The FindMe Approach to Assisted Browsing

Robin D. Burke, Kristian J. Hammond & Benjamin C. Young
Intelligent Information Laboratory
University of Chicago
1100 E. 58th St., Chicago, IL 60637
{burke, kris, bcy1}@cs.uchicago.edu

May 30, 1997

Abstract

While the explosion of on-line information has brought new opportunities for finding and using electronic data, it has also brought to the forefront the problem of isolating useful information and making sense of large multi-dimensional information spaces. In response to this problem, we have developed an approach to building data "tour guides," called FindMe systems. These programs know enough about an information space to be able to help a user navigate through it, making sure that the user not only comes away with items of useful information but also insights into the structure of the information space itself. In these systems, we have combined ideas of instance-based browsing, structuring retrieval around the critiquing of previously retrieved examples; and retrieval strategies, knowledge-based heuristics for finding relevant information. We illustrate these techniques with examples of working FindMe systems, and describe the similarities and differences between them.

1 Introduction

What do buying a car, selecting a video, renting an apartment, choosing a restaurant, and picking out a stereo system have in common? They are all tasks that require an individual to pick from a large collection of similar items one which best meets that person's unique needs and tastes. Because

there are many interacting features of each item to consider, such selection tasks typically require substantial knowledge to perform well. Our aim is to build systems that can help users perform such tasks, even when they do not have a lot of specific knowledge. Our approach, called *assisted browsing*, combines searching and browsing with knowledge-based assistance.

Suppose you want to rent a video. You are in the mood for something like *Back to the Future*. What are your options? You might want to see the sequel, *Back to the Future II*. Or maybe you want to see another movie about a person dropped into an unfamiliar setting, such as *Crocodile Dundee*, or *Time After Time*, another time-travel movie. If you really enjoyed the way *Back to the Future* was directed, maybe you would like *Who Framed Roger Rabbit?* another Robert Zemeckis picture. Or perhaps, you want to see another film starring Michael J. Fox, such as *Doc Hollywood*. No computer system can tell you what movie to see, but an intelligent assisted-browsing environment can present you with these choices (and others), getting you to think about what you liked in *Back to the Future*.

The aim of assisted browsing is to allow simplified access to information along a multitude of dimensions and from a multitude of sources. Since browsing is the central metaphor, we avoid as much as possible forcing users to create specific queries. Knowledge-based retrieval strategies can be employed to consider all of the dimensions of the information and present suggestions that lead the user's search in reasonable directions. We have implemented our assisted browsing approach in a series of systems called FindMe systems. They are

Car Navigator: Selecting a new car,

Video Navigator & PickAFlick: Choosing a rental video,

RentMe: Finding an apartment,

Entree: Selecting a restaurant,

Kenwood: Configuring a home audio system.

We see the FindMe approach as applicable to any domain in which there is a large, fixed set of choices and in which the domain is sufficiently complex that users would probably be unable to fully articulate their retrieval criteria. In these kinds of areas, person-to-person interaction also takes the form of trading examples, because people can easily identify what they want when they see it.

Figure 1 shows the entry point for ENTREE, a restaurant guide for the city of Chicago. Users can pick from a set of menu options to describe what they are looking for in a restaurant: a casual seafood restaurant for a large group, for example, or they can, as shown here, type in the name of a restaurant in some other city for which they are seeking a local counterpart.

---- Figure 1 goes here ----

Figure 1: The initial screen for Entree

The system retrieves restaurants in the Chicago area that are considered to be similar to the user's choice of Boston's "Legal Seafood," the top contender being "Bob Chinn's Crabhouse" as shown in Figure 2. The user can now continue to browse the space of restaurants by using any of the seven *tweaks*, modifications to the example. The user can ask for a restaurant that is nicer, or less expensive, one that is either more traditional or more creative, one that is quieter or more lively, and also has the option of looking for a similar restaurant but with a different cuisine.

---- Figure 2 goes here ----

Figure 2: Tweaking in Entree

This example shows some of the kind of intelligent assistance and other interface techniques that are used in FINDME systems:

Similarity-based retrieval: As has frequently been found in other information retrieval contexts, it is useful to allow a user to retrieve new items that are similar to an example currently being viewed [12, 14]. We found that in most cases overall similarity of features was a poor metric for providing examples, because users attached different significance to features depending on their goals. For example, if your goal is to buy a car that will pull a big trailer, you will weight engine size more heavily when comparing cars than other features such as passenger leg room. So, the system should regard engine size as more significant in assessing similarity in this context.

