Analyzing the Input Stream for Character-Level Errors in Unconstrained Text Entry Evaluations

JACOB O. WOBBROCK
University of Washington
and
BRAD A. MYERS
Camegie Mellon University

Recent improvements in text entry error rate measurement have enabled the running of text entry experiments in which subjects are free to correct errors (or not) as they transcribe a presented string. In these "unconstrained" experiments, it is no longer necessary to force subjects to unnaturally maintain synchronicity with presented text for the sake of performing overall error rate calculations. However, the calculation of *character-level* error rates, which can be trivial in artificially constrained evaluations, is far more complicated in unconstrained text entry evaluations because it is difficult to infer a subject's intention at every character. For this reason, prior character-level error analyses for unconstrained experiments have only compared presented and transcribed strings, not input streams. But input streams are rich sources of character-level error information, since they contain all of the text entered (and erased) by a subject. The current work presents an algorithm for the automated analysis of character-level errors in input streams for unconstrained text entry evaluations. It also presents new character-level metrics that can aid method designers in refining text entry methods. To exercise these metrics, we perform two analyses on data from an actual text entry experiment. One analysis, available from the prior work, uses only presented and transcribed strings. The other analysis uses input streams, as described in the current work. The results confirm that input stream error analysis yields richer information for the same empirical data. To facilitate the use of these new analyses, we offer pseudocode and downloadable software for performing unconstrained text entry experiments and analyzing data.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation / methodology, theory and methods, input devices and strategies (e.g., mouse, touchscreen)*

This work was funded in part by the NEC Foundation of America, Microsoft Corporation, General Motors, and by the National Science Foundation under grant number UA-0308065. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.
Authors' addresses: J. O. Wobbrock, The Information School, University of Washington, Box 352840, Mary Gates Hall Ste 370, Seattle, WA 98195-2840; email: wobbrock@u.washington.edu; B. A. Myers, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3891; email: bam@cs.cmu.edu.

General Terms: Algorithms, Design, Experimentation, Measurement

Additional Key Words and Phrases: Text entry, text input, error rate, presented string, transcribed string, input stream, optimal alignment, stream alignment, minimum string distance, confusion matrix, gesture, stroke, recognizer, character recognition, nonrecognition, substitution, insertion, omission, deletion, EdgeWrite

1. INTRODUCTION

Text entry research has seen a revival in recent years due to the advent of numerous mobile devices. Many of these devices are communicators that would benefit from rapid and accurate text entry for the sending of information from one party to another [Zhai et al. 2005]. For example, mobile phones are used to send more than 15 billion SMS messages per month in Europe [GSM World 2004]. New communication devices appear all the time, many requiring a text entry method of some kind.

With the advent of new devices and text entry methods, however, comes new pressure on evaluation methodologies. Rigorous evaluations of new methods are required to validate their strengths and weaknesses; informal, subjective results do not warrant trust or comparison [MacKenzie and Soukoreff 2002a]. For this reason, current practice is to rival two or more text entry methods against one another in controlled experiments. In these experiments, subjects are told to transcribe presented phrases as quickly and accurately as possible [MacKenzie and Soukoreff 2003] while data is logged for later analysis.

Hand tabulation of log files is tedious and prone to error, since experiments may contain upwards of 400,000 character entries [MacKenzie and Zhang 1999]. To avoid hand tabulation, automated procedures for computing speed and accuracy are essential. While the computation of speed is straightforward, the computation of error rates is not [Soukoreff and MacKenzie 2001]. For example, in Figure 1, P is a string presented to a subject and T is the subject's transcription (i.e., the final text the subject produced). How many errors are there?

In Figure 1, a simple pairwise comparison suggests that everything after the "c" in T is in error. But this is probably not the case. More likely, there was one insertion error: the "x" in T. Automatically detecting such errors between P and T has been the subject of recent work [Soukoreff and MacKenzie 2001, 2003].

What if we consider not just the final transcribed string, but the entire *input stream*, the record of all input events produced by the subject? If errors were made and corrected, the input stream would hold more error information than the transcribed string. This information could be useful to designers and evaluators for improving techniques. For example, the transcription in Figure 1 could have been produced from the input stream (IS) in Figure 2.

In the input stream in Figure 2, "<" symbols indicate backspaces and bold letters compose the final transcribed phrase T. Clearly, there were more errors in transcribing this text than T alone reveals. Thus, an analysis of T without IS paints an impoverished picture of errors. We could try to analyze IS for

```
P: the quick brown
T: the quicxk brown
                ^^^^^^^^
```

Fig. 1. An example presented (P) and transcribed (T) string.

```
IS: f<tn<he p<qul<ik<cxk bfo<<rown
```

Fig. 2. An example input stream (IS) resulting in T from Figure 1. Transcribed letters are in bold and backspaces are represented by "<".

errors, but input streams are messy and ambiguous, and determining errors within them is difficult. The core challenge in determining input stream errors is assessing *intention*. What is the subject trying to enter at every character position? Because of the difficulty in answering this question, many previous text entry evaluation strategies unnaturally constrained text entry experiments so as to make the determination of errors, particularly character-level ones, trivial. But this comes at the cost of so totally altering the natural transcription process that results from such studies are unavoidably cast into doubt.

For example, one artificially constrained experimental paradigm disallows erroneous characters completely. As the subject transcribes text on a line beneath the presented text, any attempted character that does not match the character directly above it is not displayed. Often in this paradigm, the entry of an erroneous character results in an audible "beep." Subjects may incur many successive beeps, without ever seeing characters appear because their entries do not match the presented character at their current position. What's worse, the backspace is rendered irrelevant, even though backspace is the second most common keystroke in real desktop text entry after space [MacKenzie and Soukoreff 2002a]. However, inferring intention in this paradigm is trivial because we assume that a subject is always trying to enter the next character in the presented text, even though this is often not the case. Not surprisingly, the experience for subjects entering text in these types of evaluations is potentially frustrating, since each error effectively creates a "road block" that stops them abruptly, often for many entries. Nevertheless, this paradigm's ease of use has caused many to employ it [Venolia and Neiberg 1994; Isokoski and Kaki 2002; Evreinova et al. 2004; Ingmarsson et al. 2004].

Another way to unnaturally constrain text entry experiments for the sake of measuring errors is to allow erroneous entries, but to prevent error correction, usually by disabling backspace. Two examples are in the evaluations of the OPTI keyboard [MacKenzie and Zhang 1999] and the Half-Qwerty [Matias et al. 1996]. In this paradigm, subjects are free to enter any letter, but once entered, letters can not be backspaced. Characters are deemed erroneous if they do not agree with the presented letter at their position. Erroneous characters are often accompanied by an audible "beep" and accuracy is calculated as a pairwise comparison between the letters of P and T. Thus, when subjects make errors, they are forced to mentally "catch up" to the presented text by entering the next letter at the current position, rather than by trying the missed letter again. Maintaining synchronicity is therefore of utmost importance, and single insertion errors often result in multiple successive out-of-synchronization

errors known as "error chunks" [Matias et al. 1996]. Given the importance and prevalence of error correction during real text entry, this paradigm, like the one before it, is quite artificial and often frustrating for users.

A third approach has been simply to ignore errors. In this paradigm, errors are not recorded, analyzed, or reported. Indeed, some published studies fall into this category [Lewis et al. 1999; MacKenzie et al. 1999; Rodriguez et al. 2005]. The obvious drawback is that speed and errors are tradeoffs, and analyzing one without the other encourages unreliable and uninformative comparisons.

Unfortunately, all three of these contrived paradigms are uninformative and rather unnatural. This fact has motivated recent developments in automated error rate calculation [Soukoreff and MacKenzie 2001, 2003] and the advent of the *unconstrained* text entry evaluation paradigm. In unconstrained text entry experiments, subjects are presented with phrases and told to transcribe them "quickly and accurately." Subjects are neither forced to maintain synchronicity with the presented text nor required to fix errors, but they may (and usually do). In essence, subjects are free to enter text nearer to how they would in "the real world." Importantly, subjects' subjective experiences more closely match their customary text entry experiences, where error beeps, stalled text cursors, and error chunks are not the norm. The data acquired in unconstrained evaluations is automatically analyzed by algorithms designed to accommodate it. The current work contributes to this set of algorithms by improving our ability to discern intention within the unconstrained paradigm.

1.1 Aggregate and Character-Level Errors

Aggregate error rates give an indication of a text entry method's accuracy over all entered characters. A commonly used aggregate measure for accuracy is *keystrokes per character* (KSPC) [Soukoreff and MacKenzie 2001], which is a ratio of all entered characters, including backspaces, to final characters appearing in the transcribed string. Although the calculation of KSPC requires the input stream, it only requires a *count* of characters entered, not the discovery of what these characters were *intended to be*. This is also true of a widely used definition of "corrected errors" [Soukoreff and MacKenzie 2003] which treats *any* backspaced character as an error, even if the erased character was, in fact, correct. A later article acknowledged this limitation [Soukoreff and MacKenzie 2004], separating backspaced letters into two categories: *corrected-but-right* and *corrected-and-wrong*. However, this article did not offer any algorithms for performing this separation, which is something the current work provides.

Many evaluators have reported KSPC error rates based on the number of backspaces made during entry [Sears and Zha 2003; Wobbrock et al. 2003; Gong and Tarasewich 2005], but these are aggregate, not character-level, error rates. While aggregate error rates provide a baseline for comparison, they are not sufficiently fine-grained to aid designers in targeting problematic characters. Designers and evaluators need more than aggregate measures; they need an indication of what is happening at the level of *individual* characters. For example, a character-level error analysis can tell us the probability of entering a "y" when attempting a "g". It can also tell us which characters are prone to errors

of insertion, omission, or substitution, or which are particularly slow or fast to produce.

Until now, however, the unconstrained text entry paradigm has lacked a formal character-level error analysis that handles input streams. While the unconstrained experimental paradigm is a significant advance over more artificial paradigms mentioned in the previous section, the lack of a character-level error analysis for input streams is a major drawback. Thus far, character-level error analyses for the unconstrained paradigm have focused on presented and transcribed strings [MacKenzie and Soukoreff 2002b], not on input streams. As mentioned, the difficulty in analyzing input streams is that they are fraught with ambiguity, making it hard to discern the subject's intention at each character position. The current work addresses this ambiguity by using a small set of reasonable assumptions. We argue that over the course of a text entry study, the number of times the assumptions are invalid will be far outweighed by the number of times they are valid, a claim supported by empirical results. These carefully formulated assumptions enable us to unlock the rich character-level information of input streams.

