Efficient Web Browsing on Handheld Devices Using Page and Form Summarization

Orkut Buyukkokten, Oliver Kaljuvee, Hector Garcia-Molina, Andreas Paepcke, Terry Winograd

Digital Libraries Lab(InfoLab), Stanford University

We present a design and implementation for displaying and manipulating HTML pages on small handheld devices such as personal digital assistants (PDAs), or cellular phones. We introduce methods for summarizing parts of Web pages and HTML forms. Each Web page is broken into text units that can each be hidden, partially displayed, made fully visible, or summarized. A variety of methods are introduced that summarize the text units. In addition, HTML forms are also summarized by displaying just the text labels that prompt the use for input. We tested the relative performance of the summarization methods by asking human subjects to accomplish single-page information search tasks. We found that the combination of keywords and single-sentence summaries provides significant improvements in access times and number of required pen actions, as compared to other schemes. Our experiments also show that our algorithms can identify the appropriate labels for forms in 95% of the cases, allowing effective form support for small screens.

Keywords

Personal Digital Assistant, PDA, Handheld Computers, Mobile Computing, Summarization, WAP, Wireless Computing, Ubiquitous Computing, Forms, WML

1. INTRODUCTION

Wireless access to the World-Wide Web from handheld personal digital assistants (PDAs) is an exciting, promising addition to our use of the Web. Much of our information need is generated on the road, while shopping in stores, or in conversation. Frequently, we know that the information we need is online, but we cannot access it, because we are not near our desk, or do not wish to interrupt the flow of conversation and events around us. PDAs are in principle a perfect medium for filling such information needs right when they arise.

Unfortunately, PDA access to the Web continues to pose difficulties for users [16]. The small screen quickly renders Web pages confusing and cumbersome to peruse. Entering information by pen, while routinely accomplished by PDA users, is nevertheless time consuming and error-prone. The download time for Web material to radio linked devices is still much slower than landline connections. The standard browsing process of downloading entire pages just to find the links to pursue next is thus poor for the context of wireless PDAs.

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 [23]. 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].

We have been exploring solutions to these problems in the context of our Power Browser Project [3,4,5,6]. The Power Browser provides displays and tools that facilitate Web navigation, searching, browsing, and input entry from a small device. The browser uses a very different approach to support navigation and the 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 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]. 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. However, once users approach the page of interest, seeing only the links on each page is insufficient. During this ‘end-game’ phase of the information task, more of each page must be revealed, without overwhelming the screen. A special case arises with HTML forms, such as product orders, search panels, or online hotel registration facilities. Such forms are often bulky and consume more screen real estate than is necessary while users study what information is being requested of them. This article addresses the problems of Web page summarization and the summarization of HTML forms for small devices. We will show that, while the necessary system support is very different for page and form summarization, the two can be presented to users in an integrated interface.

Figure 1: Navigation Screenshot of Power Browser

2. PAGE SUMMARIZATION

The page summarization facility is employed after a user has searched and navigated the Web, and wishes to explore in more detail a particular page. At this point, the user needs to gain an overview of the page, and needs the ability to explore successive portions of the page in more depth. Figure 2a shows as an example the Palm operating system’s home page. This page would require extensive scrolling if displayed on a PDA. Instead, our proxy server, in collaboration with the PDA, provides two levels of summarization: a macro level, and a micro level.

2.1 Macro-Level Summarization

The proxy begins by partitioning the page into 'Semantic Textual Units' (STUs). The result of this step is shown by the rectangles around sections of the page in Figure 2a, which are not present in the actual HTML. In summary, STUs are page fragments such as paragraphs, lists, or ALT tags that describe images.

In a second step, the proxy then uses font and other structural information to identify a hierarchy of STUs. For example, the elements within a list are considered to be item STUs nested within a list STU. Similarly, elements in a table, or frames on a page, are nested. Hiding the nested STUs finally completes macro summarization. Figure 2b exemplifies the result. Initially, each STU is represented by a single line on the PDA screen. The arrows in Figure 2 denote the correspondence of STUs with their single-line representations. (The line numbers on the PDA are only for convenience of explanation and do not appear on the actual display.)

The ‘+’ symbols next to some STU representations indicate the presence of hidden STUs at a deeper nesting level. Users may view STUs at those deeper levels by tapping their pen on the appropriate ‘+’ symbol, or by performing a left-to-right pen gesture. In response, the ‘+’ turns into a ‘-‘ symbol, and the nested STUs are displayed indented. For example, the STU of line 4 in Figure 2b has been expanded, revealing lines 5-9. Then the STU of line 9 was expanded to reveal lines 10-13. The STU of line 3 has not been expanded, hence the '+' on that line. Initially, only the top level of the STU hierarchy is shown on the screen. In Figure 2b this top level consists of four STUs in lines 1-4. Lines 5-13 were initially blank.

Note that this macro level summarization does not require special formatting at the Web sources. This freedom from intrusion is a significant advantage of our approach over schemes that rely on pages to be specially structured for PDAs. Reference [5] provides more detail on how STUs are extracted from pages, and how they are ordered into a hierarchy.

2.2 Micro-Level Summarization

Notice that each STU is initially "truncated" and displayed in a single line. This is one example of our micro-level summarization. In Figure 2b we only see the first portion of each STU's first sentence. If an STU contains more text, a 'line marker' (black bubble) indicates that more information is available. For example, the STU of line 6 only shows the text "The Palm m100 handheld is the f". The user can progressively open the STU by tapping on the bubble marker (see Figure 3). In particular, after the first tap, the first three lines of the STU