Tweaking: Browsing is typically driven by differences: if a user were totally satisfied with the particular item being examined he or she would stop there. But, an unsatisfactory item itself can play a useful role in articulating the user's goals. For example, if you are looking for a science fiction movie to rent, you might look at *Terminator II*, but think "That would be good, but it's too violent for my kids." The examination of a particular example can bring to mind a new feature, such as level of violence, that becomes an explicit part of further search.

Retrieval using high-level categories: The menus in the ENTREE interface present abstract categories that are likely to be of interest to the user, not the specific low-level features found in the system's data. For example, if the user asks for a seafood restaurant with a casual atmosphere, ENTREE must use knowledge of what "casual" means in the context of the database to create a query – restaurants are not described by degree of casualness.

Multiple similarity metrics: Although not shown in the above example, ENTREE also incorporates several different similarity metrics used in different contexts. If the user asks for a "very expensive" restaurant, instead of using this feature to retrieve, ENTREE actually invokes a different similarity metric, one that does not consider price. Essentially, we interpret the user's request as meaning "Money no object." Multiple similarity metrics also let a FINDME system arrive at several different suggestions, relevant for different reasons.

Although not present in ENTREE, there are other categories of interaction we have found useful:

Explanations of trade-offs: Users, especially in unfamiliar domains, may fail to understand certain inherent trade-offs in the domain they are exploring. A car buyer might not understand the trade-off between horsepower and fuel efficiency, and attempt to search for a high-powered car that also gets 50 miles to the gallon.

Browsing using low-level features: Some of our early systems allows users to browse using direct manipulation of the low-level features by which data elements were described. We found in informal evaluations that few users took advantage of these capabilities.

These mechanisms are part of a dialogue between system and user in which the user comes to a better understanding of the domain of examples (through learning about trade-offs and seeing many examples) and the system helps the user find specific items of interest by gradually refining the goal.

2 Technical Overview

At the highest level of abstraction, all FindMe systems are very similar. They contain a database, they retrieve from it items that meet certain constraints, and they rank the retrieved results by some criteria. What gives each FindMe system its character is the details of how this general pattern is instantiated for any given domain, particularly in what criteria are used for retrieval, what criteria are used for ranking results, what tweaking transformations are incorporated into the system, and what additional knowledge is brought to bear in addition to the database itself.

It is important in building a FindMe system to understand the relationship between features of the data and the selection task itself. We cannot make use of all features available in a database in a uniform way. The hours of operation of a restaurant are rarely as important as how much a typical meal costs, for example. Also, it does not make sense to build every possible tweaking option into the interface. Rarely would a user look at an apartment and say "I want something just like that, but more expensive." FindMe systems concentrate on a single possible use for the data in a database, but because of this focus, they can provide more assistance to the user.

2.1 The Entree implementation

Entree contains the essence of the FindMe idea, stripped down to its essentials. It therefore makes a good example with which to explain the functionality of a FindMe system. Entree consists of a handful of perl scripts that handle the output of web pages, and several C++ programs that implement FindMe functionality. Conceptually, a restaurant r is represented in the system as a tuple $< i, d, N, F >$, where i is a unique integer identifier for each restaurant used to index the tuple, d contains the name of the restaurant and other descriptive text about it to be displayed for the user's benefit, N is a set of indexed *trigrams*, a decomposition of the restaurant's name into three letter sequences (see below), and F is the set of features of the restaurant itself.

When the user enters ENTREE from the initial page (Figure 1), there are two possibilities (a) a particular restaurant has been entered as a model, or (b) a set of high-level features has been selected from the set of menus.

In the first case, the system must attempt to find a restaurant with the name the user has supplied. Since users are likely to mistype, misremember or misspell such names, we have a fuzzy string comparison routine that uses trigrams, looking for the restaurant name in the database that shares the most trigrams with what the user typed. The comparison is also sensitive to the location in the name where the sequence occurs. We find that this enables many misspellings to match with the correct restaurant. The name matcher returns the id for the restaurant that the user has named, which is used to lookup the corresponding feature set.

