_S reveal the highest correction and completion capabilities a keyboard can reach, while P_{W+S} is the most balanced solution with equal weights for correction and completion. The parameter sets, correction and completion capabilities for P_w, P_S and P_{W+S} are reported in Table 2.

The optimization results showed the default parameter set taken from the AOSP keyboard in Android 4.3 (Sep. 16, 2013) was sub-optimal in both correction and completion according to the criteria and the development dataset used in this study P_w, P_S, and P_{W+S} all improve the word score and keystroke saving ratio by at least 10% over $P_{Baseline}$.

It is more illuminating to compare the three Pareto optimal solutions P_w, P_S, and P_{W+S}, since they are all optimized under identical conditions with the same dataset. Table 2 shows that P_{W+S} is close to P_S in keystroke saving ratio, and close to P_w in word score. It indicates simultaneously optimizing for both objectives causes only minor performance degradation for each objective. These results were later verified with the separate test dataset.

Parameters moved in various directions after the optimization. For such a complex optimization problem with 21 free parameters, it was difficult to precisely explain why each parameter moved in such a direction after the optimization. However, we observed some distinct patterns of parameter value changes, which partially explain the performance differences across keyboards.

For examples, the parameters pertinent to the cost of proximity of touch points to letters (i.e., PROXIMITY_COST, FIRST_PROXIMITY_COST) decreased substantially from $P_{Baseline}$to P_w, P_S, and P_{W+S}, indicating that the optimized algorithms tend to be more tolerant to errors in spatial signals. The cost of untyped letters in a word (i.e., COST_FIRST_LOOKAHEAD) also decreased, indicating that the optimized algorithms were more likely to predict untyped letters as the user's intention, especially after typing the first letter, to save keystrokes.

EVALUTING KEYBOARD ALGORITHMS

To test the external validity of the optimization results, we evaluate the keyboard algorithms before and after the optimization using the test dataset, which was collected from 24 users not participating in the development dataset collection.

Correction

Word Score. We first analyzed the word scores for different keyboards. An ANOVA showed that the keyboard algorithm had a significant main effect on word score ($F_{3, 69} = 58.379$, $p < 0.001$). The mean (SD) scores were 82.7% (9.1%) for $P_{Baseline}$, 91.0% (4.9%) for P_{W+S}, and 86.9% (7.9%) for P_S, and 91.2%(4.6%) for P_w. Pairwise mean comparison showed the differences were significant for each pair ($p < 0.001$) except for P_w vs. P_{W+S}. ($p = 0.509$). Figure 4 also shows the mean (SD) word score per word length. As shown, P_w, and P_{W+S} performed almost equally well across all word lengths. The biggest gap between these

	$P_{Baseline}$	P_w	P_S	P_{W+S}
Word Score	**77.5%**	**88.4%**	**81.7%**	**88.0%**
Keystroke Saving Ratio	**20%**	**30.9%**	**38.3%**	**37.3%**
Parameters				
DISTANCE_WEIGHT_LENGTH	0.132	0.052	0.412	0.172
PROXIMITY_COST	0.086	0	0	0
FIRST_PROXIMITY_COST	0.104	0.01	0.08	0.05
OMISSION_COST	0.458	0.048	0.458	0.108
OMISSION_COST_SAME_CHAR	0.491	1.371	0.991	0.951
OMISSION_COST_FIRST_CHAR	0.582	0.092	0.582	0.382
INSERTION_COST	0.730	1.17	1.13	0.740
INSERTION_COST_SAME_CHAR	0.586	0.01	0.08	0
INSERTION_COST_FIRST_CHAR	0.623	0.753	0.613	0.583
TRANSPOSITION_COST	0.516	0.576	0.436	0.436
SPACE_SUBSTITUTION_COST	0.319	0.22	0.210	0.06
ADDITIONAL_PROXIMITY_COST	0.380	1.20	1.08	1.06
SUBSTITUTION_COST	0.403	1.603	1.143	1.143
COST_NEW_WORD	0.042	0.12	0.02	0.332
COST_SECOND_OR_LATER_WORD_FIRST_CHAR_UPPERCASE	0.25	0.69	0.370	0.09
DISTANCE_WEIGHT_LANGUAGE	1.123	1.783	1.463	1.573
COST_FIRST_LOOKAHEAD	0.545	0.053	0	0.095
COST_LOOKAHEAD	0.073	0.08	0	0
HAS_PROXIMITY_TERMINAL_COST	0.105	0.20	0.11	0.04
HAS_EDIT_CORRECTION_TERMINAL_COST	0.038	1.098	1.418	0.998
HAS_MULTI_WORD_TERMINAL_COST	0.444	1.434	0.994	1.224

