Accelerating Browsing by Automatically Inferring a User's Search Goal

Chris Drummond — Computer Science Dept. — cdrummon@csi.uottawa.ca
Robert Holte — Computer Science Dept. — holte@csi.uottawa.ca
Dan Ionescu — Electrical Engineering Dept. — ionescu@trix.genie.uottawa.ca
University of Ottawa, Ottawa, Ontario, Canada, K1N 6N5

Abstract

This paper discusses a novel method called active browsing which increases the speed and accuracy with which a user may browse libraries for reusable software. Information inferred solely from the user's normal actions is employed by the system to locate software items relevant to the user's search goal. This paper describes our active browsing system and illustrates its operation with an example using typical browsing steps. An experiment, using an automated browsing agent, is described demonstrating that active browsing accelerates search.

1 Introduction

The general aim of our research is to reduce the cost of locating reusable software artifacts in large libraries. The approach we adopt is to increase the effectiveness of human-computer interactive search, often termed browsing. Browsing has a very broad range of applications, including knowledge bases [9], information retrieval [5,12], hypertext [2,11] and even relational databases [10]. In browsing software libraries, the "state" of the search is a particular item in the library, and one is permitted to "move" to items that are related to the current item in particular ways. A computer system that aids in browsing is referred to as a browsing SYSTEM and the human that does the browsing is a browsing AGENT.

Research dealing with searching software libraries has principally focused on improving indexing [15,16,17]. We focus on browsing because we feel that indexing, although effective, has certain limitations which browsing can overcome. For instance, indexing methods are appropriate only if the user has a good goal definition. Without a good goal definition the user must browse. Fischer et al. state [13], when describing natural language interfaces for information retrieval, " ..... they do not assist users who are unable to describe precisely what they want at the beginning of an information seeking process". Even a good indexing scheme is not effective if the user is unfamiliar with the indexing language. A frequent cause of such unfamiliarity is that different users are not consistent in the way they name or classify items [14]. The two methods, browsing and indexing, are by no means mutually exclusive. If the user can accurately describe the whole or even a significant part of the target, indexing is the more powerful approach. Ideally indexing and browsing facilities would be incorporated in the same system.

The implementation and experimental results discussed in the rest of the paper are for searching libraries of code. Actual code reuse is felt by some to give limited returns [18,19]. They propose the reuse of much higher level software design information. An important idea within the knowledge base community is the "Knowledge-Based Software Assistant" [20]. This approach aims to formalize the whole process of software development and use correctness preserving transforms to aid the user in producing executable code from specifications. As new software is synthesized, not composed from already existing code, this would seem to obviate the need for libraries. In practice, however, there are likely to be libraries of software artifacts even if they are of a radically different form from code. As W. L. Johnson [21] points out "Reuse is essential at the requirements level, just as it is at the program level". The principles of the system outlined in this paper should be readily transferable to libraries of any software artifacts.

Even with a well structured library normal browsing can be a time consuming process. It is also knowledge intensive as the user must have some idea of not only what is in the library but also of its structure. We have developed a method, called active browsing, for increasing the speed and success rate of interactive search [3,4]. As the name suggests, the essence of the method is to have the browsing system play a more active role. Previous methods for enhancing the computer's role [2,6,7,12] all require input from the agent in addition to the agent's normal search actions. The unique feature of active browsing is that the search goal is inferred automatically from the sequence of moves made by the agent in the normal course of browsing. The browsing system then suggests to the browsing agent specific items in the library that it judges to be of interest, specifically those that best match the inferred search goal.

In the following, section 2 describes our active browsing system, illustrating its operation with typical browsing behaviour in a hierarchically organized library. This example demonstrates the main benefit of active browsing, namely, that in a given amount of time a much larger portion of the library is searched than by ordinary browsing. Section 3 describes an automated browsing agent used in the experiments in place of human subjects. Section 4 details these experiments and the results obtained which measure how often, in practice, active browsing decreases the time required to find a particular item in a library.

2 The active browsing system

The development of our active browsing system began with a commercial browsing system for libraries of object-oriented software written in Objective-C. An item in this library is a "class", in the object-oriented sense. A class consists of a set of “methods” which define its functionality and a set of “instance variables” which define its state. Both methods and instance variables can be implemented in the class itself or inherited from an ancestor class.

