CueTIP: A Mixed-Initiative Interface for Correcting Handwriting Errors

Michael Shilman, Desney S. Tan, Patrice Simard
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
{shilman, desney, patrice}@microsoft.com

ABSTRACT
With advances in pen-based computing devices, handwriting has become an increasingly popular input modality. Researchers have put considerable effort into building intelligent recognition systems that can translate handwriting to text with increasing accuracy. However, handwritten input is inherently ambiguous, and these systems will always make errors. Unfortunately, work on error recovery mechanisms has mainly focused on interface innovations that allow users to manually transform the erroneous recognition result into the intended one. In our work, we propose a mixed-initiative approach to error correction. We describe CueTIP, a novel correction interface that takes advantage of the recognizer to continually evolve its results using the additional information from user corrections. This significantly reduces the number of actions required to reach the intended result. We present a user study showing that CueTIP is more efficient and better preferred for correcting handwriting recognition errors. Grounded in the discussion of CueTIP, we also present design principles that may be applied to mixed-initiative correction interfaces in other domains.

ACM Classification: H.5.2 [Information Interfaces and Presentation]: User Interfaces - Graphical user interfaces, Input Devices and Strategies, Interaction styles, User-centered design; I.7.m [Document and Text Processing]: Graphics recognition and interpretation.

General terms: Design, Human Factors, Performance.

Keywords: Correction interface, mixed initiative, handwriting recognition, constraints, user study.

INTRODUCTION
The study of handwriting has been a topic of research in many fields, including psychology, neuroscience, physics, computer science, anthropology, education, forensic documentation, and others [11]. Much of this work has led to significant advances in recognition systems that automatically transform handwritten language into its symbolic representation. Because of the inherent ambiguities that exist in human handwriting, we believe that these systems will always make recognition errors. While many researchers continue to work on improving recognizer accuracy, there has been much less work exploring recovery mechanisms with which users can correct errors once they are made. Within this work, researchers have made heuristic and interface innovations that allow users to more efficiently manually correct text [e.g. 4,7]. Unfortunately, because correction has traditionally been viewed as an editing task, these systems often require laborious corrections that derail the flow of the text input task.

In this paper, we focus on exploring a mixed-initiative approach that reduces the cost of correction. Rather than treating correction as a manual editing task to be performed solely by the user, we construct a system that continues to assist the user even as they make corrections (see Figure 2 for the augmented correction flow). We characterize recognition as an optimization process and show that by treating user corrections as constraints to this optimization, the system can continue to 'steer' its guesses and attain the right answer in many fewer steps than would be possible with traditional alternatives (see Figure 1). This allows us not only to accelerate the correction process, but also to streamline the interface and interaction.

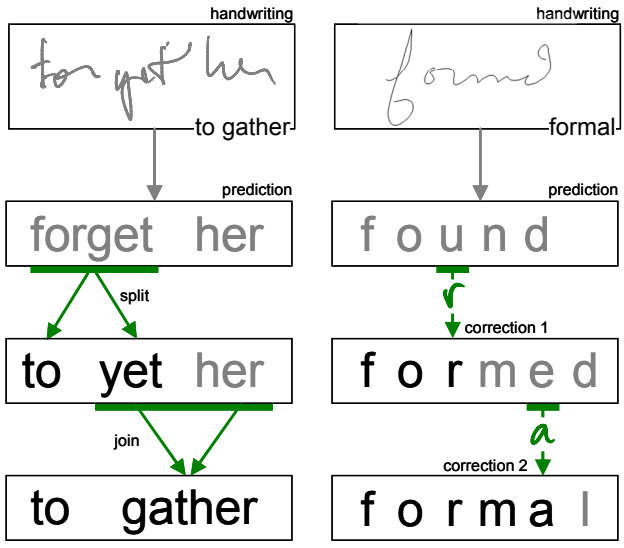


Figure 1: User handwrites some text and in order to attain the intended phrase, has to (left) split a word and then join two; (right) correct two letters. Using standard techniques to correct these same corrections would take many more operations.

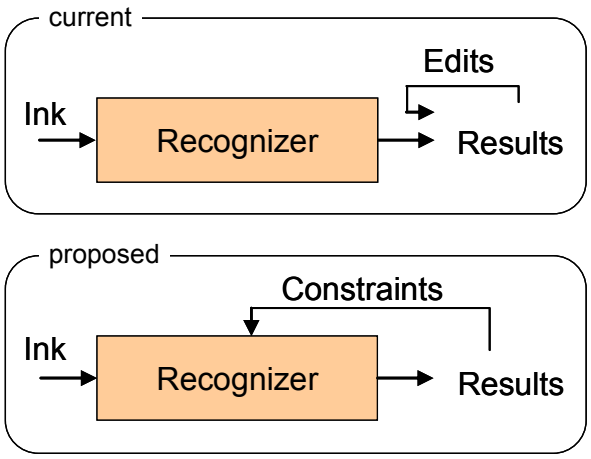


Figure 2: (top) While current methods allow users to make manual edits to fix errors; (bottom) we propose leveraging the intelligent recognizer to further assist users as they do this.

The contributions of this paper are threefold. First, we present CueTIP, a novel mixed-initiative system that dramatically improves handwriting error correction. In this system, the recognizer updates its recognition results as a function both of the original handwriting, but also of the implicit constraints added by each new correction the user makes. Second, situated in our discussion of the CueTIP interface and implementation, we describe general design principles for correction interfaces. These principles may be applied domains that generate ambiguous interpretations of user intent, including speech, optical character recognition, and search. Finally, we report a user study we conducted, suggesting that CueTIP is indeed more effective and better preferred over a traditional correction interface.