Table 2. Word Scores, Keystroke Saving Ratios and Parameters for the algorithm before ($P_{Baseline}$), and after (P_{W+S}, P_w, P_S) the optimization. The blue indicates the parameters decreased and red means the values increased after the optimization.

two keyboards was for words with 7+ characters: the word score of P_w was 1.5% higher. Both P_w, and P_{W+S} performed better than $P_{Baseline}$ and P_S across all word lengths, especially for words with 5 or more letters. The word scores for P_w, and P_{W+S} were at least 14% (absolute) higher than $P_{Baseline}$ and 6% higher than P_S.

Ratio of Error Reduction. In addition to word score, we also examined the keyboard's capability of correcting users' erroneous input strings, which is measured in *Ratio of Error Reduction (RER)* [3]. RER is defined as:

$$RER = \frac{E_{literal} - E_{transcribed}}{E_{literal}} \times 100\% \qquad \text{Eq. 6}$$

$E_{literal}$ is the error rate of the literal text, which reflects a user's uncorrected keyboard output. The literal text is generated from the closest key labels on the keyboard to users' actual touch points. This is a naïve key-detection algorithm that offers no error correction, and literally transcribes the user's touch points. Since the test data set was segmented into words, $E_{literal}$ is measured as:

$$E_{literal} = \frac{\text{number of incorrect words in the literal text}}{\text{number of tatal words}} \qquad \text{Eq. 7}$$

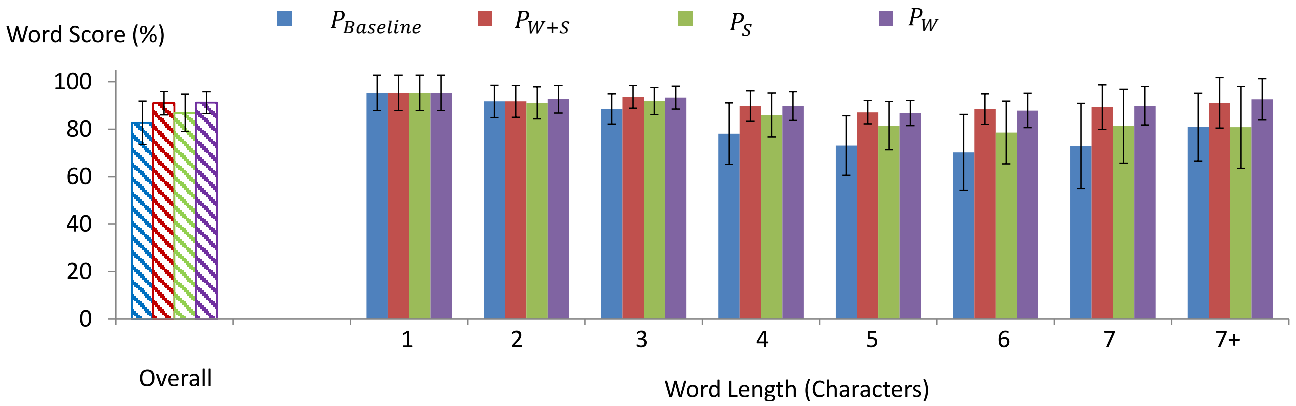


Figure 4. Mean (SD) word scores across 24 participants for $P_{Baseline}$, P_{w+S}, P_S, and P_w. The striped bars show overall results and solid bars show results per word length.

$E_{transcribed}$ is the error rate of the transcribed text, the output of a keyboard. It is defined as:

$$E_{transcribed} = 1 - \text{word score} \qquad \text{Eq. 8}$$

Note that *RER* is applicable only when $E_{literal} > 0$.

The analysis showed that $E_{literal}$ is 28.8%. The RERs were 40.9% for $P_{Baseline}$, 70.0% for P_{W+S}, 54.5% for P_S and 70.4% for P_w. P_{W+S} was close to P_w in correcting users' input errors. Both of them corrected 30% more errors than $P_{Baseline}$, and 15% more than P_S.

Table 3 shows examples of correction for different keyboards.