In the case of the second entry point, the user selects a set of high-level features describing their dining interests. For example, a casual seafood restaurant for a large group. These high-level features are decomposed into a set of low-level database features. In either case, the entry point provides the system with a set of features, F. In the menu case, ENTREE also gets some goal information it can later use to tune its ranking of results.

The next step is retrieval of R, the set of all restaurants containing one or more features from F. R is a large set, typically 20-50% of the entire database. For example, when the user selects "Legal Seafood," we retrieve all Chicago restaurants that serve seafood, but also all that charge about $15, all that are similarly casual, etc.

On R, we perform a hierarchical sort. Suppose we have an ordered list of goals $\{G_1, \ldots, G_k\}$, we can apply the goal-related metric M_{G_1} to each retrieved example. Since the metric is discrete, we can create equivalence classes or buckets based on the score returned by this metric: $B^1_{G_1}, B^2_{G_1}, \ldots$. Then the examples are ranked within each bucket with respect to the next most important goal, creating $B^{1,1}_{G_{1,2}}, B^{1,2}_{G_{1,2}}, \ldots, B^{2,1}_{G_{1,2}}, \ldots$, another series of more finely-discriminated buckets. We can repeat this process until either all possible ranking operations have been performed or there is a totally-ordered set of examples to show. To make the process efficient, we continuously truncate the set of buckets so that it contains only the minimum number of buckets needed to answer the query.

ENTREE has a default ordering of goals that is assumed if the user enters a restaurant by name: cuisine is the first priority, followed by price. So the first two passes on sorting will return all of the seafood restaurants ordered inversely by price, starting from the $15 price bracket. The next goal the system assumes is atmosphere: the feel of the dining experience. Finally, we

use ratings of quality to rank the final list.

Hierarchical sort is preferred over other ordering schemes, such as weighted sum scoring, because of the efficiencies allowed by working with one feature at a time. Our retrieval method is extremely promiscuous, so evaluating a complete similarity metric on every retrieved item would be prohibitive. With a hierarchical method, we can use a succession of simple tests that quickly trim the set of items under consideration to a manageable size. The other benefit of hierarchical sorting is that preserves an absolute ordering of goal consideration. Sorting by the second most important goal will impose an ordering only on those items already determined to be equivalent with respect to the most important goal.

The restaurants are returned on a "results" page as shown in Figure 2, with a single restaurant highlighted, and a list of links to other similar restaurants below. Each of these links returns another results page, differing only in that the chosen restaurant is highlighted instead.

All of the tweaks are implemented in essentially the same way. We perform retrieval based on similarity, just as described above, however, we then filter out of R all of the restaurants that do not satisfy the tweak given. For example, if the user looking at Figure 2 decides to look for something "nicer," the system would calculate how "nice" it thinks "Bob Chinn's Crab House" is, and then creates a subset R' of all of the restaurants in R that are nicer than "Bob Chinn's." It then performs the ranking in exactly the same way as before, looking at cuisine, price, atmosphere, etc. In some cases, the result of the tweak filter will be empty, in which case, we report to the user that there are no more restaurants along the given dimension within the preferred cuisine. The system will not switch from "seafood" to "French" in order to continue along the "nicer" dimension, because cuisine is so basic to the restaurant-finding task. This option is available to the user via the "Cuisine" tweak.

3 Some FindMe Systems

In general, as we have built FindMe we have worked from domains with small spaces of examples in which features are well-defined and user goals are straightforward, to larger domains with fuzzier features and more complex user goals. Each of the systems is profiled in this section.

3.1 Car Navigator

The first FindMe system was the Car Navigator, an assisted browsing system for new car models. Using the interface, which resembles a car magazine, the user flips to the section of the magazine containing the type of car he or she is interested in. Cars are rated against a long list of criteria such as horsepower, price or gas mileage, which are initially set by default for the car class, but can be directly manipulated. Retrieval is performed by turning the page of the magazine, at which point the criteria are turned into a search query and a new set of cars is retrieved. Depending on how the preferences have changed, the system may suggest that the user move to a different class of cars. For example, if the user started with economy cars and started to increase the performance requirement, the system might suggest sports cars instead. Figure 3 shows the user interface for Car Navigator.

