Efficient Web Form Entry on PDAs

Oliver Kaljuvee
Digital Libraries Project (InfoLab), Stanford University, Stanford, CA, 94305
kaljuvee@cs.stanford.edu

Orkut Buyukkokten
Digital Libraries Project (InfoLab), Stanford University, Stanford, CA, 94305
orkut@db.stanford.edu

Hector Garcia-Molina
Digital Libraries Project (InfoLab), Stanford University, Stanford, CA, 94305
hector@db.stanford.edu

Andreas Paepcke
Digital Libraries Project (InfoLab), Stanford University, Stanford, CA, 94305
paepcke@db.stanford.edu

ABSTRACT
We propose a design for displaying and manipulating HTML forms on small PDA screens. The form input widgets are not shown until the user is ready to fill them in. At that point, only one widget is shown at a time. The form is summarized on the screen by displaying just the text labels that prompt the user for each widget's information. The challenge of this design is to automatically find the match between each text label in a form, and the input widget for which it is the prompt. We developed eight algorithms for performing such label-widget matches. Some of the algorithms are based on n-gram comparisons, while others are based on common form layout conventions. We applied a combination of these algorithms to 100 simple HTML forms with an average of four input fields per form. These experiments achieved a 95% matching accuracy. We developed a scheme that combines all algorithms into a matching system. This system did well even on complex forms, achieving 80% accuracy in our experiments involving 330 input fields spread over 48 complex forms.

Keywords
PDA, Mobile Computing, WAP, Forms, Wireless Access

1. INTRODUCTION
Web browser designers face special challenges when trying to enable users to browse the World-Wide Web from handheld devices, such as Palm Pilots. Among these challenges, screen size limitations and the inconvenient pen-based input of URLs, keywords, and other information stand out. Screen size limitations are an issue because most HTML pages are designed to be viewed on desktop displays. Their page layout assumes that users can see large portions of each page at once. The much smaller page excerpts displayed on any one handheld device screen can interfere with users' comprehension, and the resulting scrolling activity is time consuming. Similarly, the pen-based input common to most handheld devices remains error prone and time consuming, even though character recognition has improved over the years.

One solution to the problem is the creation of special Web pages for small devices. Such pages would be laid out for optimal viewing on small screens. One example of such an approach is the Wireless Access Protocol's Markup Language (WML). While effective, this solution requires information to be prepared separately for display on both standard Web browsers, and on handheld devices. Many Web site administrators are hard pressed even now to maintain their sites for just standard browser viewing, so adding additional maintenance load is often infeasible.

Another approach to presenting information on personal digital assistants (PDAs) has been the automatic miniaturization of standard HTML pages. One such system is the ProxiWeb browser [6]. It adjusts images to work well on PDAs, and otherwise automatically formats page displays with small screen requirements in mind. This approach works well when every detail on a page is needed. A drawback of the approach is the large amount of necessary scrolling action. This need for frequent scrolling can seriously degrade the navigation phase of Web searches [3].

1.1 Our Previous Related Work
Our Power Browser is a Web browser for handheld devices [3, 4]. This browser uses a very different approach to support navigation and viewing of pages. Figure 1 shows a screen shot of the system as reported in the above references.

Instead of displaying an entire page, only the link anchors of pages are displayed by default. For example, "Concurrent VLSI Architecture," and "Database Group" are links on one of Stanford's Web pages. The user may tap on one of the links. This action will display the text of the corresponding page. Alternatively, a left-to-right swiping pen gesture over the same link description would list the target page's links on the PDA. For example, the lines "Archival Repositories" through "Hobbies" in Figure 1 are links on the "Arturo Crespo" page. These lines were added to the display in

Figure 1: Navigation Screenshot of Power Browser

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

response to a user's left-to-right pen gesture over "Arturo Crespo." Notice that the links on the Crespo page are indented. The resulting nested navigation levels are displayed like the folders in many graphical file browsers. Successive levels are indented, with thin vertical bars indicating the level of browsing depth for each row of displayed text.

In addition to this navigation support, we dynamically create inverted indexes of Web sites as users browse the Web from their PDAs. The indexes are used for site-specific searching, and to provide keyword completion as users write search keywords with their pen [4]. User studies in [1, 5], finally, examine different techniques for summarizing Web pages for display on PDAs. We are using a Web proxy server to prepare the modified PDA views. When preparing information for display on a PDA, we can therefore afford computational expenses beyond those possible on the PDA itself.

We were able to show that the facilities outlined above are effective for searching and browsing. They do not, however, address the display and manipulation of HTML forms on small displays. This paper focuses on the forms problem.

1.2 Filling Forms on PDA Screens

Many HTML pages ask users to provide information that is uploaded to a Web server. In many cases, the forms used for this purpose are simple, small collections of text input fields. For example, search engines often have just one text input field for entering search keywords. Web sites for exploring company product information might have a pull-down widget from which the user chooses products. Other sites use much more complex forms. For example, *www.garden.com*, which offers advice on planting, contains extensive forms that include check boxes, radio buttons, pull-down lists, and other input mechanisms.

The problem with forms on PDAs is that the input widgets and associated textual explanations consume too much screen real-estate. It is difficult for users to gain an overview of what the form is about, and which information is requested of them. We are therefore designing a form filling component for our Power Browser.

Rather than showing all of the input widgets at once, we want our Power Browser initially to just display minimal textual prompts for the fields. This approach would allow users to scan and understand the form quickly. When the user is ready to fill in information, pen gestures over a textual prompt would cause the associated input widget to be displayed. All other widgets would remain hidden. Figure 2 shows a mock-up example of how the display will work. We focus here not on how exactly this interface would look and behave, but on the underlying technology that will enable this type of user interface approach.