Literal	Intended	$P_{Baseline}$	P_S	P_w and P_{W+S}
agsim	again	agsim	agsim	again
popilariry	popularity	popular	popularity	popularity
quivj	quick	quivj	quivj	quick

Table 3. Correction examples for keyboards. Words in red are erroneous literal text or incorrect corrections.

Completion

We examined the keystroke saving ratio for different keyboards. An ANOVA showed a significant main effect of keyboard on keystroke saving ratio ($F_{3,\,69} = 7531$, $p < 0.001$). The mean (SD) keystroke saving ratios were 18.5% (1.0%) for $P_{Baseline}$, 36.2% (9.8%) for P_{W+S}, and 37.7% (1.2%) for P_S, and 30.3%(1.2%) for P_W. Pairwise mean comparison showed that the differences were significant between every two conditions ($p < 0.005$).

Figure 5 also shows the mean(SD) keystroke saving ratio per word length. The keystroke saving ratio for P_S was slightly higher than P_{W+S}, but the difference between them was small. The biggest difference was for words with 7 characters, where the keystroke saving ratio of P_S was 3% higher than that of P_{W+S}. Both P_{W+S} and P_S saved more keystrokes than $P_{Baseline}$ and P_W, especially for words with 5 or more characters: P_{W+S} and P_S saved more than 20% keystrokes than $P_{Baseline}$, and 7% more than P_W for these words. Table 4 showed examples of keystroke savings for these keyboards.

$P_{Baseline}$	P_{W+S}	P_S	P_W
provide	**prov**ide	**prov**ide	**prov**ide
favorite	**fav**orite	**fav**orite	**favo**rite
watch	**watc**h	**wat**ch	**watc**h

Table 4. Examples of completion for different keyboards Letters predicted by a keyboard are illustrated in gray and typed letters are in bold black.

Dataset-Independent Keystroke Saving Ratio. The keystroke saving ratios illustrated in Figure 5 were calculated based on the test dataset collected from a prior user study. In addition, we estimated keystroke saving ratios based on an assumption of perfect input: each touch point was placed on the center of the intended key. This method separated the effect of a particular data set, and also simulated the behaviors of one type of users who type extremely carefully and accurately. Such results were complementary to the results in Figure 5, which simulated the behavior of experts who usually typed ahead and always trusted keyboards' correction and completion capabilities.

Assuming perfect input, the keystroke saving ratios were 20.9% for $P_{Baseline}$, 37.9% for P_{W+S}, 39.9% for P_S, and 32.5% for P_W, based on the same set of words in the test data set. They were approximately 2% higher than the corresponding values estimated from the test data set. The results also echoed the finding shown in Figure 5: P_{W+S} and P_S were close in keystroke saving, and both of them were considerably better than $P_{Baseline}$ and P_W.

Discussion

Correction and completion have only a minor conflict. The algorithm simultaneously optimized for two objectives (i.e., P_{W+S}) performs almost equally well with the algorithms optimized for one objective only (i.e., P_w and P_S), in the corresponding measure. The word score of P_{W+S} has no significant difference from that of P_w: both of them are able

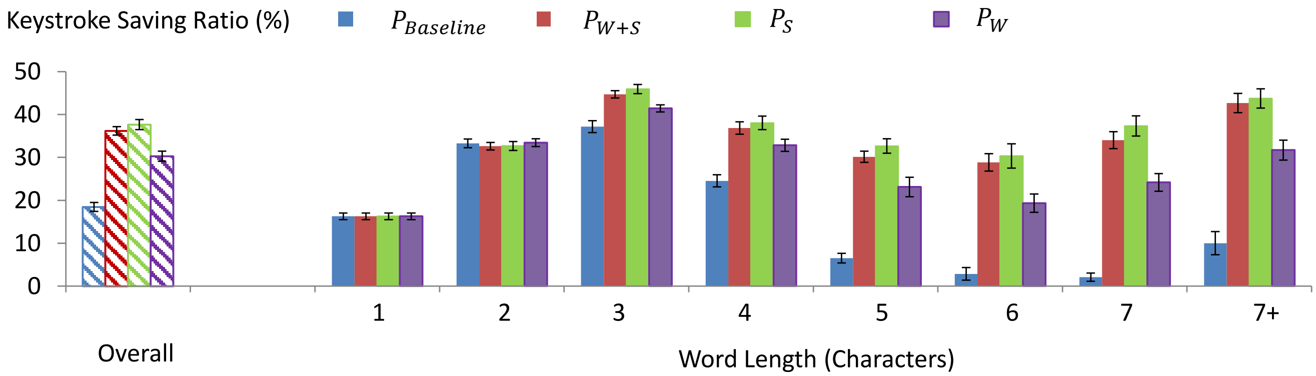


Figure 5. Mean (SD) keystroke saving ratios across 24participnts for $P_{Baseline}$, P_{w+S}, P_S and P_W. The striped bars show overall results and solid bars show results per word length.

to correctly recognize around 91% of the words. P_S is slightly better than P_{W+S} in completion (1.5% higher), but at the cost of having a significantly lower word score (4.1% lower).