---- Figure 3 here ----

Figure 3: The interface for Car Navigator

It is possible for the user to set the preferences to an impossible feature combination: one that violates the constraints present in the car domain. This triggers an explanation of the trade-off that the user has encountered. For example, if a user requests good gas mileage and then requests high horsepower the yellow light will come on next to the gas mileage and horsepower features. The system explains that there is a trade-off between horsepower and gas mileage, and the user will have to alter his or her preferences in one area or the other.

In addition to the fine-grained manipulation of preferences, Car Navigator permits larger jumps in the feature space through buttons that alter many variables at once. If the user wants a car that is "sportier" than the one he is currently examining, this implies a number of changes to the feature set: larger engine, quicker acceleration, and a willingness to pay more, for example. For the most common such search strategies, Car Navigator supplies four buttons: *sportier*, *roomier*, *cheaper*, and *classier*. Each button modifies the entire set of search criteria in one step. Although direct manipulation of the features was appealing in some situations, we found that most users preferred to use the retrieval strategies to redirect the search.

The implementation details for this system are outlined in Table 3.1.

System	CAR NAVIGATOR
Platform	Macintosh
Language	Macintosh Common Lisp (4000 lines)
	C (3800 lines)
Database	Lisp internal
Data Size	1 MB (600 cars)

Table 1: Implementation details for CAR NAVIGATOR

The interface was implemented in C, and the database in Lisp, using TCP streams to pass retrieval requests. This design made the interface very responsive, but still allowed us the maximum flexibility in our handling of data.

3.2 VIDEO NAVIGATOR and PICKAFLICK

We used our experience in building CAR NAVIGATOR in the construction of a system for browsing movie videos for rental. This system, VIDEO NAVIGATOR, draws on a database of 7500 movies from a popular video reference work [13]. The system is organized as a sequence of shelves divided into categories. The user has several tools that can be used to make queries into the shelves. Once at a particular shelf, the user can select movies and look at additional information about them, such as plot summaries, cast lists, etc.

The retrieval mechanism in VIDEO NAVIGATOR is implemented in a set of interface agents, called *clerks*. This design choice was due to the nature of the movie domain. Users have seen more movies than they have cars. They know more points in the information space, so need less help from the system in getting around. The clerks remain passive until the user selects a particular movie to examine. There are four clerks: one recalls movies based on their genre, one recalls movies based on their actors, another on directors, and still another arrives at suggestions by comparing the user against the profiles of other users. Whenever the user picks a movie to inspect, each clerk retrieves and suggests another related movie. It is as if the user has a few knowledgeable movie buffs following her around the store, suggesting movies based on their particular area of expertise. The user can choose to follow up or ignore the suggestions. Figure 4 shows the interface for VIDEO

System	VIDEO NAVIGATOR
Platform	Macintosh
Language	Macintosh Common Lisp (4700 lines)
Database	Lisp internal
Data Size	1.9 MB (7500 movies)
	+ 1.4 MB knowledge base

Table 2: Implementation details for VIDEO NAVIGATOR

NAVIGATOR.

It turned out to be difficult to implement tweaking in the movie domain. While we could easily derive buttons that might be useful: "less violence," for example, it quickly became clear that there were too many possible tweaks to have buttons for each. Ultimately, we would like to have users supply tweaks in natural language phrases and use simple natural language processing techniques to allow the system to recognize tweaks such as "too violent," "I hate musicals," or "Not Mel Gibson."

---- Figure 4 goes here ----

Figure 4: The interface for VIDEO NAVIGATOR

The implementation of VIDEO NAVIGATOR is summarized in Table 2. We built the system entirely in Macintosh Common Lisp, using the built-in interface development tools. The knowledge base in VIDEO NAVIGATOR consists of similarity relations between actors and influence relationships between directors.

Using the same knowledge base and algorithms, we created PICKAFLICK, an adaptation of VIDEO NAVIGATOR to the World-Wide Web. Instead of a browsing interface with a map and shelves, we allow the user to enter the name of a known movie in free text. This movie is used to generate suggestions using retrieval strategies like the clerks in VIDEO NAVIGATOR.