Figure 2a shows a Web page for finding people's telephone, address, and other information by name, as it would be seen on a desktop-size browser. This example is taken from Yahoo!'s site.

Figure 2b shows the top form, with only the labels being displayed. In general, all the text other than the labels will be ignored in this view. Consequently, every visible string is the label for some form item. Buttons are identified explicitly. Users may tap on the corresponding text to activate the button. Links within the form are handled as normal hyperlinks. For instance, "State" in the above example is a hyperlink to another page.

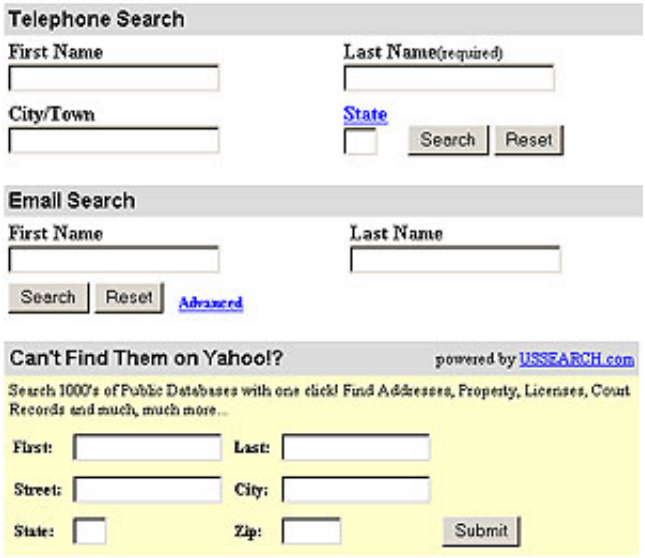

Figure 2a: Original HTML Page

Figure 2b: Form With No Widgets Revealed

Figure 2c: Form With One Widget Revealed

Figure 2c shows the top form expanded. The user has performed a left-to-right pen gesture over the last name of the telephone search. Now an input widget is available for the user to fill in. A right-to-left pen gesture would close the form again, returning the display to the state of Figure 2b.

For the Palm Pilot's operating system, Palm OS, we need to convert HTML text and password input fields to the equivalent dotted lines used on the Palm Pilot. Text areas similarly are converted to multiple dotted lines. Checkboxes, radio buttons, and selection widgets also have their Palm OS equivalents, and we need make the conversion for each input field.

The difficulty in generating the space-efficient display of Figure 2c from the original Web page is that we need algorithms that find the proper text labels for each input widget. For example, it is easy for a human being to recognize that the "First Name" at the top of Figure 2a is the text label for the input widget below it. Making this association automatically, however, is not always easy.

To illustrate this difficulty, consider the following piece of HTML, which produces a portion of the screen display in Figure 2a.

```
...
First Name <input type="text" name="FirstName">
Last Name <input type="text" name="LastName">
City/Town <input type="text" name="City">
...
```

Each line of HTML will produce one text label and one text input widget when rendered by a browser. We call the expressions in angle brackets *input elements*. Each input element includes the type of widget the browser is to display (in all these three cases type="text"), and a name attribute (e.g. name="FirstName"). At the machine level, the matching task is to associate the name attribute value of each input element with the best of its surrounding strings.

As human observers, we easily recognize the text "First Name", "Last Name", "City/Town" as labels for their respective input elements. In the HTML snippet's second line, for example, it is clear to a human that the input element with the name attribute 'LastName' is intended to be matched with the "Last Name" text label on the left. We make this association because the name attribute and the text label match well, and because we are used to reading from left to right. For the above HTML code, an automatic matching process could perform the string match the same as a human would, and if the match would not be confused by the extra space in the label, the result would be successful. A variation of this approach is indeed one of the mechanisms we tested below.

Unfortunately, HTML designers do not always match name attributes with corresponding labels as clearly as in the case above. Instead of using 'FirstName' and 'LastName' as name attributes within the input widgets, an HTML designer might have used 'fld1' and 'fld2'. The browser display would then have been identical. But in that case, the string matching technique would have failed. An automatic matching algorithm might instead make the match by blindly using the string on the left of each input element as the match. This approach sometimes does work. However, consider that certain labels, especially for the radio buttons and the checkboxes, may appear to the right of the form item. For example, consider Figure 3, which is an excerpt of the *garden.com* Web site.

We see in Figure 3 that the words "early", "mid", and "late" are each placed to the right of the input widgets they are supposed to label. Using a naive algorithm of left-association, we would associate the 'early' widget with "Spring", the "mid" label with the 'early' widget, etc.

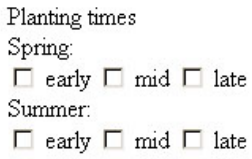

Figure 3: Text Labels Are Not Always Placed Before Their Input Widgets

Figure 3 illustrates another problem that often occurs with check boxes. In many forms, check boxes are grouped, and a common label describes the entire group. In Figure 3, the "Spring" label describes the three planting time seasons. In addition, each checkbox has its own label ("early", "mid", etc.). In the Figure, there are two such groupings, "Spring" and "Summer", which in turn have a common label ("Planting times"). Form summarization must attempt to detect and display such labels.

These challenges are just a sample of the difficulties faced in rendering forms and their labels on PDAs. We must contend with a design tradeoff: mismatches can lead to incorrectly transmitted forms. This danger would encourage us to include more, rather than less of the original textual information on the PDA page. On the other hand, the inconvenience and danger of user confusion associated with long PDA screens would will us towards careful pruning of the text that surrounds form widgets on the screen. This design tension can be resolved only by making the selection of text labels for input widgets as reliable as possible. We call the process of matching text labels with input widgets "form analysis."