1.2 Advantages of Using Input Streams

Analyzing input streams from unconstrained text entry evaluations has a number of practical benefits for designers and evaluators of text entry methods. Among these benefits are:

—The input stream usually yields more data per trial than the transcribed string, since by definition $|IS| \geq |T|$. Therefore, depending on the research questions being asked, analyzing the input stream for character-level errors may allow us to run fewer trials and save time and money on evaluations, which are often time-consuming and expensive [Jeffries et al. 1991]. Such savings will be possible if it is error data that we are after. However, if we are interested in learning rates as measured by speeds over sessions, analyzing *IS* will not reduce the number of sessions required.

—When instructed to "enter the text quickly and accurately" [Soukoreff and MacKenzie 2003], subjects tend to fix most, if not all, of their errors in text entry trials. For example, in the study accompanying a character-level error analysis from the prior work [MacKenzie and Soukoreff 2002b], subjects left only 2.23% errors in T. Other studies show even fewer uncorrected errors: 0.79% [Soukoreff and MacKenzie 2003], 0.53% [Wobbrock et al. 2004], and 0.36% [Wobbrock et al. 2003]. In the extreme case, if subjects correct *all* errors, P and T will be identical and no character-level error information will be available. Such a contingency does not reduce the value of *IS*, however, since *corrected errors* (the errors subjects made but fixed) are still captured therein.

—Speed and uncorrected errors are tradeoffs in text entry. Therefore, to equitably compare speeds, some experiments [Lewis 1999] have required *perfect transcription*, where leaving errors in T is not permitted. But character-level error analyses of perfect transcription studies are useless when employing

only P and T, since they will always be identical. Analyzing IS, on the other hand, allows for the extraction of character-level results, even in perfect transcription studies.

—For stroke-based or handwritten text entry, such as Graffiti [Palm Inc. 1995], one possible outcome of an attempted character is a *nonrecognition*. By definition, T cannot contain nonrecognitions, but IS can. Therefore, looking at IS can be valuable to designers who are trying to identify characters that are difficult to recognize in stroke-based text entry methods.

1.3 Limitations of This Analysis

Like any automated analysis, the current work has limitations. It is designed to analyze data from unconstrained text entry experiments in which subjects transcribe, rather than generate, text. Although generating text is more natural, it has numerous problems when used in experiments, such as introducing thinking time, complicating the identification of errors, and abdicating control of letter and word distributions [MacKenzie and Soukoreff 2002a, 2003]. For these reasons, text entry evaluations nearly always involve text transcription.

In the unconstrained paradigm, text is assumed to flow serially forward with the entry of new characters and backward with the correction operation (i.e., backspace). Therefore, the current analysis does not accommodate "random access" editing using cursor keys, multicharacter selection, or the mouse cursor. Indeed, a comprehensive analysis of text entry performance is a current goal of researchers, but even the metrics for such an analysis are not yet understood, let alone algorithms for their automated measurement.

The current analysis does not accommodate *method-specific* tokens in the input stream, such as the individual keypresses in Multitap (e.g., 222-NEXT-2-8 = "cat"). Rather, the current analysis examines input streams containing all entered characters and backspaces in a method-agnostic fashion that makes it suitable for any character-level text entry method. Method-specific analyses would have to be implemented in a method-specific fashion, and are viewed as supplements to this work.

The current work functions well for any type of character-level method. These methods include typing, unistrokes, stylus keyboards, eye-tracking keyboards, thumbwheels, character recognizers, and so on—any technique that produces one character at a time. The results are less informative, however, for *word-level* methods that produce multicharacter chunks. Examples of such methods are word-producing *sokgraphs* [Zhai and Kristensson 2003], word prediction systems [Wobbrock et al. 2006], and speech recognition systems [Karat et al. 1999]. Although the algorithms we present will work for such methods, character-level errors are less relevant, since the "character" is not the unit of production. Our algorithms could be adapted to accommodate word-level entry by treating multicharacter chunks as the individual units of production. Such an extension is not a theoretical complication, but is beyond the scope of this work.

In the remainder of this article, we first describe a character-level error analysis for the unconstrained text entry paradigm from prior work [MacKenzie and Soukoreff 2002b]. This prior analysis examines only P and T, but is built upon

```
P: quickly
T: qucehkly
```

Fig. 3. The example used in the prior analysis [MacKenzie and Soukoreff 2002b]. We continue with this example for continuity.

directly by the current work. Next, we describe our extensions to this prior work that allow us to analyze input streams. We present a taxonomy of input stream error types, including the assumptions underlying each, and give algorithms for their detection. Finally, we offer two analyses of real experimental data, one with the prior analysis of just P and T, and one with our new analysis using P, T, and IS. With our new analysis and the software that implements it, designers and evaluators stand to benefit from richer character-level error information which, in turn, will result in more refined text entry techniques and rigorous evaluations.

2. RELATED WORK

There is much work in the general area of string sequence comparisons [Sankoff and Kruskal 1983]. Algorithms have been developed in computer science and computational biology to find common sequences within a single string [Karp et al. 1972], within a family of strings [Landraud et al. 1989], and within a set of molecules [Waterman 1984]. An in-depth review of this work is beyond the scope of this article, but it should be noted that our current problem, inferring the intention for each character in the input stream, does not reduce to finding common substrings or repeating patterns. Rather, it is specific to the activity of transcribing a presented string in an unconstrained text entry experiment. This involves, for example, the unique treatment of backspace as a special character that removes the previous character. The current work is therefore not just an application of prior algorithms, but a new procedure developed for use in the domain of text entry evaluation.

The current work builds on a prior character-level analysis of just P and T [MacKenzie and Soukoreff 2002b]. This technique must first be understood before describing the current work. A brief overview is therefore presented. Readers are directed to prior work for more details [Soukoreff and MacKenzie 2001; MacKenzie and Soukoreff 2002b].

The prior analysis began by asking, "How many errors are in the transcription shown in Figure 3?"

A pairwise comparison of the letters in Figure 3 suggests that all letters after the "qu" are in error. But intuition tells us that the "kly" at the end seems correct. How can we determine the correct number of errors?

The answer is by using the *minimum string distance* (MSD) [Levenshtein 1965; Wagner and Fischer 1974; Soukoreff and MacKenzie 2001].[1] The MSD statistic tells us the shortest "distance" between two strings, which can be characterized as the minimum number of errors between them. This is also equivalent to the minimum number of simple editing operations, so-called

[1]The MSD algorithm is often attributed to Vladimir I. Levenshtein [1965] but was discovered independently by others (e.g., Wagner and Fischer [1974]. The algorithm appears in Figure 28.

```
P1: qu-ickly
T1: qucehkly

P2: qui-ckly
T2: qucehkly

P3: quic-kly
T3: qucehkly

P4: quic--kly
T4: qu-cehkly
```

Fig. 4. The optimal alignments of "quickly" and "qucehkly" [MacKenzie and Soukoreff 2002b].

"Morgan's operations" [Morgan 1970], required to turn one string into the other.

In Figure 3, MSD = 3. This means that there are a minimum of 3 errors in T relative to P. It also means that in no less than 3 simple editing operations we can make "quickly" and "qucehkly" into equivalent strings.[2] The error types and associated corrections described next are treated in greater detail elsewhere [Gentner et al. 1984]. Note that these are "method-agnostic" types that may occur in any character-level text entry system.

Insertion: occurs when a letter appears in T, but not in P. For example, if P is "cat" and T is "cart", there is an insertion for "r". We can represent an insertion by placing a hyphen "-" in P where the insertion appears in T. For example, we would write "cat" as "ca-t".

Omission: occurs when a letter appears in P, but not in T. For example, if P is "cat" and T is "ct", there is an omission of "a". We can represent an omission by placing a hyphen in T where the omitted letter appears in P. For example, we would write "ct" as "c-t". Other work has coined these errors "deletions" [MacKenzie and Soukoreff 2002b], but "omission" will be used here, as in earlier articles [Gentner et al. 1984].

Substitution: occurs when corresponding letters in P and T do not agree. For example, if P is "cat" and T is "kat", there is a substitution of "k" for "c".

As stated, MSD = 3 for our example. But although we know *that* 3 errors exist between P and T, and therefore 3 editing operations are necessary to equate them, we do not know *which* 3 errors occurred or which 3 operations reflect the subject's intentions. For example, did the subject substitute the "c" for "i" in Figure 3 or did he omit the "i" and correctly enter the "c"?

To handle this ambiguity, we first generate all possible operation sets of cardinality 3 that make P and T equivalent. These are called the *optimal alignments* of P and T [MacKenzie and Soukoreff 2002b]. There are 4 optimal alignments for our example in Figure 3, shown in Figure 4.

To detect errors after identifying the optimal alignments, we simply move through the (P_n, T_n) pairs, comparing letters in a pairwise fashion at each position. If two letters agree, a *no-error* is the result. If two letters disagree,

[2]These simple editing operations were noted from examining the most common mistakes in writing computer programs [Damerau 1964]. They were later applied to automatic spelling correction [Morgan 1970], and automated with the MSD algorithm [Wagner and Fischer 1974].

a *substitution* occurred. If a hyphen appears in P, an *insertion* occurred. If a hyphen appears in T, an *omission* occurred.

Ambiguity is handled by weighting each error by the inverse of the number of alignments. For example, in our aforementioned 4 alignments, each of the 3 substitutions for "i" is tallied as 1×0.25. Summed together, we get 0.75 as the total substitution error rate for "i". That is, we can say that there is a 75% chance that the user committed a substitution for "i" while entering "qucehkly". Accordingly, there is only a 25% chance that the user omitted "i", as seen in the fourth alignment.

The main limitation of this prior analysis is that it ignores all backspaced characters. In most unconstrained text entry experiments, corrected errors greatly outnumber uncorrected errors. This means that much richer error data is available to us if we include corrected errors in our analyses. This is particularly true for character-level errors, since uncorrected character-level errors may be infrequent for any given character. Including corrected character-level errors allows us to see which characters really are error-prone during entry.

3. MAKING SENSE OF INPUT STREAMS

The analysis of input streams yields new types of errors. It also entails new complexities due to ambiguity. This section describes both.

3.1 Input Stream Error Types

When we move from an analysis of just P and T to an analysis of P, T, and IS, new types of errors arise. These new error types provide more detail about the text entry process and give us a more powerful scope under which to view errors. This section presents a taxonomy of input stream error types with relevant examples. The assumptions used in this section are made explicit in Section 3.2.