BACKGROUND AND RELATED WORD

Handwriting Recognition

Recognition is the interpretation of ambiguous sensor measurements into hypotheses about user action or intent. Handwriting recognition techniques generally fall into one of two categories, structural and rule-based methods, or statistical classification methods [14]. In structural and rule-based methods, the recognizer leverages the idea that character shape may be described in an abstract manner, independent of irrelevant variations that occur during execution. Because finding reliable abstract rules is generally very difficult, researchers have recently shifted much of their efforts to exploring the second category, statistical classification methods. In these methods, the system represents shape by features on the ink that define a multidimensional space, and creates probability distributions that map points in this space onto discrete classes.

Many such recognition systems employ some variant of *likelihood maximization*, in which the recognizer outputs the hypothesis that is *most likely* given the observed inputs [5]. In likelihood maximization, the goodness of a hypothesis given the input is expressed with a scoring function, such as probability. Under this formulation, the recognition process is implemented as an optimization over the output space with respect to the likelihood scoring function. Many researchers are working on better scoring metrics, faster or more accurate optimizations, or techniques that do both at the same time [5] in order to improve recognition rates.

Improving recognition rates and decreasing errors is extremely important because error recovery remains an expensive and frustrating process for the user. In doing this, researchers have attempted to characterize errors so that they can specifically address them. For example, Schomaker described errors as stemming from discrete noises, badly formed shapes, input legible by humans but not recognizers, badly spelled words, cancelled material, and device generated errors [12]. Other researchers have proposed various schemes in order to address these errors. For example, Oviatt proposes multimodal interfaces both for the recognition task as well as the correction [9]. In such interfaces, the system would take recognition hypotheses from one modality (e.g. speech) in order to constrain or reorder recognition results from another (e.g. handwriting). The hope is that the complementary information will make recognition more accurate. For more complete surveys of recognition techniques, see [11,14].

Regardless of the recognition technique employed, we do not believe these systems will ever completely eliminate errors in interpreting user intent given a set of pen strokes. Furthermore, small recognition errors often cascade to much larger ones, each of which the user has to tediously correct. In fact, even correction gestures are misrecognized, leading to what Oviatt and VanGent term error spirals, or series of multiple attempts to correct the same error [10]. In our work, we focus on designing interfaces that allow users to efficiently correct errors once they have been made.

Error Correction Interfaces

Error correction is not a new topic. In fact, researchers have explored error correction mechanisms for misrecognized handwriting within several different domains. Early work by Goldberg and Goodisman [1] tested a system that allowed users to correct characters by tapping on options from an n-best list. They found in informal observations that searching for the appropriate correction was too expensive and that users preferred to just tap on the incorrect character repeatedly to cycle through the list. Alternatively, if the list was not accurate enough for the user to get to the correct option quickly, they preferred to simply write over the error and have the recognizer try again.

Smithies, Novins, and Arvo worked on correcting recognition errors with handwritten mathematical equations [13]. They provided a modal correction mechanism for stroke segmentation that allowed users to draw a line over strokes in order to indicate that they should be recognized as a single symbol. They used this mechanism both for joining strokes as well as splitting them. Users could also click on individual characters and choose the correct one from an n-best list. If the list did not contain the correct option, the user would fall back onto using a soft keyboard.

In their work, Mankoff, Hudson, and Abowd provide a survey of correction techniques, which they classify into repetition and choice [7]. They present OOPS, a toolkit that supports resolution of input ambiguity through mechanisms that allow the user to specify correct interpretation of their input. Among other techniques, they demonstrate the utility of their toolkit with a new interaction that coupled word prediction with online handwriting recognition. While their prediction mechanisms were built into the process of writing (much like T9 Text Input® does with mobile text input), we explore similar prediction ideas while the user is actually making the correction. Also, we believe that there are other classes of corrections such as grouping strokes that are not adequately covered in the repetition/choice dichotomy. Rather than characterizing the mechanisms by which corrections are typically made, Huerst, Wang, and Wiabel classified error repair into overwriting, deletion, completion, and insertion [4]. In this paper, we extend this body of work by presenting a mixed-initiative approach to correction interfaces as well as principles that should be considered in designing them.

Mixed-Initiative Correction Interfaces

While recognition engines have gotten more 'intelligent,' many of the correction mechanisms we have discussed so far have involved interface innovations made completely independently of recognizers. These interfaces are designed so that users can fully (re)specify their intent as efficiently as possible. We believe that there is potential to significantly improve these correction mechanisms by better leveraging the recognizer. Specifically, we propose that the system should continue to assist the user even as corrections are made, to reduce the number of actions required to reach the intended result. Such a system builds upon many of the general principles derived for mixed-initiative interfaces, which propose coupling automated services with direct user manipulation. Horvitz, for example, presents a set of eleven principles, which includes developing value-added automation, inferring ideal action in light of costs, benefits, and uncertainties, as well as employing dialog to resolve uncertainty, among others [2].

In line with these principles, Kristiansson, Culotta, Viola, and McCallum introduce the concept of *correction propagation*, which proposes that after each user correction action, the remaining recognition should be re-recognized in order to yield the best results constrained to match the current set of corrections [6]. They apply this to the use of conditional random fields as a method of performing automatic form filling. As the user corrects mistakes, the system continues to make changes to other fields in response to the actual corrected one. The authors show, using a simulated quantitative calculation that the number of actions and, by inference, the effort, that would be required by the user is drastically reduced with such a system. This positive result is not surprising given the heavy constraints that the restricted form schema places on the possible outputs. In this paper, we build upon this work by extending it to the relatively unconstrained problem of handwriting correction. We present a novel handwriting correction interface, which we call CueTIP, as well as general principles for thinking about mixed-initiative correction interfaces. We also conduct a study to examine how the simulated expectations compare to actual user performance.