For example, suppose the user enters the name of the movie *Bringing Up Baby*, a classic screwball comedy starring Cary Grant and Katharine Hepburn. PICKAFLICK locates similar movies using three different strategies. First, it looks for movies that are similar in genre: other fast-paced come-

System	PickAFlick
Platform	Web (Sun Solaris)
Language	Allegro Common Lisp (2500 lines)
	perl (1500 lines)
Database	Lisp internal
Data Size	12 MB (80,000 movies)

Table 3: Implementation details for PickAFlick

dies. As Figure 5 shows, it finds *His Girl Friday*, another comedy from the same era starring Cary Grant, as well as several others. The second strategy looks for movies with similar casts. This strategy will discard any movies already recommended, but it finds more classic comedies, in particular *The Philadelphia Story*, which features the same team of Grant and Hepburn. The director strategy returns movies made by Howard Hawks, preferring those of a similar genre.

```
---- Figure 5 goes here ----
```

Figure 5: A search result from PickAFlick

We expanded our movie database when moving to the web platform, handling an order of magnitude more movies, as shown in Table 3. Since the Lisp image containing this database took about 60 seconds to load and initialize, it was impractical to load and run it in response to each web request. We set up the Lisp image as a server, responding to requests from a TCP stream. This stream is created and managed by a set of perl scripts that handle the web requests using the CGI protocol to interact with our web server.

3.3 RentMe

All of our subsequent FindMe systems have been web-based. RentMe is an interface to a database of classified ads for rental apartments. A typical apartment seeker might have a goal like "I'd like a place like what I have now but a little bigger and in a neighborhood with more stuff to do

nearby." Notions such as "like the apartment I live in now" are idiosyncratic and can only be evaluated by the user examining a particular apartment listing. Another important aspect of the goal stated above is its reference to knowledge outside of the domain of the apartment listings themselves. To know whether a neighborhood has "more things to do," one must know something about the city itself.

The entry point for RentMe is a set of menus: for neighborhood, price and size. The list of apartments meeting these constraints forms the starting point for continued browsing. As shown in Figure 6, the user can improve the search by selecting any apartment and using it as the basis for further retrieval by tweaking. The "Cheaper" button is used to tell the system to find similar apartments that are cheaper. The system performs another round of retrieval, keeping in mind the features of the apartment the user originally selected. As shown in Figure 7, it only finds one acceptable apartment in the same neighborhood, so it relaxes the neighborhood constraint and begins to look at other, similar, neighborhoods for cheaper apartments.

---- Figure 6 goes here ----

Figure 6: Tweaking an apartment in RentMe

---- Figure 7 goes here ----

Figure 7: The result of applying the "cheaper" tweak

RentMe starts not from a database, but from a text file of classified ads for apartments. It builds the database from this text using an expectation-based parser [10] to extract features from the very terse and often-agrammatical language of the classified ads. Expectation-based parsing makes it possible to distinguish between "No dogs" and "Dogs welcome," a distinction lost to many keyword-based approaches.

RentMe was implemented much like PickAFlick with a Lisp server containing the database and perl scripts running on a Unix platform. The Lisp code base is large because RentMe also contains the code for the natural language parser.

System	RentMe
Platform	Web (Sun Solaris)
Language	Allegro Common Lisp (9500 lines)
	perl (1200 lines)
Database	Lisp internal
Data Size	2.8 MB (3700 apartments)

Table 4: Implementation details for RentMe

3.4 Entree

Entree was our first attempt to build a FindMe system that was sufficiently stable, robust and efficient to survive as a public web site. Our previous systems were implemented in Common Lisp, and could not be made available for public access without regular monitoring. Also, all of the FindMe systems discussed so far keep the entire corpus of examples in memory (the Lisp workspace). This technique has the advantage of quick access and easy manipulation of the data, but it is not realistic in that it cannot be easily updated or scaled up to very large data sets. We use a combination of `dbm`, a free database package for Unix, and flat text files to store the data for Entree.

The system has been operation on the World-Wide Web since August 1996 in the configuration described in Table 5. It was first used by attendees of the Democratic National Convention in Chicago. In addition to its database of restaurants, Entree also has knowledge of cuisines — in particular, the similarities between cuisines. This enables it to smooth over some of the discontinuities that exist between our different data sources. In some sources, "Tex-Mex" was considered a cuisine, in others only "Mexican" was used.