The browsing system supported a small set of actions, the most important of which was to move through the class inheritance hierarchy. Active browsing cannot succeed if the normal browsing actions are highly "ambiguous", i.e., give very little indication about WHY an agent chooses a particular action. Moving down the inheritance hierarchy is a reasonably unambiguous action: from it, one can plausibly infer that the browsing agent has some interest in the methods implemented in the parent class. Many of the other actions in the original system were highly ambiguous, and were replaced with less ambiguous actions that maintained or enhanced the system’s functionality.

Two browsing actions, which were modified to reduce ambiguity, are of particular note. These are used by the heuristic browsing agent in the experiments detailed in the later sections and are as follows:

1. view the methods implemented in a specific class by:
   a. expanding a class to show a list of the method names that form its interface.
   b. expanding a method to show its argument types.
   c. further expanding the method to show the methods and instance variables used in its processing.
   d. further expanding the method to show its code.
2. request a list of classes that implement a specified collection of methods. This action computes, for every class in the library, the degree to which the class implements the methods and returns a list of all classes ordered from highest to lowest degree.

The active component of the browsing system monitors the browsing agent’s actions. From the sequence of actions, the system constructs an "analogue" of features plus associated certainties, that represents what it believes to be the agent’s search goal. From the analogue the system constructs a template, called a "relevancy measure", which consists of a set of weighted terms that can be directly compared to individual information fields within a library item. With this template the active browsing system scans the library, and computes the (numerical) strength of the match with each library item. The items are ordered according to match strength and presented to the browsing agent in a special window, called the suggestion box.

The inference of the analogue from the action sequence, and the construction of the relevancy measure, are done using production rules and a forward-chaining inference engine supporting MYCIN-like certainty factors [1]. A rule’s antecedent involves browsing actions and/or existing features of the analogue and its consequent causes new features to be generated or old ones updated. In this way the analogue is updated to reflect the browsing agent’s current goal. The analogue’s features are properties of the items in which the user has demonstrated specific interest. The certainties reflect the degree and currentness of that interest. The former is inferred from the action type, the latter is realised by decaying the certainties with time (for further details see [4]). The principle difference between the terms of the relevancy measure template and those of the analogue is that the former has an importance factor as well as a confidence factor. This reflects the significance of the term in locating code in the library. It is multiplied by the confidence factor to assign the term a weight, used when matching the relevancy measure to library items.

2.1 An example of goal extraction

The following example demonstrates that active browsing can help the agent by locating relevant classes, remote in the class hierarchy. Figure 1 shows all classes in one part of the hierarchy. “OTHERS” represents the children of “Object” that are not shown and their descendants.

The main points of interest in this example are four groups of classes, all kinds of collection. They are not all children of a general collection class as the hierarchy in this library is based on code, not specification, inheritance. The central cluster, “Cltn” and its children, are collections whose elements are added and extracted sequentially or by indexing using the list position. There are also two special types: those with methods specific to integers “IntCltn” and “IntOrdCltn” and those with methods specific to pointers “PtrCltn” and “PtrOrdCltn”. The fourth type “SortCltn” has elements that are added and au-