3.1.1 *Uncorrected No-Errors, Substitutions, Insertions, and Omissions.* Hereafter, Gentner et al.'s errors [Gentner et al. 1984] are prefaced with the term "uncorrected" to indicate that these errors remain in the transcribed string. So, we now have uncorrected no-errors, uncorrected substitutions, uncorrected insertions, and uncorrected omissions. The definitions of uncorrected errors remain unchanged from prior work [Gentner et al. 1984; MacKenzie and Soukoreff 2002b].

We now turn to the new error types introduced by the current work, namely, corrected character-level errors.

3.1.2 *Corrected No-Errors.* Characters that are already correct are erased surprisingly often in the text entry process, particularly when touch-typing [Soukoreff and MacKenzie 2004]. We call these correct-but-erased characters *corrected no-errors*. When combined with uncorrected no-errors, they compose the set of all correctly entered characters.

3.1.3 *Corrected Substitutions.* Consider the input stream in Figure 5, where "<" is a backspace.

```
 P: quickly
IS: qv<w<uickly
```

Fig. 5. An input stream showing difficulty entering "u".

```
 P: quickly
IS: qv<w<xickly
```

Fig. 6. The "v" and "w" are corrected substitutions for "u", while the "x" is an uncorrected substitution for "u".

```
 P: quickly
IS: qvlck<<<<uickly
```

Fig. 7. The "vl" is erroneous, but the subject did not correct these letters until after correctly entering the first "ck".

This input stream shows a correctly entered "q", but apparent trouble producing the following "u". The subject first entered, and then backspaced, a "v" and a "w" before correctly entering the "u". This might be because "u", "v", and "w" are keys that are physically too close together on a mobile device's miniature keyboard, or because the user easily confused strokes for "u", "v", and "w" in a stylus method. For example, "u"-"v" confusion is common for novices when writing Graffiti [MacKenzie and Zhang 1997].

In Figure 5, "v" and "w" are considered *corrected substitutions* for "u" because if either "v" or "w" (but not both) were left in lieu of "u", we would have had an *un*corrected substitution for "u". Note that the term "corrected" refers to the fact that "v" and "w" were *backspaced*, not to the fact that "u" correctly ended up in T. For example, "v" and "w" are corrected substitutions for "u" in Figure 6, despite an "x" persisting as an *un*corrected substitution for "u".

Subjects in text entry experiments often go a few letters past erroneous entries before noticing their errors and backspacing to fix them [Soukoreff and MacKenzie 2004]. This can result in an input stream like that of Figure 7.

In Figure 7, the subject may not have spotted the erroneous "vl" until after the first "ck". The "v" should be classified as a corrected substitution for "u", the "l" as a corrected substitution for "i", and the erased "ck" as corrected no-errors, since they were initially correct, despite having to be erased.

3.1.4 *Nonrecognition Substitutions.* A *nonrecognition substitution* occurs when an attempt to produce a character yields no result. Although nonrecognitions are more applicable to stroke-based entry than to keys or buttons, virtual keyboards may regard clicks or taps in the "dead space" between keys or along their margins as nonrecognitions, since these are also unproductive attempts to enter characters. Thus, nonrecognitions are applicable to methods beyond those that use recognizers. Indeed, any method in which an attempt to produce a character can produce nothing at all is relevant.

We can represent nonrecognitions in the input stream as "ø". Consider the input in Figure 8. In this example, the first attempt at "u" produced no actual character ("ø"). The second attempt produced a "u", and the subject proceeded

```
 P: quickly
IS: qøuickly
```

Fig. 8. The input stream contains a nonrecognition "ø".

```
 P: quickly
IS: qxui<<<uickly
```

Fig. 9. The "x" in the input stream is a corrected insertion.

```
 P: quickly
IS: quicklxa<
```

Fig. 10. The input stream shows an "a" inserted beyond the corresponding length of the presented string.

correctly thereafter. Note that no backspace is required to remove a nonrecognition, since it does not represent any printed character.

There are various ways to add nonrecognitions to input streams (*IS*). One is for a text entry technique to send a special nonprintable character code to be trapped by the user test software and logged directly as a nonrecognition. This approach may be called *explicit nonrecognition handling*. A second approach is for a log file analyzer to *infer* a nonrecognition when it sees that a stroke (or other effort) occurred, but did not produce a character. This approach can be called *implicit nonrecognition handling*. Our test software, described in Section 5, supports both approaches.

3.1.5 *Corrected Insertions.* Insertions are extra characters in T or *IS* that lack a corresponding character in P. Consider the input in Figure 9. Here, it seems the subject inserted an "x" before the first "u", but later noticed and erased it. If the "x" were not erased, it would have resulted in an *un*corrected insertion. Note that this determination is independent of the fact that a "u" is ultimately transcribed in T. It is *not* independent, however, of the fact that a "u" immediately follows the inserted "x". It relies on this fact to determine that the "x" was inserted and not an attempted "u", and the "u" an attempted "i", which is possible, but not likely.

Note that in Figure 9, the first "u" and "i" are treated as errors by aggregate measures that only count backspaces in the input stream [Soukoreff and MacKenzie 2003]. But clearly, the first "u" and "i" are correct, and the current analysis treats them as such, classifying them as corrected no-errors.

A second type of corrected insertion is when characters are entered beyond the length of P. Figure 10 shows an example. In this example, it seems that the "a" was inserted at the end of *IS* and then erased. Since all the letters in P and *IS* are already matched, the "a" is deemed a corrected insertion.

Yet a third type of corrected insertion occurs when we have two identical letters in a row, and the first is correct, but the second incorrect. Consider Figure 11. In this example, it seems likely that the third "e" is not an attempt at "c", but an accidental double-entry of the previous "e". This type of corrected insertion is probably more common to keypad or keyboard entry than to

```
P: speech
IS: spee<ch
```

Fig. 11. The third "e" in the input stream could be the result of an accidental doubling of the correct "e" before it.

```
P: speech
IS: spedd<<ech
```

Fig. 12. The "d"s are both deemed corrected substitutions. The second "d" is not a corrected insertion because the "d" before it is an error.

```
P: cat
IS: catøø
```

Fig. 13. The input stream shows two nonrecognition insertions.

```
P: quickly
IS: quikl<<ckly
```

Fig. 14. The "c" is initially omitted, resulting in a corrected omission.

stroke-based entry, since double-entries can occur when physical buttons are pressed too firmly, held down too long, or because the key-repeat rate is too fast. Double-entries sometimes occur with a stylus on a virtual keyboard for subjects with tremor [Wobbrock et al. 2003]. Input techniques can be "debounced" to help protect against these kinds of insertions.

A requirement for this type of corrected insertion is that the character prior to the potential double-entry is correct (the second "e" in Figure 11). This increases our confidence that the character under consideration (the third "e") is indeed a double-entry.

Now consider Figure 12. This input should probably not be treated as having a corrected insertion for the second "d" because the first "d" itself is erroneous as a corrected substitution for "e". In this case, the second "d" is treated as a corrected substitution for "c".

3.1.6 *Nonrecognition Insertions.* Only the second of the three types of corrected insertion, those caused by entering letters beyond the length of P (Figure 10), applies to *nonrecognition insertions*. For example, see Figure 13. The input stream in Figure 13 shows two nonrecognition insertions, both of which occur after each letter in P is already paired with a letter in IS.

3.1.7 *Corrected Omissions.* Omissions occur when characters in P are skipped in T or IS. *Corrected omissions* occur when a character in P is initially skipped, but then later replaced, thereby remedying the omission. Figure 14 shows an example. It seems the "c" is initially skipped, but later replaced, resulting in a corrected omission (corrected no-errors are tallied for the first "kl"). Note that the classification of "c" as a corrected omission does not depend on a "c" ultimately being transcribed. For example, consider Figure 15. The input in Figure 15 contains a corrected omission for "c" because "c" was initially

```
P: quickly
IS: quikl<<xkly
```

Fig. 15. The "c" is initially omitted, resulting in a corrected omission, even though an "x" takes its place as an uncorrected substitution for "c".

```
P: cats
IS: cuf<<ats
```

Fig. 16. Our first assumption says that subjects proceed sequentially through the presented string, so "uf" in *IS* is matched with "at" in *P*.

```
P: cat
IS: cx<at
```

Fig. 17. Was the "x" an attempted "a" or merely a corrected insertion? Our first assumption favors the former.

omitted. This classification depends on the correctly entered "k" immediately following the "i". If the "k" were a different letter, we would have no reason to believe that the letter was not an attempted "c" and therefore a corrected substitution. Note that the transcribed "x" results in an uncorrected substitution for "c".

3.2 Assumptions for Resolving Ambiguity

The previous taxonomy made some implicit assumptions to resolve ambiguity. This section makes these assumptions explicit, arguing for their reasonableness and necessity if value is to be extracted from input streams at the level of individual characters.

3.2.1 *Subjects Proceed Sequentially Through P.* The first assumption is that subjects proceed sequentially through *P* as they enter *IS*. This assumption is the bedrock on which text entry transcription studies are built. Consider Figure 16. Our first assumption pairs "uf" with "at". Without this assumption, we would have to allow that any letter in *IS* could be paired with any letter in *P*, or none in *P* at all! This assumption is reasonable, given the nature of text entry experiments in which subjects are instructed to sequentially transcribe presented strings.

Now consider Figure 17. In this example, it seems the subject entered an "x" while attempting an "a". But are we sure? What if the "x" was not an attempted "a", but an insertion that was promptly corrected? In other words, do we have a corrected substitution of "x" for "a" or a corrected insertion of "x"? Our first assumption favors the former: "x" is paired with "a". Prior character-level evidence shows that substitutions are much more common than insertions [MacKenzie and Soukoreff 2002b].

However, this assumption does not prevent us from detecting corrected insertions and corrected omissions. Examine Figure 18. In the first pair, was "x" an attempted "a" and "a" an attempted "t"? Or was the "x" simply inserted? Because the first "a" in IS_1 matches the "a" in P_1, we favor the latter and report a corrected insertion for "x" and a corrected no-error for "a".

```
P1: cat
IS1: cxa<<at

P2: cat
IS2: ct<at
```

Fig. 18. These two examples show a corrected insertion and a corrected omission, respectively.

```
P1: cats
IS1: cxfa<<<ats

P2: cats
IS2: cs<ats
```

Fig. 19. Our second assumption allows for only one corrected insertion or omission in a row. Thus, these examples contain only corrected substitutions.

```
P: cats
IS: cxfs<<<ats
```

Fig. 20. Our second assumption keeps us from treating the "at" in P as initially omitted and the "xf" in IS as initially inserted. Instead, our assumption simply gives us corrected substitutions.