We examined eight algorithms that can be used together to accomplish heuristic form analysis. Conceptually, each algorithm takes one HTML form as input. For each input element within that form, the algorithms produce as output one string excerpt from the form to use as an explanatory label. We describe these algorithms in Section 2. In Section 3 we describe the results of experiments we conducted to test each algorithm's performance. A discussion in Section 4 explains remaining failure modes, and our plans for addressing them.

2. HEURISTIC MATCHING ALGORITHMS

Each of our matching algorithms proceeds in two steps. The first step is common to all algorithms:

1. **Chunk Partitioning**: the entire HTML page is broken into "chunks." Chunks are small pieces of HTML code that are delimited by HTML tags, such as text paragraphs, table cells, or input elements. The result of this first step is an ordered list of all the chunks in the HTML page.
2. **Label Selection**: for each input element chunk within the chunk list, one text string from neighboring chunks is selected as the match that best describes the element's purpose.

Our eight algorithms differ in how they accomplish Step 2. We first describe chunk partitioning and then each of our algorithms in turn.

2.1 Chunk Partitioning

Chunks are identified by finding bounding HTML tags. For example, a paragraph of text that is delimited by a <p> tag is considered a chunk. Text strings delimited by table cell tags are also taken to be chunks, as are text within header tags, images, input elements, and horizontal rules (i.e. HTML-generated lines across the page).

A simplified chunk sequence (composed from a commercial gardening page) might look as follows:

```
1.  [<br>]
2.  ["Welcome to Your"]
3.  [<br>]
4.  ["Garden Environment"]
5.  [<br>]
6.  [<font SIZE="1" face="Arial,Helvetica">]
7.  [<br>]
8.  [<img src="flwr.jpg" alt="Flower">]
9.  [<p>]
10. ["Sun Exposure"]
11. [<SELECT name="SQL_LIKE_Optimum_Sun">]
12. [<option value="1">]
13. ["full shade"]
14. [<option value="2">]
15. ["partial shade"]
```

Each expression within square brackets constitutes one chunk. The initial string "Welcome to Your" is in a separate chunk from the string "Garden Environment" because a line break tag (
) separates the strings. Also, any HTML tag is in a separate chunk. For example, the "full shade" string in line 13 is a separate chunk from the 'option' chunks that precede and follow it. Incidentally, notice that chunks 11-15 will show on a browser screen as check boxes labeled "full shade" and "partial shade". Label "Sun Exposure" would describe these two check boxes as a group.

These sequences of chunks allow us to introduce the notion of distance within forms. For example, the string chunk "Sun Exposure" is one unit of distance away from the 'select' chunk on line nine. This distance notion is useful when delimiting the scope of a matching algorithm's activity. For example, our algorithms do not consider string chunks as matches, if they are more than *n* chunks away from the input element chunk they are examining. All of our algorithms use *n=10*, a value obtained empirically.

Each chunk is represented as an 8-tuple [C,T,Fid,LP,GP,M?,TM, Addl] with the following contents:

- **C**: Content of the chunk. For a text chunk, the content is the string itself. For an input element, the content includes the name attribute.
- **T**: Chunk type: string, input element, break, etc.
- **Fid**: Identifier of the form in which the chunk is embedded
- **LP**: Local position: chunk's distance from the beginning of the form
- **GP**: Global position: chunk's distance from beginning of Web page
- **M?**: Boolean indicating whether the chunk was already matched, and is therefore unavailable as candidate for further matches.
- **TM**: Table membership. If the chunk is embedded in a table, this field is a pointer into a table list (see below)
- **Addl**: Additional information, such as the values of input element attributes.

In addition to the chunk list, a list of HTML table chunk indexes are maintained (Figure 4).

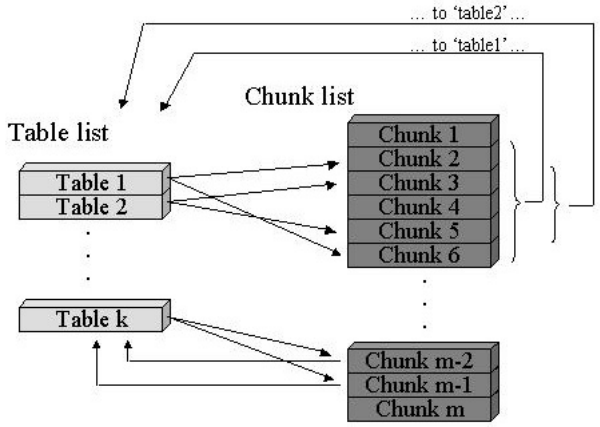

Figure 4: Chunk and Table Index Lists

Each entry in the table list identifies the range of chunks in that table. Notice that tables can be nested. For instance, in Figure 4, table 2 is within table 1. All chunks embedded within a table in turn identify the inner-most table they are in (TM field). For example, in Figure 4, chunk 3 would point back to table 2. For each table entry, a two-dimensional array is also maintained to track how chunks are laid out in the table. (These arrays are not shown in Figure 4.) As the HTML is scanned, the layout array is filled out, so that each entry in the array points to the first chunk (in the chunk list) that is located in that cell. This layout information is important for algorithms that consider page layout in their matching heuristics (see below).

In setting up the chunk list, the chunk partitioning algorithm discards some "stop phrases" that are known to make poor input widget labels, but that are commonly encountered in HTML forms. These stop phrases include "optional," "required," and the asterisk character, among others, which are often used to mark optional fields.