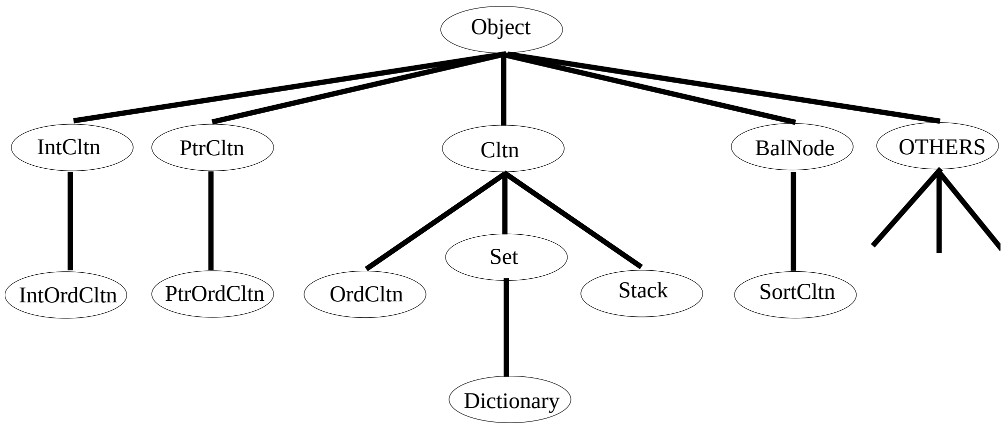
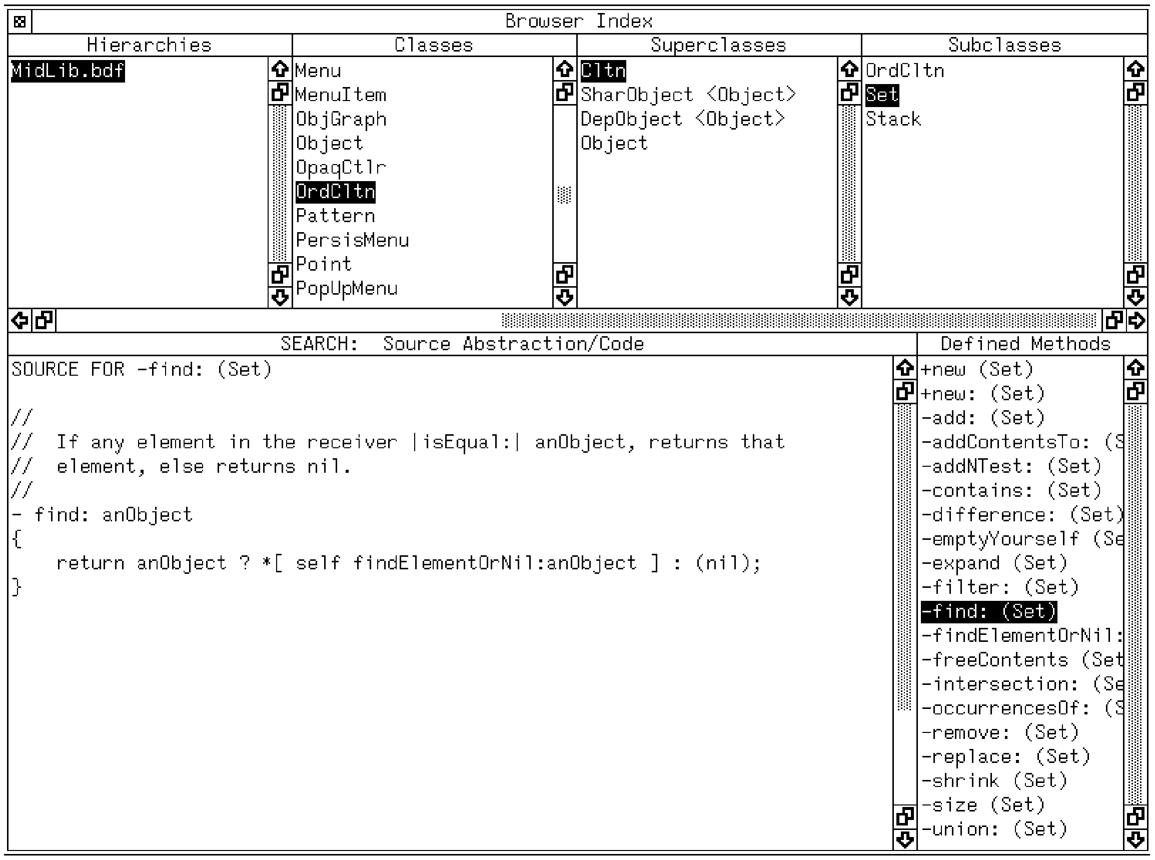


Figure 1 Class hierarchy



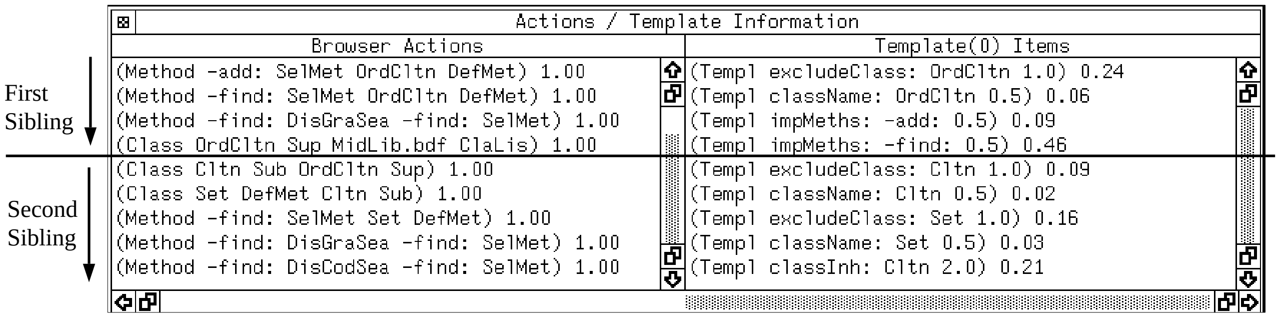
```
SOURCE FOR -find: (Set)

//
//  If any element in the receiver |isEqual:| anObject, returns that
//  element, else returns nil.
//
- find: anObject
{
    return anObject ? *[ self findElementOrNil:anObject ] : (nil);
}
```

Figure 2 Exploring sibling classes

tomatically positioned alphabetically. This type is based on a balanced tree for rapid update and retrieval.