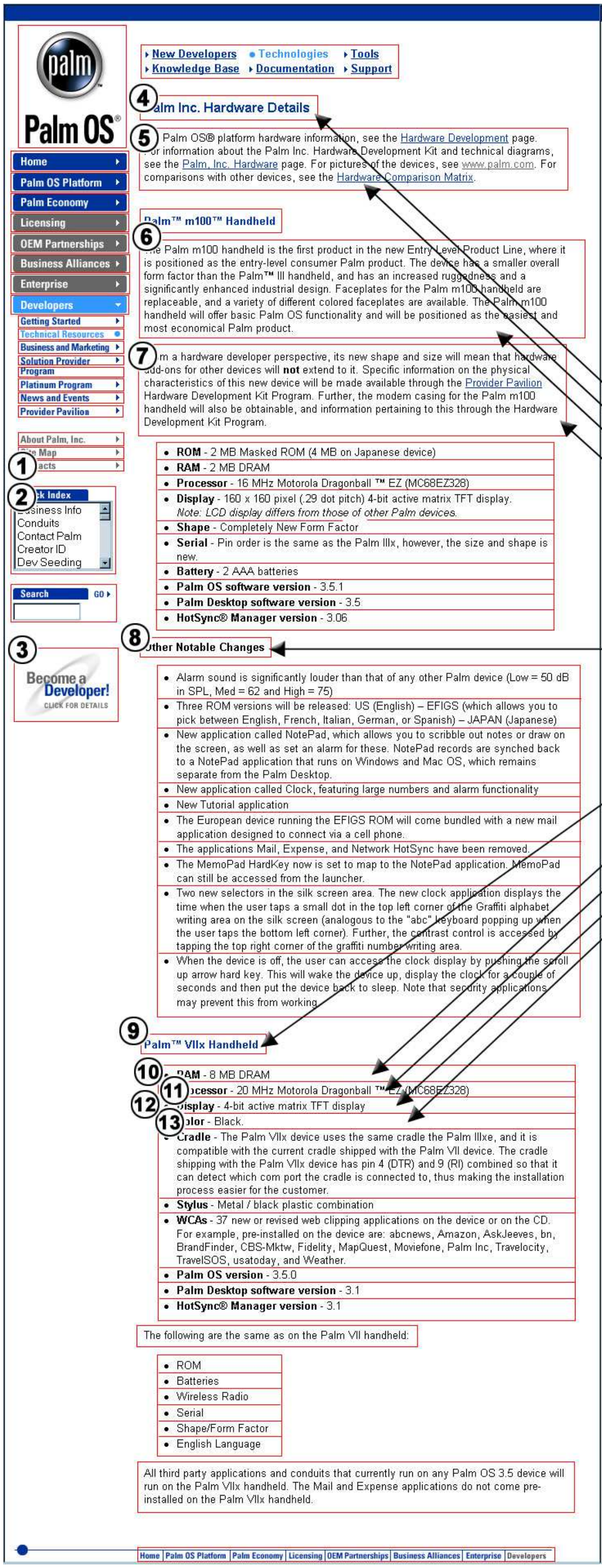

Figure 2a: Palm OS Hardware Web Page

Figure 2b: Screenshot of Same Page with Power Browser

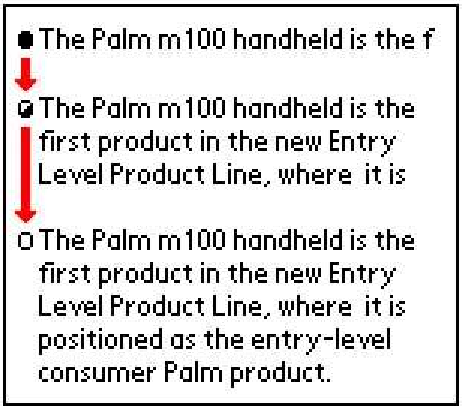

Figure 3: An STU Progressively Displayed in Three States

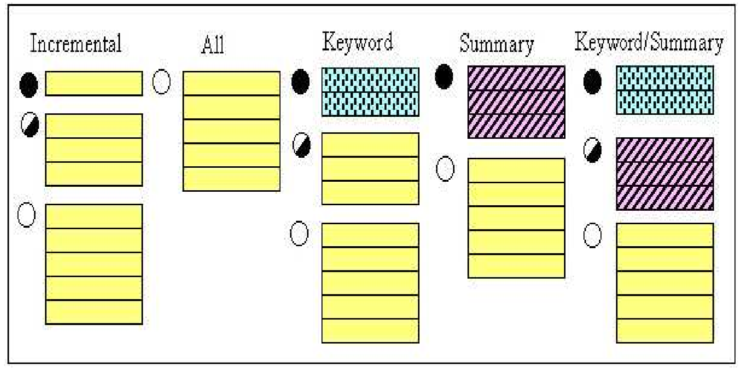

Figure 4: Five Methods for Progressively Disclosing STUs

are shown. A half-empty line marker signals that text is still available. A second tap reveals all of the STU. In this case, an empty line maker indicates that the entire STU is revealed. The system thus reveals each STU in up to three display states (two if the STU was smaller than or equal to three lines, or one state if the entire STU fits on a single line).

With our two-level summarization, an STU like the one in line 7 of Figure 2b may potentially be "opened" in two ways: tapping the bubble reveals its textual content (e.g., text in a paragraph), while tapping on its '+' symbol reveals nested STUs (e.g., list items under this paragraph). Using this two-level, *'accordion'* approach to Web browsing, users can initially get a good high-level overview of a Web page, and then "zoom into" the portions that are most relevant.

The micro-level summarization scheme of Figure 3 is simple to implement, but its effectiveness is limited. The first sentence of a paragraph is not necessarily the best representation of the paragraph's content. It is important that users open as few irrelevant STUs as possible. The micro-level summarization scheme is therefore key for ensuring optimal user performance.

We have explored five methods for micro-level summarization, and performed user testing to learn how effective each of them are in helping users solve information tasks on PDAs quickly. All of the methods we tested retain our macro-level accordion browser approach of opening and closing large structural sections of a Web page. However, the methods differ in how they summarize and progressively reveal the STUs at the micro-level.

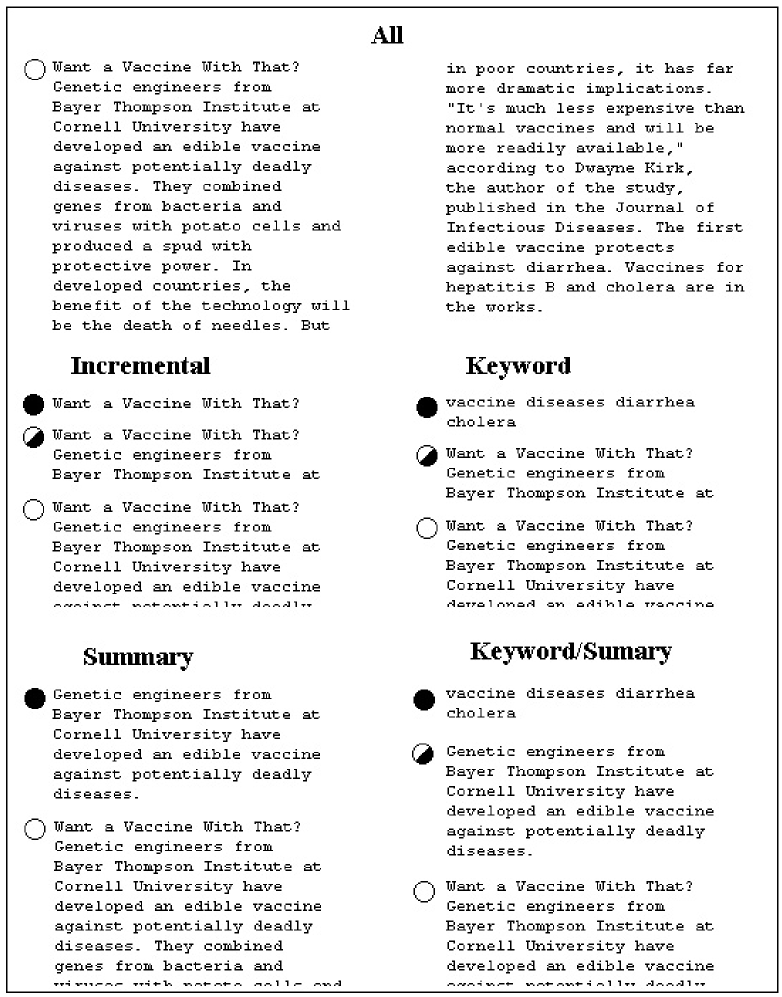

Figure 5: Examples for Each Progressive Disclosure Method

Every method we tested displays each STU in several states, just as described above for the simple case. But the information for each state is prepared quite differently in each method. All displays are textual. That is, none of the STUs displays images. (Existing work on image compression for other PDA browsers [13], has not yet been incorporated into our browser.) The methods we tested are illustrated in Figure 4. They work as follows:

• *Incremental*: The first method is the same as described above: each STU is revealed gradually in three states; the first line, the first three lines and the whole STU.

• *All*: This display method shows the text of an entire STU in a single state. No progressive disclosure is enabled.

• *Keywords*: The third method displays in its first state the 'important' keywords that occur in the STU. We will describe below how we determined which of the STU's words are considered important keywords. We show all of the keywords on the display, even if they extend beyond a single line and wrap down to additional lines. The second state shows the first three lines of the STU. The third state shows the entire STU.

• *Summary*: This method consists of only two states. In the first state the STU's 'most significant' sentence is displayed. The second state shows the entire STU. We describe below how significant sentences are selected.

• *Keyword/Summary*: This method combines the previous two methods. The first state shows the keywords. The second state shows the STU's most significant sentence. Finally, the third state shows the entire STU.

There are of course many other ways to mix keywords, summary sentences, and progressive disclosure. However, in our initial experience, these 5 schemes seemed the most promising, and we hence selected them for our experiments. Also note that in all of these methods, only one state is used if an entire STU happens to fit on a single line. Similarly, if an STU consists of only one sentence, the most significant sentence is the entire STU and there are no additional state transitions.

Figure 5 shows an example that applies all five methods to one STU on *www.onhealth.com*. The *ALL* method at the top of Figure 5 is shown in two columns for reasons of presentation in this publication only. On PDAs and cellular phones, the display is arranged as a single column. The *ALL* method displays all of the STU's text. The empty line marker on the left indicates to the user that the STU cannot be expanded further.

All of our states, except *Keywords*, display hyperlinks when encountered. For example, if a summary sentence contains a link, it is displayed underlined and is active. If the user clicks on the link, the top-level view of the new page is shown. In the *Incremental* method, if the link starts at the end of a truncated line, the visible portion of the link is shown and is active. With *Keywords* summarization, no links are displayed, even if a keyword is part of some anchor text. In this case we felt that a single keyword was probably insufficient to describe the link. Furthermore, making a keyword a link would be ambiguous when the new keyword appears in two separate links.

Stepping back, Figure 6 shows how users' requests for Web pages are processed, and how summarized pages are generated. The components of Figure 6 are located in a Web proxy through which Web page requests from PDAs are filtered. We will provide detailed explanations for the dark-colored components in subsequent sections. The User Manager keeps track of PDA user preferences (e.g., preferred summarization method, timeout for downloading Web pages), and of information that has already been transmitted to each active user's PDA. This record keeping activity is needed, because the proxy acts as a cache for its client PDAs. Once a requested Web page, possibly with associated style sheet, has been downloaded into the proxy, a Page Parser extracts all the page tokens. Using these tokens, the Partition Manager identifies the STUs on the page, and passes them to the Organization Manager, which arranges the STUs into a hierarchy. In Figure 2, the results of the Organization Manager's work are the entries that are preceded by the '+' and '-' characters.