In the second pair in Figure 18, was "t" an attempted "a" or was "a" omitted and then promptly added? Since the first "t" in IS_2 matches the letter after the "a" in P_2 (the "t"), we favor the latter interpretation and report a corrected omission for "a" and a corrected no-error for the first "t".

3.2.2 *Subjects Insert or Omit Only One Character in a Row.* The second assumption states that subjects insert or omit only one character at a time. Consider these variations on the input streams from Figure 18, as depicted in Figure 19.

In the first pair, we could regard both "x" and "f" in IS_1 as corrected insertions because they are followed by an "a". But then, at how many consecutive insertions do we draw the line? For tractability, we choose to draw it at one and deem "xfa" an attempted "ats", reporting them all as corrected substitutions.

In the second pair, we might regard both "a" and "t' in P_2 as corrected omissions. But again, at how many consecutive omissions do we draw the line? We choose to allow one omission in a row, since subjects are instructed to progress sequentially through P, and skipping multiple letters is uncommon in practice.

Our assumptions so far allow us to avoid considering unlikely possibilities, such as that of Figure 20. The limitation of one insertion or omission in a row prevents us from treating the "at" in P as omissions and "xf" in IS as insertions. Instead, we simply have corrected substitutions of "xf" for "at", which are much more likely and straightforward.

3.2.3 *Backspaces Are Made Accurately and Intentionally.* The third assumption is that backspaces are made both accurately and intentionally. Of course, backspace ("<") is fundamental to text entry. Keyboards usually have relatively large backspace keys. Unistroke alphabets such as Graffiti [Palm Inc.

```
 P : cat
IS1: cxø<at
IS2: cxz<<at
```

Fig. 21. Our third assumption asserts that backspaces are made accurately, but "ø<" and "<<" patterns may indicate exceptions.

```
 P: cat
IS: ca<at
```

Fig. 22. Our third assumption asserts that backspaces are made intentionally, but "$x< \ldots <x$" patterns may indicate exceptions.

1995] and EdgeWrite [Wobbrock et al. 2003] assign simple straight-line strokes to backspace. Designers are motivated to make backspace quick and accurate due to its frequency and importance. As previously stated, backspace has been shown to be the second most common keystroke in actual desktop PC use, after the spacebar [MacKenzie and Soukoreff 2002a].

The current analysis assumes that backspaces are made accurately, that is, an attempted "<" results in a "<". Since we do not have any backspaces in P to compare to those in IS, there is no implied intention on which we can rely. Accommodating the possibility of erroneous backspaces greatly complicates the analysis and is an ambitious topic for future work.

However, not all backspaces are going to be made accurately in a large text entry experiment. Consider the cases in Figure 21. In IS_1, we regard the nonrecognition ("ø") as an attempted "t", after which the subject notices and erases the erroneous "x". But what if the nonrecognition was a failed *first* attempt at a backspace? It is difficult to say, since subjects often notice errors only after they've gone past them, and "ø" could easily have been an attempted letter.

Similarly, in IS_2 we regard the "z" as an attempted "t", after which the subject notices the erroneous "x" and erases the "xz". But what if the "z" was an attempted backspace to begin with? Again, there is no way to know, but the "<<" pattern may indicate a failed attempt at backspace. Then again, subjects often move a few letters past an error before noticing and backspacing to fix it, so perhaps the "z" was an attempted "t", after all. There is simply no way to know, so we rely on our assumption.

The third assumption also says that backspaces are made *intentionally*, that is, an attempt at something other than "<" does not result in a "<". Consider the example in Figure 22. Perhaps the subject initially thought the "a" was in error. Or perhaps the backspace was an attempted "t", which erased the already correct "a", and therefore the "a" had to be replaced. Under this assumption, we treat the backspace as intentional. Corrected no-errors like the first "a" in Figure 22 can occur when subjects enter text quickly, since they will occasionally anticipate an error and enter a backspace, even when their entry turns out to have been correct.

To assess the feasibility of these assumptions about backspace for a stroke-based method, we manually examined logs from a unistroke text entry study

```
 P: cats
IS: cax<y<z
 T: caz
```

Fig. 23. This input is ambiguous as to whether the "z" should be aligned with "t" or with "s". Both possibilities are represented in the optimal alignment set and weighted accordingly.

[Wobbrock et al. 2003]. If backspaces were error-prone, we should see many "ø<" and "<<" patterns in the input streams. If backspaces were unintentional, we should see many "$x < \cdots < x$" patterns, where x is a correct entry and "..." are optional intervening characters.

Together, subjects in the study attempted 4,932 characters. Of these, 4,660 produced characters and 272 were nonrecognitions. Of the 4,660 entered characters, 4,207 were alphanumeric and 453 were backspaces. The sequence "ø<" was observed 55 times, and "<<" was observed 38 times. In scrutinizing the strokes by hand, however, we saw that only 8 "ø<" were due to unrecognized attempts at backspace. Similarly, only 2 "<<" were due to misrecognized backspaces. Thus, only 10/453, or 2.21%, of attempted backspaces appeared to violate the first part of our assumption.

Because recognition-based methods are often less accurate than selection-based methods during entry [Költringer and Grechenig 2004], the first part of our third assumption is likely to hold even more reliably for stylus keyboards, on-screen keyboards, mini-Qwerty keyboards, and others, since backspace is selected and not the result of a potentially sloppy stroke.

The sequence "$x < \cdots < x$", where x is a correct entry, was observed 6 times. This means that only $\frac{6}{4207-2+6}$, or 0.14%, of attempted alphanumeric characters were unintentional backspaces, supporting the reasonableness of the second part of our third assumption.[3]

This data was from a study of first-time users after minimal practice. The aforementioned percentages would probably be even smaller for subjects with more practice. In any event, a technique with a highly inaccurate or inadvertent backspace should first remedy that rather dire problem before performing a more in-depth character-level error analysis.

3.2.4 *Omissions in T Are Also Omitted in IS.* The fourth assumption says that letters skipped in T are also skipped in IS. Consider Figure 23. The transcription T for this input aligns with P as both "caz-" and "ca-z". Two alignments are needed because we cannot be sure whether "z" was an attempted "t" or an attempted "s". It may be that the subject was attempting "t" all along and forgot the "s" at the end. Or it may be that the subject did not see the "t" and was attempting the "s". As discussed previously, weighting by the number of alignments accommodates this uncertainty. Thus, in the first alignment ("caz-"), "x" and "y" are treated as attempts at "t", while in the second ("ca-z"), they are treated as attempts at "s".

[3]The denominator is the total number of attempted alphanumerics: the total number of produced alphanumerics (4,207) minus those that were attempted backspaces (2) plus the backspaces that were meant to be alphanumerics (6).

```
 P: cats
IS: cax<y<s
 T: cas
```

Fig. 24. The fourth assumption treats the "x" and "y" as attempts at "s".

```
 P: cats
IS: caf<<ts
 T: cts
```

Fig. 25. The subject overshot with backspace past the erroneous "f" and through the correct "a", which he or she then neglected to replace.

On the other hand, what if we had an "s" in the final transcription, instead of a "z"? This is shown in the example in Figure 24. In this case, there is only one optimal alignment of T with P: "ca-s", which contains an uncorrected omission for "t". Because the "t" was skipped, we assume that the "x" and "y" were not attempts at "t", but attempts at "s". After all, it is with the entry of an "s" that the subject becomes satisfied with leaving a character in this position. To assume otherwise is to allow that "x" was perhaps first an attempt at "t", and that "y" was another attempt at "t" —or perhaps suddenly an attempt at "s"? We would have to assume that subjects left nothing to show for their attempts at "t" and became misaligned as a result.

In our experience over many text entry studies, subjects are rarely this capricious. Regardless of the text entry method being used, subjects try hard to stay aligned with P while transcribing T. If they fail to obtain a correct letter after many tries, they often leave their final incorrect attempt and proceed, thereby remaining aligned with P, rather than erasing their final attempt and becoming misaligned thereafter. In other words, if the "y" in Figure 24 was indeed an attempt at "t", then it is unlikely that subjects would first erase it before attempting the "s" because this throws them out of alignment. If indeed "y" was a failed attempt at "t", subjects would usually leave it and simply try the "s". In short, subjects eschew gaps and strive for alignment.

One complication with this assumption, however, is when subjects overshoot with backspace and neglect to replace letters. For example, Figure 25 shows the erasure of both an erroneous "f" and correct "a". But nothing is replaced for the "a" that was accidentally erased, so the aligned transcription is "c-ts", which contains an uncorrected omission for "a". Since the subject omitted "a" in T, it is assumed that he or she omitted it in IS, but that is probably not true here. This complication arises because over-backspacing without replacement injects ambiguity into discerning intention. Fortunately, over-backspacing without replacement is relatively uncommon, and is discouraged by well designed user test software that employs fixed-width fonts for P and T and displays P above T in close proximity. Manual inspection of data from the aforementioned study [Wobbrock et al. 2003] revealed that only 2/453, or 0.44%, of backspaces were overshoots. For both of these overshoots, the letter erased *was* replaced, meaning that our fourth assumption held for all 4,207 characters. Still, relaxing this assumption is a topic for future work.

```
count: 12100010001000121000010 ←
   IS: pv<<quc<cøk<ehly<<klyz< ←
```

Fig. 26. The first step is to flag the letters in the input stream that compose the transcribed string using a backward pass.

```
FLAG-STREAM(IS)
 1  count ← 0
 2  for i ← |IS| − 1 to 0 do
 3     if IS[i] = '<' then
 4        count ← count + 1
 5     else if IS-LETTER(IS[i]) then
 6        if count = 0 then FLAG(IS[i])
 7        else count ← count − 1
 8  return IS
```

Fig. 27. Algorithm for flagging the characters in IS that compose T.

4. ERROR DETECTION AND CLASSIFICATION

We have developed an algorithm to automate the detection and classification of the 10 error types described earlier. Automatically generated error reports can aid designers and evaluators in improving text entry techniques by revealing troublesome characters.