We eliminate these terms from the chunk list along with the terms containing text-formatting HTML tags, although they are still considered in determining the notion of distance. The final PDA screen layout manager might still decide to include these stop phrases on the display. But the phrases are guaranteed not to be the sole labels for input fields.

Incidentally, notice that the chunking strategy we have described here is different from the strategy used in [1, 5] to partition Web pages for display. Here, our chunks are used to determine how "far" (measured in chunks) potential labels are from input elements. In [1, 5], on the other hand, Web pages are partitioned into "semantic units" that users would like to see on their screens. Since the goals are different, the rules for partitioning are different.

2.2 Matching Algorithms

Once the chunk and table lists have been constructed, the first step of the matching procedure is complete. The lists are passed on to the

matching algorithms, which find each input element chunk in the chunk list, and attempt to find a good label for it. Our matching algorithms fall into two main categories. In the first category (Sections 2.2.1 through 2.2.3), the algorithms try to find labels that are textually "similar" to the variable name used in the input element. These algorithms yield a score that measures how similar the two strings are. In the second category (Sections 2.2.4 and 2.2.5), algorithms look at the form layout and search for labels that would be displayed close to the input element. These algorithms yield a score of 1 if a label is found, and 0 if not. Notice that the algorithms often yield different matching labels, with different scores. Later on in Section 4 we describe how the "predictions" of each algorithm may be combined into a more reliable one.

2.2.1 N-Gram

Our *N-gram* matching algorithm uses an information retrieval technique, called "n-gram matching." This technique was originally conceived to overcome unwanted text comparison mismatches when two words are compared that differ only by a small typographical error. In general, when comparing any two words, A and B through the n-gram technique, both words are divided into overlapping substrings of length n. This procedure results in two sets of equal-length substrings. The cardinality of the intersection between these two sets serves as a similarity score for A and B.

For example, consider the words "ants" and "grants" with $n=3$. This technique would generate tri-grams gra/ran/ant/nts from "grants," and ant/nts from "ants." The resulting matching score would be 2, from the common tri-grams "ant" and "nts." Notice that we do not perform any "normalization" to compensate for different string lengths. (That is, we do not divide the score by the sum of the number of n-grams, as is typically done.) We found the simple number of matching n-grams a better predictor for label/input-name matching, especially if the strings involved are of very different lengths.

We use n-grams to yield our *N-gram* algorithm in the following way.

1. For each input element chunk *C*, obtain the *s* chunks that precede, and the *s* chunks that follow *C*. The result of this step is a set of up to 2*s* chunks that are *C*'s neighbors. There may be fewer than 2*s* neighbors for input elements that are closer than distance *s* from the top or bottom of the page that contains the element. The optimal span *s* was empirically determined to be 10.
2. Obtain the name attribute of chunk *C* (from its content field).
3. For each neighbor chunk *N* of Step 1:
   a. If *N* is not a string chunk, discard *N*.
   b. Canonicalize *N*'s string content by removing all spaces and non-alpha characters.
   c. Perform an n-gram match between *C*'s name attribute (from Step 2), and *N*'s canonicalized string content.
4. Choose (the string content of) the neighbor chunk with the highest n-gram score as the best match, provided that this score is greater than or equal to a constant threshold *T*. This constant is the lowest score that can still be considered significant enough to be a match. Empirical methods led us to set $T=2$. If no score meets the threshold, then the algorithm reports failure (score = 0).

The literature suggests that tri-grams ($n=3$) tend to work well for successful matching [2], and we use tri-grams for our *N-gram* algorithm.

2.2.2 Letter/Word and Word/Letter

The *Letter/Word* algorithm is a variation of the *N-gram* algorithm. The difference lies in the canonicalization Step 3b. Rather than removing spaces and non-alpha characters, the first two words of chunk *N*'s string content are extracted. Any remaining words in the string are discarded. The first letter of the first word is then concatenated with the second word. Thus, "First Name" would become "FName," and "Work Phone" would become "WPhone." If *N*'s string contains fewer than two words, the *Letter/Word* algorithm is not applicable. We developed this algorithm, because we noticed that such contractions are frequently used by HTML designers.

Word/Letter is the dual of *Letter/Word*. For example, "Min Height" would become "MinH," and 'Min Width' would be transformed to "MinW."

2.2.3 Substring

The *Substring* algorithm is another variation on the *N-gram* algorithm. It is designed to match words in which one word is a contraction of the other. For example, the *Substring* algorithm would successfully match "password" with 'pwd', or "height" with 'hgt'.

The modification to the *N-gram* algorithm occurs in Steps 3b and 3c. The canonicalization Step 3b is replaced with the following procedure. After removing spaces and non-alpha characters from neighbor N, the lengths of N and the input element chunk's name attribute (*C*) are compared, and their ratio is recorded. For example, consider a input element's name attribute 'pwd', and a neighbor string chunk "Password." Comparing their lengths, the algorithm would obtain the approximate ratio $R=3:1=3$.

The modified Step 3c then partitions the longer of *N* and *C* into *R* substrings of roughly the same length. The shorter word is partitioned into *R* letters. For example, "Password" would be partitioned into "Pas," "swo," and "rd." The shorter string, "pwd," would be partitioned into its three component letters.

Finally, the *Substring* algorithm tests for the presence of each of the R letters in their corresponding substrings. Continuing with our example, the presence of "p" in "Pas," "w" in "swo," and "d" in "rd," leads to three matches, or a raw score of three.