If a Web page contains a form, the form-related tokens are passed from the Page Parser output to the Form Processor, where the tokens are analyzed and filtered as described in Section 3.

The Summary Generator (second module up from the bottom of Figure 6) operates differently for our five STU display methods. For the *Incremental* and *ALL* methods, this module passes STUs straight to the Representation Manager for final display. For the *Keyword* and *Keyword/Summary* methods, the Summary Generator relies on the Keyword Extractor module. This module uses a dictionary that associates words on the Web with word weights that indicate each word's importance. The module scans the words in each STU and chooses the highest-weight words as keywords for the STU. These keywords are passed to the Summary Generator.

For the *Summary* and *Keyword/Summary* methods, the Summary Generator relies on the Sentence Divider and the Sentence Ranking

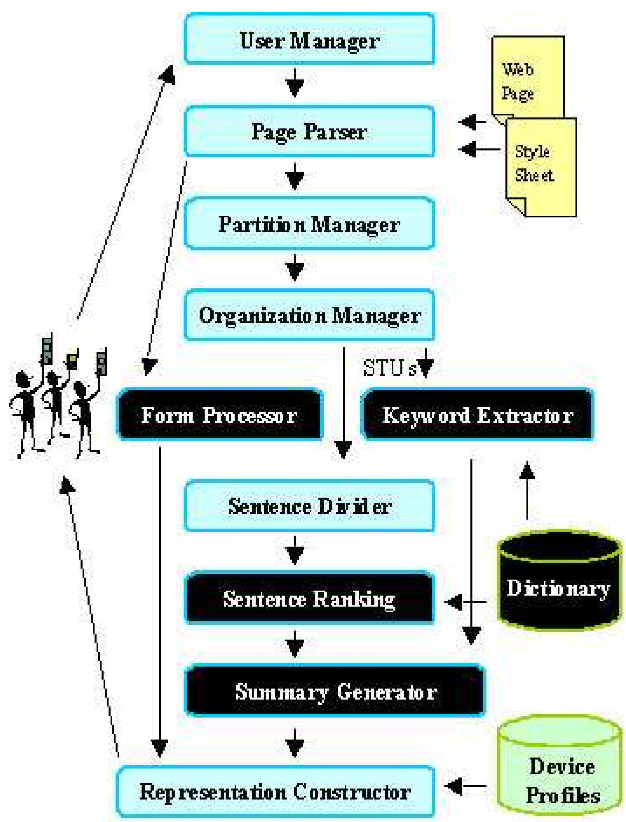

Figure 6: Processing a Web Page Request from a PDA

modules. The Sentence Divider partitions each STU into sentences. This process is not always trivial [21, 22, 26]. For example, it is not sufficient to look for periods to detect the end of a sentence, as abbreviations, such as "e.g." must be considered. The Sentence Ranking module uses word weight information from the dictionary to determine which STU sentence is the most important to display (see Section 2.3 below).

The Representation Constructor, finally, constructs all the strings for the final PDA display, and sends them to the remote PDA over a wireless link. The Representation Constructor draws target device information from the Device Profiles database (e.g., how many lines in the display, how many characters per line). This database allows the single Representation Constructor to compose displays for palm-sized devices and for cellular phones. The respective device profiles contain all the necessary screen parameters.

We now go into more detail on how the summarization process works. Again, this process involves the dark-colored modules in Figure 6. This process includes summary sentence and keyword extraction.

2.3 The Page Summarization Process

Our work builds on well know techniques for text summarization [19]. We use the well-known *TF/IDF* and within-sentence clustering techniques to find keywords and summary sentences. However, these techniques have traditionally been used on relatively homogeneous, limited collections, such as newspaper articles. We found that the Web environment required some tuning and adaptation of the algorithms. We begin with a discussion of our keyword extraction.

2.3.1 Extracting Keywords

Keyword extraction from a body of text relies on an evaluation of each word's importance. The importance of a word *W* is dependent on how often *W* occurs within the body of text, and how often the word occurs within a larger collection that the text is a part of. Intuitively, a word within a given text is considered most important if it occurs frequently within the text, but infrequently in the larger collection. This intuition is captured in the *TF/IDF* measure [27] as follows:

$$w_{ij} = tf_{ij} \times \log_2 \frac{N}{n} \text{ where}$$

w_{ij} =weight of term T_j in document D_i

tf_{ij} =frequency of term T_j in document D_i

N = number of documents in collection

n = number of documents where term T_j occurs at least once

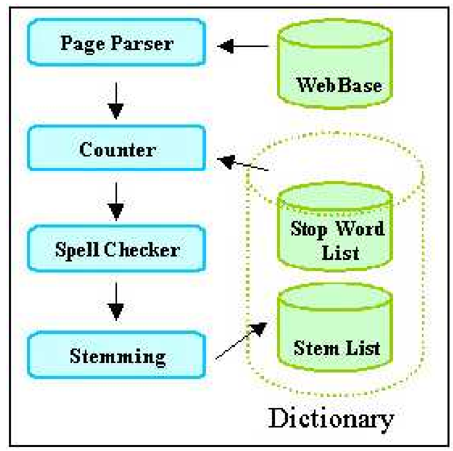

Figure 7: Creating a Dictionary of Weighted Words

Parameter *n* in this formula requires knowledge of all words within the collection that holds the text material of interest. In our case, this collection is the World-Wide Web, and the documents are Web pages.

Given the size of the Web, it is impossible (at least for us) to construct a dictionary that tells us how frequently each word occurs across Web pages. Thus, the system of Figure 6 uses an approximate dictionary that contains only some of the words, and for those only contains approximate statistics. As we will see, our approximation is adequate because we are not trying to carefully rank the importance of many words. Instead, typically we have a few words in an STU (recall that STUs are typically single text paragraphs), and we are trying coarsely to select a handful of important words. Because our dictionary is small, we can keep it in memory, so that we can evaluate keywords and sentences quickly at runtime.

To build our approximate dictionary, we analyzed word frequencies over 20 million Web pages that we had previously crawled and stored in our *WebBase* [15]. Figure 7 illustrates how the dictionary was created, and Figure 8 shows the number of words in the dictionary after each step. The Page Parser in Figure 7 fetches Web pages from our WebBase and extracts all the words from each page. The Page Parser sends each word to the Counter module, unless the word is a stop word, or is longer than 30 characters. Stop words are very frequent words, such as "is", "with", "for", etc.

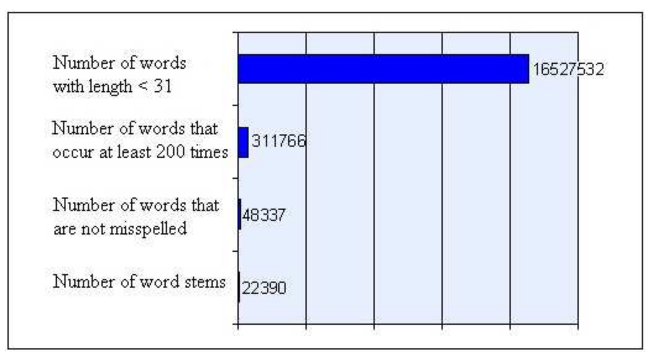

Figure 8: Trimming the Dictionary Collected from 20 Million Web Pages

The Counter module tags each unique word with a number and keeps track of the number of documents in which the word occurs. The top bar in Figure 8 shows how many words we extracted in this counting procedure.

Once counting is complete, the words that occur less than 200 times across all the pages are eliminated. This step discards 98% of the words (second bar in Figure 8). Notice that this step will remove many person names, or other rare words that may well be very important and would make excellent keywords for STUs. However, as discussed below, we will still be able to roughly approximate the frequency of these missing words.

The remaining words are passed through a spell checker, which eliminates another 84% of these remaining words. The size of the dictionary has now shrunk to 48 thousand words (Figure 8).

Finally, words that have the same grammatical stem are combined into single dictionary entries. For example, 'jump' and 'jumped' would share an entry in the dictionary. We use the Porter stemming algorithm for this step of the process [24]. The resulting dictionary, or 'stem list' contains 22,390 words, compared to 16,527,532 of the originally extracted set. The words, and the frequency with which each word occurs in the 20 million pages are stored in a dictionary lookup table. The frequencies are taken to be approximations for the true number of occurrences of words across the entire Web.

At runtime, when 'significant' keywords must be extracted from an STU, our Keyword Extractor module proceeds as follows. All the words in the STU are stemmed. For each word, the module performs a lookup in the dictionary to discover the approximate frequency with which the word occurs on the Web. The word's frequency within the Web page that contains the STU is found by scanning the page in real time. Finally, the word's *TF/IDF* weight is computed from these values. Words with a weight beyond some chosen threshold are selected as significant.