3.5 Kenwood

Our most recent FindMe system allows users to navigate through various configurations for home theater systems. The user can enter the system two ways: by selecting a budget or by identifying particular components they already own. The user also must specify the type of room the system will operate in. The user can browse among the configurations by adjusting the budget constraint, the features of the room or by adding, removing or

System	ENTREE
Platform	Web (Sun Solaris)
Language	C++ (3200 lines)
	perl (1200 lines)
Database	`dbm` and flat text
Data Size	2.2 MB (4400 restaurants)
URL	`http://infolab.cs.uchicago.edu/entree/`

Table 5: Implementation details for ENTREE

replacing components. Since we are dealing with configurations of items, it is also possible to construct a system component by component and use that system as a starting point. This makes the search space somewhat different than the other systems discussed so far, in that every combination of features that can be expressed actually exists in the system.

Figure 8 shows the system after the user has asked to look at a systems around $1500. The bottom part of the screen has button to alter the parameters around which the configuration was built: the price tag, the room size, and particular components involved.

`---- Figure 8 goes here ----`

Figure 8: Results retrieved by the KENWOOD system.

Our database in KENWOOD is not of individual stereo components and their features, but rather entire configurations and their properties. Although the database is large as indicated in Table 6, each entry in the database is very simple, just the price for the overall configuration, the constraints it satisfies, and a flag for each of the possible components. An adapted version of KENWOOD is currently part of the web presence for Kenwood, USA, and is accessible from `<URL:http://www.mykenwood.com/Build/>`. (Choose "Build System.")

KENWOOD was our first experience using standard database techniques for data access. In ENTREE, we used `dbm` and flat text files of our own design, obviously not a method easily scaled or extended. The KENWOOD system uses a subset of the Standard Query Language (SQL), as implemented in

System	KENWOOD
Platform	Web (Sun Solaris)
Language	C++ (2800 lines)
	perl (2700 lines)
Database	mSQL
Data Size	3 MB (340k configurations)
URL	`http://www.mykenwood.com/Build`

Table 6: Implementation details for KENWOOD

`mSQL`.[1] The structure of the data and the constraints of the interface to the database meant that we could not use the same retrieval techniques as in our other systems. Instead we create queries for specific values, and then relax the constraint in steps if no answers are found. This is an inefficient method, but the domain of examples in KENWOOD is sufficiently compact that the system rarely has to relax more than a single increment. The knowledge in the KENWOOD system is in the system's relaxation techniques, and in the construction of the database itself, which is essentially an encoding of a similarity metric over the different configurations.

3.6 Summary of Implemented Systems

Table 7 gives an overview of the different FINDME searching and browsing techniques that are employed in each of the systems we have discussed. Not every technique is appropriate in every domain and as the table shows no system actually makes use of all of the techniques we have explored.

In general, we have tried to minimize interface complexity, particularly in our web-based applications. Usually that precludes the use of low-level features in retrieval because any interface that presents all such features would be cumbersome. Similarly, in the movie domain, the set of high-level features (particularly all the different genres and subgenres known to the system) was simply too large to be easily presented: the example-trading interface for PICKAFLICK enabled us to provide the FINDME functionality cleanly and simply.

[1] The `mSQL` implementation that is freely available for non-commercial use from `ftp://bond.edu.au/pub/Minerva/msql/`. A nominal licensing fee is required for commercial use.

	User Input				System Responses	
FindMe System	Low-level features	High-level features	Examples	Tweaks	Multile metrics	Trade-offs
Car Navigator	X		X	X		X
Video Navigator	X	X	X		X	
PickAFlick			X		X	
RentMe	X			X		X
Entree		X	X	X	X	
Kenwood		X		X		

Table 7: Summary of FindMe systems and their capabilities

Interface constraints also entered in the decision not to employ tweaking in Video Navigator and PickAFlick. There are simply too many things that the user might dislike about a movie for us to present a comprehensive set of tweak buttons. The natural language tweaking capacity discussed above is the most likely candidate for a tweaking mechanism in this domain, but that remains a goal for future research.

One important distinguishing characteristic between domains in the degree of "naming" that one can expect. Some objects like restaurants and movies have well-defined names, but apartments and stereo systems do not. Naming is essential in using examples as retrieval cues: we would otherwise have to require that users completely define the features of an example they want the system to work from.