2.2.4 Tables

HTML forms make heavy use of tables as a visual layout tool. Visual layout of form items on the page are a very strong clue that human users pay attention to when performing the matching task we are trying to accomplish automatically. Tables can force elements to be grouped visually on the page, to be stacked on top of each other, or to be lined up neatly in a row (Figure 5). HTML pages without tables are much less predictable in their appearance to the eventual human user. Our *Table* algorithm attempts to emulate how human onlookers tend to match text labels with input widgets on a screen.

Recall that the table and chunk lists enable matching algorithms to determine where input elements are located within a table. When

trying to match an input element chunk *C* that is located within a table, the table algorithm first checks whether a string chunk is located within the same table cell with *C*. In Figure 5, this condition is true, for instance, in the case of the 'E-mail address' text widget in Figure 5, but not for the text widget intended for entering a password. If a string chunk is located within the same cell with *C*, that string is chosen to be the label for *C*.

If no string is found within the same cell, the *Table* algorithm checks whether a string chunk is located in a cell immediately to the left of the input element it is trying to match. If such a string is present, it is chosen as the match. If no such string is located to the left of *C*, the reason could be that *C* is the left-most cell of the table, or that the cell on the left does not contain a string. In this case, the table algorithm checks first the cell above, then the cell below, and finally the cell to the right of *C*, choosing the first string it finds in that sequence of trials. If a label is found, the algorithm yields a score of 1. Otherwise, if the Table algorithm fails, it reports a score of 0.

2.2.5 Previous and Following

When choosing a text label match for an input element chunk *C*, the Previous algorithm examines in order up to *s* chunks that precede *C*. As soon as a string chunk is found, that chunk's string content is chosen as the match for *C*. If no string is found within span *s*, then the Previous algorithm reports failure.

The *Following* algorithm similarly examines up to *s* chunks that follow C.

Experimental results made us choose 10 as the value of the matching span *s*.

2.2.6 NULL Algorithm

The *NULL* algorithm simply returns the value of an input element's 'name' attribute as that input element's match. (The algorithm always returns a score of 1.) For example, consider the task of matching the text input element in the following chunk sequence:

```
["Please enter your electronic mail address."]
[<br>]
[<input type="text" name="email">]
```

The *NULL* algorithm would choose "email" as the matching text label, no matter which strings are found in neighboring chunks.

2.3 Finding Group Labels

As mentioned in Section 1, groups of checkboxes often have one label for the entire group, and one label for each checkbox. Consider the following example:

```
<form ...>
"Which plants do you like?"<br>
<input type="radio" name="plants" value="tul">
Tulips<br>
<input type="radio" name="plants" value="rose">
Roses<br>
<input type="radio" name="plants" value="beg">
Begonias<br>
"What did we forget?"<br>
<input type="text" name="other_plant">
</form>
```

Please log on before using the system.		
Login:	E-mail address	
Password:		

Figure 5: Positioning Form Elements With Tables

This HTML snippet would display "Which plants do you like?", followed by three radio buttons, labeled "Tulips", "Roses", and "Begonias", respectively. A text input box labeled "What did we forget?" would be placed below the radio buttons. The name attribute "plants" is common to all radio buttons. This property identifies the three buttons as a group. This grouping ensures that only one of the radio buttons is checked at any time. When rendering grouped buttons on a PDA, a screen layout manager might also wish to place them in close proximity to each other.

Notice that the *N-gram* algorithm would easily identify the labels for the radio buttons. But we also need to ensure that the group as a whole is labeled. If the PDA display were to discard the question about plant preference, then the radio buttons, even with their labels, would not make sense to the user.

Before finding individual labels for grouped checkbox and radio button input elements, our system therefore attempts to identify a group label for each group. The algorithm for finding group labels is as follows.

1. We apply the *N-gram* algorithm to the 'name' attribute value of the first button in the group. In this example, the 'name' attribute has the value "plants". The best matching label in this case is "Which plants do you like?" with a score of 4.
2. We apply the *N-gram* algorithm to the first button's 'value' attribute (in this case "tul"). In this case, there is no matching label nearby, so the score is 0.
3. The highest-scoring match is taken to be the label for the button group. This example would correctly yield "Which plants do you like?" as the label for all buttons with 'name' attribute "plants".

Notice that the group label is now "used up." The respective text chunk should not subsequently be identified as the label for some individual input element. For example, the text box at the bottom of the above HTML snippet contains 'name' attribute "other_plant". This name would match well with the "Which plants do you like?" string, but that match would be disallowed. Thus, the *N-gram* algorithm would find no match for the bottom text box. However, the Previous algorithm would find the correct string "What did we forget?" To register "used up" labels, each chunk record includes a Boolean (called M? in section 2.1) which is set to TRUE when a chunk is no longer available for matching.

3. PERFORMANCE EXPERIMENTS

In an initial experiment, we tested the effectiveness of our algorithms on 100 forms. The forms were selected at random. In particular, we selected random Web pages from a cache of 40 million pages. If a page did not have a form, it was discarded; else it was kept. If a page was located at the same site as a previously selected page, it was discarded, so that HTML design conventions

on any one site would not bias our performance studies. In the 100 forms we collected, there were an average of four input elements. We call these forms our "simple" forms, to distinguish them from the "complex" forms used for our second experiment.

For each of the input fields in the simple forms, a human being manually matched the correct text label. We then applied all of our matching algorithms to each input element. As mentioned, we used a maximum chunk span of 10, and an n-gram matching threshold of 2. That is, we only considered string chunks within a neighborhood of plus/minus 10 as candidates for input element labels, and we required a score of at least 2 for algorithms that involve n-gram matching.