This finding suggests that it is possible to simultaneously optimize a keyboard algorithm for both correction and completion at the same time, with little compromise compared to the specialized models that are optimized for only one objective.

Pareto optimization improves a keyboard's correction and completion capabilities. The results (Figure 4) shows that P_{W+S} improves the word score by 8.3%, and keystroke saving ratio by 17.7% over $P_{Baseline}$ based on the test dataset, echoing the findings from the development dataset (Figure 5). These results indicate that the proposed method based on the Pareto optimization and the Metropolis algorithm is an effective approach of improving keyboard algorithm quality.

The improvements are more pronounced for long words. P_{W+S} improves the word score by 14% and the keystroke saving by 20% for words with 5 or more letters, over $P_{Baseline}$. P_{W+S} is also more effective than $P_{Baseline}$ in correcting users' input errors: it reduces users' input error by 70% while $P_{Baseline}$ only reduces input error by 40.9%.

LIMITATIONS AND FUTURE WORK

The datasets used this paper were collected in a Wizard of Oz setting, and covered a vocabulary of common English words. As a result, the optimal values for the parameters may change as the usage environment and/or vocabulary become more diverse We plan to further expand this work to larger and more diverse data sets, both from lab studies and long-term longitudinal studies.

We employed a remulation-based approach [3] to evaluate keyboard algorithms, which measured the correction and completion power a keyboard algorithm can offer, but did not evaluate the entire user experience of a touchscreen keyboard. How and to what degree a user can take advantage of a keyboard's correction and completion power depends on the UI design, users' preferences and bias in motor, visual and cognitive effort trade-offs, However, based on the success of popular mobile and for example CJK (Chinese, Japanese, Korean) text input methods, it is safe to say both correction and completion are desirable features. One question was whether completion be done without significant loss of correction power for the user, which this work is focused on.

The same proposed Pareto optimization process may be applicable for various keyboard algorithms. Due to the limited scope one paper can cover and the accessibility of the keyboard algorithms, we only applied it to a variant of the AOSP keyboard. It would be interesting to apply it to other keyboards, to examine the shape of the Pareto frontiers, and to investigate the improvements it can bring.

CONCLUSIONS

Although correction and completion are featured in almost all modern touchscreen keyboards, their relationship has not been studied in the research literature. The present work explores the optimality issues of them, resulting in the following contributions.

First, the work suggests that correction and completion have only a minor conflict with each other. It is possible to optimize a keyboard algorithm for both at the same time, with no compromise to correction, and only a minor compromise to completion. Before this investigation, it was unknown whether this would be possible because one could imagine that optimizing for one objective would come at a large cost for the other.

Second, we demonstrated the effectiveness of applying a Pareto optimization method based on the Metropolis algorithm to improve the correction and completion of a keyboard algorithm. Our investigation based on an variant of AOSP keyboard showed that a keyboard after the optimization (i.e., P_{W+S}) improved the word score by 8.3% and keystroke saving ratio by 17.7% over the keyboard before the optimization (i.e., $P_{Baseline}$) . The methods developed in this work can be applied to other keyboards with minor modifications.

REFERENCES

1. Android Open Source Project http://source.android.com/

2. Azenkot, S. and Zhai, S. (2012). Touch Behavior with Different Postures on Soft Smart Phone Keyboards. *Proc. of MobileHCI'12*. 251-260.

3. Bi, X., Azenkot, S., Partridge, K., Zhai, S. (2013) Octopus: Evaluating Touchscreen Keyboard Correction and Recognition Algorithms via *Remulation*, *Proc. of CHI'13,* 543 – 552.