For the purposes of browsing, the inheritance structure is just one of many types of relationship between classes. In object oriented programming this is the principal structure for any library and a likely one for the user to explore. We suppose in this example the user starts by studying some methods in one class and then reviews its siblings as alternatives that have the same general properties. The active browser suggests other classes with similar properties not included in the list of siblings.

Figure 2 shows the state of the user interface of the



```
Browser Actions                                  Template(0) Items
(Method -add: SelMet OrdCltn DefMet) 1.00        (Templ excludeClass: OrdCltn 1.0) 0.24
(Method -find: SelMet OrdCltn DefMet) 1.00       (Templ className: OrdCltn 0.5) 0.06
(Method -find: DisGraSea -find: SelMet) 1.00     (Templ impMeths: -add: 0.5) 0.09
(Class OrdCltn Sup MidLib.bdf ClaLis) 1.00       (Templ impMeths: -find: 0.5) 0.46
(Class Cltn Sub OrdCltn Sup) 1.00                (Templ excludeClass: Cltn 1.0) 0.09
(Class Set DefMet Cltn Sub) 1.00                 (Templ className: Cltn 0.5) 0.02
(Method -find: SelMet Set DefMet) 1.00           (Templ excludeClass: Set 1.0) 0.16
(Method -find: DisGraSea -find: SelMet) 1.00     (Templ className: Set 0.5) 0.03
(Method -find: DisCodSea -find: SelMet) 1.00     (Templ classInh: Cltn 2.0) 0.21
```

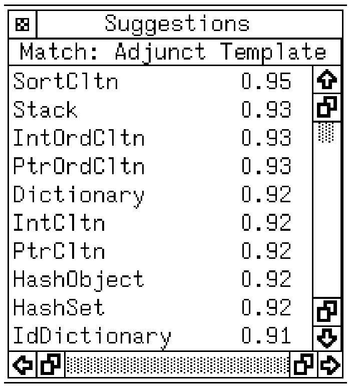
Figure 3 Action and template predicates

browser after the complete sequence of actions described below. The example begins with the user selecting the class “OrdCltn” , from the pane labelled “Classes”, and expanding it to show the method names that define its interface. The methods "-add:" and "-find:" are selected from the list to show their argument types. The latter is expanded to show the other methods and instance variables used in its processing. These actions, in predicate form, and the inferred template predicates are shown in the first four lines of figure 3.

The operator “Superclasses” is then applied to “OrdCltn”, which is highlighted, and the ancestor list containing “Cltn” returned in the pane labelled “Superclasses”. The operator “Subclasses” is then applied to “Cltn”, which is highlighted, and a list of its children displayed in the pane labelled “Subclasses”. The result is a list of sibling classes including “OrdCltn”. The class “Set” is selected, its methods displayed in the pane labelled “Defined Methods” and a method, with the same name, "-find:" is viewed in the pane labelled “SEARCH: Source Abstraction/Code”. On this occasion the user expands it one more level to show the actual code of this variant of the method. The bottom five lines in figure 3 shows these actions and the inferred template predicates.

The template includes the names of the three classes visited and the methods inspected. The method "-find:" has a higher confidence factor than "-add:" as it was selected in both classes and examined more closely. There are template terms to exclude three classes. Those for classes “OrdCltn” and “Set” have higher confidences than that for “Cltn” because the user explored them in detail. The last template term shown in figure 3 is added because the user has explored a number of sibling classes. This term directs the template matcher to compare classes with all methods implemented in, or inherited by, the parent class “Cltn”. Each template predicate has two numbers; a certainty value associated with the user’s interest in a particular feature and a scale factor determining the importance of the term in the matching process. These values are multiplied together to give the weight for the specific template term.

To match maximally with this template a class should have certain properties. It should have an interface similar to class "Cltn" and have a name that matches with "OrdCltn", "Cltn" and "Set". It should implement methods "add" and more importantly "find" and be a class the user has not shown a lot of interest in previously.