A special situation arises when a word is not in the dictionary, either because it was discarded during our dictionary-pruning phase, or it was never crawled in the first place. Such words are probably more rare than any of the ones that survived pruning and were included in the dictionary. We therefore ensure that they are considered as important as any of the words we retained. Mathematically, we accomplish this prioritization by multiplying the word's document frequency with the inverse of the smallest collection frequency that is associated with any word in the dictionary. Given that we are only searching for keywords with *TF/IDF* weight above a threshold, replacing the true small weight by an approximate but still small weight, has little effect. Thus, given this procedure, we can compute the approximate *TF/IDF* score for all words on any Web page.

Finally, notice that in our implementation we are not yet giving extra weight to terms that are somehow "highlighted." We believe that when a term is in italics, or it is part of an anchor, it is more likely to be a descriptive keyword for an STU. We plan to extend the weight formula given earlier to take into account such highlighting.

2.3.2 Extracting a Summary Sentence

Two of our methods, *Summary*, and *Keyword/Summary* require the Sentence Ranking module of our system to extract the most important sentence of each STU. In order to make this selection, each sentence in an STU is assigned a significance factor. The sentence with the highest significance factor becomes the summary sentence. The significance factor of a sentence is derived from an analysis of its constituent words. Luhn suggests in [18] that sentences in which the greatest number of frequently occurring distinct words are found in greatest physical proximity to each other, are likely to be important in describing the content of the document in which they occur. Luhn suggests a procedure for ranking such sentences, and we applied a variation of this procedure towards summarization of STUs in Web pages. The procedure's input is one sentence, and the document in which the sentence occurs. The output is an importance weight for the sentence.

The procedure, when applied to sentence *S*, works as follows. First, we mark all the significant words in *S*. A word is significant if its *TF/IDF* weight is higher than a previously chosen *weight cutoff W*. *W* is a parameter that must be tuned (see below). Second, we find all 'clusters' in *S*. A cluster is a sequence of consecutive words in the sentence for which the following is true: (i) the sequence starts and ends with a significant word. And (ii) fewer than *D* insignificant words must separate any two neighboring significant words within the sequence. *D* is called the *distance cutoff*, and is also a parameter that must be tuned. Figure 9 illustrates clustering.

In Figure 9, *S* consists of nine words. The stars mark the four words whose weight is greater than *W*. The bracketed portion of *S* encloses one cluster. The assumption for this cluster is that the distance cutoff *D>2*: we see that no more than two insignificant words separate any two significant words in the figure. We assume that if Figure 9's sentence were to continue, the portions outside brackets would contain three or more insignificant words.

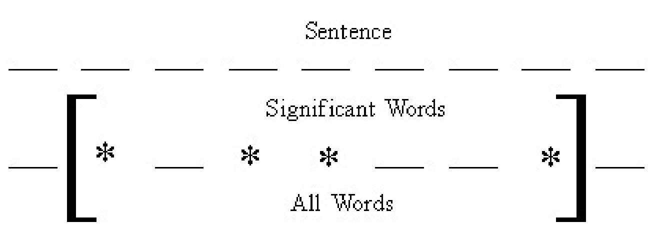

Figure 9: Finding Word Clusters within Sentences

A sentence may have multiple clusters. After we find all the clusters within *S*, each cluster's weight is computed. The maximum of these weights is taken as the sentence weight. Luhn [18] computes cluster weight by dividing the square of the number of significant words within the cluster by the total number of words in the cluster. For example the weight of the cluster in Figure 9, would be 4x4/7.

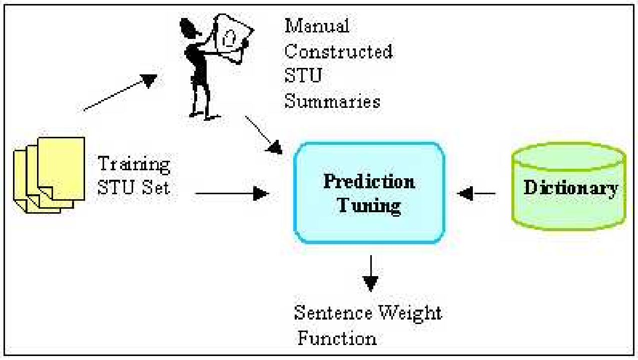

Figure 10: Tuning Cluster Selection

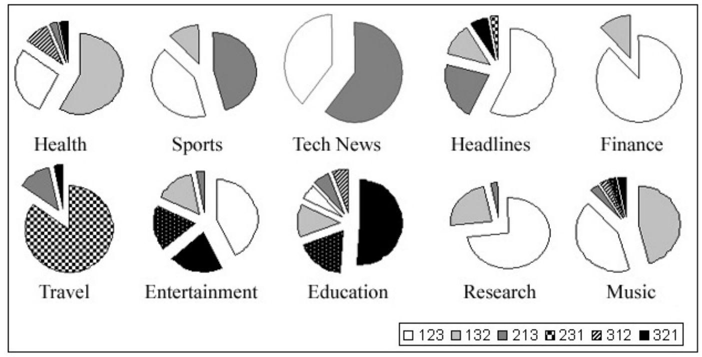

Figure 11: Results of Human-Generated STU Sentence Ranking

However, when we tried to apply Luhn's formula, we achieved poor results. This was not surprising, since our data set is completely different from what Luhn was working with. Therefore we tried several different functions to compute cluster weight. We achieve the best cluster weighting results by adding the weights of all significant words within a cluster, and dividing this sum by the total number of words within the cluster.

We conducted user tests to help us tune the weight *W* and distance *D* cutoffs for cluster formation and to inform our selection of the above cluster weighting function. Figure 10 shows the steps we took.

We selected ten three-sentence STUs from Web pages of ten different genres. We asked 40 human subjects to rank these sentences according to the sentences' importance. We then passed the STU set and the results of the human user rankings to a Prediction Tuning Unit. It used the dictionary and these two inputs to find the parameter settings that make the automatic rankings best resemble the human-generated rankings.

Figure 11 summarizes the results of the human-generated rankings. For example, for the "Sports" STU, about 44% percent of the human subjects said the most descriptive sentence was number 1 (of that STU), and that the second most descriptive sentence was number 2. (Thus, the sentence ranking was 1-2-3.) Another 44% preferred the sentence ordering 2-1-3, while about 12% liked 1-3-2. Clearly, ranking is subjective. For example, subjects disagreed in six ways on the ranking of the three education sentences, although about half of the subjects did settle on a 3-2-1 ranking. Finance clearly produced a 1-2-3 ordering, while the result for technical news is almost evenly split between a 1-2-3, and a 2-1-3 order. In most cases, however, there is a winning order.

These results in hand, the task was to tune the cutoffs and the cluster weighting formula so that automatic ordering would produce rankings that matched the human-generated results as closely as possible. Figure 12 illustrates this optimization problem. The two axis represent the parameters, distance and weight cutoff. The lightness of each area is proportional to how many of the most-popular rankings (or second most popular rankings) are selected at that setting. For instance, with a weight cutoff of 2 and a distance cutoff of 3, we get a very dark area, meaning that with these parameter values almost none of the two most-popular human rankings are selected.

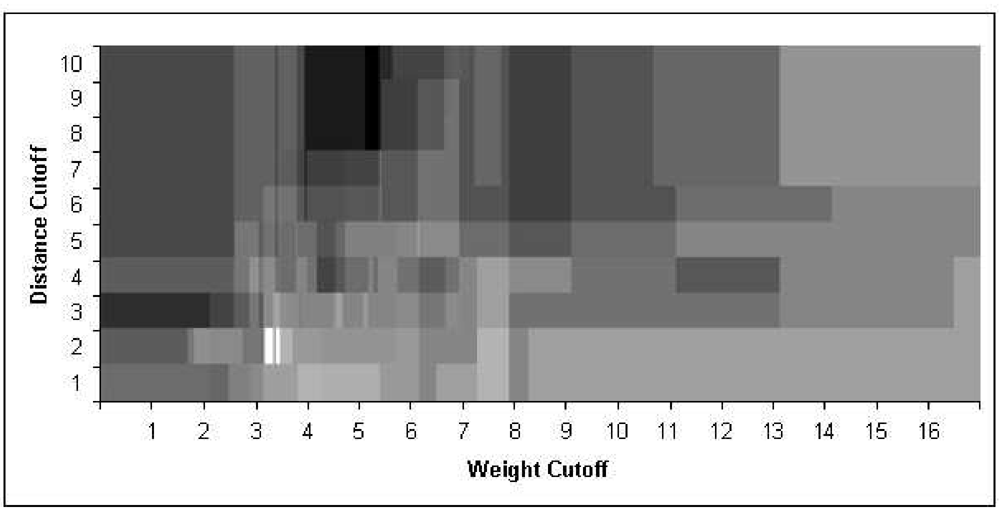

Figure 12: Finding the Proper Cutoffs for Word Weight and Distance

The brightest region in Figure 12 has the optimum cutoff values, 2 for the *distance cutoff* and 3.16 for the *weight cutoff*. These are the values used by our system. With these values, the automatic ranking agreed with the most preferred human-generated ranking 70% of the time, and with the second-most preferred ranking 20% of the time.