Explanations of trade-offs are only useful in domains where the trade-off is meaningful and where tweaks are implemented. In the domain of movies, it isn't particularly meaningful to report that user has tried to move into a corner of the space where no movies exist. If you want something more violent than "Texas Chainsaw Massacre" and no such movie exists, there probably is little useful feedback the system could give.[2]

While the FindMe research project has demonstrated the wide applicability of our research ideas to many information access domains, and given us confidence that they apply in many more, we have not to date demonstrated the effectiveness of the systems in any formal sense. It would be

[2]On the other hand, a movie producer might be quite interested in identifying parts of the space of possible movies where users keep coming up empty-handed.

useful to create a "standard" database system incorporating the same data as ENTREE for example, and compare the performance of users given the same task to perform on a FINDME system versus the standard.

One issue to be addressed in such an evaluation is the design of an appropriate evaluation task. One of our research goals in FINDME systems is to help users find items that are meaningful and useful to them, even if they cannot fully articulate why the item is appropriate. This makes artificial evaluation tasks inappropriate: for example, if users were assigned the task of "finding a good steakhouse for dinner," different users might be expected to find different restaurants, since they would probably evaluate them differently. We would also need to evaluate cognitive change as a result of using the system to determine if the system has succeeded in teaching users about the structure of the data space.

One important result of our continued research has been the refinement of the core FINDME engine. This shell forms the computational component of ENTREE and could be easily adapted with a different knowledge base to operate in any FINDME domain. In its current state, the knowledge base must be encoded in the system as large constant data structures. We are developing a more flexible version of the shell that would operate on declarative knowledge structures, and on the knowledge acquisition tools needed to create those structures for any database and domain.

4 Related Work

The problem of navigating through complex information spaces is a topic of active interest in the AI community. (See, for example, [2, 5, 6].) Much of this research is directed at browsing in unconstrained domains, such as the World-Wide Web, where pages can be on any topic and users' interests are extremely varied. As a result, these systems must use knowledge-poor methods, typically statistical ones.

Our task in FINDME systems is somewhat different. We expect users to have highly-focused goals, such as finding a suitable apartment to rent. The data being browsed all represents the same type of entity, in the case of RENTME, apartment ads. As a result, we can build substantial, detailed knowledge into our systems that enables them to identify trade-offs, compare entities in the information space, and respond to user goals. All of these properties make FINDME systems more powerful than general-purpose browsing assistants.

In the area of information retrieval, browsing is usually a poor cousin to retrieval, which is seen as the main task in interacting with an information source. The metrics by which information systems are measured do not typically take into account their convenience for browsing. The ability to tailor retrieval by obtaining user response to retrieved items has been implemented in some information retrieval systems through relevance feedback [9] and through retrieval clustering [3].

Our approach differs from relevance feedback approaches in both explicitness and flexibility. In most relevance feedback approaches, the user selects some retrieved documents as being more relevant than others, but does not have any detailed feedback about the features used in the retrieval process. In FindMe systems, feedback is given through the use of tweaks. The user does not say "Give me more items like this one," the aim of relevance feedback and clustering systems, but instead asks for items that are different in some particular way.

Examples have been used as the basis for querying in databases since the development of Query-By-Example [12]. Most full-feature database systems now offer the ability to construct queries in the form of a fictitious database record with certain features fixed and others variable. The RABBIT system [14] took this capacity one step further and allowed retrieval by incremental reformulation, letting the user incorporate parts of retrieved items into the query, successively refining it. Like these systems, FindMe uses examples to help the user elaborate their queries, but it is unique in the use of knowledge-based reformulation to redirect search based on specific user goals.

Another line of research aimed at improving human interaction with databases is the "dynamic query" approach [11]. These system use two-dimensional graphical maps of a data space in which examples are typically represented by points. Queries are created by moving sliders that correspond to features, and the items retrieved by the query are shown as appropriately colored points in the space. This technique has been very effective for two-dimensional data such as maps, but only when the relevant retrieval variables are scalar values representable by sliders.

Like FindMe, the dynamic query approach has the benefit of letting users discover trade-offs in the data because users can watch the pattern of the retrieved data change as values are manipulated. However, dynamic query systems have no declarative knowledge about trade-offs, and cannot explain to users how they might modify their search or their expectations in light of the trade-off. Also, as we found in Car Navigator, direct manipulation is less effective when there are many features to be manipu-

lated, especially when users may not be aware of the relationships between features.