SortCltn	0.95
Stack	0.93
IntOrdCltn	0.93
PtrOrdCltn	0.93
Dictionary	0.92
IntCltn	0.92
PtrCltn	0.92
HashObject	0.92
HashSet	0.92
IdDictionary	0.91

Figure 4 Suggested classes

Each class in the library is compared term by term, and the product of the degree of match and the term weight is formed into a normalised sum for all terms. The result of this template being matched is shown in figure 5, the numbers on the right being the normalised sums. The third sibling, “Stack” , as yet unexplored by the user, is not the best match. There is a higher scoring class, “SortCltn” (Sorted Collection). In addition the classes “IntOrdCltn” and “PtrOrdCltn” score above “Dictionary”, a child of “Set”, and several others tie. These and the remaining classes in the list are situated in unrelated parts of the library. The only common ancestor is “Object” the root of the inheritance tree. The high score is due to the extensive polymorphism deliberately designed into the library. These classes are intended to fulfill similar roles to the children of “Cltn” but because of their implementation differences are remote in the hierarchy.

This example demonstrates a plausible situation where active browsing can aid the user in search. The particular actions discussed are unlikely, in reality, to occur in isolation. Rather they would form a subsequence of a much longer series of actions in an actual search. The following section presents experiments to demonstrate the system's effectiveness in longer searches.

3 The heuristic browsing agent

In general, to test the effectiveness of an improvement in an interactive tool, the natural experimental method is to require human subjects to perform tasks using the tool with and without the improvement. There are several practical difficulties in performing an experiment of this kind. The first is that a large number of experimental subjects are required to produce significant results. This difficulty is exacerbated by the fact that human subjects learn during an experiment and cannot be reused in closely related experiments. The use of human subjects also has the disadvantage that it is impossible to run carefully controlled or repeatable experiments. To overcome these difficulties we have used an automated heuristic browsing agent, a notion analogous to using "artificial data" [8].

The heuristic browsing agent has not been designed to accurately simulate the complex behavioural characteristics of a human searcher. Rather it encompasses the general heuristics that a human might be expected to follow. It consists of two parts: a "fuzzy oracle" that represents the agents's current knowledge of the search goal, and a heuristic search strategy that consults the oracle and selects browsing actions. The fuzzy oracle contains a target class selected by the experimenter from the classes in the library. This represents the actual goal of the agent's search. The oracle gives YES/NO answers to questions about if a given library item matches the target class in certain ways. The oracle is "fuzzy" because its answers are not always correct; for each type of question, the probability can be set that the oracle will give an incorrect response. This noisiness represents the browsing agent's uncertainty in recognizing the search goal.

The heuristic search strategy is a combination of depth-first search and hill-climbing. The agent scans the lists of class names from top to bottom and selects the first one on that list that seems of sufficient interest. The selected class is expanded to allow an assessment of its functionality by a closer inspection of its methods. If a group of its methods is of sufficient interest the agent requests a new list of classes that implement these methods. This list is itself searched and the process repeated if a class is found whose functionality is more interesting than all previously inspected classes. If no such class can be found the agent returns to the previous list and carries on the search with the remainder of this list. The search terminates when the agent find the target class.

Two factors combine to cause the top classes in each new list to be more similar to the target, on average, than those of previous lists. The first factor is the hill-climbing. The second is the fact that the noise in the fuzzy oracle is reduced each time a new class is expanded. Thus as search proceeds the heuristic agent gets progressively more accurate answers to its questions. This represents a user's growing certainty in being able to recognize the search goal i.e. the target class.

4 Experimental results

The experiments are in two parts. The first part measures how successful the active browsing system is in inferring the target class from the agent's actions. Two measures are used; the frequency that the active browsing system identifies the target class prior to its being found by the browsing agent and a comparison of the "closeness" to the target class of the agent and the system, across the whole search. The second part measures the effect on the agent's search of using the suggestion box. The number of steps required to find the target class is compared to that required without the use of the suggestions.

To obtain the "closeness" measure, each time the browsing agent creates a new class list or backtracks to a previous list, we record the rank of the target class in the agent's current list (henceforth called the agent's ranking) and the rank, at that moment, of the target class in the suggestion box produced by the active browsing system (henceforth called the system's ranking). For example, the information shown in figure 5, is recorded for the target class "ProbableMethodsList".