3. FORM SUMMARIZATION

A special challenge arises when the page we are summarizing contains a form. In many cases, the form is complex and consumes a lot of screen real estate. Thus, it is difficult for users to gain an overview of what the form is about and which information is required of them.

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 allows users to scan and understand the form quickly. When the user is ready to fill in information, pen gestures over a textual prompt cause the associated input widget to be displayed. All other widgets remain hidden. Figures 13-15 shows how the display works. 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 13 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 13: Original HTML Page

Figure 14 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 15 shows the top form partially 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 14.

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 to convert for each input field.

The difficulty in generating the space-efficient display of Figure 15 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 13

Figure 14: Form With No Widgets Revealed

Figure 15: Form With One Widget Revealed

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 13.

Example 1:

```
...
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, and a name attribute (e.g. name="FirstName"). In all these three cases type="text". 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 radio buttons and checkboxes may appear to the right of the form item. For example, consider Figure 16, which is an excerpt of the *garden.com* Web site.

We see in Figure 16 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.

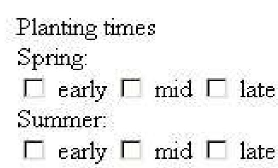

Figure 16: Text Labels Are Not Always Placed Before Their Input Widgets

Figure 16 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 16, 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."

3.1 Form Analysis

We developed and evaluated eight form analysis algorithms. Each algorithm’s input is the internal representation of a form, and the input field for which a label match is needed. The output is a piece of text that is excerpted from somewhere within the form. That text is intended to describe the input field.

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. Chunks differ from STUs in that they include additional bookkeeping information (e.g., the identifier used for an input element), and are customized to the needs of form analysis.

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. The matching string is selected from a chunk that is at most *n* chunks away. All of our algorithms use *n=10*, a value obtained empirically.

For label selection, we used eight different matching algorithms. We call these algorithms: N-Gram, Letter/Word, Word/Letter, Substring, Tables, Previous, Following, and NULL Algorithm. The algorithms differ in how they accomplish the matching of labels to input fields. All algorithms are detailed in [6]. Here we explain just them briefly.

One set of algorithms exploits the fact that HTML programmers must provide an internal name to each input field. This name is needed for the Web Server's HTML processing. Often, these internal names resemble at least a piece of the correct label text (see Example 1 above). The algorithms that use the internal name attempt to find the surrounding text that is most similar to the internal name.

One example is our *N-gram* algorithm that uses an information retrieval technique called "n-gram matching." When comparing any two words, A and B, 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 [8] 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 cardinality of intersection a better predictor for proper matching, especially if the strings involved are of very different lengths. If the score is greater than or equal to a threshold *T*, then the score is considered significant and a match is made. Otherwise, the algorithm reports a failure (score = 0).

The *Substring* algorithm also uses the internal input field name, but attempts to find form text of which the internal name is a substring or an acronym, such as 'Password'→pwd. *Letter/Word* and *Word/Letter* attempt matches of the form 'First Name'→Fname, and 'Phone Work'→PhoneW, respectively. The *NULL* algorithm attempts no matching at all, and simply returns the internal field name as the label.

Other algorithms use the fact that the placement of labels on forms often conforms to certain layout conventions. Our *Table* algorithm is an example. HTML 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. 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.

To see how this is done, 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 *E* that is located within a table, the *Table* algorithm first checks whether some text string is located within the same table cell with *E*. If such a string exists, it is chosen to be the label for *E*. If no string is found within the same cell, the *Table* algorithm checks sequentially whether a string is located in a cell immediately to the left, above, below, or to the right. The first string found in these probes is taken to be *E*'s label, and a score of 1 is returned. If no label is found, a score of 0 results.

Previous and *Following* are also layout-related algorithms. They choose text that, respectively, precedes or follows an input field as the appropriate label.

4. EXPERIMENTS

We conducted a variety of performance experiments to evaluate the effectiveness of page and form summarization. These experiments included both system-level evaluations, and user tests that helped us refine our designs. In this section, we summarize our results. We focus first on the experiments that probed our page summarization performance. In Section 4.2 we summarize form-related results.

4.1 Page Summarization

For our page summarization evaluations, we tuned our algorithms as described earlier, and then designed user experiments that would reveal which of the five methods of Figure 4 worked best for users. In particular, we wanted to determine which method would allow users to complete a set of sample information exploration tasks fastest, and how much I/O (pen gestures) users needed to perform for each method.

We constructed an instrumented Palm Pilot and Nokia cellular phone emulator and added it as a user front-end to the test system described in Figure 6. The emulator does not simulate a complete Palm Pilot or cellular phone in the sense that it could run programs written for these devices. It rather performs only the functions of our browser application. The emulator does maintain a live connection to our Web proxy, which in turn communicates with the Web. If users were to follow links on the emulator display (which they did not for this set of experiments), then the emulator would request the page from the proxy and would display the result. We can toggle the display between the Palm Pilot and the cellular phone look-alike, so that we can assess the impact of the cellular phone's smaller screen. We have not performed the cellular phone experiments yet.

The emulator displays a photo-realistic image of a 3COM Palm Pilot or Nokia phone on a desktop screen. (See Figure 17.) Instead of using a pen, users perform selection operations with the mouse. We consider this substitution acceptable in this case, because our experiments required no pen swiping gestures. Only simple selection was required. The emulator is instrumented to count selection clicks, and to measure user task completion times.

When using the emulator for an experiment, the subject or the operator selects one of the methods (*Incremental*, *Keyword*, etc.) from the pull-down list in the view panel. A task is selected in the task/URL selection field. The start button begins the experiment, the stop button ends it. Each task for the series of experiments reported on here involved a single Web page and one question about that page. We limited tasks to cover only a single page to ensure that we restricted measurements to cover summarization issues, as opposed to browsing artifacts, such as network delays, false trails, and subjects’ adjustment to different page styles. Subjects used the mouse to expand and collapse portions of each page, and to open or close STUs as they looked for the answers to questions we posed about the page.

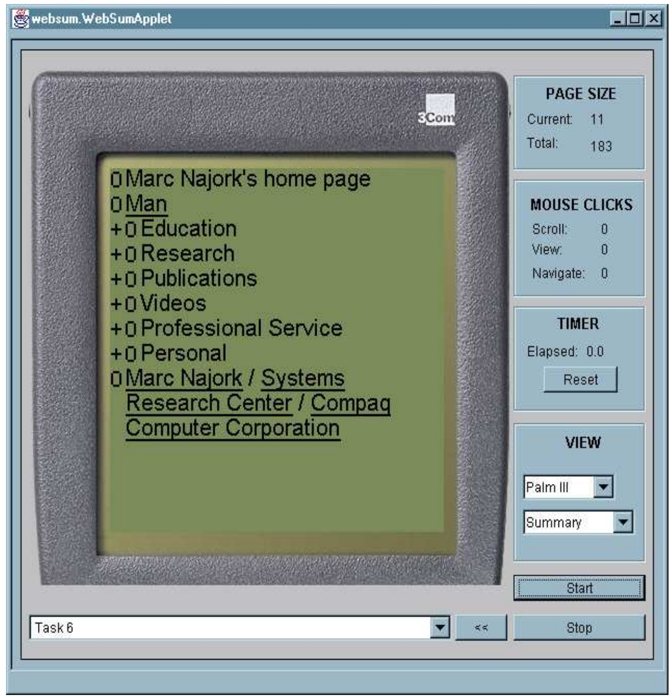

Figure 17: Instrumented PDA Emulator for our User

We selected Web pages for these tasks to be of varying length, but large enough not to fit on a single PDA screen. The questions we asked varied as well. Some questions requested subjects to find a particular link on the page. Others asked subjects to find particular pieces of content within the page. Each question had a well-defined answer, rather than being open-ended. We selected Web pages and questions without first viewing the results of the summarization. Table 1 shows the list of 10 tasks that we asked users to perform.