4.1 Algorithm Walkthrough Step-by-Step

As noted, the algorithm from prior work on which the current work builds compared only P and T [MacKenzie and Soukoreff 2002b]. The current work compares P, T, and IS. We give pseudocode so that technique designers and evaluators can incorporate this analysis into their own work. In the code, both space and letters are referred to as just "letters" and backspace is the only error correction. In **for** loops, the **to** keyword includes the upper bound. For readability, string indices are not bounds-checked, and the comparison of $A[i]$ to $B[j]$ is taken to be **false** if either index i or j is beyond the end of its respective string A or B.[4] The notation $|S|$ denotes the length of string S or the cardinality of set S. The "←" symbol means "assign", while the symbol "$\leftarrow^{+}$" means "append to string" or "add to set." The style of this pseudocode is based on that of a popular algorithms book [Cormen et al. 1990].

4.1.1 *Flag the Input Stream.* In the first step, we "flag" the transcribed letters in the input stream (i.e., we flag the letters in IS that compose T). We do this with a backward pass over IS, incrementing a counter after passing "<", decrementing it (but not below zero) after passing a letter, and leaving it unchanged after passing a nonrecognition ("ø"). We flag letters for which the counter is zero *before* decrementing. Figure 26 shows the hypothetical input stream for T = "qucehkly" from Figure 3. Flagged letters are bold. The pseudocode for doing the flagging is given in Figure 27.

[4]Where necessary, one can add checks so that tests fail if an index is out of bounds. Thus, **if** $(A[i] = B[j])$ becomes **if** $(i < |A|$ **and** $j < |B|$ **and** $A[i] = B[j])$.

```
MSD-MATRIX(P, T)
 1  D ← new matrix of dimensions |P| + 1, |T| + 1
 2  for i ← 0 to |P| do
 3     D[i, 0] ← i
 4  for j ← 0 to |T| do
 5     D[0, j] ← j
 6  for i ← 1 to |P| do
 7     for j ← 1 to |T| do
 8        D[i, j] ← MIN(D[i – 1, j] + 1,
 9                      D[i, j – 1] + 1,
10                      D[i – 1, j – 1] + P[i – 1] ≠ T[j – 1])
11  return D[|P|, |T|] and D
```

Fig. 28. Algorithm for computing the minimum string distance. In this case, the ≠ comparator returns integer '1' if **true** and integer '0' if **false**.

```
ALIGN(P, T, D, x, y, P', T', ref alignments)
 1  if x = 0 and y = 0 then
 2     alignments ←+ (P', T')  // add a new aligned pair
 3     return
 4  if x > 0 and y > 0 then
 5     if D[x, y] = D[x – 1, y – 1] and P[x – 1] = T[y – 1] then
 6        ALIGN(P, T, D, x – 1, y – 1, P[x – 1] + P', T[y – 1] + T')
 7     if D[x, y] = D[x – 1, y – 1] + 1 then
 8        ALIGN(P, T, D, x – 1, y – 1, P[x – 1] + P', T[y – 1] + T')
 9  if x > 0 and D[x, y] = D[x – 1, y] + 1 then
10     ALIGN(P, T, D, x – 1, y, P[x – 1] + P', "-" + T')
11  if y > 0 and D[x, y] = D[x, y – 1] + 1 then
12     ALIGN(P, T, D, x, y – 1, "-" + P', T[y – 1] + T')
```

Fig. 29. Algorithm for computing the optimal alignments of P and T. Reproduced with permission [MacKenzie and Soukoreff 2002b].

4.1.2 *Compute the MSD Matrix.* The second step is to compute the minimum string distance (MSD) matrix. The MSD algorithm fills a matrix of integers as it determines the minimum number of simple editing operations required to equate two strings. We adjust a prior version of the MSD algorithm [Soukoreff and MacKenzie 2001] to return not just the string distance, but also the filled MSD matrix, which we will use in the next step. Readers wishing further details on this algorithm are directed to that article or earlier ones [Levenshtein 1965; Wagner and Fischer 1974].

4.1.3 *Compute the Set of Optimal Alignments.* The third step is to compute the set of optimal alignments of P and T, as in Figure 4. In Figure 29, D is the MSD matrix, and x and y are initialized with the lengths of P and T, respectively. P' and T' are initially empty strings and the "+" operation on them is "string concatenation." For more information, readers are directed to prior work [MacKenzie and Soukoreff 2002b].

4.1.4 *Stream-Align IS with P and T.* In the fourth step, we add a copy of *IS*, still flagged, to each optimal alignment pair computed by ALIGN, turning them into optimal alignment *triplets*. We then *stream-align* the triplets so that P, T, and *IS* are all aligned. We do this by aligning the flagged letters in *IS* with their correspondents in P and T. The last alignment from Figure 4 looks like Figure 30 when stream-aligned.

```
P4:  _____qui__c___--____kly__
T4:  _____qu-__c___eh____kly__
IS4: pv<<qu_c<cøk<ehly<<klyz<
```

Fig. 30. A stream-aligned triplet of (P, T, IS). This is the fourth of the optimal alignments from Figure 4.

```
STREAM-ALIGN(IS, alignments)
 1  foreach aligned pair (P', T') in alignments do
 2     IS' ← COPY(IS)
 3     for i ← 0 to MAX(|T'|, |IS'|) – 1 do
 4        if T'[i] = '-' then
 5           INSERT('_', IS'[i])
 6        else if not IS-FLAGGED(IS'[i]) then
 7           INSERT('_', P'[i])
 8           INSERT('_', T'[i])
 9     triplets ←+ (P', T', IS')  // add a new aligned triplet
10  return triplets
```

Fig. 31. Algorithm for aligning P, T, and IS.

```
   → 01100000000000000011000000
IS4: pv<<qu_c<cøk<ehly<<klyz<
```

Fig. 32. Position values are shown atop characters in the input stream. They are assigned using a forward pass.

The triplet in Figure 30 uses underscore spacers ("_") in P and T where flagged letters in IS are absent, and in IS where uncorrected omissions are in T (i.e., where "-" symbols appear in T). Figure 31 gives the pseudocode for stream-aligning P, T, and IS.

4.1.5 *Assign Position Values to Characters in the Input Stream.* The fifth step is to assign "position values" to each character in IS. Position values help determine the intended target letter in P for each letter in IS. Flagged letters always receive a position value of zero. Within each unflagged substring between two flags in IS, a letter's position value is the substring index it *would* have had if the substring were transcribed unto this point. A backspace's position value, by contrast, is the position value of the letter that the backspace erases. The position values for our example are shown in Figure 32. The algorithm for assigning position values is shown in Figure 33. The function receives the set of triplets from STREAM-ALIGN.

4.1.6 *Proceed Through IS to Detect and Classify Errors.* The sixth and final step of the algorithm proceeds through each IS in each stream-aligned triplet and classifies each input stream character. The procedure for doing this is shown in Figures 35–37. It takes successive substrings between flagged letters in IS, where the substrings include a flagged character on their right ends but not on their left. Thus, the first three substrings of IS_4 from Figure 30 are " pv<<**q**", "**u**", and "_c<**c**". Letters within a substring are then compared to corresponding letters in P. These corresponding letters are determined using the position values assigned in Figure 33.

```
ASSIGN-POSITION-VALUES(ref triplets)
 1  foreach aligned triplet (P, T, IS) in triplets do
 2    pos ← 0
 3    for i ← 0 to |IS| – 1 do
 4      if IS-FLAGGED(IS[i]) then
 5        SET-POSITION-VALUE(IS[i], 0)
 6        pos ← 0
 7      else
 8        if IS[i] = '<' and pos > 0 then
 9          pos ← pos – 1
10        SET-POSITION-VALUE(IS[i], pos)
11        if IS-LETTER(IS[i]) then
12          pos ← pos + 1
```

Fig. 33. Algorithm for assigning position values to characters in *IS*. Position values help determine the intended letter in *P* for each letter in *IS*.

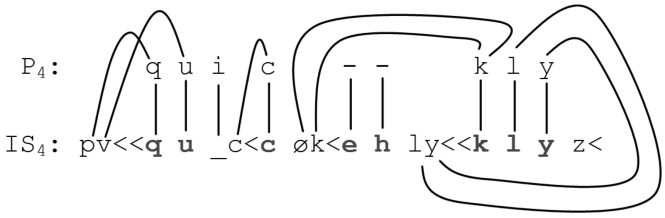

Fig. 34. Comparisons made for the fourth stream-alignment (Figure 30).

The comparisons performed for the fourth stream-alignment (Figure 30) are illustrated in Figure 34. For clarity, underscore spacers have been removed from *P*. Note that this processing is applied to each alignment triplet returned by STREAM-ALIGN, not just this one.

In Figure 35, the intended letter in *P* (i.e., *P*[*target*]) is determined by using the position value at *IS*[*i*]. This value, which in DETERMINE-ERRORS is stored in the variable *v*, is added to $|M| - |I|$ to determine the target in *P*. The set *M* stores the position values of letters that precipitate corrected omissions, and the set *I* stores position values of letters that are corrected insertions. When a backspace ('<') is processed, the sets *M* and *I* are inspected to see whether they contain the position value of the backspace, that is, to see if an omission or insertion was just corrected. If the value *v* is found, it is removed from *M* or *I* accordingly. This, in turn, appropriately affects the determination of the target index, since target is calculated as $v + |M| - |I|$.

Figure 38 gives the output of DETERMINE-ERRORS for the example triplet from Figures 30 and 34. The output shows 18 results. This compares to only 9 results available from an analysis of just *P* and *T*. Thus, using the input stream has doubled the character-level error data available for analysis.