CUETIP INTERFACE

The CueTIP system provides an intuitive mixed-initiative interface that allows users to efficiently correct errors made by the handwriting recognizer. The interface itself consists of several graphical components (Figure 3). CueTIP provides a handwriting entry box in which the user writes. Below the entry box is the results panel, the region in which recognition results are presented. To reduce the number of modes a user has to transition between to make corrections, all corrections are made directly on the results.

When a user writes into the entry box, the recognizer builds its prediction of the text and updates this in the results panel. This is identical to what most current systems do. We augment this by drawing a purple result bar above each recognized word in the panel and aligning this to the ink in the textbox. This allows users to quickly match the set of handwriting strokes that led to a particular interpretation in order to help with correction.

If the user finds any errors, they can correct the text directly in the results pane. The correction operations are simple (see rest of Figure 3). First, the user can correct segmentation errors, or errors the recognizer has made in determining word breaks, by using simple gestures on the results. To join two or more words, the user simple draws a

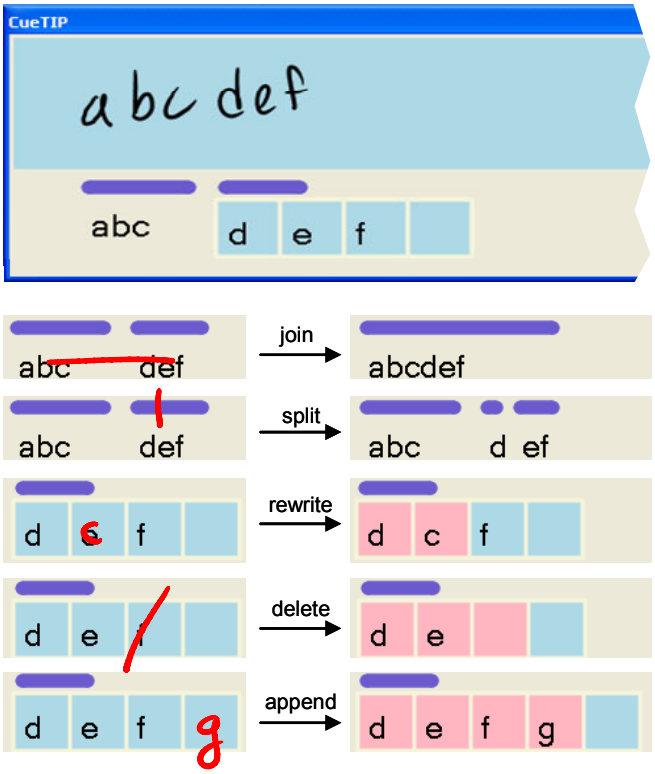


Figure 3: (top) The CueTIP interface. (rest) The full set of correction operations available to the user.

horizontal line connecting the words together. To split a word in two, the user draws a vertical line through the purple result bar. The location of the split is approximately the projection of this line up into the ink in the entry box. We did this because we thought that it would be easiest for users to determine where word breaks should occur with respect to the ink, rather than on the recognized text, which sometimes does not resemble the intended result.

When a user performs either of these operations, the recognizer is re-invoked with the original handwriting as well as any correction constraints that the user has specified (e.g. that a series of strokes were actually one word rather than two). Having new constraints leads to an adjustment of recognizer hypotheses and a single user action can lead to multiple edits within the result. A small number of actions can often lead to the desired result with high accuracy.

The user can also correct errors within individual words. To do this, the user first taps on the word to show its individual characters, which they can then correct by writing over. The user can also delete unwanted characters by using a slashing gesture through a particular character. Finally, they can extend the recognition result by writing in a blank character box at the end of the word. As with segmentation corrections, each of these corrections triggers the recognizer, so that it can take the handwriting and the new information added by the user and re-recognize the text.

Note that there is no need for an explicit 'insert' character operation in such a scheme. For example, if the user would like to insert the letter 'i' into 'pan', they would simply rewrite the 'n' in the word. Given the original ink, the recognizer would decide whether to treat this as an insert and make the word 'pain' or as purely an overwrite and further transform the word, perhaps into 'pail'.

In order to validate our assertions that CueTIP would reduce the number of operations required to correct errors, we ran simulations to quantitatively evaluate CueTIP early in our design phase. We randomly selected 5000 handwriting samples that contained errors from a much larger corpus collected by the Microsoft TabletPC group. First, we calculated the average string edit distance from the initial result to the intended result. This is the number of operations that a user, operating optimally in a traditional interface, would require. This is a lower bound and we show in our user study that users often do not operate optimally in such interfaces. We compared this to the number of corrections that would be required with CueTIP. We found that CueTIP reduced the average number of operations by about 30% (from 2.98 edits to 2.07 edits for this corpus). See Figure 4 for examples of corrections that would require a single correction in CueTIP but significantly more in a traditional correction interface.

Optimizations

From early observations of users correcting recognition errors, we make two assumptions that lead to optimizations in this system. First, we assume that users generally correct the segmentation errors before correcting individual words.

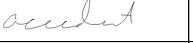
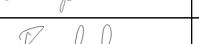
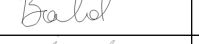
Original Ink	Predicted	Corrected
	audit	accident
	impend	inspired
	Babel	Bald
	formals	journals
	retread	returned

Figure 4: With a single character correction (in all of these examples), CueTIP is able to accurately propose the intended word, thus significantly reducing time and effort spent correcting errors.

This assumption allows us to appropriately scope character correction constraints to affect only the current word. This simplifies the propagation of corrections because we do not have to operate over the entire series of strokes. We believe that it also leads to a better interface because users generally do not expect recognition of entire sentences to change when they correct a single character in a word.

Second, when correcting within a word, we assume that the user generally corrects from left to right. In fact, when a user corrects a character, the constraint created is one stating that all characters up to and including the corrected one are correct and that the recognizer should try and find hypotheses within the ink that match this constraint. This is exposed to the user by changing the color of character tiles from blue to pink. Pink tiles indicate a constraint, or that the system will not try to change these. To create or remove a constraint, the user simply taps on the tile of interest. We did not see many users use this functionality beyond initial exploration of how it works.