Table 1. The 10 Tasks Our 15 Subjects Completed on the PDA Emulator

	Description
Task 1	From the Bureau of Census home page find a link to News for Federal Government Statistics.
Task 2	From the Lonely Planet Honk Kong Web page find when the Hong Kong Disneyland is going to open.
Task 3	From the Stanford HCI Page, find the link to Interaction Design Studio.
Task 4	From the WWW10 Conference home page, find the required format for submitted papers.
Task 5	From the ’upcomingmovies’ review of the movie Contender: How was the character "Kermit Newman" named?
Task 6	From Marc Najork’s Home page find the conference program committees he participated in.
Task 7	From the science article in Canoe find out: What percentage of bone cells can be converted to brain cells?
Task 8	From the ’boardgamecentral’ Web page find what "boneyard" means in the dominoes game.
Task 9	From the ’zoobooks’ Web page find where penguins live.
Task 10	From the Pokemon official site find the price of Pokemon Gold and Silver.

Table 2 provides statistics about each task’s Web page. The average number of STU’s on each of our tasks’ Web pages was 33. The average total page length was 155 PDA lines. The number of sentences in each STU varied from 1 to 10. The number of lines in each STU varied from 1 to 48. A Palm Pilot device can display 13 lines at a time with our browser, the cellular phone device can display eight. Given the PDA’s screen size, a 48-line STU would be displayed as 4 pages on the PDA and 6 pages on the cellular phone. The one-line STUs would fit on a single page. In short, we ensured that we exposed users to STUs of widely varying lengths. Some easily fit onto one screen, others required scrolling when expanded.

Table 2. Number and Lengths of STUs for Each Task

Task	1	2	3	4	5	6	7	8	9	10
# of STUs	31	32	26	67	32	33	33	19	18	36
# of Lines	47	169	306	140	343	120	120	60	100	145

For our experiment, we consecutively introduced 15 subjects with strong World-Wide Web experience and at least some Computer Science training to our five STU exploration methods. Each subject was introduced to the emulator, and allowed to complete an example task using each of the methods. During this time, subjects were free to ask us questions about how to operate the emulator, and how to interact with the browser for each of the methods. Once we had answered all of the subject's questions, we handed him a sheet of paper that instructed him on the sequence in which he was to run through the tasks, and which method to use for each. Subjects clicked the start button once they had selected a task and method. This action displayed the collapsed Web page for the task. Once subjects had found the answer to the task's question by opening and closing the structural hierarchy and individual STUs, they clicked the stop button.

The instructions we gave to each subject had them use each method twice (for different tasks). We varied the sequence in which subjects used the methods. In this way each task was tackled with different methods by different subjects. We took this step to exclude performance artifacts based on method order, or characteristics of the matches between particular tasks and methods.

4.1.1 User Performance

Figure 18 summarizes the average task completion time for each method. The figure shows that in six out of 10 tasks method *Incremental* performed better than the *ALL* method. The methods are thus close in their effectiveness. These results seem to indicate that showing the first line of the first sentence is often not effective, probably because STUs on the Web are not as well structured as paragraphs in carefully composed media, such as, for example, articles in high-quality newspapers. Thus, showing the full text of the STU and letting the user scroll seems to be as effective as first showing just the first sentence. Recall however, that the *ALL* method shows the entire text of a single STU, not the text of the entire page. Thus the '+/-' structural controls are still being used even for the *ALL* method.

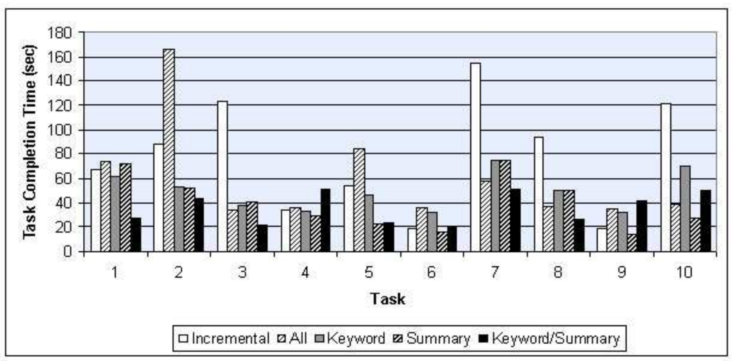

Figure 18: Task Completion Times for All Methods and All Tasks

We see that for one half of all tasks (5 out of 10), the *Summary* method gave the best task completion time, and for the other half, the *Summary/Keyword* method yielded the best time. The time savings from using one of these summarization techniques amount to as much as 83% compared to some of the other methods! Using at least one of these techniques is thus clearly a good strategy.

Notice that both pairs *Incremental/ALL*, and *Summary/Keyword-Summary* tend to be split in their effectiveness for any given task. In the case of *Incremental* and *ALL*, the completion time ratio between the methods was at least two in five of our 10 tasks. In Task 2, for example, *Incremental* took about 80 seconds, while *ALL* required 160 seconds for completion, a ratio of 2. On the other hand, *ALL* was much better than *Incremental* in Task 7. Similarly, *Keyword* and *Keyword/Summary* had completion time ratios of two or higher in five of 10 tasks. In contrast, *Keyword* and *Summary* more often yielded comparable performance within any given task. Given that *Summary* and *Keyword/Summary* are the two winning strategies, we need to understand which page characteristics are good predictors for choosing the best method. We plan to perform additional experiments to explore these predictors.

Figure 19 similarly summarizes I/O cost: the number of pen taps subjects expended on scrolling and the expansion and collapse of STUs. Notice that in most of the cases either *Summary* or *Keyword/Summary* gave the best results, reinforcing the timing results of Figure 18. The reward for choosing one of the summarization methods is even higher for I/O costs. We achieve up to 97% savings in selection activity by using one of the summarization methods.

Before processing the results of Figures 18 and 19 further to arrive at summary conclusions about our methods, we examined the average completion time for each user across all tasks. Figure 20 shows that this average completion time varied among users.

This variation is due to differences in computer experience, exploration sequence, level of concentration, and so on. In order to keep the subsequent interpretation of our raw results independent from such user differences, we normalized the above raw results before using them to produce the additional results below. The purpose of the normalization was to compensate for these user variations in speed. We took the average completion time across all users as a base line, and then scaled each user's timing results so that on the average, all task completion times would be the same. The average completion time for all users over all tasks was 53 seconds.

To clarify the normalization process, let us assume for simplicity that the average completion time was 50 seconds, instead of the actual 53 seconds. Assume that user A performed much slower than this overall average, say at an average of 100 seconds over all tasks. Assume further that user B performed at an average of 25 seconds. For the normalization process, we would multiply all of user A's individual completion times by 1/2, and all of B's times by 2.

With these normalized numbers, we summarized the timing and I/O performance for each method (Figures 21 and 22). Recall that I/O performance is the sum of all mouse/pen actions (scrolling, opening and closing STU's, etc.).

Notice that *ALL* and *Keyword* are comparable in completion time. One explanation for this parity could be that our keyword selection is not good. A more likely explanation is that for our, on the average, short STU lengths, a quick scan is faster than making sense of the keywords.

Notice that on average, *Summary* and *Keyword/Summary* produce a 39 second gain over *Incremental*, and an 18 second gain over *ALL*. The two methods are thus clearly superior to the other methods. In Figure 21 the two methods are head-to-head in timing performance.

As we see in Figure 22, however, *Keyword/Summary* requires 32% fewer input effort than *Summary*. This difference gives *Keyword/Summary* an advantage, because user input controls on PDAs are small, and users need to aim well with the input pen. On a real device, this small scale thus requires small-motor movement control. Operation in bumpy environments, such as cars, can therefore lead to errors. The combination of Figures 21 and 22 therefore give *Keyword/Summary* the lead in overall performance.