4. Bi, X., Smith, B., Zhai. S. (2010) Quasi-Qwerty Soft Keyboard Optimization , *Proc. of CHI'10,* 283~286.

5. Bi, X., Smith, B., Zhai. S. (2012) Multilingual Touchscreen Keyboard Design and Optimization. *Human-Computer Interaction* , Volume 27, Issue 4, 352-382.

6. Dunlop, M. and Levine, J. (2012) Multidimensional pareto optimization of touchscreen keyboards for speed, familiarity and improved spell checking. *Proc. of CHI'12*, 2669-2678.

7. Matthias Ehrgott (2005). *Multicriteria Optimization. Birkhäuser*. ISBN 978-3-540-21398-7.

8. Findlater, L., and Wobbrock, J.O. (2012). Personalized input: improving ten-finger touchscreen typing through automatic adaptation. *Proc. of CHI '12*. 815-824.

9. Getschow, C. O., Rosen, M. J., and Goodenough-Trepagnier, C. (1986). A systematic approach to design a minimum distance alphabetical keyboard. *Proc. of RESNA (Rehabilitation Engineering Society of North America) 9th Annual Conference*, 396-398.

10. Goel, M., Findlater, L. and Wobbrock, J. (2012) WalkType: using accelerometer data to accomodate situational impairments in mobile touch screen text entry. In *Proc. of CHI '12*. 2687-2696.

11. Goel, M., Jansen, A., Mandel, T., Patel, S., and Wobbrock. J. (2013) ContextType: using hand posture information to improve mobile touch screen text entry. *Proc. of CHI '13,* 2795-2798.

12. Goodman, J., Venolia, G., Steury, K., and Parker, C. (2002). Language modeling for soft keyboards. *Proc. of AAAI '02*, 419-424.

13. Higginbotham, J. (1992) Evaluation of keystroke savings across five assistive communication technologies *Augmentative and Alternative Communication* Vol. 8, No. 4 , 258-272.

14. Kane, K. S, Wobbrock, J., Harniss, M., and Johnson, K. (2008) TrueKeys: identifying and correcting typing errors for people with motor impairments. *Proc. of IUI'08*, 349-352.

15. Koester, H. H., Levine, S. P. (1994) Modeling the speed of text entry with a word prediction interface, *IEEE Transactions on Rehabilitation Engineering,* Volume: 2 Issue: 3 177 – 187.

16. Kristensson, P-O., Zhai, S. (2005) Relaxing Stylus Typing Precision by Geometric Pattern Matching. *Proc. IUI'05,* 151–158,

17. MacKenzie, I. S., and Soukoreff, R. W. (2003). Phrase sets for evaluating text entry techniques. *Proc. of CHI EA '03*, 754-755.

18. MacKenzie, I. S., and Tanaka-Ishii, K. (Eds.). (2007). *Text Entry Systems: Mobility, Accessibility, Universality*: Morgan Kaufmann Publishers.

19. MacKenzie, I. S., and Zhang, S. X. (1999) The design and evaluation of a high-performance soft keyboard. *Proc. of CHI '99*, 25-31.

20. Nielsen. nielsen.com/us/en/newswire/2010/u-s-teen-mobile-report-calling-yesterday-texting-today-using-apps-tomorrow.html

21. Oulasvirta, A., Reichel, A., Li, W., Zhang, Y., Bachnynskyi, M., Vertanen, K. and Kristensson, P.O. (2013) Improving two-thumb text entry on touchscreen devices. *Proc of CHI*. 2765 – 2774.

22. Time.com techland.time.com/2011/07/21/study-fewer-than-50-of-smartphone-users-make-calls/

23. Wobbrock, J.O. (2007). Measures of text entry performance. Chapter 3 in I.S. MacKenzie and K. Tanaka-Ishii (eds.), Text Entry Systems: Mobility, Accessibility, Universality. San Francisco: Morgan Kaufmann, pp. 47-74.

24. Yin, Y., Ouyang, T., Partridge, K., and Zhai, S. (2013) Making touchscreen keyboards adaptive to keys, hand postures, and individuals: a hierarchical spatial backoff model approach. *Proc. of CHI '13*. 2775-2784.

25. Zhai, S., Hunter, M., & Smith, B. A. (2000). The Metropolis Keyboard - an exploration of quantitative techniques for virtual keyboard design. *Proc. of UIST'00*. 119-218.

26. Zhai, S., Hunter, M., & Smith, B. A. (2002). Performance optimization of virtual keyboards. Human-Computer Interaction, 17(2,3), 89-129.