HIGH-LEVEL DESIGN PRINCIPLES

Although CueTIP has a relatively simple interface, its specific design reflects a number of high-level guiding principles. We believe that understanding these will be useful to researchers trying to build new mixed-initiative correction interfaces, both for handwriting correction, but also in other domains. The key design considerations include:

Minimize decision points by choosing appropriate operators. Current recognition systems provide several options for users to transform errors into the intended text. Users can usually rewrite individual characters or words, type with the stylus on a soft keyboard, or pick words from an n-best word list [e.g. 14]. Unfortunately, the cognitive load imposed by such schemes is relatively high, as shown in speech correction work [2]. This is because users have to choose between multiple correction strategies, each of which may be more or less effective in different scenarios. Then, even within a given strategy, users have to carefully plan how to do the correction. For example, if a user decides to rewrite individual characters and would like to make the minimum number of edits possible, they have to

calculate where to overwrite existing characters or to insert and delete new ones. In an unpublished internal survey, the Microsoft TabletPC group found that many expert users quickly started skipping the correction mechanisms that entailed rewriting and could lead to ambiguous corrections. Instead they resorted to retyping the entire word using the more certain, but less efficient, soft keyboard.

In CueTIP, we reduce the core correction mechanisms available and provide only two operators with orthogonal functionality, correcting segmentation and correcting characters. This was derived from informal observations of how users think about the correction process and we believe drastically reduces cognitive effort required in our interface. Additionally, we offload the decision of whether the correction requires an insert or an overwrite to the recognizer, which makes the task easier. At any given point while correcting a word, the user only has to focus on overwriting the leftmost error within that word.

Design seamless transition between modes. Similarly, in cases for which mode switches are required, designers should try to make transitions as seamless and invisible as possible. For example, in the early instantiations of CueTIP, we had a checkbox which allowed users to specify whether or not they wanted the system to use a dictionary optimization to constrain available recognition results. While knowing whether or not to use a dictionary could theoretically have improved results drastically, users found it cumbersome to use. Hence, in subsequent iterations we allowed the system to make this decision, automatically using a dictionary optimization if it could not attain a reasonable result by looking at the handwriting. Users did not know the system was doing this, but were much more satisfied when using the system with this change.

Provide reachability of all states. We assert that the age-old HCI principle of "make the most frequently used operations easy to do, and the rest possible" applies equally well to transformations required for corrections. When designing CueTIP, we aimed to get the user to the intended text as quickly as possible. Most of the time, we assumed that the handwriting contained enough information that the system could make useful inferences about the result. However, occasionally, the user would like to transform an arbitrary word or phrase into another, regardless of the original handwriting. While we do not optimize for this case because early observations showed it to be the exception rather than the rule, we designed CueTIP such that in its limit, it is no worse than using a manual correction interface in order to get from any state to any other state. This is not always a natural characteristic of mixed-initiative interfaces in which the recognizer is constantly trying to 'pull' the answer towards its conception of the right answer.

Appropriately scope cascading changes. Another goal is to scope the impact of each change so that constraints applied to one part of the output do not result in sweeping changes to a distant part of the output. This is important so that users can mentally map impact of their actions and do not encounter too many surprising, and hence frustrating, changes. The two assumptions described above allow us to appropriately scope changes to the word or words being operated on and users seemed to find this intuitive.

Expose clear user model. While the system model and user model often do not and need not agree, we believe that it is important to provide the user with a clear model of how they should expect things to work. This increases predictability while using the interface and provides feeling of empowerment within the system-user dialog. In CueTIP, we do not formally expose the underlying constraint-based recognition system, but we do provide feedback about which characters the system will not change by appropriately coloring the tiles in which they are displayed. While constraints require some mathematical understanding of the system model, knowing which characters the system is going to help with versus not is intuitive for users.

Possible Correction Strategies

In the abstract correction task, there are several possible high-level correction strategies, including:

N-Best Selection. The simplest correction strategy is selection-based, in which the system returns an *n*-best list from which the user can select. In handwriting correction, this is typically an n-best word list for each word in a given segmentation. Unfortunately, an n-best list can usually only reveal a small portion of the possible output hypotheses. Furthermore, since recognition results are highly sensitive to misrecognition of individual characters, even small errors can cause the intended result to fall off the n-best list.

Transformation. A second common correction strategy is one in which the user applies a set of edits to the output to transform it into the desired result. In handwriting this includes inserting, deleting, and modifying characters and/or spaces in the output string. While this approach guarantees that the user can get to any point in the output space, this could require a large number of operations.

Hints. A third option is one in which the user provides hints to the recognizer, which the recognizer can use to reinterpret the inputs. For example, in handwriting recognition, rewriting a word could provide extra evidence of user intent, but does not guarantee anything about the revised output (except that perhaps it should be different from the original output). A hint-based approach has the potential to maximally leverage the recognizer's intelligence, but risks surprising the user, as well as discarding information that the user may have intentionally specified.

Constraints. A final option is one in which the user creates constraints, and the recognizer continuously tries to optimize within these constraints. In handwriting this could be constraining individual characters, words, or segmentations. This approach combines the best characteristics of transformations and hints. As the user starts adding constraints to the system, the recognizer has considerable freedom to help the user 'jump' to the right answer. However, as the user continues to correct, the recognizer becomes

over-constrained, and the behavior gracefully transitions to something more like a transformation-based system.

CUETIP V.1 IMPLEMENTATION

CueTIP is built in .NET C# on top of the Microsoft Tablet PC Platform SDK, a publicly available toolkit for building digital ink applications. This toolkit includes a set of facilities for capturing and displaying digital ink as well as access to a best-of-breed handwriting recognizer and can be downloaded at [www.microsoft.com/windowsxp/tabletpc].