The difference in timing vs. I/O performance for *Keyword/Summary* is somewhat puzzling, as one would expect task completion time to be closely related to I/O effort. We would therefore expect *Keyword/Summary* to do better in timing performance than *Keyword*. We believe that the discrepancy might be due to the cognitive burden of interpreting keywords. That is, looking at the complete summary sentence is easier than examining the keywords, as long as the summary sentence is not too long.

In summary, we conclude from our studies that the *Keyword/Summary* method is the best method to use for finding answers to questions about individual Web pages on PDAs. While the keywords require some mental interpretative overhead, the savings in input interaction tips the balance to *Keyword/Summary*, even though this method's timing performance is comparable with that of *Summary*.

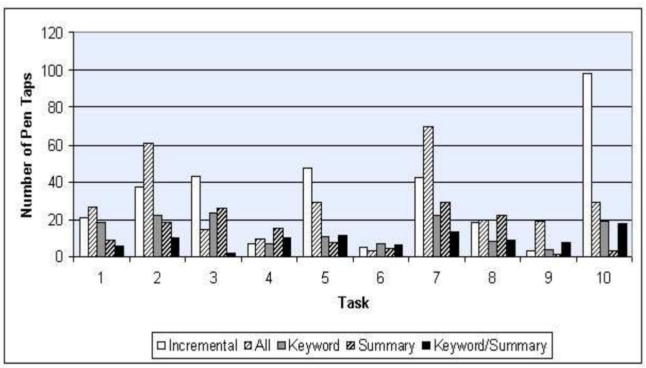

Figure 19: I/O Activity Required for All Methods Over All Tasks

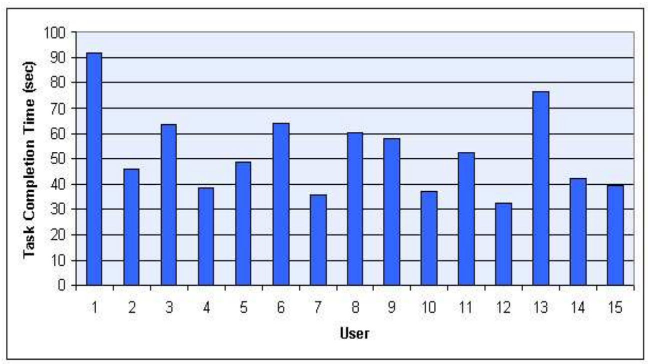

Figure 20: Differences in Average Task Completion Times

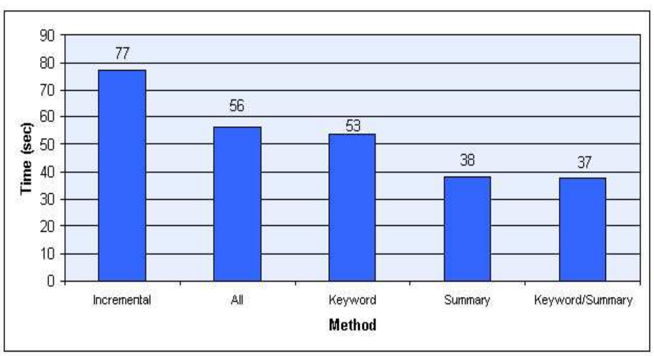

Figure 21: Average Completion Time for Each Method Across All Tasks

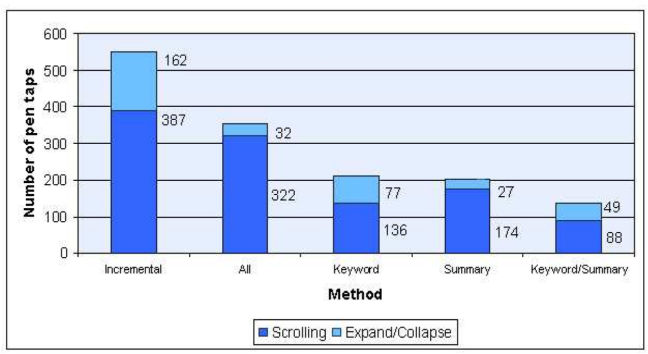

Figure 22: Average I/O Expenditure for Each Method Across All Tasks

4.1.2 System Performance

Recall that the deployment platform for our system is a wirelessly connected PDA. The amount of information that is transferred from the Web proxy to the PDA is therefore an important system-level parameter that must be considered in an overall evaluation. This information flow impacts the bandwidth requirements, which is still in short supply for current wireless connections.

Table 3 summarizes the bandwidth-related properties of each task's Web page. Column 1 shows the total number of bytes occupied by a fully displayed HTML page, when images and style sheets are included. Column 2 shows the size once images and style sheets are removed from the total. The third column lists the number of bytes our system sends when transmitting STUs. The average 90% savings of Column 3 over Column 1 stem from stripping HTML formatting tags, and the discarded images. If we just consider the HTML and ignore images, the average savings is 71%. Note that these transmission times are not included in our timing data, since we were using the emulator for our experiments. The numbers in Column 3 are for the *ALL* method. The *Keyword*, *Summary*, and *Keyword/Summary* methods require additional data to be transmitted: the keywords, and the start and end indexes of the summary sentences in the transmitted data. On average over all tasks, this additional cost is just 4% for *Summary*, 24% for *Keyword*, or 28% for *Keyword/Summary*. Even for the latter worst case this still leaves a 87% savings in required bandwidth for our browser.

Table 3. Bandwidth Requirements for Different Browsing Alternatives

Task	Page Size (Total Bytes)	Page Size (HTML Bytes)	Packet Size (ALL)	Size Savings (Compared to Full Page)
1	51,813	18,421	1193	97.7%
2	45,994	18,309	4,969	89.2%
3	66,956	12,781	9,762	85.4%
4	17,484	11,854	3,736	78.7%
5	55,494	21,276	10,913	80.3%
6	23,971	6,583	1,079	95.5%
7	75,291	35,862	5,877	92.2%
8	44,255	9,394	1,771	96.0%
9	19,953	7,151	3,042	84.8%
10	114,678	17,892	4,342	96.2%

Notice, that an 87% reduction in required bandwidth is highly significant when operating our browser in a wireless environment. To see this significance, consider that in terms of transmission time over wireless links, an average size page (over the 10 tasks) would take seven seconds for the *ALL* method on one popular wireless network. Sending all of the HTML as well would take 24 seconds over the same network. If images and style sheets were added in addition, transmission of an average page would take up 77 seconds! Compared to a browser that sends the full page, our browser's bandwidth parsimony would therefore amount to an 11-fold improvement. Even a browser that discarded images and style sheets, but transmitted all of the HTML tags would require three times more bandwidth than our solution. The computation time for transforming the original Web pages on the fast proxy is negligible, compared to the transmission time.

4.2 Form Summarization

The experiments for evaluating our form summarization aimed at gauging the effectiveness of our algorithms for matching Web page text to form input labels. Matching results are objective enough that no user experiments were required. All experiments were run over unmodified forms that we retrieved from the Web.

4.2.1 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 randomly selected 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 $n = 10$.

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 threshold $T=2$) will always beat a table algorithm (with maximum score of 1). (A better scheme to combine algorithms is described in Section 4.2.3)

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 label-matching algorithm when used by itself. The second experiment examined the effectiveness of combining the algorithms with a score maximization scheme.

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.

4.2.2 Experimental Results

Table 4 shows the results of our 115 element, individual algorithms experiment on complex forms. Each row of Table 4 describes the performance of our algorithms for the forms on one Web site. Each column contains the results for one algorithm.

Each column shows four summary results for the respective algorithm. The first summary (row TOTALS) consists of two numbers. The first number indicates how often the respective algorithm produced a correct match. The second number indicates how often the algorithm concluded that it had a match. For example, the *N-gram* algorithm concluded 60 times that it had a match, and its match was correct for 57 of these 60 matches. For the 55 (115-60) remaining input elements, the *N-gram* algorithm concluded that it could not produce a good match (the score was below the threshold *T*).

The second row in Table 4 is the success rate of the respective algorithm when measured against the entire collection. The *N-gram* algorithm, for example, had 57 successful matches, which translate to an approximate success rate of 49% over all 115 input elements.

The third row 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, amounting to a failure rate of 3%.

The fourth row, finally, is the algorithm's "pass" rate. It shows how often the algorithm did not venture a match at all. Note that the score for *Previous, Following* and *NULL* is zero because they can always be applied.

Table 4: Each Algorithm Matching 115 Input Fields

Site (Forms)	N-Gram	Letter/Word	Word/Letter	Substring	Tables	Previous	Following	Null
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%

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

4.2.3 Combining Matching Algorithms

Observe in Table 4 that in many cases, multiple algorithms come up with the correct answer. The total number of successful matches in Row 1 of Table 4 is 272, significantly more than the 115 input fields to be matched. At the same time, we found cases when one algorithm performs significantly better than others.

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 5 shows the results of our combined strategy running over the full 330 input element complex-form data set.

Table 5: Matching Performance for Algorithm Combination over 330 Input Elements

Site (Forms)	N-Grams	Letter/Word	Word/Letter	Substring	Tables	Previous	Following	Null	Total Matches
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%	

Table 5 is organized similarly to Table 4. The differences are as follows. On the far right, a new column shows that 264 of the 330 input fields were successfully matched. This corresponds to an 80% success rate for these complex forms.

Notice that, in contrast to Table 4, there is no overlap in the success numbers for each algorithm. For any given input element, only one algorithm is 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.

In summary, we found that for the simple forms as they most frequently occur on the Web, the algorithms described achieve 95% accuracy in generating understandable form summaries. For complex forms with many input fields, summarization success is around 80%. For failure cases, information beyond the summary must be provided for the user to understand the form. The *N-gram* algorithm is the single best matching method. But using a combination of the described algorithms yields significantly improved results.

4.2.4 Common Breakdowns

Although the combined matching algorithm performs well, it is instructive to see why failures occur. The three main reasons are 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 23):