Step Number	System's Ranking	Agent's Ranking	Difference
1	16	20	4
2	13	20	7
3	9	33	24
4	6	20	14
5	6	28	22
6	3	28	25
7	3	28	25
8	4	39	35

Figure 5 Record of ranking during search

The first row gives the rankings after the browsing agent has made its first step, i.e., created its first new class list. The target class is 20th in this list. The target class is ranked slightly higher (16th) by the active browsing system. As the agent proceeded in its search, either by

Definition Set	Low Noise			Moderate Noise			High Noise		
	Wins	Losses	Draws	Wins	Losses	Draws	Wins	Losses	Draws
#1	84	55	50	78	66	45	61	76	52
#2	44	79	66	40	90	59	29	99	61

Table 1 Inferring the target class: Wins-Losses-Draws

creating new class lists or backtracking to previous ones, its ranking of the target class actually got worse, dropping at step 8 to 39th place. This is not a surprising effect. The agent may well evaluate a number of different variants on a theme. For instance, different collections of methods might be investigated or methods with similar but not identical names. Thus the search will, at least temporarily, move in the “wrong” direction. Despite this fact, the history of the actions carries enough information for the active browsing system to filter out the characteristic features of the target class. This is demonstrated by the fact that from the third step onwards, the target class is in the system’s top ten.

The search for a given target class is terminated when either (a) the browsing agent finds the target class, or (b) the target class is ranked in the top ten by the active browsing system for six consecutive steps. Criterion (b) applies in the example above.

The experiment consists of successively using each class in the library as the target class. The library is the combination of two commercially available libraries plus classes developed as part of this implementation. It contains 189 classes in total. By varying the noisiness of the oracle’s answers to questions posed by the heuristic agent the effect of a human searcher’s uncertainty can be investigated. The experiment results presented below are for several different noise levels.

4.1 How often is the target class inferred ?

Intuitively, a target class is a "win" (for the active browsing system) if the active browsing system correctly infers it before it is found by the browsing agent. Likewise, a target class is a "loss" if the browsing agent finds the class before the active browsing system infers it. There are a variety of different ways of defining a “win” and a “loss”; we will consider two possibilities.

Definition Set #1: A target class is a win for the active browsing system if it is ranked in the top ten by the active browsing system for five consecutive steps OR if the system’s ranking is higher (better) than the agent’s ranking when the agent finds the target class. A target class is a loss if the agent finds the target class and, in that final step, the target class is not in the system’s top ten. All other target classes are draws.

Definition Set #2: A target class is a win only if it is ranked in the top ten by active browsing system for five consecutive steps. A target class is a loss if the agent finds the target class and, in that final step, the agent’s ranking is higher than the system’s ranking. All other target classes are draws.

The definitions in set #2 are less generous towards the active browsing system, awarding it fewer wins and more losses. There are other realistic definitions of "win" that are more generous than definition #1: for example, one could award a win if the active browsing system inferred a class that was extremely similar to the target class.

Table 1 shows wins-losses-draw for the two sets of definitions and three representative values of the one noise parameter that was varied. The number of wins, according to definition #2, is much smaller, but in assessing the success rate it is important to bear in mind that a win, defined in this way, is impossible for target classes that are found by the browsing agent in fewer than five steps. There are 75 such target classes; if these are disregarded, active browsing succeeds on almost exactly the same percentage of classes with either definition.

4.2 Step-by-step comparison of the rankings

The measurement of wins-losses-draws directly indicates how often the target class is highly ranked by the active browsing system at the termination of the search, but it gives no indication of how the ranking evolved as the search progressed. If a browsing agent is to benefit from an active browsing system in practice, it must be true that throughout the search the system’s ranking of the target class is consistently significantly higher than the agent’s own ranking of the target.

The difference between the agent’s ranking and the system’s ranking is plotted in Figure 6 for two different values of the noise parameter. The results for low noise are plotted with the solid line and the results for moderate noise are plotted with the dotted line. Each step in the agent’s search for the target class corresponds to a different point on the X-axis. The Y-axis indicates the difference between the agent’s and the system’s ranking