Once all matching scores for a given input element had been computed, we took the match with the highest score as the "winner." In this first set of experiments, we made no attempt to normalize the scoring mechanism across the algorithms. Thus, for example, a successful *N-gram* algorithm (with its score greater than *T=2*) will always beat a table algorithm (with maximum score of 1). (A better scheme to combine algorithms is described in Section 4.)

Finally, we compared the winner with the correct answer. We found that for this initial large batch of forms, our matching success was 95%. A large number of forms on the Web that one might wish to fill out on a PDA are as simple as the forms we matched in this first experiment. Examples are search forms, and simple order forms.

Nevertheless, the high success rate with simple forms encouraged us to tackle more complex forms. As the base data for two additional experiments, we therefore manually selected 21 Web sites that contained 48 fairly complex forms with between two and 30 elements. All forms taken together, this collection included 330 input elements, of which 307 were embedded in HTML tables. Of all input elements, 40% were one-line text input fields, 2% were password input fields, 22% were select fields (pull-down lists of choices), 5% were editable, multi-line text areas, 15% were radio buttons, and 16% were checkboxes.

For each input element in each form, we again manually found the correct labels. Then we ran two experiments on these complex forms. The first experiment aimed at measuring the effectiveness of each individual algorithm when used by itself. The second experiment examined the effectiveness of combining the algorithms with a score maximization scheme. This second experiment is described in Section 4.1.

For the first experiment, we used 115 of our 330 input elements as the test set. We ran all of our algorithms on each of the 115 input elements. We recorded how many times each algorithm believed that it had generated a successful match, and how often this match was correct, as compared with the human-generated match.

We also recorded failure reasons when none of the algorithms succeeded.

3.1 Experimental Results

Table 1 shows the results of our 115 element, individual algorithms experiment on complex forms.

Each row of Table 1 describes the performance of our algorithms for the forms on one Web site. The left-most column summarizes each site's statistics. The first number in parentheses is the number of forms on the site. The second number is the total number of input elements to be matched on these forms. For example, on site 2, we tested our algorithms on four forms that contained a total of 12 input fields. We include these numbers to show how the forms were spread across different sites, and how HTML conventions on any one site might influence the algorithms.

Each of the remaining eight columns contains the results of one algorithm. The first of the two numbers in each table cell indicates the number of times the respective algorithm produced a correct match. The second number indicates how often the algorithm concluded that it had a match. For example, within site 3, the *N-gram* algorithm "thought" nine times that it had a match, and its match was correct for seven of these nine matches. Given that site 3 contained a total of 17 input elements, we can see that for eight input elements the *N-gram* algorithm decided to pass on matching. Such a pass condition occurs when an algorithm's matching score is below the matching threshold.

The bottom of each column shows four summary results for the respective algorithm. The first result is the total of the column. In the *N-gram* column, for example, we see that when confronted with our 115 input elements to match, the algorithm concluded that it found a

Table 1: Each Algorithm Matching 115 Input Fields

Site (Forms)	N-Grams	Letter/Word	Word/Letter	Substring	Tables	Previous	Following	Null
1 (1,26)	9/9	1/1	0/0	2/2	17/25	15/26	0/26	6/26
2 (4,12)	5/5	0/0	0/0	3/3	0/3	5/12	0/12	6/12
3 (1,17)	7/9	0/0	0/0	4/4	0/0	11/17	0/17	2/17
4 (3,8)	8/8	2/2	0/0	7/8	6/8	8/8	0/8	8/8
5 (1,21)	11/12	1/1	0/0	11/12	10/13	14/21	5/21	7/21
6 (1,7)	0/0	0/0	0/0	0/0	6/7	3/7	1/7	0/7
7 (1,13)	13/13	0/0	0/0	6/6	0/0	9/13	4/13	10/13
8 (1,11)	4/4	0/0	0/0	4/4	9/9	6/11	0/11	6/11
TOTALS	57/60	4/4	0	37/39	48/65	71/115	10/115	45/115
Success	49%	3%	0%	32%	42%	62%	9%	39%
Failure	3%	0%	0%	2%	15%	38%	91%	61%
Pass	48%	97%	100%	66%	43%	0%	0%	0%

match in 60 of the cases. The algorithm produced the correct match in 57 of these cases.

The second summary is the success rate of the respective algorithm when measured against the entire collection. The *N-gram* algorithm's 57 successful matches, for example, translate to a (approximate) 49% success rate over the 115 input elements.

The third total is the failure rate. It shows how often the respective algorithm produced an incorrect result. For example, when processing the 115 input elements, the *N-gram* algorithm produced 3 incorrect results (60-57), amounting to a failure rate of 3%.

The fourth total, finally, is the algorithm's "pass" rate. It shows how often the algorithm did not venture a match at all. Notice that the *Previous* and *Following* algorithms can almost always be applied. The only exceptions occur when an input element has no string chunk to its left or right, respectively. The *NULL* algorithm is always applicable. This near-universal applicability of the three algorithms is reflected in their pass rate of zero.

4. DISCUSSION

We see from Table 1 that Previous has a high success rate, as one might expect. Notice, however, that it also features a high failure rate. The *N-gram* algorithm, while featuring a lower success rate, also fails much less. When comparing the success/failure rates, *N-gram* stands out with a ratio of 17. The *Letter/Word* and *Word/Letter* algorithms did not make any mistake in this data set, but most of the time they were not applicable.

Upon conclusion of the experiments, we analyzed the most frequent failure modes, and thought about how our various algorithms might be combined to form an optimized matching system. The following sections discuss these issues.