Figure 23: 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 24 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 chose 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 24: 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 25 shows an example.

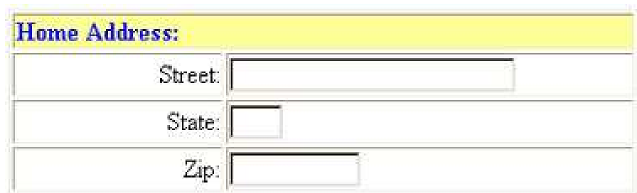

Figure 25: Text Field Group With Group Label

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 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 16, 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.

5. RELATED WORK

Our Power Browser draws on two research traditions. The first is the search for improving user interaction with text by designing non-linear approaches to text displays and document models. Projects in the second tradition have examined design choices for displays on small devices.

One body of work in the first tradition has explored effective ways of displaying documents and search results through the use of structured browsing systems. See for example [7, 11, 25]. The long-standing Hypertext community [10] has focused on tree structures for interacting with multiple documents [12] and large table of contents [9]. The Cha-Cha system allows users to open and collapse search results. In this sense that system is similar to our displaying individual Web pages as nested structures. But Cha-Cha applies this concept over multiple pages, and the display is pre-computed. The part of our Power Browser that we introduced in this paper focuses on a single Web page, and all displays are dynamically computed.

Similarly, Holophrasting interfaces [28] have aimed to provide visualization of textual information spaces by providing contextual overviews that allow users to conceal or reveal the display of textual regions. We use the Holophrasting principle for our STUs. But rather than progressively disclosing a fixed body of text, some of the methods we explored here apply Holophrasting to transformations of the text, such as summaries or keywords.

Numerous approaches to browsing the Web on small devices have been proposed in work of the second above mentioned tradition. Digestor [2] provides access to the World-Wide Web on small-screen devices. That system re-authors documents through a series of transformations and links the resulting individual pieces. Our technique is more in the tradition of Fisheye Views [14], where a large body of information is displayed in progressively greater detail, with surrounding context always visible to some extent.

Ocelot [1] is a system for summarizing Web pages. Ocelot synthesizes summaries, rather than extracting representative sentences from text. The system's final result is a static summary. Ocelot does not provide progressive disclosure where users can drill into parts of the summary, as we do in the Power Browser. Another system, WebToc [20], uses a hierarchical table of contents browser; that browser, however, covers entire sites, and does not drill into individual pages. More information about our own summarization of forms is found in [6].

Similar to our Partition Manager, the system described in [17] applies page partitioning to Web pages. The purpose of that system's partitioning efforts, however, is to convert the resulting fragments to fit the 'decks' and 'cards' metaphor of WAP devices.

6. CONCLUSION

We developed a new approach to summarize and browse Web pages. We apply 'Macro-level' summarization, which relies on structural analysis of Web pages. These summaries allow users to expand and contract pages based on their relative structural nesting. An additional, integrated 'Micro-level' summarization uses information retrieval techniques to outline portions of the text for the user.

Our user experiments showed that a combination of keyword extraction and text summarization gives the best performance for discovery tasks on Web pages. For instance, compared to a scheme that does not summarize, we found that for some tasks our best scheme cut the completion time by a factor of 3 or 4.

A special problem with Web page summarization is the display of HTML forms on small screens. We introduced algorithms needed for finding short string extractions among a form's text. These extractions are then used as labels for input fields. We introduced eight algorithms that in combination produce good form summaries in 80%-95% of the cases. Success rates depend on the details of the form design. Where summarization fails, more of the form's expository text must be revealed on the PDA screen.

Our overall results suggest that summarization approaches are key to successful PDA-based user interactions with the World-Wide Web. Information task completion times and input effort can be significantly reduced if summarization techniques of various forms are employed.

7. REFERENCES

[1] A.L. Berger, V.O. Mittal, OCELOT: A System for Summarizing Web Pages, Proc. of 23rd Annual Conf. on Research and Development in Information Retrieval (ACM SIGIR), 2000, pp. 144-151.

[2] T.W. Bickmore and B.N. Schilit, Digestor: Device-independent Access to the World-Wide Web, In Proc. of 6th Int. World-Wide Web Conf., 1997.

[3] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd, Power Browser: Efficient Web Browsing for PDAs, In Proc. of the Conf. on Human Factors in Computing Systems, CHI'00, 2000, pp. 430-437.

[4] O. Buyukkokten, H. Garcia-Molina, and A. Paepcke, Focused Web Searching with PDAs, In Proc. of 9th Int. World-Wide Web Conf., 2000, pp. 213-230.

[5] O. Buyukkokten, H. Garcia-Molina, A, Paepcke, Accordion Summarization for End-Game Browsing on PDAs and Cellular Phones, , In Proc. of the Conf. on Human Factors in Computing Systems, CHI'01, 2001.

[6] O. Kaljuvee O. Buyukkokten, H. Garcia-Molina, and A. Paepcke, Efficient Form Entry on PDAs, In Proc. of 10th Int. World-Wide Web Conf., 2001.

[7] M. Chen, M. Hearst, J. Hong and J. Lin, Cha-Cha: A System for Organizing Intranet Search Results, In Proc. of 2nd USENIX Symposium on Internet Technologies and SYSTEMS (USITS), 1999.

[8] 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.

[9] R. Chimera, K. Wolman, S. Mark and B. Shneiderman, An Exploratory Evaluation of Three Interfaces for Browsing Large Hierarchical Tables of Contents, ACM Transactions on Information Systems, 12, 4, Oct. 94, pp. 383-406.

[10] J. Conklin, Hypertext: An Introduction and Survey, IEEE Computer, 20(9), pp. 17-41,1987.

[11] D.E. Egan, J.R. Remde, T.K. Landauer, C.C. Lochbaum and L.M. Gomez, Behavioral Evaluation and Analysis of a Hypertext Browser, In Proc. of CHI'89, pp. 205-210.

[12] S. Feiner, Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structure, Conf. on Office Information Systems, New York: ACM, 1988, pp. 205-212.

[13] A. Fox and E.A. Brewer, Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation, Proc. of 5th Int. World-Wide Web Conf., 1996.

[14] G.W. Furnas, Generalized Fisheye Views, In Human Factors in Computing Systems III, Proc. of the CHI'86 Conf., 1986, pp. 16-23.

[15] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, WebBase: A Repository of Web Pages, In Proc. of 9th Int. World-Wide Web Conf., 2000, pp. 277-293.

[16] M. Jones, G. Marsden, N. Mohd-Nasir, K. Boone and G. Buchanan, Improving Web Interaction on Small Displays, In Proc. of 8th Int. World-Wide Web Conf., 1999, pp. 51-59.

[17] E. Kaasinen, M. Aaltonen, J. Kolari, S. Melakoski and T. Laakko, Two Approaches to Bringing Internet Services to WAP devices, In Proc. of 9th Int. World-Wide Web Conf., 2000, pp. 231-246.

[18] H.P. Luhn, The Automatic Creation of Literature Abstracts, IBM Journal of Research & Development, 2 (2), 1958, pp. 159-165.

[19] I. Mani and M.T. Maybury (editors), Advances in Automatic Text Summarization, MIT Press, 1999.

[20] D.A. Nation, C. Plaisant, G. Marchionini and A. Komlodi, Visualizing Web Sites using a Hierarchical Table of Contents Browser: WebToc. In Proc. of 3rd Conf. on Human Factors and the Web, 1997.

[21] D.D. Palmer and M.A. Hearst, SATZ: An Adaptive Sentence Boundary Detector. http://elib.cs.berkeley.edu/src/satz/.

[22] D. D. Palmer and M.A. Hearst, Adaptive Multilingual Sentence Boundary Disambiguation, In Computational Linguistics, 23(2), 1997, ACL. pp. 241-269.

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

[24] M.F. Porter, An Algorithm for Suffix Stripping, Program, 14(3), pp. 130-137, 1980.

[25] W. Pratt, M.A. Hearst and L.M. Fagan, A Knowledge-Based Approach to Organizing Retrieved Documents, In Proc. of 16th National Conf. on AI (AAAI-99), 1999.

[26] J.C. Reynar and A. Ratnaparkhi, A Maximum Entropy Approach to Identifying Sentence Boundaries. In Proc. of the 5th Conf. on Applied Natural Language Processing, 1997.

[27] G. Salton, Automatic Text Processing, Addison-Wesley, Chapter 9, 1989.

[28] S.R. Smith, D.T. Barnard and I.A. Macleod, Holophrasted Displays in an Interactive Environment, Int. Journal of Man-Machine Studies, 20:343-355, 1984.