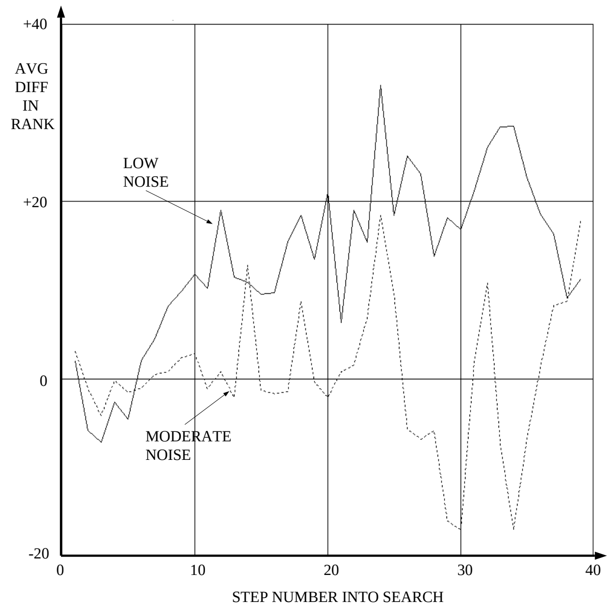


Figure 6 Step by step comparison of rankings

on a given step, averaged over all the target classes. A positive Y-value indicates that the target class is ranked higher by the active browsing system than by the agent.

Note that the search for some target classes involves fewer steps than the search for others. For example, 75 target classes have a search involving 4 or fewer steps. Thus, as one moves right along the X-axis, the number of classes contributing to the Y-value decreases rapidly; the data for large values of X are based on the small number of target classes that require many steps to find.

The graph for low noise shows that the target class is consistently ranked much higher by the active browsing system than by the browsing agent on all steps except the first few. A similar pattern occurs for other noise values, but as the noise value increases the difference diminishes and becomes more erratic. As the noise level is further increased the rank difference gradually returns to being consistent and significant, but with the browsing agent having a better ranking than the active browsing system.

Instead of measuring the numerical difference in rankings, one could simply measure the sign of the difference in rankings at each step [4]. This indicates, for each step, how frequently the target class is ranked higher by the active browsing system than by the browsing system. A similar pattern emerges. For low to moderate noise levels, after the first few steps of search, roughly 70% of the target classes are ranked higher by the active browsing system. Unlike the numerical rank difference, this measure declines quite slowly as the noise level increases; for the high noise level, the system's ranking is higher than the agent's on 45% of the target classes.

4.3 Selecting from the suggestion box

One extension to the experiment, more akin to how the tool would be used in practice, is to have the agent make selections from the suggestion box. Here, the agent makes a selection from either the suggestion box or the previous class list with equal probability, every time it backtracks. This follows the intuitive notion that the suggestions are most likely to be used when a human user is unsure of how to proceed. At this juncture the user may well look at the suggestion box or unexplored classes in earlier lists.

The results in table 2 are for the low and moderate noise values. For each value the experiment is run twice, with and without the use of the suggestion box respectively. The search is terminated when the agent finds the target class. The number of steps required to find each class in the library is compared. In this case only a single criterion is used. A win is assigned to the active browser if it takes fewer steps when turned on, a loss if it takes more and a draw if they are equal. The results show a significant benefit at the lower noise values, some of which is lost at the higher values. The ratio of wins to losses is comparable to that given by the upper bound of the simple comparison experiments.

Low Noise			Moderate Noise		
Wins	Losses	Draws	Wins	Losses	Draws
52	32	105	51	44	94

Table 2 Using the suggestion box

5 Discussion of results

The results show that the active browsing system frequently infers the target class before it is found by the browsing agent, and that during search the active browsing system consistently ranks the target class significantly higher than the browsing agent. Further, when the agent makes use of the suggestion box the speed of search is improved, on average. There are two conditions under which active browsing does not outperform the browsing agent: during the early steps of search, and when the browsing agent is highly uncertain about the search goal (simulated in the experiments by a high noise level).

That active browsing does not outperform the agent in the early steps of search can be explained by it having insufficient information upon which to base its rankings.

There are numerous factors that might cause the performance of active browsing system to be degraded by increased noise levels more than the agent's performance. The most likely explanation is that the agent's behaviour consisted of a large number of uninformative actions followed by a relatively small number of highly informative actions that led the agent to the target. If the agent behaves in this way, the active browsing system will lag behind the agent in moving towards the target because the final few informative actions will not immediately outweigh the mass of previous uninformative actions. This kind of "misleading" initial behaviour by the agent is promoted by the fact the oracle's noise level decreases as the agent's search progresses, and is further promoted by the fact that in the test library, as in most software libraries, there are some fairly large groups of highly similar items. This latter situation can be readily detected. Both the analogue and the relevancy measure template terms could be adjusted to reduce the effect, although this idea has not been experimentally validated at the present time.