4.2 Character-Level Metrics

After processing data from an experiment, we have error tallies for each letter. We also have counts: How many times was each letter presented? Transcribed? Entered? Intended? Correct? Unrecognized? Note that only the first two of these counts are available from an analysis of just *P* and *T*.

```
DETERMINE-ERRORS(triplets)
 1  foreach aligned triplet (P, T, IS) in triplets do
 2    a ← 0
 3    for b ← 0 to |IS| – 1 do
 4      if T[b] = '-' then
 5        UNCORRECTED-OMISSION(P[b])
 6      else if IS-FLAGGED(IS[b]) or b = |IS| – 1 then
 7        M ← new integer set  // corrected omissions
 8        I ← new integer set  // corrected insertions
 9        for i ← a to b – 1 do  // iterate over a substring between flags
10          v ← GET-POSITION-VALUE(IS[i])
11          if IS[i] = '<' then
12            if CONTAINS(M, v) then REMOVE(M, v)
13            if CONTAINS(I, v) then REMOVE(I, v)
14          else if IS[i] ≠ '_' then
15            target ← LOOK-AHEAD(P, b, v + |M| – |I|, IS-LETTER)
16            if IS[i] = 'ø' then
17              if target ≥ |P| then NONREC-INSERTION('ø')
18              else NONREC-SUBSTITUTION(P[target], 'ø')
19            else  // IS[i] is a letter
20              next_P ← LOOK-AHEAD(P, target, 1, IS-LETTER)
21              prev_P ← LOOK-BEHIND(P, target, 1, IS-LETTER)
22              next_IS ← LOOK-AHEAD(IS, i, 1, IS-NOT('ø', '_'))
23              prev_IS ← LOOK-BEHIND(IS, i, 1, IS-NOT('_'))
24              if IS[i] = P[target] then
25                CORRECTED-NOERROR(IS[i])
26              else if target ≥ |P| or IS[next_IS] = P[target]
27                    or (IS[prev_IS] = IS[i] and IS[prev_IS] = P[prev_P]) then
28                CORRECTED-INSERTION(IS[i])
29                I ←+ v  // track this corrected insertion
30              else if IS[i] = P[next_P] and IS-LETTER(T[target]) then
31                CORRECTED-OMISSION(P[target])
32                CORRECTED-NOERROR(IS[i])
33                M ←+ v  // track this corrected omission
34              else CORRECTED-SUBSTITUTION(P[target], IS[i])
35        end for // i from a to b – 1
36        if P[b] = '-' then UNCORRECTED-INSERTION(T[b])
37        else if P[b] ≠ T[b] then UNCORRECTED-SUBSTITUTION(P[b], T[b])
38        else if P[b] ≠ '_' then UNCORRECTED-NOERROR(T[b])
39        else if IS[b] = 'ø' then NONREC-INSERTION('ø')
40        a ← b + 1  // a starts the next substring
```

Fig. 35. Algorithm for classifying errors in the input stream. This algorithm finds all errors from previous work and the new input stream error types described in Section 3.1.

4.2.1 *Three Error Rates.* Each character has three separate error rates: uncorrected, corrected, and total. These are defined as follows for a given character i:

$$Uncorrected\ Error\ Rate_i = 1 - \frac{Uncorrected\ NoErrors_i}{Transcribed_i} \tag{1}$$

$$Corrected\ Error\ Rate_i = 1 - \frac{Corrected\ NoErrors_i}{Entered_i - Transcribed_i} \tag{2}$$

and

$$Total\ Error\ Rate_i = 1 - \frac{Uncorrected\ NoErrors_i + Corrected\ NoErrors_i}{Entered_i} \tag{3}$$

```
LOOK-AHEAD(S, start, count, CONDITION-FN)
 1  index ← start
 2  while 0 ≤ index < |S| and not CONDITION-FN(S[index]) do
 3     index ← index + 1  // proceed until the condition is met
 4  while count > 0 and index < |S| do
 5     index ← index + 1
 6     if index = |S| then break
 7     else if CONDITION-FN(S[index]) then
 8        count ← count – 1
 9  return index
```

Fig. 36. Procedure used by DETERMINE-ERRORS (Figure 35) that looks forward in string S from a zero-based index *start* until a count number of CONDITION-FNs have been satisfied, returning the index of the countth successful test.

```
LOOK-BEHIND(S, start, count, CONDITION-FN)
 1  index ← start
 2  while 0 ≤ index < |S| and not CONDITION-FN(S[index]) do
 3     index ← index – 1  // go back until the condition is met
 4  while count > 0 and index ≥ 0 do
 5     index ← index – 1
 6     if index < 0 then break
 7     else if CONDITION-FN(S[index]) then
 8        count ← count – 1
 9  return index
```

Fig. 37. Procedure used by DETERMINE-ERRORS (Figure 35) that is analogous to LOOK-AHEAD, but operates in the reverse direction.

```
 P4: ____qui__c___--____kly__
 T4: ____qu-__c___eh____kly__
IS4: pv<<qu_c<cøk<ehly<<klyz<

1.  corrected substitution (q, p)
2.  corrected substitution (u, v)
3.  uncorrected no-error (q, q)
4.  uncorrected no-error (u, u)
5.  uncorrected omission (i, -)
6.  corrected no-error (c, c)
7.  uncorrected no-error (c, c)
8.  non-recognition substitution (k, ø)
9.  corrected no-error(k, k)
10. uncorrected insertion (-, e)
11. uncorrected insertion (-, h)
12. corrected omission (k, -)
13. corrected no-error (l, l)
14. corrected no-error (y, y)
15. uncorrected no-error(k, k)
16. uncorrected no-error (l, l)
17. uncorrected no-error (y, y)
18. corrected insertion (-, z)
```

Fig. 38. Classification output for the triplet from Figure 30. Each line can be read as `classification` (*intended, produced*).

Eq. (1) answers the question, "of the i's remaining in the transcription, what percent were erroneous?" Eq. (2) answers, "of the erased i's, what percent were erroneous?" Eq. (3) answers, "of all entered i's, what percent were erroneous?" Note that a high rate for Eq.(2) means that most of the backspaced i's were, in fact, errors and should ideally have been corrected. Conversely, a low rate for Eq. (2) means that subjects were erasing already correct i's, which might indicate that they were too hasty to backspace.

The total error rate for a letter (Eq. (3)) refers only to actual entries of this letter. It says, "given that an i was entered, what are the chances that i was correct?" Omissions of i are therefore not captured by Eq. (3). Instead, omissions are handled by a separate error rate, described in Section 4.2.3.

4.2.2 *Substitutions versus Intentions.* Our data allows us to answer the question, "what is the probability of getting i when trying for i?" For this, we take the ratio of substitutions to intentions. The number of times a letter was intended is obtained by adding its substitutions and no-errors. These five error types (uncorrected, corrected, and nonrecognition substitutions; and uncorrected and corrected no-errors) have intended target letters and actual produced letters (or nonrecognitions).

$$Uncorrected\ Substitution\ Rate_i = \frac{Uncorrected\ Substitutions_i}{Intended_i} \tag{4}$$

$$Corrected\ Substitution\ Rate_i = \frac{Corrected\ Substitutions_i}{Intended_i} \tag{5}$$

$$Nonrecognition\ Substitution\ Rate_i = \frac{Nonrecognition\ Substitutions_i}{Intended_i} \tag{6}$$

and

$$Total\ Substitution\ Rate_i = \frac{\begin{array}{c}Uncorrected\ Substitutions_i + \\ Corrected\ Substitutions_i + \\ Nonrecognition\ Substitutions_i\end{array}}{Intended_i} \tag{7}$$

Eq. (4) answers the question, "when trying for i, what is the probability that we produce an uncorrected substitution for i?" Similar questions can be asked of corrected substitutions (Eq. (5)) and nonrecognition substitutions (Eq. (6)). Eq. (7) answers, "when trying for i, what is the probability that we don't get i?"

4.2.3 *Omissions versus Presentations.* We can also determine whether some letters are prone to omission. For this, we take the ratio of omissions to presentations—the number of times a letter was omitted compared to the number of times it was presented to the subject for transcription.

$$Uncorrected\ Omission\ Rate_i = \frac{Uncorrected\ Omissions_i}{Presented_i} \tag{8}$$

$$Corrected\ Omission\ Rate_i = \frac{Corrected\ Omissions_i}{Presented_i} \tag{9}$$

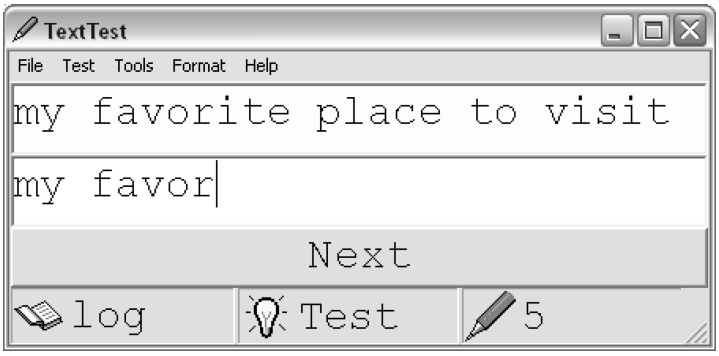

Fig. 39. The TextTest text entry evaluation program. This program writes XML log files that can be analyzed by StreamAnalyzer (Figure 40).

and

$$\textit{Total Omission Rate}_i = \frac{\textit{Uncorrected Omissions}_i + \textit{Corrected Omissions}_i}{\textit{Presented}_i} \quad (10)$$

Eqs. (9) and (10) are special in that they can be over 100% if a single presented letter is omitted repeatedly. Although this is a theoretical possibility, subjects are unlikely to exhibit this behavior.

4.2.4 *Insertions versus Entries.* We can also discover whether a letter i is prone to insertion. For this, we take the ratio of i's insertions to i's entries.

$$\textit{Uncorrected Insertion Rate}_i = \frac{\textit{Uncorrected Insertions}_i}{\textit{Entered}_i} \quad (11)$$

$$\textit{Corrected Insertion Rate}_i = \frac{\textit{Corrected Insertions}_i}{\textit{Entered}_i} \quad (12)$$

and

$$\textit{Total Insertion Rate}_i = \frac{\textit{Uncorrected Insertions}_i + \textit{Corrected Insertions}_i}{\textit{Entered}_i} \quad (13)$$

5. USER TEST SOFTWARE AND LOG FILE ANALYZER

To facilitate the automation of the analyses described in this article, we built two complementary applications. The first is TextTest (Figure 39), a program designed for conducting text entry evaluations. It runs on Microsoft Windows systems with any text entry method, provided that the method sends characters through the low-level keyboard input stream. For example, the SendInput() and keybd_event() Win32 functions or the SendKeys methods in Visual C# or Visual Basic .NET do this on Windows systems. TextTest can also send or receive characters over TCP for conducting studies on mobile devices.