Constraints

CueTIP's correction user interface is designed entirely in terms of constraints. To simplify the interaction, only two types of constraints are possible in CueTIP, segmentation constraints and character constraints. While these specific constraints are tailored to the handwriting correction task, we assert that this approach is not specific to handwriting, and similar sets of operations can be designed for other optimization-based recognition tasks.

Split / Join. When the user splits a word composed of a set of strokes into a multiple sets of strokes, CueTIP creates a set of segmentation constraints specifying that all recognition results must respect the user-specified segmentation. These constraints do not just apply to the ink that was split or joined, but to the entire ink stream. This ensures that the segmentation for other parts of the ink will not change, and therefore enforces the *scoping* design principle. Similarly, when the user joins multiple sets of strokes into a single word, CueTIP creates a segmentation constraint to represent the join.

Character. Similarly, when the user specifies that the j'th character of a word represented by the set of strokes should be the character *c*, this creates a constraint. The system also creates a set of segmentation constraints corresponding to the current segmentation. As in the split or join case, this scopes the recognition change to the current word.

Cascaded Constrained Recognizer

Unfortunately, the standard recognizer that ships with the SDK, and in fact most recognizers, do not support specifying constraints. We built the first version of CueTIP on top of the existing recognizer, and despite its shortcomings were able to demonstrate a reasonable improvement over the existing correction interface. At the end of the paper we describe a more graceful and performant implementation that involves augmenting the recognizer.

In order to implement CueTIP using the out-of-the-box Microsoft handwriting recognizer, we had to appropriately pre-process its inputs and post-process its outputs. The recognizer accepts a set of strokes as input and returns a list of hypothesis strings as output, ordered by confidence. It can be invoked in *standard mode*, in which it performs both word segmentation and recognition on the input ink, or in *word mode*, in which it only considers the given ink as a single word.

Additionally, it can be configured to use a custom user dictionary or to a regular expression (called a *factoid*) to further constrain its output. The customized dictionary is useful for domain-specific word recognition tasks, and factoids are useful for recognizing input with a known structure, such as telephone numbers, email addresses, or URLs.

When our system first tries to recognize the user's handwriting, it has no segmentation or character constraints, and the recognizer is called directly. As the user starts correcting, CueTIP adds segmentation and character constraints and re-invokes the recognizer with these. When segmentation constraints are added, the system breaks the ink up into individual words and feeds these words to the recognizer one at a time. It is for recognizing each word that CueTIP uses a cascaded heuristic:

1. The recognizer is initially run unconstrained on the word strokes. If there are no character constraints for that ink, the result is returned. Otherwise, the cascaded recognizer post-processes the output of the recognizer, iterating through the results until it finds one that matches the character constraints. Since the recognizer only emits a maximum of 32 words, we are not guaranteed a solution that matches the constraints.
2. If the system does not find a result in the first level of the cascade, the constraints are run over a dictionary word list, preprocessing the word list down to a set of words that match the constraints. This constrained word list is then used as the custom dictionary for the recognizer. For example, if the user has overwritten the first two letters of a word, the recognizer will only use words that start with those letters. This step is slow, but typically produces high-quality results.
3. Finally, if the word list in the previous section is empty, or the most confident match falls beneath a threshold, the recognizer reverts to a manual editing mode, where it takes any constraints that have been added and applies them to its previous output for that given set of strokes.

This cascaded approach gracefully handles both in and out-of-dictionary words through a smooth transition between constraint-based and transformational correction models. In informal observations of this system, users did not realize that this was happening and felt like the system was always doing the right thing. The drawback of this cascading approach is that it is computationally expensive and time consuming, so each correction made using this heuristic produced considerable lag. We ran a user study with this prototype as proof of concept and to show that further work in making the system more efficient was indeed warranted.

USER STUDY

We conducted a user study comparing the effectiveness of our mixed-initiative CueTIP interface to traditional correction mechanisms in current interfaces. These traditional correction mechanisms include correcting misrecognized characters by writing over them, correcting segmentation errors by treating spaces as characters that can be added or deleted, and correcting incorrect words by selecting options in a list that contains the recognizer's n-best guesses of the

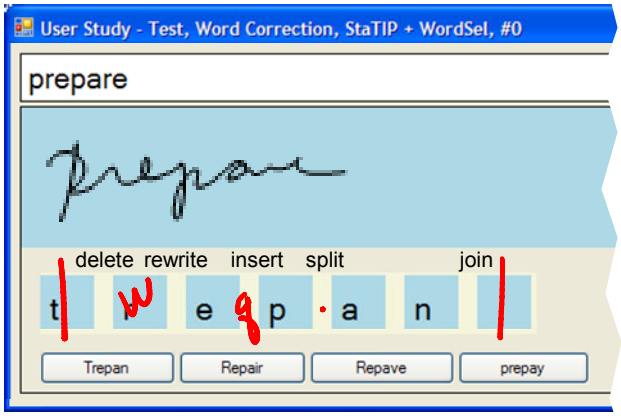


Figure 5: The StaTIP interface looked similar to the CueTIP one. As shown in the figure, users could delete, rewrite, insert, and append characters, as well as split and join. The recognizer did not assis in corrections, and none of these cascaded to other corrections.

writing. We replicated this functionality in a prototype we call StaTIP (for Standard Tablet Input Panel), which we implemented in the same framework as our system. See Figure 5 for a view of the interface within the experimental apparatus as well as the functionality that it provided.

CueTIP, as described earlier, extends the first two mechanisms by adaptively proposing other changes as the user corrects characters. It also provides a novel mechanism that allows users to correct segmentation errors with simple gestures to either split or join words. These segmentation corrections, like the character corrections, cause the recognizer to be called with more information and may cause cascading changes. To test the effects of n-best lists, we also created a version of CueTIP that adapts its n-best lists as the user starts to correct characters.