6 Conclusion

This paper has described a system for active browsing and experimentally demonstrated that a browsing agent's search goal can be inferred from normal browsing actions. The inferred goal provides a reliable way of estimating the "relevance" of a library item to the agent's actual search goal. Active browsing increases the effectiveness of browsing without imposing any restrictions or "cost" on the user. The agent is able to make full use of the facilities of the standard browser, and is not required to take special actions or learn any new tools.

Future experiments should use a rich variety of realistic browsing agents. Further work on agents using the system's suggestions would permit measurement of the relative effectiveness of different ways of presenting the inferred goals to the browsing agent.

7 Acknowledgments

The work was supported in part by a Strategic grant and operating grants from NSERC of Canada. Colleagues on the "Machine Learning Applied to Software Reuse" project were helpful at all stages of this research.

8 References

[1] J.A. Alty & M.J. Coombs. Expert Systems: Concepts And Examples National Computing Centre Pubs. (1984)

[2] G. A. Boy. Indexing Hypertext Documents In Context. Proc. 3rd ACM Conference on Hypertext (1991).

[3] C. Drummond, D. Ionescu & R. Holte, Automatic Goal Extraction from User Actions when Browsing Software Libraries. Proc. Canadian Conference on Electrical and Computer Engineering, pp WA6.31.1-WA6.31.4 (1992)

[4] C. Drummond, Automatic Goal Extraction from User Actions to Accelerate the Browsing of Software Libraries, M.A.Sc. Thesis, University of Ottawa, December 1992.

[5] R. Godin, J. Gecsei & C. Pichet. Design Of A Browsing Interface For Information Retrieval. SIGIR 89 Proc. 12th Int. Conf. on Research and Development in Information Retrieval (1989)

[6] D. Harman. Relevance Feedback Revisited. SIGIR 92 Proc. 15th International Conference on Research and Development in Information Retrieval (1982)

[7] S. Henninger. CodeFinder: A Tool For Locating Software Objects For Reuse. Automating Software Design: Interactive Design Workshop Notes AAAI–91 pp 40–47 (1991)

[8] D. Kibler & P. Langley. Machine Learning As An Experimental Science. Proc. 3rd Working Session On Learning (1989)

[9] T. P. Martin, HK. Hung & C. Walmsley. Supporting Browsing Of Large Knowledge Bases. Dept of Computing and Information Science, Queen's Uni. Kingston Canada. Unpublished (1991)

[10] A. Motro. BAROQUE: A Browser For Relational Databases. ACM Trans. on Office Information Systems Vol 4 No. 2 April 1986 pp 164–181. (1986)

[11] R. H.C. Seabrook & B. Shneiderman. The User Interface In A Hypertext Multiwindow Browser. Interacting with Computers Vol 1 No. 3 1989 pp 301–337 (1989)

[12] R. H. Thompson & W. B. Croft. Support for Browsing In An Intelligent Text Retrieval System. Int. J. Man-Machine Studies Vol. 30 pp 639–668 (1989)

[13] G. Fischer & H. Nieper-Lemke. HELGON: Extending The Retrieval By Reformulation Paradigm. Proc. CHI-89 Human Factors In Computing Systems pp 357–362 (1989)

[14] G.W. Furnas, T.K. Landauer, L.M. Gomez & S.T. Dumais. The Vocabulary Problem In Human-System Communication. CACM. Nov 1987 Vol 30 No 11 pp 964–971 (1987)

[15] S. D. Fraser, J. M. Duran & R. Aubin. Software Indexing For Reuse. Proc. 1989 IEEE International Conference On Systems, Man and Cybernetics pp 853–858 (1989)

[16] R. Prieto-Diaz. Implementing Faceted Classification For Software Reuse. CACM Vol 34 1991 pp 89–97 (1991)

[17] Y. S. Maarek, D. M. Berry & G. E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. IEEE Transactions On Software Engineering Vol. 17 No. 8 Aug. 1991 pp 800–813 (1991)

[18] T. J. Biggerstaff, C. Richter. Reusability Framework, Assessment And Directions. Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis, ACM Press pp 1–17(1987)

[19] E. Horowitz & J.B. Munson. An Expansive View Of Reusable Software. Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis ACM Press pp 19–41 (1984)

[20] D. A. White. The Knowledge-Based Software Assistant: A Program Summary. Tutorial Overview. Proc. 6th Knowledge Based Software Engineering Conference (1991)

[21] W. L. Johnson. Interactive Acquisition Of Requirements For Large Systems. Automating Software Design: Interactive Design Workshop Notes AAAI–91 pp 61–70 (1991)