Table 2: Matching Performance for Algorithm Combination over 330 Input Elements

Site (Forms)	N-Grams	Letter/Word	Word/Letter	Substring	Tables	Previous	Following	Null	Total
1 (1,26)	9/9	0/0	0/0	0/0	12/17	0/0	0/0	0/0	21
2 (4,12)	5/5	0/0	0/0	0/0	0/3	2/3	0/0	1/1	8
3 (1,17)	7/9	0/0	0/0	0/0	0/0	7/8	0/0	0/0	14
4 (3,8)	6/6	2/2	0/0	0/0	0/0	0/0	0/0	0/0	8
5 (1,21)	11/12	1/1	0/0	0/0	3/6	2/2	0/0	0/0	17
6 (1,7)	0/0	0/0	0/0	0/0	6/7	0/0	0/0	0/0	6
7 (1,13)	13/13	0/0	0/0	0/0	0/0	0/0	0/0	0/0	13
8 (1,11)	4/4	0/0	0/0	0/0	7/7	0/0	0/0	0/0	11
9 (2,18)	12/13	0/0	0/0	0/1	0/3	0/0	1/1	0/0	13
10 (1,15)	3/3	0/0	0/0	0/0	0/0	0/1	11/11	0/0	14
11 (8,11)	1/1	0/0	0/0	0/0	0/0	4/8	0/0	2/2	7
12 (7,64)	30/42	2/2	0/0	0/5	0/0	10/12	0/0	2/3	44
13 (3,17)	8/8	0/0	1/1	0/0	2/6	0/0	0/0	0/2	11
14 (1,30)	22/22	0/0	0/0	0/0	4/6	2/2	0/0	0/0	28
15 (4,4)	4/4	0/0	0/0	0/0	0/0	0/0	0/0	0/0	4
16 (1,4)	0/1	0/0	0/0	0/0	3/3	0/0	0/0	0/0	3
17 (2,4)	2/2	0/0	0/0	0/0	1/2	0/0	0/0	0/0	3
18 (2,25)	12/12	0/0	0/0	0/0	8/11	0/0	0/0	0/2	20
19 (2,3)	2/3	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2
20 (1,13)	6/6	0/0	2/2	0/0	3/5	0/0	0/0	0/0	11
21 (1,8)	6/8	0/0	0/0	0/0	0/0	0/0	0/0	0/0	6
TOTALS	163/183	5/5	3/3	0/6	49/76	27/36	12/12	5/10	264/330
Success	49%	2%	1%	0%	15%	8%	4%	1.5%	80%
Failure	6%	0%	0%	2%	8%	3%	0%	1.5%	
Not Applied	45%	98%	99%	98%	77%	89%	96%	97%	

4.1 Common Breakdowns

The three main failure reasons in our matching scheme were overly limited scope, image faults, and group labels for radio buttons or check boxes. An example of overly limited scope would occur in the following case (Figure 6):

Figure 6: Limited Scope Failure

The corresponding simplified HTML looks as follows:

```
<tr>
<td>Full Name <br>(e.g. John H. Doe)</td>
<td><input type="text" name="fname"></td>
</tr>
```

Our *Table* algorithm would note that the input element is in a cell all by itself. It would therefore retrieve the chunk to the left of the element. Since "First Name" and "e.g. John H. Doe" are separated by a break tag (
), our chunk partitioning procedure will separate the two pieces of text into two chunks. The label chosen for the input field would thus incorrectly be chosen to be "e.g. John H. Doe", which is much less desirable than the obviously correct match "First Name". In our algorithm combination experiment (see below), about 7% of our matches failed due to a limited scope breakdown.

An image fault occurs when the correct match is text within an image. We currently do not perform optical character recognition (OCR) on images. None of our algorithms will therefore find such text. About 3% of our matches failed due to image faults.

A radio button error occurs when groups of radio buttons do not have a common label, or when they are missing labels for the individual buttons. As described, our algorithms process radio buttons by matching the first button's 'name' and 'value' attributes to surrounding text chunks. This process attempts to find a group label for all the buttons, as well as an appropriate label for each individual button. Many radio buttons and checkboxes are indeed organized in groups with such a group label. However, when there are no group or individual labels, our system tends to produce mismatches as it insists on finding these labels. Figure 7 shows an example for missing individual labels. None of our algorithms finds a label for the drop-down menus, since there are no labels to choose from. However, notice that even though we label this instance as a "failure," in reality the user will see the menu values in the widget displayed on the PDA, and will be able to make a selection just as with a full display.

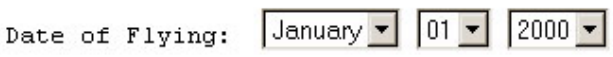

Figure 7: Missing Individual Labels

A dual case occurs when text or select fields are grouped, as might be the case with an HTML form for inputting postal addresses. Figure 8 shows an example.

The text "Home Address" is a group label for the following input fields. Each input field has its own label. A PDA display that filters

Figure 8: Text Field Group With Group Label

out the "Home Address" group label would leave the user wondering which address is being requested. Unfortunately, contrary to groups of radio buttons, groups of text fields are not identified in the HTML code by common input element 'name' attribute values. Automatic analysis therefore easily misses such groupings. We have so far not systematically addressed this problem of group labels for text fields in our system. In some cases, special circumstances cause us to produce good results anyway. We will not go into detail on these cases.

Among these group-related failures, the missing group label for check box groups is the most common. Our 330 input element test set included 102 radio buttons and check boxes. Of these, 14 were missing a group label (14%). Of our 203 text fields, 15 (7%) had group labels, of which we missed nine. Overall, about 48% of our matching failures were due to grouping problems.