CueTIP incrementally updates its model of user intent and continually provides assistance (in the form of new recognition hypotheses for the handwriting) at each interaction within the correction process. Since we expect that each of these iterations will, on average, bring the user closer to the intended word or phrase faster than without the assistance, we hypothesized that users would require a smaller number of operations to make corrections with CueTIP than with StaTIP. In fact, since the number of operations is loosely associated with the amount of time spent on the correction, we also hypothesized that users would spend less time, on average, correcting words with CueTIP than with StaTIP, even with the additional processing time that is required in our particular implementation to regenerate hypotheses for each step within the correction interaction in CueTIP.

Similarly, the n-best list could potentially reduce the number of correction operations to one (simple selection of the correct option). Since we expected that users would utilize this functionality when they could, we expected that they would require a smaller number of operations to make corrections when they had the n-best list. However, it was unclear what this would, on average, do to completion time, since there is a cost associated with scanning the list, especially when the intended word is either late in the list or not there at all. Additionally, we set out to explore the effect that CueTIP had on the usage of the n-best list. We wanted to know if the reduced cost of correction offered by CueTIP would reduce the use of the list.

To explore these hypotheses, we decomposed the interfaces into the distinct correction mechanisms, character or segmentation correction as well as choosing results from the n-best list. We specifically examined only the correction interactions as users utilized the traditional and CueTIP implementations both on single words as well as on phrases.

Participants

Twelve volunteers (5 female) from the Greater Puget Sound area participated in this study. Users averaged 43.1 years of age (ranging from 29 to 54 years). Users were screened such that half of them were novice tablet handwriting input users who had used this functionality for less than 3 hours overall, and half were experts, who used it for at least 3 hours a week over the course of at least the 3 months preceding the study. All users had normal or corrected-to-normal eyesight, and all were physically able to use the handwriting interfaces for the 1.5 hour study. Users received software gratuities for their participation.

Tasks and Materials

We used two tasks to examine the various components of the interfaces. In the first task, we explicitly tested the efficiency of the interfaces for *single word correction*. For each trial in this task, we presented users with a target word they had to input using one of the interfaces. Since we were primarily interested in use of the correction mechanism and not in the initial handwriting recognition accuracy, we provided users with sample handwriting of the target word that would always lead to an initial recognition error. Hence, users were forced to make corrections on every trial in this task. We constructed five sets of words, one practice and four test sets. The practice set contained 16 words and test sets contained 12 words each.

We used a corpus of several hundred thousand text segments and associated handwriting samples collected by the Microsoft Tablet PC group to train and test the Windows Tablet Input Panel. In order to pick our questions, we first filtered the corpus to contain only samples that would lead to an initial misrecognition. We calculated the histogram of string edit distances, or the minimal number of inserts, deletes, or substitutions, needed to transform the recognized word into the intended one. We then randomly selected samples to create sets of words that matched the overall histogram. This led to test sets which had approximately half of its words with a string edit distance of three or less, and the rest with a distance of greater than three.

In the second task, we tested the efficiency of the interfaces when users had to perform *phrase correction*. This task included both word as well as segmentation correction. We used the same procedure as in the first task to force correc-

tions on every trial. We also used an equivalent procedure to create five sets of words, a practice set with 12 phrases and four test sets with 8 phrases each. In addition to the process we used for word selection, we also imposed the limitation that our phrase samples would contain at least one segmentation error.

We collected trial times for each correction. Trial time was the time taken from presentation of the initial incorrect result until the user corrected and confirmed the word or phrase. Although this did not include the time of the initial presentation and recognition of the handwriting, it did include any time required to recognize correction overwrites and potentially to generate new hypotheses. We also measured the number of character or segmentation correction actions required to get to the desired result. This included the number of inserts, deletes, word splits and joins, character toggles, n-best list selects, as well as successful overwrites. Finally, since we were interested in how users would utilize the n-best list, we measured the number of times users invoked this list when it was present.

Setup

Users performed the study on a Toshiba M200 TabletPC with a 2GHz PentiumM processor and 1GB of memory. The display ran at a resolution of 1400 × 1050 and was used in landscape orientation in the slate mode. Users interacted with the system using the stylus that comes with the tablet. Before beginning the study, users sat at a table and adjusted their chair as well as the position and orientation of the tablet so that they were comfortable writing on it. Some users preferred using the tablet on the desk while others placed them in their laps or cradled them.

Design and Procedure

The study was a within-subjects design, with each user performing all the tasks in all four conditions, created by crossing Interface (StaTIP vs. CueTIP) and n-best list (absent vs. present). After balancing for expertise, we counterbalanced the order of interface and n-best list independently across users, with each user having the same order across both tasks. All users performed the tasks in the same order, starting with the word correction task on both interfaces and moving to the phrase correction task. Test sets within each task were performed in the same order so that they would be balanced across interfaces.

At the beginning of the experiment, we instructed users on how to use each interface in the study. We provided written instructions, demonstrated the interface, and allowed users to ask questions they had. Prior to beginning each task, we further provided written instructions explaining the task to the user. For each set of trials with each interface, users practiced on a subset of practice questions, and when they were comfortable with the interface and task, performed the task on the test set. After performing each task on both interfaces, users filled out a questionnaire indicating subjective preference and providing free-form comments for the interfaces within that particular task. They also filled out a final subjective questionnaire at the end of the study.

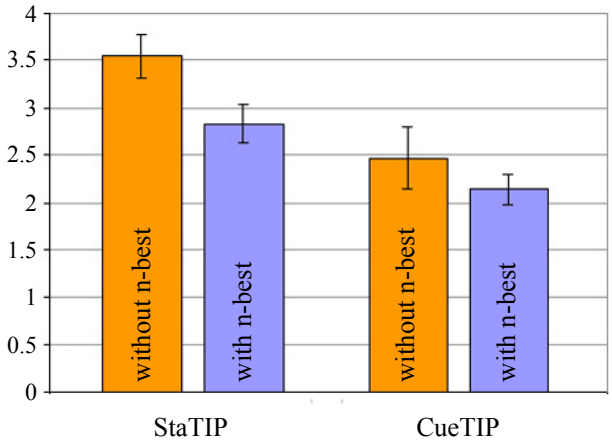


Figure 6: CueTIP required significantly fewer manual operations to make successful corrections. The n-best list also significantly reduced this number. Note that the average string edit distance of all trials was 3.

Results

Performance

We analyzed data at the summary level. For each of our metrics we conducted a 2 (Task: Word correction vs. Phrase correction) × 2 (Interface: StaTIP vs. CueTIP) × 2 (n-best list: absent vs. present) repeated measures analysis of variance (RM-ANOVA). We had initially included Expertise (Novice vs. Expert) as a factor, but dropped this from our final analysis because we found no effects due to it. Results remain equivalent without this factor.

The first metric we tested was the average number of operations required to correct the words. For this metric, we found a significant main effect of Interface ($F(1,11)=92.72$, $p<.001$), with CueTIP requiring fewer operations than StaTIP (2.31 vs. 3.19, respectively). We also found a significant main effect of n-best list ($F(1,11)=33.175$, $p<.001$), with the n-best list reducing the number of operations required (2.49 vs. 3.01). We also saw a main effect of Task ($F(1,11)=33.33$, $p<.001$), reaffirming that phrase correction required more correction operations than word correction. Finally, we saw a borderline significant interaction between Interface and n-best list ($F(1,11)=4.683$, $p=.053$). Paired comparisons using Bonferonni correction indicated that each condition, except for the CueTIP without n-best list, was significantly different from all others. See Figure 6 for a summary of these results.

In looking at the number of uses of the n-best list, we found a significant main effect of n-best list, though this is a relatively uninteresting finding given that the n-best list was not available in one set of those conditions. We saw no interactions, indicating either that n-best list was not used differentially across Task or Interface, or more likely that users utilized this functionality so infrequently (slightly under 15% of the time when it was present) that it was not a very sensitive metric.

Finally, we examined average completion time for each trial. Here, we found a significant main effect of Interface

($F(1,11)=49.25$, $p<.001$), again with the CueTIP leading to faster times than StaTIP, on average (18.7 vs. 21.9 s, respectively). This represents about a 15% speed up using CueTIP. We also observed a significant main effect of Task ($F(1,11)=560.99$, $p<.001$), with phrase correction taking much longer than word correction (28.4 vs. 12.2 s). We saw no main effects or interactions with n-best list.

Subjective Preference

After each task, we had users provide ratings on a 5-point likert scale for efficiency and ease of each interface. The higher end of the scale represented favorable responses. We ran a similar 2 (Task) × 2 (Interface) × 2 (n-best list) RM-ANOVA on this data. We observed a significant effect of Interface ($F(1,11)=12.44$, $p=.005$), with users preferring the CueTIP interface over StaTIP (mean rating of 3.6 vs. 2.8). We also observed a main effect of Task ($F(1,11)=38.46$, $p<.001$), suggesting that people felt both interfaces were less efficient in the phrase correction than the word correction task. When we had users rank order the interfaces at the end of the experiment, 10 of the 12 users had one of the CueTIP interfaces (with or without n-best list) as the top on their list. In fact, CueTIP without the n-best list topped that list with an average rating of 1.9, followed by CueTIP with the n-best list at 2.1. StaTIP with and without the n-best list were at 2.9 and 3.1 respectively.

When users were asked to comment on the various interfaces, many (eight) stated that had been skeptical before using CueTIP that they would have liked the system making indirect changes for them. However, all of these users claimed to have been pleasantly surprised and the strongest proponents of the system were in fact the ones who had not expected it to be very satisfactory. One confusion that several users (five) brought up with CueTIP had to do with the words lining up with the ink, as it created unevenly spaced words. Also, several (six) users commented that they did not like how they had to match their gesture to the ink when they wanted to split a word. They wanted to perform the correction entirely within the result and had totally forgotten about the ink at that point. We will implement and examine these improvements in future versions of CueTIP.

CUETIP V.2 IMPLEMENTATION

The user study showed our proof-of-concept CueTIP system, which implemented a cascaded constrained recognizer, helped users correct recognition errors with fewer operations and in less time, and was preferred over a traditional interface. This was extremely encouraging, given that the specific implementation added considerable computational overhead which resulted in lag at each step in the correction procedure. In this section, we discuss our second implementation, which improved efficiency and generality of the system. Rather than treating the recognizer as a black box, as in the first implementation, we modified the recognizer to integrate and incorporate constraints natively.

Integrated Constrained Recognizer

In this implementation, we established a different abstraction for the recognizer. Our goal was to be able to communicate the constraints *declaratively*, rather than procedurally, to allow for greatest flexibility in the implementation. To see why this makes sense, consider word segmentation constraints. In our first implementation of segmentation constraints, we divided the ink up into multiple stroke sets and call the recognizer in word mode for each stroke set. However, doing this deprives the recognizer of context that could improve recognition. For example, the recognizer would not be able to leverage its language model that exploits the frequency of word co-occurrences if it were being fed ink word-by-word.

As mentioned in the introduction, many recognition algorithms are implemented as optimizations, where the goal of the algorithm is to find an output that optimizes a likelihood function. In handwriting recognition, the optimization is over sequences of ink fragments, where each fragment is created by cutting strokes at their local minima and maxima. Each fragment is interpreted as a piece of a character, and the optimization stitches together these interpretations to come up with word hypotheses. Interpretations that include words that are not in the dictionary are penalized over words that are in the dictionary as a way of encoding the language model. There are many ways to perform this optimization.