Recognition-based text entry methods can send special nonprinting character codes to TextTest for explicitly logging character starts, character ends, and nonrecognitions. TextTest randomly presents phrases from a published corpus of 500 [MacKenzie and Soukoreff 2003], or from any custom phrase set, and

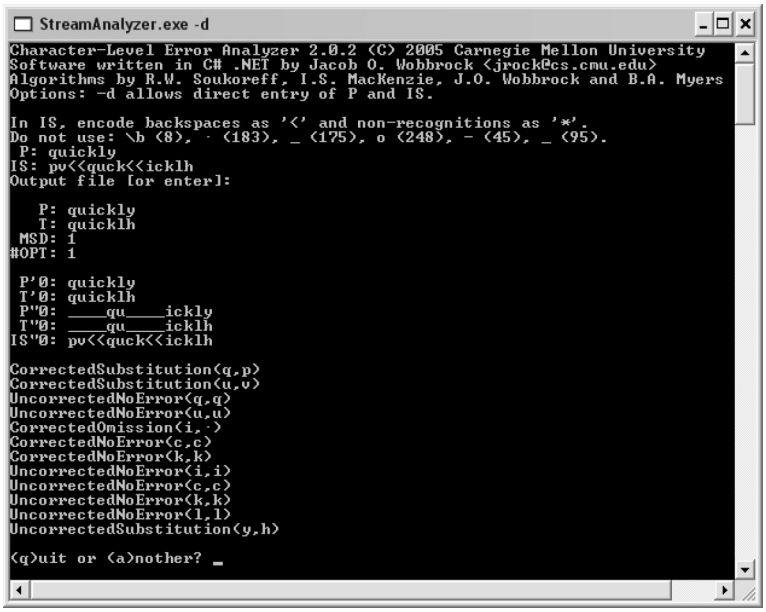

Fig. 40. The StreamAnalyzer log file analyzer. This program reads the XML logs written by TextTest (Figure 39) and computes various measures. Here, the program is being used to analyze input given directly on the console.

writes XML log files for easy parsing. TextTest itself is implemented in Visual C#.

Accompanying TextTest is a log file analyzer named StreamAnalyzer (Figure 40) that parses TextTest's XML logs and computes various measures of text entry performance. Alternatively, when run with the "-d" switch, StreamAnalyzer can be used to analyze any P and IS given directly from the console. StreamAnalyzer performs all of the analyses described in this article and in articles on which this work builds. It writes its output to a text file, which can be pasted into a spreadsheet for statistical analysis. Like TextTest, the analyzer is written in Visual C#.

6. COMPARISON OF ANALYSIS TECHNIQUES

To illustrate the advantages of analyzing input streams over merely analyzing transcribed strings, both analysis methods were used on text entry data from a real experiment in which five first-time novices entered a total of 75 sentences for 3,750 presented characters. They made 4,932 attempts for 4,660 produced characters and 272 nonrecognitions. Of the characters, 4,207 were letters (alphanumerics or spaces) and 453 were backspaces. The technique under investigation was the unistroke method EdgeWrite using a stylus on a PDA [Wobbrock et al. 2003].

6.1 Error Rate Tables

Character-level probability tables show error rates for each letter [MacKenzie and Soukoreff 2002b]. The prior analysis of just P and T enables the production

Table I. Character-Level Results

Character	Presented	Transcribed	Insertion	Substitution	Omission	Sum
'r'	145	144	0.0%	1.1%	1.0%	2.1%
'c'	60	59	0.0%	1.7%	0.0%	1.7%
'b'	50	50	0.0%	0.7%	0.6%	1.3%
'e'	325	327	0.0%	0.0%	0.0%	0.0%
Total	3750	3750	0.2% Avg.	0.2%	0.1%	0.5%

An excerpt of character-level results using a prior analysis of transcribed strings [MacKenzie and Soukoreff 2002b]. The bottom row is for the whole table, not just for the excerpt shown here.

Table II. Character-Level Results Using the Current Analysis

Character	Entered	Intended	Correct	Insertion	Substitution	Omission	Sum
'r'	14	7	5	0.0%	28.5%	0.0%	28.6%
'f'	80	46	41	0.0%	10.9%	2.5%	13.4%
'j'	15	11	10	6.7%	9.1%	0.0%	15.8%
'e'	360	344	330	1.4%	4.0%	0.0%	5.4%
Total	4479	4446.83	3808.17	0.7% Avg.	14.4%	0.8%	15.9%

An excerpt of character-level results using the current analysis. The bottom row is for the whole table, not just for the excerpt shown here.

of tables that list error rates for uncorrected insertions, substitutions, and omissions. Space precludes a full table here, so Table I shows the letters with the three highest error rates and the letter "e" for comparison.

Note the relatively few total uncorrected errors examinable by this analysis (0.5%). In fact, only 8 of the 37 characters (*a*–*z*, 0–9, space) in the full table have nonzero error rates. Perhaps subjects were careful to correct errors, or perhaps they did not make many errors to begin with. With this analysis, there is no way to know.

A much bigger table results from analyses of input streams. The table has a row for each letter and a column for each count and character-level measure. This table, which is automatically generated by StreamAnalyzer (Figure 40), reports that "e" was intended 344 times, entered 360 times, correct 330 times, and unrecognized 8 times. The chance of getting another character when intending an "e" was 4.0%. There were no omissions of "e" and 5 insertions of "e", or 1.4% of all entered "e"s. These are the types of results available from our character-level analysis of input streams.

Table II is an excerpt from the full table, containing only a subset of the full table's rows and columns. It shows the letters with the three highest error rates and the letter "e" for comparison. The results in Table II differ substantially from those in Table I because of the inclusion of corrected errors. Overall, an attempt to make a letter had a 14.4% chance of being a substitution, not just a 0.2% chance, as reported in Table I. This rate is high because the test subjects were first-time users of the technique under investigation.

6.2 Confusion Matrices

The error rate tables tell us which characters are prone to different types of substitution errors. But they do not tell us what these substitution errors are. For this, we use a confusion matrix [Grudin 1984; MacKenzie and Soukoreff

2002b]. This is similar, but not identical to, the confusion matrices used in handwriting recognition. Those confusion matrices indicate where a gesture recognizer has trouble differentiating between written characters. By contrast, the confusion matrices we refer to show where the *user* has intended some letter but produced another. This may indeed be due to a poor recognizer, or may be due to user confusion, forgetting how to make a letter, or guessing incorrectly. For example, in Graffiti, "k" and "x" are mirror images. Although the Graffiti recognizer has no trouble distinguishing between them, novices often mistake one for the other [MacKenzie and Zhang 1997].

In the confusion matrix, the x-axis is the *intended* character and the y-axis is the *produced* character. For a character, "total attempts" is the sum of its matrix column and "total entries" (excluding insertions) is the sum of its matrix row.

Values along the diagonal (where the intended and produced characters match) represent correct entries and dwarf the values off the diagonal. We therefore omit values along the diagonal so that the substitution errors are more visible in Figure 41.

Figure 41(a) has a maximum value of just 1 off of its diagonal. Figure 41(b) has a maximum value of 18 for alphabetic substitutions (u, l) and 32 for nonrecognition substitutions $(m, \emptyset)$.

One unexpected finding from Figure 41(b) that is not available in Figure 41(a) is the high number of "l"s produced when trying for "u"s, since $(u, l) = 18$. In EdgeWrite, if subjects lift their stylus prematurely while trying for a "u", an "l" can result, and apparently this happened.

Another unexpected finding was the high nonrecognition rate for "n", since $(n, \emptyset) = 28$. The reason for this appears to be that in making the diagonal portion of "n", subjects sometimes caught the bottom-left corner of the input area accidentally, which resulted in a nonrecognition since that corner sequence, which EdgeWrite uses to recognize strokes, is not defined. A subsequent redesign of the corner regions changed them from rectangles to triangles and remedied this problem [Wobbrock et al. 2003].

Another interesting character-level finding was the high substitution rate of "i" for "l", since $(l, i) = 16$. This tells us that when subjects were trying to produce an "l", they often thought of a lowercase "l" shape, which is a line straight down. Such a stroke is an "i" in EdgeWrite, whereas an "l" is a capital "L" stroke down and then across.

Yet another finding of interest was the confusion of "f" for "t", since $(t, f) = 15$. The strokes for "t" and "f" are mirror images of each other. Apparently, this was confusing to subjects when they intended to write a "t". However, the reverse was not the case, since $(f, t) = 1$, meaning that when trying for "f", subjects were not confusing it with the shape of "t". Incorporating linguistic information in the form of letter digraph probabilities, for example, might be one way to remedy this confusion by allowing the "f" stroke to enter "t" if the previous character indicates that "t" is much more likely than "f".

These results have influenced the redesign of EdgeWrite and its accompanying instructional material. Without a rich character-level analysis, the aforementioned insights (and others) would have been forfeit and fine-grain improvements would have been more difficult to make.

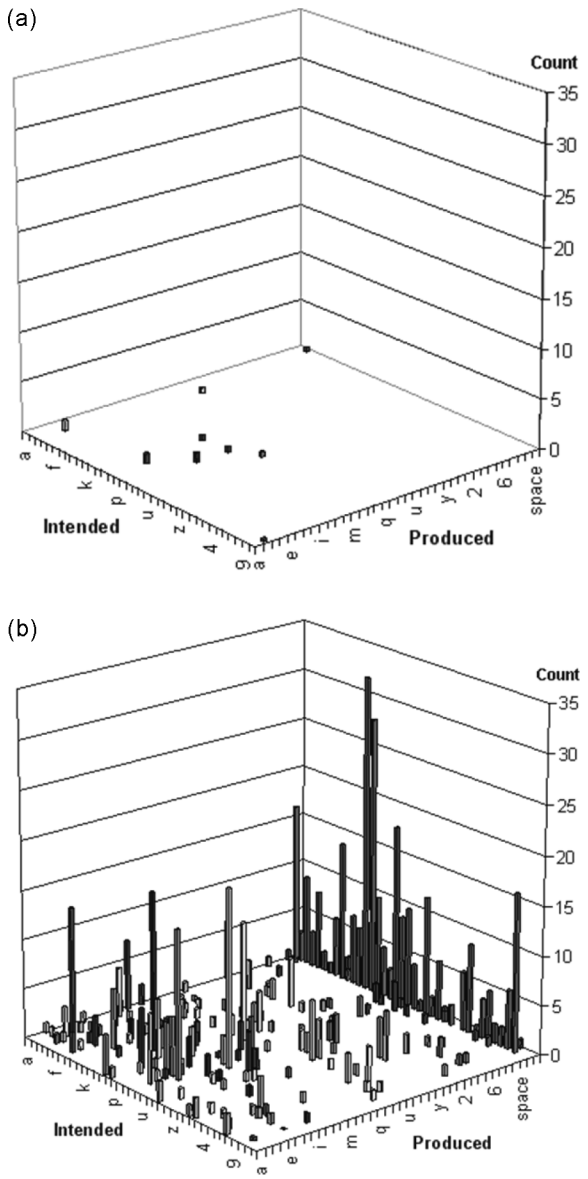

Fig. 41. (a) Confusion matrices from the previous analysis of P and T and (b) from the current analysis of P, T, and IS for the same empirical data. These matrices are automatically produced by the StreamAnalyzer program. The large discrepancies in the two graphs are due to corrected substitutions, since these appear in IS, but not in T. The high values against the back wall in (b) are nonrecognition substitutions, which are also not captured in (a).

7. FUTURE WORK

Future work includes adding algorithmic extensions to further relax or remove the assumptions used to resolve ambiguity in the input stream. Such extensions may involve clustering or data mining to assess errors in a complementary way. A wider review of data from other text entry experiments would help inform this.

Although this work helps designers understand which character-level errors happen, it offers little help in determining *why* they happen, since this determination is method-dependent. For stroke-based methods, a graphical error report coupling stroke traces with the errors they produced could facilitate iteration and redesign. The algorithms presented here would still be required, however, to show the designer where to look. Method-specific extensions could also be made to the input stream, such as augmenting input streams with mode setting actions (e.g., the capitalization up-stroke in Graffiti, the SHIFT key on keyboards, or the multiple key presses in Multitap).

Yet another avenue for future work is to enable TextTest and Stream-Analyzer to produce results in real-time during a text entry study. As they stand, these tools support the running of a text entry study first and then later analysis of the log files. Analyzing the logs during a study would provide immediate feedback to the experimenter, which could be useful in motivating subjects or for practicing until a certain level of proficiency is reached.

8. CONCLUSION

The unconstrained text entry evaluation paradigm has changed the way rigorous text entry experiments can be conducted. However, with the advent of this paradigm comes a major challenge in capturing character-level errors, particularly in the input stream, where such errors are most prevalent. Although they were artificial and often frustrating for users, former constrained text entry evaluation paradigms made assessing character-level errors trivial, making the use of the new unconstrained paradigm a tradeoff, rather than an outright win, over older, more artificial paradigms.

We have presented a technique and tools for the analysis of character-level errors in the input stream that is independent of the character-level text entry method under investigation. We have shown the value of performing character-level error analyses on input streams, rather than just on transcribed strings. Despite the inherent ambiguity in input streams, we are able to extract error information using four testable assumptions. Our taxonomy of ten input stream error types provides a means of describing character-level errors in a well defined fashion at a fine-grained level. The current measures, the algorithms that automate them, and the software that implements them can aid designers and evaluators of text entry methods by providing more character-level error data. In short, we have brought the unconstrained experimental paradigm one step closer to having the comprehensive description of text entry behavior eagerly sought by text entry researchers.

ACKNOWLEDGMENTS

The authors thank R.W. Soukoreff for comments on earlier drafts and I.S. MacKenzie for permission to include MSD-MATRIX and ALIGN.

REFERENCES

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. The MIT Press. Cambridge, MA.

DAMERAU, F. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM 7*, 171–176.

EVREINOVA, T., EVREINOV, G., AND RAISAMO, R. 2004. Four-Key text entry for physically challenged people. *Adjunct Proceedings of the 8th ERCIM Workshop on User Interfaces for All (UI4ALL'04)*. Vienna, Austria (June 28–29).

GENTNER, D. R., GRUDIN, J. T., LAROCHELLE, S., NORMAN, D. A., AND RUMELHART, D. E. 1984. A glossary of terms including a classification of typing errors. In *Cognitive Aspects of Skilled Typewriting*, W. E. Cooper Ed. Springer Verlag, 39–43.

GONG, J. AND TARASEWICH, P. 2005. Alphabetically constrained keypad designs for text entry on mobile devices. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Portland, OR (Apr. 2–7). 211–220.

GRUDIN, J. T. 1984. Error patterns in skilled and novice transcription typing. In *Cognitive Aspects of Skilled Typewriting*, W. E. Cooper Ed. Springer Verlag, 121–143.

GSM WORLD. 2004. *The Netsize Guide*. http://www.gsmworld.com.

INGMARSSON, M., DINKA, D., AND ZHAI, S. 2004. TNT—A numeric keypad-based text input method. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Vienna, Austria (Apr. 24–29). 639–646.

ISOKOSKI, P. AND KAKI, M. 2002. Comparison of two touchpad-based methods for numeric entry. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Minneapolis, MN (Apr. 20–25). 25–32.

JEFFRIES, R., MILLER, J. R., WHARTON, C., AND UYEDA, K. M. 1991. User interface evaluation in the real world: A comparison of four techniques. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. New Orleans, LA (Apr. 27–May 2). 119–124.

KARAT, C.-M., HALVERSON, C., HORN, D., AND KARAT, J. 1999. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. Pittsburgh, PA (May 15–20). 568–575.

KARP, R. M., MILLER, R. E., AND ROSENBERG, A. L. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the ACM Symposium on Theory of Computing*. Denver, CO (May 1–3). 125–136.

KÖLTRINGER, T. AND GRECHENIG, T. 2004. Comparing the immediate usability of Graffiti 2 and virtual keyboard. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI)*. Vienna, Austria (Apr. 24–29). 1175–1178.

LANDRAUD, A. M., AVRIL, J.-F., AND CHRETIENNE, P. 1989. An algorithm for finding a common structure shared by a family of strings. *IEEE Trans. Pattern Anal. Mach. Intell. 11*, 8, 890–895.

LEVENSHTEIN, V. I. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR 163*, 4, 845–848.

LEWIS, J. R. 1999. Input rates and user preference for three small-screen input methods: Standard keyboard, predictive keyboard, and handwriting. In *Proceedings of the Human Factors and Ergonomics Society 43rd Annual Meeting*. Houston, TX (Sept. 27–Oct. 1). Santa Monica, CA. 425–428.

LEWIS, J. R., LALOMIA, M. J., AND KENNEDY, P. J. 1999. Evaluation of typing key layouts for stylus input. In *Proceedings of the Human Factors and Ergonomics Society 43rd Annual Meeting*. Houston, TX (Sept. 27–Oct. 1). Santa Monica, CA. 420–424.

MACKENZIE, I. S. AND ZHANG, S. X. 1997. The immediate usability of Graffiti. In *Proceedings of the Graphics Interface Conference*. Kelowna, BC (May 21–23). Toronto, ON. 129–137.

MACKENZIE, I. S. AND ZHANG, S. X. 1999. The design and evaluation of a high-performance soft keyboard. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Pittsburgh, PA (May 15–20). 25–31.

MacKenzie, I. S., Zhang, S. X., and Soukoreff, R. W. 1999. Text entry using soft keyboards. *J. Behav. Inf. Technol. 18*, 4, 235–244.

MacKenzie, I. S. and Soukoreff, R. W. 2002a. Text entry for mobile computing: Models and methods, theory and practice. *Human-Comput. Interact. 17*, 2, 147–198.

MacKenzie, I. S. and Soukoreff, R. W. 2002b. A character-level error analysis technique for evaluating text entry methods. In *Proceedings of the 2nd Nordic Conference on Human-Computer Interaction (NordiCHI)*. Århus, Denmark (Oct. 19–23). 243–246.

MacKenzie, I. S. and Soukoreff, R. W. 2003. Phrase sets for evaluating text entry techniques. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI)*. Ft. Lauderdale, FL (Apr. 5–10). 754–755.

Matias, E., MacKenzie, I. S., and Buxton, W. 1996. One-Handed touch-typing on a QWERTY keyboard. *Human-Comput. Interact. 11*, 1, 1–27.

Morgan, H. L. 1970. Spelling correction in systems programs. *Commun. ACM 13*, 2, 90–94.

Palm, Inc. 1995. *The Graffiti Alphabet*. http://www.palm.com/us/products/input/Palm_Graffiti.pdf

Rodriguez, N. J., Borges, J. A., and Acosta, N. 2005. A study of text and numeric input modalities on PDAs. In *Proceedings of the 11th International Conference on Human-Computer Interaction (HCI Int'l)*. Las Vegas, NV (July 22–27).

Sankoff, D. and Kruskal, J. B. 1983. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA.

Sears, A. and Zha, Y. 2003. Data entry for mobile devices using soft keyboards: Understanding the effects of keyboard size and user tasks. *Int. J. Human-Comput. Interact. 16*, 2, 163–184.

Soukoreff, R. W. and MacKenzie, I. S. 2001. Measuring errors in text entry tasks: An application of the Levenshtein string distance statistic. In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI)*. Seattle, WA (Mar. 31–Apr. 5). 319–320.

Soukoreff, R. W. and MacKenzie, I. S. 2003. Metrics for text entry research: An evaluation of MSD and KSPC, and a new unified error metric. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Ft. Lauderdale, FL (Apr. 5–10). 113–120.

Soukoreff, R. W. and MacKenzie, I. S. 2004. Recent developments in text-entry error rate measurement. In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI)*. Vienna, Austria (Apr. 24–29). 1425–1428.

Venolia, D. and Neiberg, F. 1994. T-Cube: A fast, self-disclosing pen-based alphabet. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '94)*. Boston, MA (Apr. 24–28). 265–270.

Wagner, R. A. and Fischer, M. J. 1974. The string-to-string correction problem. *J. ACM 21*, 1, 168–173.

Waterman, M. S. 1984. General methods of sequence comparison. *Bull. Math. Biol. 46*, 4, 473–500.

Wobbrock, J. O., Myers, B. A., and Kembel, J. A. 2003. EdgeWrite: A stylus-based text entry method designed for high accuracy and stability of motion. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. Vancouver, BC (Nov. 2–5). 61–70.

Wobbrock, J. O., Myers, B. A., and Aung, H. H. 2004. Writing with a joystick: A comparison of date stamp, selection keyboard, and EdgeWrite. In *Proceedings of the Graphics Interface Conference*. London, ON (May 17–19). 1–8.

Wobbrock, J. O., Myers, B. A., and Chau, D. H. 2006. In-stroke word completion. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. Montreux, Switzerland (Oct. 15–18). 333–336.

Zhai, S. and Kristensson, P. 2003. Shorthand writing on stylus keyboard. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Ft. Lauderdale, FL (Apr. 5–10). 97–104.

Zhai, S., Kristensson, P., and Smith, B. A. 2005. In search of effective text input interfaces for off the desktop computing. *Interact. Comput. 17*, 3, 229–250.

Received January 2005; revised December 2005; accepted April 2006