To conclude our discussion of failure modes, we observe that failures will not be catastrophic. If a user is confused by the labels displayed on the PDA (or is puzzled by the results obtained after filling out the form), he can switch the display to a full HTML rendering. As an intermediate option, we plan to provide progressive disclosure of text around the input field. Transmitting the full page to the PDA, and scrolling through the full page, will be more time consuming than manipulating our "synthesized" forms, but will be an option. Of course, keep in mind that even the full HTML rendering may be error prone on a small display. For example, if the complete form is not visible on the screen, the user may get disoriented. In Figure 3, for instance, if all the checkboxes cannot be seen, the user may not be able to tell if a label corresponds to the checkboxes on its left or on its right.

4.2 Combining Matching Algorithms

Observe in Table 1 that in many cases, multiple algorithms come up with the correct answer. For example, the total number of successful matches in row 1 of Table 1 is 50, significantly more than the 26 input fields to be matched. At the same time, we find cases when one algorithm performs significantly better than others. For example, in row 6, the *N-gram* algorithm features no match at all, while the *Table* algorithm successfully matches six of the seven input elements.

These observations raise the question of how best to combine the algorithms such that overall matching performance is optimized. Several strategies for combination are available, and in particular we empirically developed one that orders the strategies according to their observed usefulness.

With this simple score maximization strategy, algorithms *N-gram*, *Letter/Word*, *Word/Letter*, *Substring*, and *Tables* are run, one at a time, for each input element. If any of these algorithms produces a match for an input element, we select the highest-scoring match as the winner. If the first five algorithms did not produce a match, we attempt *Previous*, *Following*, and *NULL* in order, picking the first match we find. We isolated Previous and Following from the other

five algorithms in this way, because of their high failure rate. The NULL algorithm is the last in the application sequence, because it can always be applied, and therefore does not have a "natural" ability to be discriminating. Also, recall that the label this last algorithm produces is the internal HTML input element name attribute, which is not normally intended for display on the screen. We therefore gave lowest priority to this algorithm.

Table 2 shows the results of our combined strategy running over the full 330 input element data set.

Table 2 is organized similarly to Table 1. The differences are as follows. On the far right, a new column has been added. It lists the total number of input elements that were successfully matched in each row. Thus, in row one, 21 of the 26 input elements were successfully matched by the combined strategy. At the lower right of Table 2, the total matching result is 264/330, or 80%.

Notice that, in contrast to Table 1, there is no overlap in the success numbers for each algorithm. For example, in the first row, the *N-gram* algorithm's nine successful matches are distinct from the 12 matches reported for the *Table* algorithm. For any given input element, only one of the algorithms is thus ever credited for a successful match.

We see that the *N-gram* algorithm provides the lion's share of all successful matches (49%). The *Table* algorithm is the second-most successful, with a failure rate comparable to *N-gram*. *Letter/Word*, *Word/Letter*, and *Following* are "safe" algorithms to try, in that they did not produce failures, even though their success rate was small as well.

While this optimization strategy works reasonably well for complex forms, and quite well for simple ones, it is an open question whether some other strategies might perform better. We could, for example, weight the results of each algorithm, depending on the algorithm's success/failure ratio. The results of high-ratio algorithms would carry more votes in the voting process than results from low-ratio algorithms. We plan to continue our experiments, especially the development of these strategies for optimizing overall performance.

5. CONCLUSION

Our goal is to automatically and dynamically "summarize and organize" Web pages for display on small hand-held devices. In this paper we have focused on how best to display forms on these devices. In particular, we have shown various strategies for selecting descriptive labels for the various input elements that appear on the summarized forms. Our experiments show that in the vast majority of cases our algorithms find labels that a human would have selected, thus making summarized form display feasible and attractive.

Of course, there are still ways to improve on our results. For example, our syntactic and structural feature analysis can be improved. In particular, we are extending our approach to take into account occurrences such as colons at the end of text strings, as well as parenthetical expressions. We might, for example, favor text chunks with trailing colons if these chunks immediately precede an input field. At the same time, we might lower the matching eligibility of any text chunks that are parenthesized.

As Web page designs grow more elaborate, labels in images will be more frequent. As reported above, we currently have no facilities for detecting images that are used as input field labels. As a first step, we will use ALT tags as substitutes for text in images. An ALT tag is a word or phrase that is associated with an image link in an HTML page. This information is displayed to users while the image is being loaded and is not yet visible on the user's screen. Beyond the use of ALT tags, we plan to examine how well we can do when we perform optical character recognition on the images themselves. We will focus on images that are in page layout positions often occupied by labels.

6. REFERENCES

[1] Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Accordion Summarization for End-Game Browsing on PDAs and Cellular Phones. In Proceedings of the Conference on Human Factors in Computing Systems, 2001.

[2] R.C. Angell, G.E. Freund, and P. Willett. Automatic Spelling Correction Using a Trigram Similarity Measure. Information Processing and Management, 19(4):255-261, 1983.

[3] Orkut Buyukkokten, Hector Garcia Molina, Andreas Paepcke, and Terry Winograd. Power Browser: Efficient Web Browsing for PDAs. In Proceedings of the Conference on Human Factors in Computing Systems, 2000.

[4] Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Focused Web Searching with PDAs. In Proceedings of the Ninth International World Wide Web Conference, 2000.

[5] Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Seeing the Whole in Parts: Text Summarization for Web Browsing on Handheld Devices. In Proceedings of the Tenth International World-Wide Web Conference, 2000.

[6] ProxiNet. ProxiWeb. ProxiNet website: http://www.proxinet.com/.