Naïve. With a set of M interpretations for each fragment (where M is proportional to the number of characters in the alphabet), a naïve optimization that tries every combination would consume time proportional to $O(K^M)$, which is impossibly large for any real input.

Viterbi. One common improvement over the naïve approach is to use dynamic programming, which, for optimizing over sequences, is the Viterbi algorithm [5]. In Viterbi, the problem is rephrased as a recurrence relation in which the optimal interpretation of the whole input sequence is the best combination of all interpretations of the first fragment with all interpretations of the rest of the fragments. The computational complexity is therefore $O(NM^2)$. This technique assumes that the cost interaction of interpreting the first and second fragments must not be dependent on any other fragments in the input. Unfortunately, fragment cost interactions are non-local when out-of-dictionary words are penalized. Techniques exist to adapt Viterbi for large-dictionary scenarios, but they are $O(V^2)$ where V is the number of words in the dictionary [5].

Beam. An efficient approach to large-lexicon recognition is Beam Search [5]. Beam Search is a left-to-right scan across the sequence of fragments, where each new character candidate is ranked as an extension to all of the hypotheses before it. Beam search is approximate because it only keeps the top K hypotheses at each point in the sequence. It tries to combine the top K next hypotheses with the previous top K previous hypotheses, and takes the top K of these results. This is an $O(NK^2)$ algorithm.

Constrained Beam. In order to implement CueTIP, we extended Beam Search and constructed a Constrained Beam Search algorithm, in which character constraints come into play. For each step in which the result path is extended, we also check the resulting path for validity against the character constraints and keep only valid paths in the beam. When added to the existing beam search implementation, this extension is extremely efficient and causes negligible performance slowdown. Furthermore, the left-to-right character constraint assumption that we have built into the recognition interface interacts very well with beam search because it ensures that the correct answer stays "on the beam" as it scans left to right.

CONCLUSION AND FUTURE WORK

In this paper, we have presented a mixed-initiative approach to designing correction interfaces. We have described CueTIP, a novel interface for correcting handwriting recognition errors. Grounded in our discussion of this interface, we have presented design principles that may be generalized and applied to interfaces in other domains for which users have to correct ambiguous interpretations of their input. We have also shown in a user study that even the inefficient implementation of such a system allows users to more effectively correct errors, and as a result, have built the more efficient implementation.

In future work, we plan on refining CueTIP based on study feedback and observations. Users seem to forget about the ink once they have written it and expressed preference for correcting entirely within the result domain. It is not obvious how to implement an output-only solution for specifying and leveraging segmentation constraints, but this is something we plan to explore. We have also observed that misrecognized character corrections are extremely frustrating. We hope to modify to the character recognition algorithm that is biased on its word context in order to minimize these errors. We also plan to explore efficient ways to leverage character n-best lists to make the character correction task completely unambiguous. Finally, we hope to build upon these results and design similar mixed-initiative interfaces in other domains.

ACKNOWLEDGMENTS

We would like to thank Ahmad Abdulkader, Kumar Chellapilla, Mary Czerwinski, Eric Horvitz, Keywon Chung, and the Microsoft TabletPC group for engaging discussions and invaluable assistance through the course of this project.

REFERENCES

1. Goldberg, D., & Goodisman, A. (1991). Stylus User Interfaces for Manipulating Text. *Proceedings of Fourth Annual ACM Symposium on User Interface Software and Technology*, 127-135.
2. Halverson, C., Horn, D., Karat, C., Karat, J. (1999). The Beauty of Errors: Patterns of Error Correction in Desktop Speech Systems. *Proceedings of Interact 1999*, 133-140.
3. Horvitz, E. (1999). Principles of Mixed-Initiative User Interfaces. *Proceedings of CHI 1999 Conference on Human Factors in Computing Systems,* 159-166.
4. Huerst, W., Yang, J., & Waibel, A. (1998). Interactive Error Repair for an Online Handwriting Interface. *Proceedings of CHI 1998 Conference on Human Factors in Computing Systems*, 353-354.
5. Jelinek, F (1997). *Statistical Methods for Speech Recognition*. The MIT Press.
6. Kristjansson, T., Culotta, A., Viola, P., & McCallum, A. (2004). Interactive Information Extraction with Constrained Conditional Random Fields. *Proceedings of the 19th AAAI International Conference on Artificial Intelligence*, 412-418.
7. Mankoff, J., Hudson, S.E., Abowd, G.D. (2000). Interaction Techniques for Ambiguity Resolution in Recognition-based Interfaces. *Proceedings of Thirteenth Annual ACM Symposium on User Interface Software and Technology,* 11-20.
8. Microsoft Tablet Input Panel. Retrieved 1 April 2006: www.microsoft.com/windowsxp/using/tabletpc/pen/correcttext.mspx.
9. Oviatt, S.L. (2000). Taming Recognition Errors with a Multimodal Interface. *Communications of the ACM*, 43(9), 45-51.
10. Oviatt, S.L., & R. VanGent (1996). Error Resolution During Multimodal Human-Computer Interaction. *Proceedings of the 1996 International Conference on Spoken Language Processing,* 1, 204-207.
11. Plamondon, R., & Srihari, S. (2000). On-line and Off-line Handwriting Recognition: A Comprehensive Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1), 63-84.
12. Schomaker, L.R.B. (1994). User-interface Aspects in Recognizing Connected-Cursive Handwriting. *Proceedings of the IEE Colloquium on Handwriting and Pen-based Input, 1994/065*, 8/1-8/3.
13. Smithies, S., Novins, K., & Arvo, J. (1999). A Handwriting-Based Equation Editor. *Proceedings of Graphics Interface*, 84-91.
14. Tappert, C.C., Suen, C.Y., & Wakahara, T. (1990). The State of the Art in On-Line Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(8),* 787-808.