TrailBlazer: Enabling Blind Users to Blaze Trails Through the Web

Jeffrey P. Bigham
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195 USA
jbigham@cs.washington.edu

Tessa Lau and Jeffrey Nichols
IBM Almaden Research Center
650 Harry Rd
San Jose, CA 95120 USA
{tessalau, jwnichols}@us.ibm.com

ABSTRACT
For blind web users, completing tasks on the web can be frustrating. Each step can require a time-consuming linear search of the current web page to find the needed interactive element or piece of information. Existing interactive help systems and the playback components of some programming-by-demonstration tools identify the needed elements of a page as they guide the user through predefined tasks, obviating the need for a linear search on each step. We introduce *TrailBlazer*, a system that provides an accessible, non-visual interface to guide blind users through existing how-to knowledge. A formative study indicated that participants saw the value of TrailBlazer but wanted to use it for tasks and web sites for which no existing script was available. To address this, TrailBlazer offers suggestion-based help created on-the-fly from a short, user-provided task description and an existing repository of how-to knowledge. In an evaluation on 15 tasks, the correct prediction was contained within the top 5 suggestions 75.9% of the time.

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces; K.4.2 Social Issues: Assistive technologies for persons with disabilities

General Terms
Human Factors, Design, Algorithms

Author Keywords
Non-Visual Interfaces, Web Accessibility, Programming by Demonstration, Suggestions, Blind Users

INTRODUCTION
For blind web users, completing tasks on the web can be time-consuming and frustrating. Blind users interact with the web through software programs called screen readers. Screen readers convert information on the screen to a linear stream of either synthesized voice or refreshable Braille. If a blind user needs to search for a specific item on the page, they either must listen to the entire linear stream until the goal item is reached or they may skip around in the page using structural elements, such as headings, as a guide. To become proficient, users must learn hundreds of keyboard shortcuts to navigate web page structures and access mouse-only controls. Unfortunately, for most tasks even the best screen reader users cannot approach the speed of searching a web page that is afforded to sighted users [5, 23].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IUI 2009, February 8 - 11, 2009, Sanibel Island, Florida, USA.
Copyright 2009 ACM 978-1-60558-331-0/09/02...$5.00.

Time Card CoScript

1. goto "http://www.mycompany.com/timecard/"
2. enter "8" into the "Hours worked" textbox
3. click the "Submit" button
4. click the "Verify" button

Figure 1. A CoScript for entering time worked into an online time card. The natural language steps in the CoScript can be intrepreted both by tools such as CoScripter and TrailBlazer, and also read by humans. These steps are also sufficient to identify all of the web page elements required to complete this task – the textbox and two buttons. Without TrailBlazer, steps 2-4 would require a time-consuming linear search for screen reader users.

Existing repositories contain how-to knowledge that is able to guide people through web tasks quickly and efficiently. This how-to knowledge is often encoded as a list of steps that must be perfomed in order to complete the task. The description of each step consists of information describing the element that must be interacted with, such as a button or text box, and the type of operation to perform with that element. For example, one step in the task of buying an airplane flight on orbitz.com is to enter the destination city into the text box labeled "To". One such repository is provided by CoScripter [15], which contains a collection of scripts written in a "sloppy" programming language that is both human- and machine-understandable (Figure 1).

In this paper, we present *TrailBlazer*, a system that guides blind users through completing tasks step-by-step. TrailBlazer offers users suggestions of what to do next, automatically advancing the focus of the screen reader to the interactive element that needs to be operated or the information that needs to be heard. This capability reduces the need for time-consuming linear searches when using a screen reader.

TrailBlazer was created to support the specific needs of blind users. First, its interface explicitly accomodates screen readers and keyboard-only access. Second, TrailBlazer augments the CoScripter language with a "clip" command that specifies a particular region of the page on which to focus the user's attention. A feature for identifying regions is not included in other systems because it is assumed that users can find these regions quickly using visual search.

Third, TrailBlazer is able to dynamically create new scripts from a brief user-specified description of the goal task and the existing corpus of scripts. Dynamic script creation was inspired by a formative user study of the initial TrailBlazer system, which confirmed that TrailBlazer made the web more usable but was not helpful in the vast majority of cases where a script did not already exist. To address this problem, we hypothesized that users would be willing to spend a few moments describing their desired task for TrailBlazer if it could make them more efficient on tasks lacking a script. The existing repository of scripts helps CoScripter to incorporate knowledge from similar tasks or sub-tasks that have already been demonstrated. Building from the existing corpus of CoScripter scripts and the short task description, TrailBlazer dynamically creates new scripts that suggest patterns of interaction with previously unseen web sites, guiding blind users through sites for which no script exists.

As blind web users interact with TrailBlazer to follow these dynamically-suggested steps, they are implicitly supervising the synthesis of new scripts. These scripts can be added to the script repository and reused by all users. Studies have shown that many users are unwilling to pay the upfront costs of script creation even though those scripts could save them time in the future [13]. Through the use of TrailBlazer, we can effectively reverse the traditional roles of the two groups, enabling blind web users to create new scripts that lead sighted users through completing web tasks.

This paper makes the following three contributions:

- **An Accessible Guide** - TrailBlazer is an accessible interface to the how-to knowledge contained in the CoScripter repository that enables blind users to avoid linear searches of content and complete tasks more efficiently.
- **Formative Evaluation** - A formative evaluation of TrailBlazer illustrating its promise for improving non-visual access, as well as the desire of participants to use it on tasks for which a script does not already exist.
- **Dynamic Script Generation** - TrailBlazer, when given a natural language description of a user's goal and a pre-existing corpus of scripts, dynamically suggests steps to follow to achieve the user's goal.

RELATED WORK

Work related to TrailBlazer falls into two main categories: (i) tools and techniques for improving non-visual web access, and (ii) programming by demonstration and interactive help systems that play back and record how-to knowledge.

Improving Web Accessibility

Most screen readers simply speak aloud a verbal description of the visual interface. While this enables blind users to access most of the software available to sighted people, they are often not easy-to-use because their interfaces were not designed to be viewed non-visually. Emacspeak demonstrated the benefits to usability resulting from designing applications with voice output in mind [21]. The openness of the web enables it to be adapted it for non-visual access.

Unfortunately, most web content is not designed with voice output in mind. In order to produce a usable spoken interface to a web site, screen readers extract semantic information and structure from each page and provide interaction techniques designed for typical web interactions. When pages contain good semantics, these can be used to improve the usability of the page, for instance by enabling users to skip over sections irrelevant to them.

Semantic information can either be added to pages by content providers or formulated automatically when pages are accessed. Adding meaningful headings tags (<H1 - 6>) has been shown to improve web efficiency for blind web users browsing structural information [24] but less than half of web pages use them [5]. To improve web navigation, in-page "skip" links visible only to screen reader users can be added to complex pages by web developers. These links enable users to quickly jump to areas of the page possibly far in linear distance. Unfortunately, these links are often broken [5]. Web developers have proven unreliable in manually providing navigation aids by annotating their web pages.

Numerous middleware systems [1] have suggested ways for inserting semantically relevant markup into web pages before they reach the client. Other systems have moved the automatic detection of semantically-important regions to the interface itself. For example, the Hearsay non-visual web browser parses web pages into a semantic tree that can be more easily navigated with a screen reader [20].

Augmenting the screen reader interface has also been explored. Several systems have added information about surrounding pages to existing pages to make them easier to use. Harper *et al.* augments links in web pages with "Gist" summaries of the linked pages in order to provide users more information about the page to which a link would direct them [10]. CSurf observes the context of clicked links in order to begin reading at a relevant point in the resulting page [17].

Although adding appropriate semantic information makes web content more usable, finding specific content on a page is still a difficult problem for screen reader users. AxsJAX addresses this problem by embedding "trails" into web pages that guide users through semantically-related elements [2]. TrailBlazer scripts expand on this trail metaphor. Because AxsJAX trails are generally restricted to a single page and are written in Javascript, AxsJAX trails cannot be created by end users or applied to the same range of tasks as TrailBlazer's scripts.

Recording and playback of how-to knowledge

Interactive help systems and programming by demonstration tools have explored how to capture procedural knowledge and express it to users. COACH [22] and Eager [8] are early systems in this space that work with standard desktop applications instead of the web. COACH observes computer users in order to provide targeted help, and Eager learned and executed repetitive tasks by observing users.

Expressing procedural knowledge, especially to assist a user who is currently working to complete a task, is a key issue for interactive help systems. Kelleher et al.'s work on stencil-based tutorials demonstrates a variety of useful mechanisms [12], such as by blurring all of the items on the screen except for those which are relevant to the current task. Sticky notes adding useful contextual information was also found to be effective. TrailBlazer makes use of analogous ideas to direct the attention of users to important content in its non-visual user interface.

Representing procedural knowledge is also a difficult challenge. *Keyword commands* is one method, which uses simple psuedo-natural language description to refer to interface elements and the operations to be applied to them [16]. This is similar to the sloppy language used by CoScripter to describe web-based activity [15]. TrailBlazer builds upon these approaches because the stored procedural knowledge represented by TrailBlazer can be easily spoken aloud and understood by blind users.

A limitation of most current systems is that they cannot generalize captured procedural knowledge to other contexts. For example, recording the process of purchasing a plane flight on orbitz.com will not help perform the same task on travelocity.com. One of the only systems to explore generalization is the Goal-Oriented Web Browser [9], which attempts to generalize a previously demonstrated script using a database of common sense knowledge. This approach centered around data detectors that could determine the type of data appearing on web sites. TrailBlazer incorporates additional inputs into its generalization process, including a brief task description from the user, and does not require a common-sense knowledgebase.

An alternate approach to navigating full-size web pages with a script, as TrailBlazer does, is to instead shrink the web pages by keeping only the information needed to perform the current task. This can be done using a system such as Highlight, which enables users to reauthor web pages for display on small screen devices by demonstrating which parts of the pages used in the task are important [18]. The resulting simplified interfaces created by Highlight are more efficient to navigate with a screen reader, but prevent the user from deviating from the task by removing content that is not directly related to the task.

AN ACCESSIBLE GUIDE

TrailBlazer was designed from the start for non-visual access using the following three guidelines (Figure 2):

- **Keyboard Access**. All play back functions are accessible using only the keyboard, making access for those who do not use a mouse feasible.

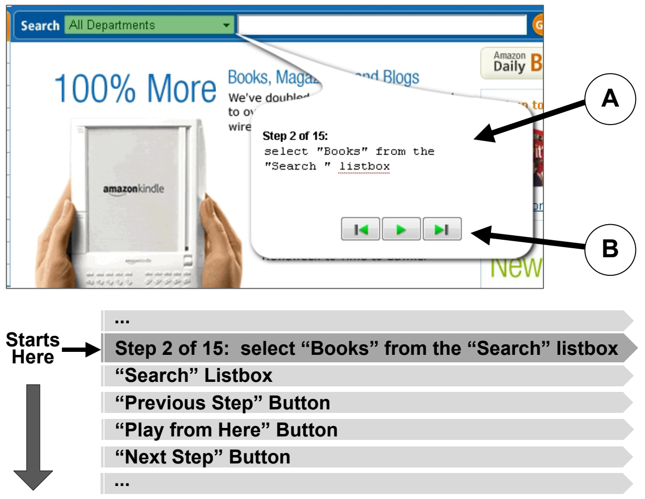

Figure 2. The TrailBlazer interface is integrated directly into the page, is keyboard accessible, and directs screen readers to read each new step. A) The description of the current step is displayed visually in an offset bubble but is placed in DOM order so that the target of a step immediately follows its description when viewed linearly with a screen reader. B) Script controls are placed in the page for easy discoverability but also have alternative keyboard shortcuts for efficient access.

- **Minimize Context Switches**. The playback interface is integrated directly into the web pages through which the user is being guided. This close coupling of the interface into the web page enables users to easily switch between TrailBlazer's suggestions and the web page components needed to complete each step.

- **Directing Focus**. TrailBlazer directs users to the location on each page to complete each step. As mentioned, a main limitation of using a screen reader is the difficulty in finding specific content quickly. TrailBlazer directs users to the content necessary to complete the instruction that it suggests. If the user wants to complete a different action, the rest of the page is immediately available.

The bubbles used to visually highlight the relevant portion of the page and provide contextual information were inspired by the "sticky notes" used in Stencil-Based Tutorials [12]. The non-visual equivalent in TrailBlazer was achieved by causing the screen reader to begin reading at the step (Figure 2). Although the location of each bubble is visually offset from the target element, the DOM order of the bubble's components was chosen such that they are read in an intuitive order for screen reader users. The visual representation resembles that of some tutoring systems, and may also be preferred by users of visual browsers, in addition to supporting non-visual access with TrailBlazer.

Upon advancing to a new instruction, the screen reader's focus is set to the instruction description (e.g., "Step 2 of 5: click the "search" button"). The element containing that text is inserted immediately before the relevant control (e.g., the search button) in DOM order so that exploring forward from this position will take the user directly to the element mentioned in the instruction. The playback controls for previous

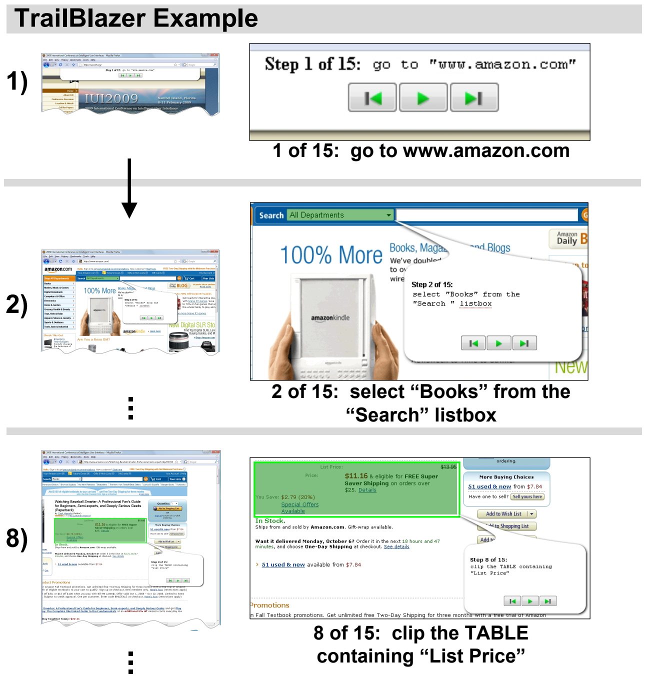
TrailBlazer Example
1)
Step 1 of 15: go to "www.amazon.com"
1 of 15: go to www.amazon.com
2)
Step 2 of 15: select "Books" from the "Search " listbox
2 of 15: select "Books" from the "Search" listbox
8)
Step 8 of 15: clip the TABLE containing "List Price"
8 of 15: clip the TABLE containing "List Price"

Figure 3. TrailBlazer guiding a user step-by-step through purchasing a book on Amazon. 1) The first step is to goto the Amazon.com homepage. 2) TrailBlazer directs the user to select the "Books" option from the highlighted listbox. 8) On the product detail page, TrailBlazer directs users past the standard template material directly to the product information.

step, play, and next step are represented as buttons and are inserted following the relevant control. Each of these functions can also be activated by a separate keyboard shortcut - for example, "ALT+S" advances to the next step.

The TrailBlazer interface enables screen reader users to move from step to step, verifying that each step is going to be conducted correctly, while avoiding all linear searches through content (Figure 3). In the event that the user does not want to follow a particular step of the script they are using, the entire web page is available to them as normal. TrailBlazer is a guide but does not override the user's intentions.

CLIPPING

While examining the scripts in the CoScripter repository, we noticed that many scripts contained comments directing users to specific content on the page. Comments are not interpreted by CoScripter however, and there is no command in CoScripter's language that can identify a particular region of the screen. Whether users were looking up the status of their flight, checking the prices of local apartments or searching Google, the end goal was not to press buttons, enter information into text boxes, or follow links; the goal was to find information. A visual scan might locate this information quickly, but doing so with a screen reader would be a slower process.

Coyne *et al.* observed that blind web users often use the "Find" function of their web browsers to address this issue [7]. The find function provides a simple way for users to

1-a. "2008 season stats"
1-b. "The highlighted region is of statistics. This is a table that has multiple numbers describing a player's achievements and records of what he has accomplished."

2-a. "This region lists search results for your query."
2-b. "This area contains the heading 'Search Results' along with the returns from a search of a term."

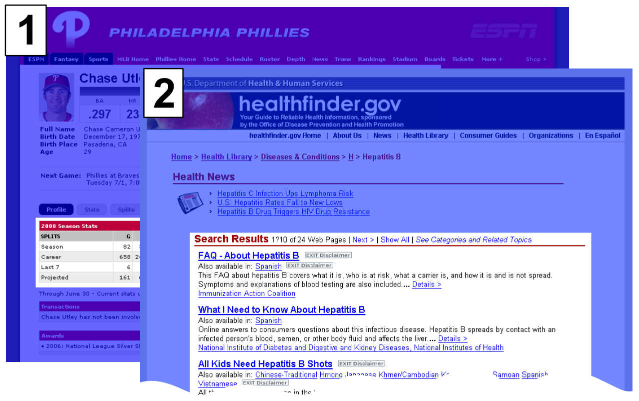

Figure 4. The descriptions provided by two participants for the screenshots shown illustrating diversity in how regions were described. Selected regions are 1) the table of statistics for a particular baseball player, and 2) the search results for a medical query.

quickly skip to the content, but requires them to know in advance appropriate text for which to search. The "clip" command that TrailBlazer adds to the CoScripter language enables regions to be described and TrailBlazer users to be quickly directed to them.

Region Description Study

Existing CoScripter commands are written in natural language. In order to determine what language would be appropriate for our CoScripter command, we conducted a study in which we asked 5 participants to describe 20 regions covering a variety of content (Figure 4). To encourage participants to provide descriptions that would generalize to multiple regions, two different versions of each region were presented.

Upon an initial review of the results of this study, we concluded that the descriptions provided fell into the following 5 non-exclusive categories: high-level semantic descriptions of the content (78%), descriptions matching all or part of the headings provided on the page for the region (53%), descriptions drawn directly from the words used in the region (37%), descriptions including the color, size, or other stylistic qualities of the region (18%), and descriptions of the location of the region on the page (11%).

The Syntax of the Clip Command

We based the formulation of the syntax of the clip command on the results of the study just described. Clearly, users found it most convenient to describe the semantic class of the region. While future work may seek to leverage a data detector like Miro to automatically determine the class of data in order to facilitate such a command [9], our clip command currently refers to regions by either their heading or the content contained within them.

When using a heading to refer to a region, a user lists text that starts the region of interest. For instance, the step "clip the 'search results"' would begin the clipped region at the text "search results." This formulation closely matched what many users wrote in our study, but does not explicitly specify an end to the clip. TrailBlazer uses several heuristics to end the clip. The most important part is directing the user to the general area before the information that is valuable to them. If the end of the clipped region comes too soon, they can simply keep reading past the end of the region.

To use text contained within a region to refer to it, users write commands like, "clip the region containing "flight status"". For scripts operating on templated web site or for those that use dynamically-generated content, this is not always an ideal formulation because specific text may not always be present in a desired region. By using both commands, users have the flexibility to describe most regions, and, importantly, TrailBlazer is able to easily interpret them.

FORMATIVE EVALUATION

The improvements offered in the previous sections were designed to make TrailBlazer accessible to blind web users using a screen reader. In order to investigate its perceived usefulness and remaining usability concerns, we conducted a formative user study with 5 blind participants. Our participants were experienced screen reader users. On average, they had 15.0 (SD=4.7) years of computer experience, including 11.8 (2.8) years using the web.

We first demonstrated how to use TrailBlazer as a guide through pre-defined tasks. We showed users how they could, at each step, choose to either have TrailBlazer complete the step automatically, complete it themselves, or choose any other action on the page. After this short introduction, participants performed the following three tasks using TrailBlazer: (i) checking the status of a flight on united.com, (ii) finding real estate listings fitting specific criteria, and (iii) querying the local library to see if a particular book is available.

After completing these tasks, each participant was asked the extent to which they agreed with several statements on a Likert scale (Figure 5). In general, participants were very enthusiastic about TrailBlazer, leading one to say "this is exactly what most blind users would like." One participant said TrailBlazer was a "very good concept, especially for the work setting where the scenarios and templates are already there." Another participant who helps train people on screen reader use thought that the interface would be a good way to gradually introduce the concept of using a screen reader to a new computer user for whom the complexity of web sites and the numerous shortcuts available to them can be overwhelming. Participants uniformly agreed that despite their experience using screen readers, "finding a little kernel of information can be really time-consuming on a complex web page" and that "sometimes there is too much content to just use headings and links to navigate."

Participants wondered if TrailBlazer could help them with dynamic web content, which often is added to the DOM of web pages far from where it appears visually, making it difficult to find. Screen readers can also have trouble presenting dynamically-created content to users. TrailBlazer could not only direct users to content automatically, avoiding a long linear search, but also help them interact with it.

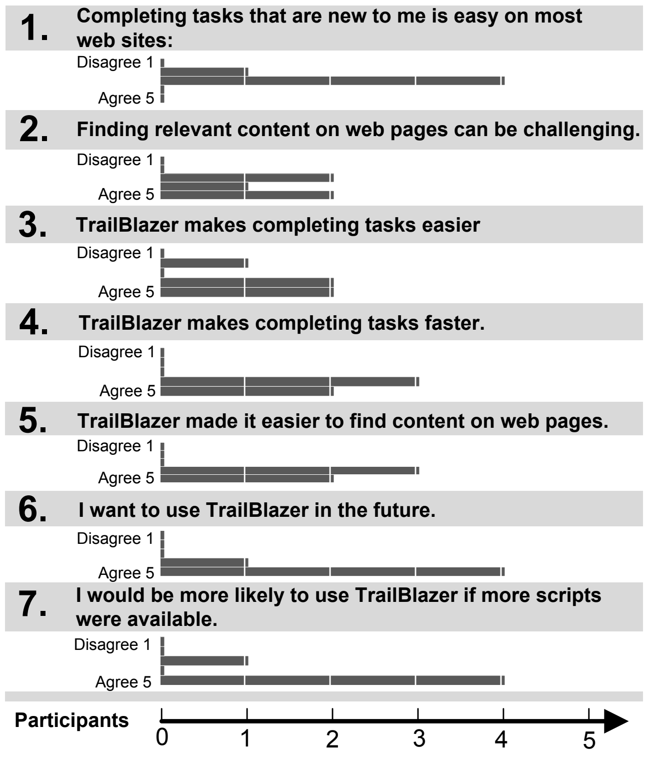

Figure 5. Participant responses to Likert scale questions indicating that they think completing new tasks and finding content is difficult (1, 2), think TrailBlazer can help them complete tasks more quickly and easier (3,4,5), and want to use it in the future (6), especially if scripts are available for more tasks (7).

Despite their enthusiasm for using TrailBlazer for tasks that were already defined, they questioned how useful it would be if they had to rely on others to provide the scripts for them to use. One participant even questioned the usefulness of scripts created for a task that he wanted to complete because "designers and users do not always agree on what is important." TrailBlazer did not support recording new tasks at the time of the evaluation, although new CoScripts could be created by sighted users using CoScripter.

Participants also had several suggestions on how to improve the interface. TrailBlazer guides users from one step to the next by dynamically modifying the page, but screen readers do not always update their external models of the pages that they read from. To fix this users would need to occasionally refresh the model of the screen reader, which many thought could be confusing to novice users. Other systems that improve non-visual access have similar limitations [2], and these problems are being addressed in upcoming versions of screen readers.

DYNAMIC SCRIPT GENERATION

TrailBlazer can suggest actions that users may want to take even when no pre-existing script is available for their current task. These suggestions are based on a short task description provided by the user and an existing repository of how-to knowledge. Suggestions are presented to users as options, which they can quickly jump to when correct but also easily ignore. Collectively, these suggestions help users dynamically create a new script - potentially increasing efficiency even when they first complete a task.

Example Use Case

To inform its suggestions, TrailBlazer first asks users for a short textual description of the task that they want to complete; it then provides appropriate suggestions to help them complete that task. As an example, consider Jane, a blind web user who wants to look up the status of her friend's flight on Air Canada. She first provides TrailBlazer with the following description of her task: "flight status on Air Canada." The CoScripter repository does not contain a script for finding the status of a flight on Air Canada, but it does contain scripts for finding the status of flights on Delta and United.

After some pre-processing of the request, TrailBlazer conducts a web search using the task description to find likely web sites on which to complete it. "goto aircanada.com" is its first suggested step, and Jane chooses to follow that suggestion. If an appropriate suggestion was not listed, then Jane could have chosen to visit a different web site or even searched the web for the appropriate web site herself (perhaps using TrailBlazer to guide her search). TrailBlazer automatically loads aircanada.com and then presents Jane with the following three suggestions: "click the 'flight status' button", "click the 'flight' button, and "fill out the 'search the site' textbox." Jane chooses the first, and TrailBlazer completes it automatically. Jane uses this interface to complete the entire task without needing to search within the page for any of the necessary page elements.

A pre-existing script for the described task is not required for TrailBlazer to accurately suggest appropriate actions. TrailBlazer can in effect apply scripts describing tasks (or subtasks) on one web site on other web sites. It can, for example, use a script for buying a book at Amazon to buy a book at Barnes and Noble, a script for booking a trip on Amtrak to help book a trip on the United Kingdom's National Rail Line, or a script for checking the status of a package being delivered by UPS to help check on one being delivered by Federal Express. Subtasks contained within scripts can also be applied by TrailBlazer in different domains. For example, the sequence of steps in a script on a shopping site that helps users enter their contact information can be applied during the registration process on an employment site. If a script already exists for a user's entire task, then the suggestions they receive can follow that script without the user having to conduct a search for that specific script in advance.

Suggestion Types

The CoScripter language provides a set number of action types (Figure 6). Most CoScripts begin with a "goto" command that directs users to a specific web page. Next, users are led through interaction with a number of links and form controls. Although not included in the scripts in the CoScripter repository, the final action implicitly defined in most CoScripts is to read the information that resulted from completion of the previous steps, which corresponds to the "clip" command added by TrailBlazer.

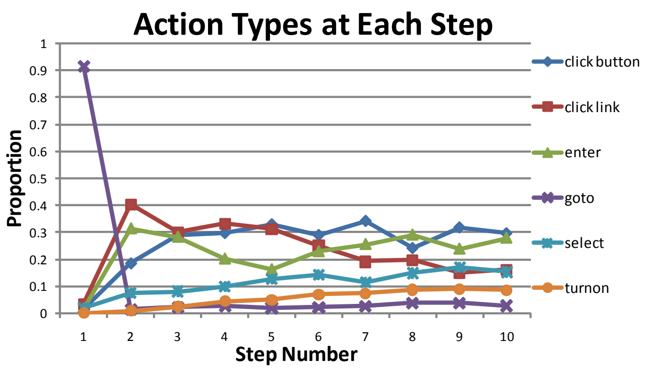

Figure 6. Proportion of action types at each step number for scripts in the CoScripter repository.

The creation of suggestions in TrailBlazer is divided into the following three corresponding components:

- **Goto Component** - TrailBlazer converts a user's task description to keywords, and then searches the web using those keywords to find appropriate starting sites.
- **General Suggestion Component** - TrailBlazer combines a user's task description, scripts in an existing repository, and the history of the user's actions to suggest the next action that the user should take.
- **Automatic Clipping Component** - TrailBlazer uses the textual history of user actions represented as CoScripter commands to find the area on the page that is most likely relevant to the user at this point using an algorithm inspired by CSurf [17]. Finding the relevant region to read is equivalent to an automatic clip of content.

The following sections describe the components used by TrailBlazer to choose suggestions.

Goto Component

As shown in Figure 6, most scripts begin with a goto command. Accordingly, TrailBlazer offers goto commands as suggestions when calculating the first step for the user to take. The goto component uses the task description input by the user to suggest web sites on which the task could be completed.

Forming goto suggestions consists of the following three steps: (i) determining which words in the task description are most likely to describe the web site on which the task it to be completed, (ii) searching the web using these most promising keywords, and (iii) presenting the top results to users. A part-of-speech tagger first isolates the URLs, proper nouns, and words that follow prepositions (e.g., "United"

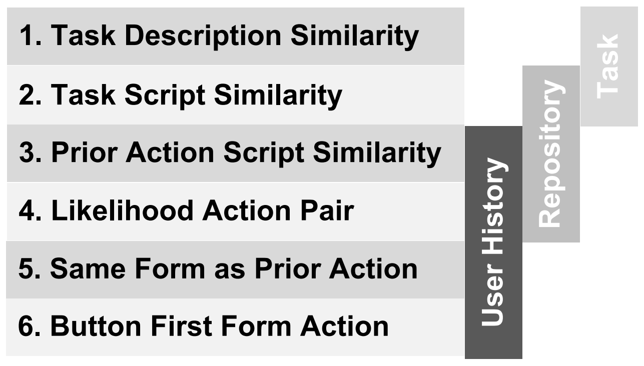
1. Task Description Similarity
2. Task Script Similarity
3. Prior Action Script Similarity
4. Likelihood Action Pair
5. Same Form as Prior Action
6. Button First Form Action
User History
Repository
Task

Figure 7. The features calculated and used by TrailBlazer in order to rank potential action suggestions, along with the three sources from which they are formed.

from the phrase "on United") in the task description. TrailBlazer proceeds in a series of rounds, querying with keywords in the order described until it gathers at least 5 unique URLs. These URLs are then offered as suggestions.

The performance of the goto component is highly dependent on the task description provided by the user and on the popularity of the site on which it should be completed. The authors have observed this component to work well on many real-world tasks, but future work will test and improve its performance.

General Suggestion Component

The main suggestion component described in this paper is the general suggestion component, which suggests specific actions for users to complete on the current page. These suggestions are presented as natural language steps in the CoScripter language and are chosen from all the actions possible to complete on the current page. TrailBlazer ranks suggestions based on the user's task description, knowledge mined from the CoScripter script repository, and the history of actions that the user has already completed.

Suggestions are first assigned a probability by a Naive Bayes classifier and then ranked according to them. Naive Bayes is a simple but powerful supervised learning method that after training on labeled examples can assign probability estimates to new examples. Although the probabilities assigned are only estimates, they are known to be useful for ranking [14]. The model is trained on recordings of tasks that were previously demonstrated using either TrailBlazer or CoScripter, which are contained within the CoScripter script repository.

The knowledge represented by the features used in the model could also have also been expressed as static rules for the system to follow. TrailBlazer's built-in machine learning model enables it to continually improve as it is used. Because tasks that users complete using TrailBlazer implicitly describe new scripts, the features based on the script repository should become more informative over time as more scripts are added.

Features Used in Making Suggestions

In order to accurately rank potential actions, TrailBlazer relies on a number of informative, automatically-derived features (Figure 7). The remainder of this section explains the motivation behind the features found to be informative and describes how they are computed.

Leveraging Action History

TrailBlazer includes several features that leverage its record of actions that it has observed the user perform. Two features capture how the user's prior actions relate to their interaction with forms (Figure 7-5,6). Intuitively, when using a form containing more than one element, interacting with one increases the chance that you will interact with another in the same form. The *Same Form as Prior Action* feature expresses whether the action under consideration refers to an element in a form for which an action has previously been completed. Next, although it can occur in other situations, pressing a button in a form usually occurs after acting on another element in a form. The *Button First Form Action* feature captures whether the potential action is a button press in a form in which no other elements have been acted upon.

Similarity to Task Description

The *Task Description Similarity* feature enables TrailBlazer to 1weight steps similar to the task description provided by the user more highly (Figure 7-1). Similarity is quantified by calculating the vector-cosine between the words in the task description and the words in each potential suggestion. The word-vector cosine metric considers each set of words as a vector in which each dimension corresponds to a different term, and in which the length is set to the frequency by which each word has been observed. For this calculation, a list of stopwords are removed. The similarity between the task description word vector v_d and the potential suggestion word vector v_s is calculated as follows.

$$VC(v_d, v_s) = \frac{v_d \cdot v_s}{||v_d|| * ||v_s||} \quad (1)$$

The word-vector cosine is often used in information retrieval settings to compare documents with keyword queries [3].

Using the Existing Script Repository

TrailBlazer uses the record of user actions combined with the scripts contained with the script repository to directly influence which suggestions are chosen. The CoScripter repository contains nearly 1000 human-created scripts describing the steps to complete a diversity of tasks on the web. These scripts contain not only the specific steps required to complete a given web-based task, but also more general knowledge about web tasks. Features used for ranking suggestions built from the repository are based on (i) statistics of the actions and the sequence in which they appear, and (ii) matching suggestions to relevant scripts already in the repository. These features represent the knowledge contained within existing scripts, and enables TrailBlazer to apply that knowledge to new tasks for which no script exists.

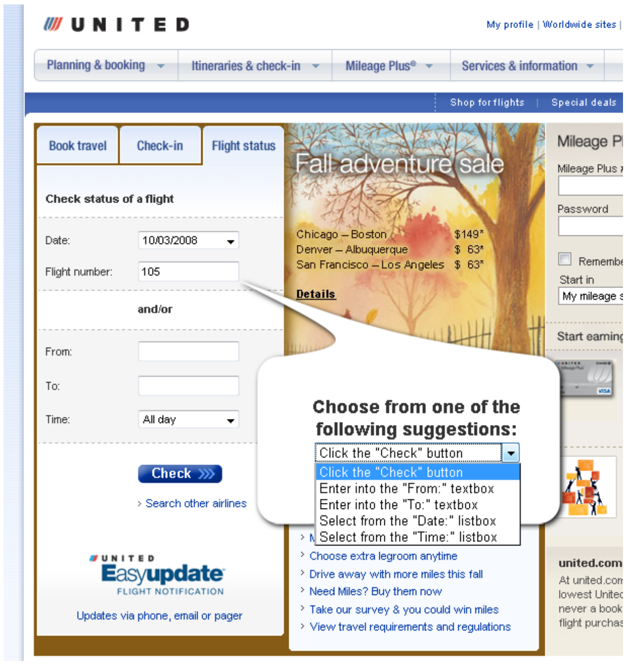

Figure 8. Suggestions are presented to users within the page context, inserted into the DOM of the web page following the last element with which they interacted. In this case, the user has just entered "105" into the "Flight Number" textbox and TrailBlazer recommends clicking on the "Check" button as its first suggestion.

Some action types are more likely than others according to how many actions the user has completed (Figure 6). For instance, clicking a link is more likely near the beginning of a task than near the end. In addition, some actions are more likely to follow actions of particular types. For instance, clicking a button is more likely to follow entering text than it is clicking a link because buttons are usually pressed after entering information into a form. Following this motivation, the *Likelihood Action Pair* feature used by TrailBlazer is the likelihood of each action given the actions that the user completed before (Figure 7-4). This likelihood is computed through consideration of all scripts in the repository.

Leveraging Existing Scripts for Related Tasks

TrailBlazer also uses related scripts already in the repository to help form its suggestions. Two sets of related scripts are retrieved and a separate feature is computed for each. First, TrailBlazer uses the task description provided by the user as a query to the repository, retrieving scripts related to the user's task. For instance, if the user's task description was "Flight status on Air Canada," matches on the words "flight" and "status" will enable the system to retrieve scripts for finding the flight status on "United" and "Delta." The procedure for checking flight status on both of these sites is different than it is on Air Canada but certain steps, like entering information into a textbox with a label containing the words "Flight Number" are repeated on all three. The *Task Script Similarity* feature captures this information (Figure 7-2).

The second set of scripts that TrailBlazer retrieves are found using the last action that the user completed. These scripts may contain subtasks that do not relate to the user's stated goal but can still be predictive of the next action to be completed. For instance, if the user just entered their username into a textbox with the label "username," many scripts will be retrieved suggesting that a good next action would be to enter their "password" into a password box. The *Prior Action Script Similarity* feature enables TrailBlazer to respond to relevant sub-tasks (Figure 7-3).

The motivation for the *Task Script Similarity* and *Prior Action Script* features is that if TrailBlazer can find steps in existing scripts similar to either the task description or an action previously completed by the user, then subsequent steps in that script should be predictive of future actions. The scores assigned to each step are, therefore, fed forward to other script steps so that they are weighted more highly. All tasks implicitly start with a goto step specifying the page on which the user first requests suggestions, so a prior action always exists. The process used is similar to spreading activation, which is a method used to connect semantically-related elements represented in a tree structure [6]. The added value from a prior step decreases exponentially for each subsequent step, meaning that steps following close after highly-weighted steps primarily benefit.

To compute these features, TrailBlazer first finds a set of related scripts S by sending either the task description or the user's prior action as a $query$ to the CoScripter repository. TrailBlazer then derives a weight for each of the steps contained in each related script. Each script s contains a sequential list of natural language steps (Figure 1). The weight of each script's first step is set to $VC(s_0, query)$, the vector-cosine between the first step and the query as described earlier. TrailBlazer computes the weight of each subsequent step, as follows:

$$W(s_i) = w * W(s_{i-1}) + VC(s_i, query) \qquad (2)$$

TrailBlazer currently uses $w = 0.3$, which has worked well in practice. The fractional inclusion of the weight of prior steps serves to feed their weight forward to later steps.

Next, TrailBlazer constructs a weighted sentence $sent_S$ of all the words contained within S. The weight of each word is set to the sum of the computed weights of each step in which each word is contained, $W(s_i)$. The final feature value is the word-vector cosine between vectors formed from the words in $sent_S$ and $query$. Importantly, although the features constructed in this way do not explicitly consider action types, the labels assigned to page elements, or the types of page elements, all are implicitly included because they are included in the natural language CoScripter steps.

Presenting Suggestions to Users

Once the values of all of the features are computed and all potential actions are ranked, the most highly-ranked actions are presented to the user as suggestions. The suggestions are integrated into the accessible guide interface outlined earlier. TrailBlazer provides five suggestions, displayed in the interface in rank order (Figure 8).

The suggestions are inserted into the DOM immediately following the target of the prior command, making them appear to non-visual users to come immediately after the step that they just completed. This continues the convenient non-visual interface design used in TrailBlazer for script play back. Users are directed to the suggestions just as they would be directed to the next action in a pre-existing script. Just as with pre-defined actions, users can choose to review the suggestions or choose to skip past them if they prefer, representing a hallmark of mixed-initiative design [11]. Because the suggestions are contained within a single listbox, moving past them requires only one keystroke.

Future user studies will seek to answer questions about how to best present suggestions to users, how many suggestions should be presented, and how the system's confidence in its suggestions might be conveyed by the user interface.

EVALUATION OF SUGGESTIONS

We evaluated TrailBlazer by testing its ability to accurately suggest the correct next action while being used to complete 15 tasks. The chosen tasks represented the 15 most popular scripts in the CoScripter repository according to the number people who have run them. The scripts contained a total of 102 steps, with an average of 6.8 steps per script (SD=3.1). None of the scripts included in the test set were included when training the model.

Using existing scripts to test TrailBlazer provided two advantages. The first was that the scripts represented a natural ground truth to which we could compare TrailBlazer's suggestions and the second was that each provided a short title that we could use as the user's description for purposes of testing. The provided titles were relatively short, averaging 5.1 words per title. The authors believe that it is not unreasonable to assume that users could provide similar task descriptions since users provided these titles.

On the 15 tasks in this study, TrailBlazer listed the correct next action as its top suggestion in 41.4% of cases and within the top 5 suggestions in 75.9% of cases (Figure 9). Predicting the next action correctly can dramatically reduce the number of elements that users need to consider when completing tasks on the web. The average number of possible actions per step was 41.8 (SD=37.9), meaning that choosing the correct action by chance has a probability of only 2.3%. TrailBlazer's suggestions could help users avoid a long, linear search over these possibilities.

Discussion

TrailBlazer suggested the correct next action among its top 5 suggestions in 75.9% of cases. The current interface enables users to review these 5 choices quickly, so that in these cases users will not need to search the entire page in order to complete the action - TrailBlazer will lead them right there. Furthermore, in the 24.1% of cases in which TrailBlazer did not make an accurate suggestion, users can continue completing their tasks as they would have without TrailBlazer. Future studies will look at the effect on users of incorrect suggestions and how we might mitigate these problems.

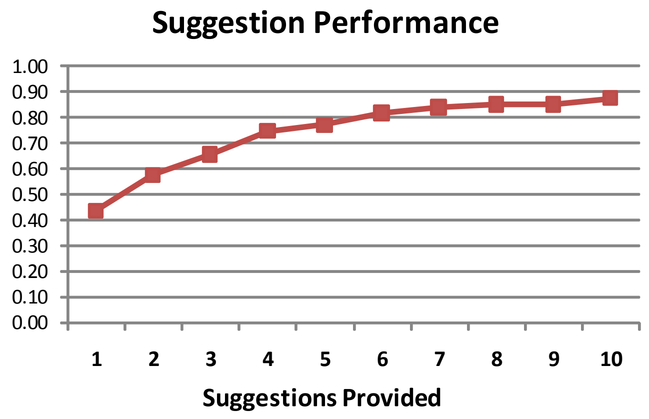

Figure 9. The fraction of the time that the correct action appeared among the top suggestions provided by TrailBlazer for varying numbers of suggestions. The correct suggestion was listed first in 41.4% cases and within the top 5 in 75.9% of cases.

TrailBlazer had difficulties making correct suggestions when the steps of the procedure seemed arbitrary. For instance, the script for changing one's Emergency Contact information begins by clicking through links titled "Career and Life" and "About me - personal" on pages on which nearly 150 different actions could be made. Because no similar scripts existed, no features indicated that it should be selected. Later, the script included the step, "click the "Emergency contacts" link," which the system recommended as its first choice. These problems illustrate importance of having scripts in the first place, and are indicative of the improvements possible as additional scripts and other knowledge are incorporated.

Fortunately, TrailBlazer is able to quickly recover upon making an error. For instance, if it does not suggest an action involving a particular form and the user completes an action on the form anyway, TrailBlazer is likely to recover on the next suggestion. This is particularly true with form elements because of the features created specifically for this case (features 5 and 6 in Figure 7), but TrailBlazer is often able to recover in more general cases. TrailBlazer benefits greatly from its design as a guide to a human who can occasionally correct its suggestions, and its operators also benefit because of the efficiency gains when it is correct.

FUTURE WORK

TrailBlazer can accurately predict the next step that users are likely to want to perform based on an initial action and prvides an accessible interface enabling blind users to leverage those suggestions. A next step will be user studies with blind web users to study how blind web users use the system and discover improvements on the provided interface to suggestions.

Guiding web users through completing web tasks has many applications beyond improving access for blind web users. The TrailBlazer approach adapts easily to commonly-used phone menu systems. Because of the reduction in complexity achieved by guiding users, the TrailBlazer approach may also be appropriate for certain users with learning or cognitive disabilities. Additionally, users of small-screen devices face many of the same problems finding relevant information in complex web pages as screen reader users and may benefit from a similar approach.

Interfaces for non-visual access could benefit from moving toward task-level assistance of the kind exposed by TrailBlazer. Current interfaces too often focus on either low-level annotations or deciding beforehand what users will want to do, taking away their control.

CONCLUSION

We introduced TrailBlazer, an accessible interface to how-to knowledge that helps blind users complete web-based tasks faster and more effectively by guiding them through a task step-by-step. By directing the user's attention to the right places on the page and by providing accessible shortcut keys, TrailBlazer enables users to follow existing how-to instructions quickly and easily. A formative evaluation of the system revealed that users were positive about the system, but that the lack of how-to scripts could be a barrier to use. We extended the TrailBlazer system to dynamically suggest possible next steps based on a short description of the desired task, the user's previous behavior, and a repository of existing scripts. The user's next step was contained within the top 5 suggestions 75.9% of the time, showing that TrailBlazer is successfully able to guide users through new tasks.

ACKNOWLEDGEMENTS

We thank T.V. Raman for inspiring the use of trails to improve non-visual access; the participants in our formative evaluation for useful and encouraging feedback; and Yevgen Borodin, Anna Cavender, Allen Cypher, Clemens Drews, James Lin, and Barton Smith for their feedback and support.

REFERENCES

1. Asakawa, C. and Takagi, H. *Web Accessibility: A Foundation for Research*, Springer, 2008.
2. AxsJAX. Google, Inc. (2008). Http://code.google.com/p/google-axsjax/.
3. Baeza-Yates, R., and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Boston, Massachusetts, 1999.
4. Bigham, J. P. and Cavender, A. C. Evaluating Existing Audio CAPTCHAs and an Interface Optimized for Non-Visual Use. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI '09)*, Boston, Massachusetts, 2009. To Appear.
5. Bigham, J. P., Cavender, A. C., Brudvik, J. T., Wobbrock, J. O., and Ladner, R. E. Webinsitu: A comparative analysis of blind and sighted browsing behavior. In *Proc. of the 9th Intl. ACM SIGACCESS Conf. on Computers and Accessibility (ASSETS '07)*, Tempe, Arizona, 2007, 51–58.
6. Collins, A. and Loftus, E. A spreading activation theory of semantic processing. *Psychological Review*, *82* (1975), 407–428.
7. Coyne, K. P. and Nielsen, J. Beyond alt text: Making the web easy to use for users with disabilities (2001).
8. Cypher, A. Eager: programming repetitive tasks by example. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI '91)*. New Orleans, Louisiana, United States, 1991, 33–39.
9. Faaborg, A. and Lieberman, H. A goal-oriented web browser. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI '06)*. 2006, 751–760.
10. Harper, S. and Patel, N. Gist summaries for visually impaired surfers. In *Proc. of the 7th Intl. ACM SIGACCESS Conf. on Computers and Accessibility (ASSETS '05)*. New York, NY, USA, 2005, 90–97.
11. Horvitz, E. Principles of mixed-initiative user interfaces. In *Proc. of the SIGCHI Conf. on Human factors in Computing Systems (CHI '99)*. New York, NY, USA, 1999, 159–166.
12. Kelleher, C. and Pausch, R. Stencils-based tutorials: design and evaluation. In *Proc. of the SIGCHI Conf. on Human factors in Computing Systems (CHI '05)*. Portland, Oregon, USA, 2005, 541–550.
13. Leshed, G., Haber, E. M., Matthews, T., and Lau, T. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proc. of the 26th SIGCHI Conf. on Human Factors in Computing Systems (CHI '08)*. Florence, Italy, 2008, 1719–1728.
14. Lewis, D. D. Naive bayes at forty: The independence assumption in information retrieval. In *Proc. of ECML-98, 10th European Conf. on Machine Learning*. Springer Verlag, Heidelberg, DE, Chemnitz, DE, 1998, 1398, 4–15.
15. Little, G., Lau, T., Cypher, A., Lin, J., Haber, E. M., and Kandogan, E. Koala: capture, share, automate, personalize business processes on the web. In *Proc. of the SIGCHI Conf. on Human factors in Computing Systems (CHI '07)*. 2007, 943–946.
16. Little, G. and Miller, R. C. Translating keyword commands into executable code. In *Proc. of the 19th annual ACM symposium on User Interface Software and Technology (UIST '06)*. New York, NY, USA, 2006, 135–144.
17. Mahmud, J., Borodin, Y., and Ramakrishnan, I. V. Csurf: A context-driven non-visual web-browser. In *Proc. of the Intl. Conf. on the World Wide Web (WWW '07)*. 31–40.
18. Nichols, J. and Lau, T. Mobilization by demonstration: using traces to re-author existing web sites. In *Proc. of the 13th Intl. Conf. on Intelligent User Interfaces (IUI '08)*. Gran Canaria, Spain, 2008, 149–158.
19. Nielsen, J. *Hypertext and hypermedia*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.
20. Ramakrishnan, I. V., Stent, A., and Yang, G. Hearsay: Enabling audio browsing on hypertext content. In *Proc. of the 13th Intl. Conf. on the World Wide Web (WWW '04)*. 2004.
21. Raman, T. V. Emacspeak—a speech interface. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI '96)*. Vancouver, Canada, 1996, 66–71.
22. Selker, T. Cognitive adaptive computer help (coach). In *Proc. of the Intl. Conf. on Artificial Intelligence*. IOS, Amsterdam, 1989, 25–34.
23. Takagi, H., Saito, S., Fukuda, K. and Asakawa, C. Analysis of navigability of web applications for improving blind usability. In *ACM Transactions on Computer-Human Interaction*, 14:3–13. ACM Press, 2007.
24. Watanabe, T. Experimental evaluation of usability and accessibility of heading elements. In *Proc. of the Intl. Cross-Disciplinary Conf. on Web Accessibility (W4A '07)*. 2007, 157 – 164.