Our use of knowledge-based methods to the retrieval of examples has its closest precedent in retrieval systems used in case-based reasoning (CBR) [4, 7, 8]. A case-based reasoning system solves new problems by retrieving old problems likely to have similar solutions. Because the retrieval step is critical to the CBR model, researchers in this area have concentrated on developing knowledge-based methods for precise, efficient retrieval of well-represented examples. For some tasks, such as case-based educational systems, where cases serve a variety of purposes, CBR systems use a variety of retrieval strategies that measure similarity in different ways [1].

5 Conclusion

FindMe systems perform a needed function in a world of ever-expanding information resources. Each system is an expert on a particular kind of information, extracting information on demand as part of the user's exploration of a complex domain. In FindMe systems, users are an integral part of the knowledge discovery process, elaborating their information needs in the course of interacting with the system. One need only have general knowledge about the set of items and only an informal knowledge of one's needs; the system knows about the tradeoffs, category boundaries, and useful search strategies in the domain.

Robustness in the face of user uncertainty and ignorance is another important aspect of FindMe systems. Most people's understanding of real world domains such as cars and movies is vague and ill-defined. This makes constructing good queries difficult or impossible. We believe therefore that an information system should always provide the option of examining a "reasonable next piece," of information, given where the user is now. These next pieces are derived through the application of retrieval strategies.

Acknowledgments

The authors would like to thank Dan Kaplan and the Chicago Reader for their contributions to the RentMe project, Tom Weiner for his assistance with Video Navigator, and Sunil Mehrotra for assistance with the Kenwood project. The FindMe interfaces, except for Car Navigator were designed and created by Robin Hunicke of the University of Chicago. Numerous other students have contributed to the FindMe effort including

Terrence Asselin, Kai Martin, and Robb Thomas. Kass Schmitt was the original programmer on the CAR NAVIGATOR system.

References

[1] Burke, R., & Kass, A. 1995. Supporting Learning through Active Retrieval of Video Stories. *Journal of Expert Systems with Applications*, 9(5).

[2] Burke, R. (ed.) 1995. *Working Notes from the AAAI Fall Symposium on AI Applications in Knowledge Navigation and Retrieval*, AAAI Technical Report FS-95-03.

[3] Cutting, D. R.; Pederson, J. O.; Karger, D.; and Tukey, J. W. 1992. Scatter/Gather: A cluster-based approach to browsing large document collections. In *Proceedings of the 15th Annual International ACM/SIGIR Conference*, 318-329.

[4] Hammond, K. 1989. *Case-based Planning: Viewing Planning as a Memory Task*. Academic Press. Perspectives in AI Series, Boston, MA.

[5] Hearst, M. & Hirsch, H. *Working Notes from the AAAI Spring Symposium on Machine Learning in Information Access*, AAAI Technical Report SS-96-05.

[6] Knoblock, C. & Levy, A. (eds.) 1995. *Working Notes from the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, AAAI Technical Report SS-95-08.

[7] Kolodner, J. 1993. *Case-based reasoning*. San Mateo, CA: Morgan Kaufmann.

[8] Riesbeck, C., & Schank, R. C. 1989. *Inside Case-Based Reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

[9] Salton, G., & McGill, M. 1983. *Introduction to modern information retrieval*. New York: McGraw-Hill.

[10] Schank, R.C., & Riesbeck, C. 1981. *Inside Computer Understanding: Five Programs with Miniatures*. Hillsdale New Jersey: Lawrence Erlbaum Associates.

[11] Schneiderman, B. 1994. Dynamic Queries: for visual information seeking. *IEEE Software* 11(6): 70-77.

[12] Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Sys-*

tems Vol 1. Computer Science Press, 1988.

[13] Wiener, T. 1993.*The Book of Video Lists*. Kansas City: Andrews & McMeel.

[14] Williams, M. D., Tou, F. N., Fikes, R. E., Henderson, T., & Malone, T. 1982. RABBIT: Cognitive, Science in Interface Design. In *Fourth Annual Conference of the Cognitive Science Society*, pp. 82-85. Ann Arbor, MI: