CBR in the Pipeline

Marc Goodman

Continuum Software, Inc.
800 West Cummings Park, Suite 4950
Woburn, Mass. 01801
marc@continuumsi.com

Abstract

In a variety of reasoning tasks, even ones for which CBR seems ideally suited, a stand-alone CBR component may not prove adequate. First, the data available in system construction may be too raw or noisy for direct processing and may require sophisticated reasoning before it is in a form suitable for CBR. Second, capacity demands and other run-time constraints may prohibit a straight CBR module from being deployed. This paper describes a pipelined architecture where one or more reasoning steps are used to preprocess data into a form suitable for use in CBR, and CBR is used as a synthesis component for the creation of a stand-alone, run-time database.

Introduction

The SideClick link referral system [Goodman 1998] is a web-based service for resource exploration. Given a URL (most often a link to a particular web page of interest), SideClick can provide a list of related URLs organized by topic as well as a list of related topics. Or, given a topic of interest, SideClick can provide a list of URLs related to that topic as well as other related topics. For example, given a URL for "The Dilbert Zone" [Adams 1998], SideClick returns links for "Over the Hedge" [Fry and Lewis 1998], "Rose is Rose" [Brady 1998], "Peanuts" [Schulz 1998], the United Media comics page [United Media 1998], "Doonesbury" [Trudeau 1998], etc. and the related topics, "Entertainment" and "Comics and Humor." Clicking on the "Entertainment" topic returns links from baseball, movies, music, magazines, etc. and over 50 related topics from Art to UFOs. By following links and topics of interest, the user is free to discover new, interesting web resources in a serendipitous fashion.

SideClick models the way users of the web link together and organize information as embodied in bookmarks files and other on-line links pages. The core observation in the system is that people who create links pages tend to group links in sections, organized by content, with other similar links. Hence, a web page can be viewed as a case, composed of a number of snippets [Redmond 1992; Kolodner 1993] or microcases [Zito-Wolf and Alterman 1994]. Each snippet contains a group of links, a content header, a pointer to a parent snippet, and a set of pointers to child snippets. For example, a particular links page might contain a snippet consisting of links to peripheral manufacturers. Its header might be something like the text string "Peripherals". It might appear on the page as a subsection under a supersection called "Computer Hardware," and it might have child sections such as "Modems, " "Printers," etc. Each of the child sections and the parent section would also be represented by snippets.

The process of recommending links, conceptually, consists of taking a particular link, retrieving all of the snippets that contain this link, synthesizing the snippets into a representative snippet, and displaying this snippet to the user. The process of listing the links that occur under a particular topic consists of retrieving all of the snippets that were indexed under an appropriate section header, synthesizing the snippets into a representative snippet, and displaying this snippet to the user. Stated more intuitively, the system is saying something like "Given that the user is interested in a particular link, other web users who have been interested in this link have tended to organize it with these other links, under these topics. Therefore, the user should find these links and topics interesting as well."

Harder than it Sounds

Unfortunately, several factors conspire to make this simple conceptual framework for link recommendation insufficient. First, data on web pages is extremely noisy. This is hardly surprising given that most of these documents are generated by hand, and many of them are generated by people who have only passing familiarity with computers and computer programming. What is surprising is the sheer variety of types of noise in web pages. Some common types of noise include:

- **Markup Noise:** Web pages reflect their organizational structure primarily via the author's choice of markup, or the way the layout of the page is expressed in terms of markup *tags*. One author, for example, might choose to create sections using delimited lists of links, with

various levels of headers used to label sections and the relative size of those headers intended to convey scoping information. Another author might present the same information in tabular form, with the headers relegated to a column along the side of the page and the links contained in separate cells in a different column. A third author might display information within free-form descriptive paragraphs that contain embedded links, separated from other sections by horizontal rules. The number of distinct markup styles approaches the number of web authors. This source of noise is further compounded by the majority of authors who use markup tags incorrectly, invent their own markup tags (which browsers simply ignore), and even introduce syntax errors within the tags themselves. Reducing the amount of markup noise is crucial for placing links correctly within snippets as well as understanding the relationships between snippets within a case.

- **URL Noise:** It is unfortunate that URL stands for "Uniform Resource Locator," not "Unique Resource Locator." In fact, there are usually several distinct ways of referring to any particular web document. For example, the Netscape Home Page can be found at any of the following URLs: http://www.netscape.com/, http://home.mcom.com/, http://mcom.com/index.html, http://www.home.netscape.com/home/, and several others. Of the 4.5 million distinct URLs referred to by documents within the SideClick casebase, over 500,000 of these URLs are redundant. Successfully canonicalizing URLs prevents the system from referring the user to the same web resource via multiple URLs, as well as increasing the number and usefulness of snippets indexed under those URLs.
- **Section Heading Noise:** As described above, markup noise can make it difficult to identify the piece of text (if any) that identifies the topic of a snippet. However, even if that piece of text is successfully located, different people tend to label the same content differently. For example, the section headings, "Search," "Search Tools," "Suchmachinen," "Suchdienst," "Metasearch," "Keyword Search," "Search Forms," "Moteurs De Recherche," and "Search Engines" all refer to the same topic. Successfully canonicalizing section headings prevents the system from referring the user to multiple versions of the same topic with different names, as well as increasing the number and usefulness of snippets indexed under those section headings. A related but unsolved problem is ambiguity in section headings. For example, some people label links about stock quotations under "Quotations," while other people label links about quotes from famous people "Quotations." Or, some people might place stock chart links under "Charts," while other people might place music charts under "Charts." The result of this ambiguity is that the system currently contains some "interesting" mixed topics.
- **Taxonomic Noise:** Those of us who have experienced the joys of knowledge representation first-hand will not be surprised to learn that what look like section/subsection relationships between snippets often do not correspond to taxonomic or partonomic relationships. For example, one web page might place "Scotland" under "Food." Perhaps the author intends the section to be "Scottish Foods." Another author will place "Recipes" under "Scotland," meaning "Scottish Recipes." A third author will place "Recipes" under "Food," and a fourth author will place "Chicken" under "Recipes." Extracting a meaningful taxonomy of topics from the raw data is currently an unsolved problem.
- **Cobwebs:** It is a big exaggeration to say that half the web is "under construction," and the other half is missing, relocated, or hopelessly out of date. In actual fact, only 18% of the URLs cited in pages on the web refer to documents that no longer exist, serve only to redirect the user to new locations, or live on servers that aren't reachable or fail DNS (based on a sampling of over one million commonly cited web documents). The fewer such "cobwebs" that are contained within a service, the more useful that service becomes.

Another factor that makes creating a link referral service difficult is the sheer *size* of the web. According to Search Engine Watch [Search Engine Watch 1997], AltaVista [AltaVista 1998] had indexed over 100 million web pages in April of 1997, and their Chief Technical Officer, Louis Monier, estimated that there were as many as 150 million distinct pages on the web. Even a small subset of the web will contain millions of documents with tens of millions of snippets. Retrieving and synthesizing these snippets can be very computationally expensive.

Finally, a successful web service is, by definition, a high-volume web service. The most popular websites generate millions of page views per day. A scant million hits a day adds up to over 11 hits per second, and peak access times can easily reach two or three times as many hits per second as the average. At 33 hits per second, 30 msecs per query is about enough time to do three disk seeks. There isn't a lot of time for complicated run-time analysis.

CBR in the Pipeline

The solution we have developed to the above problems is to divide the system into a run-time component that does fast lookup on a pre-built database (or knowledge base), and a development component that builds the database. The development component is further broken down into several distinct processing steps, featuring one or more distinct form of reasoning/analysis at each step. These processing steps can be loosely grouped into 1). Fetching the data, 2). preprocessing the raw data, 3).using CBR to synthesize the run-time database, and 4). accessing the run-time database.

Fetching the Data

The system has been bootstrapped to the point where the analysis of a body of existing documents later in the pipeline has produced a list of canonical URLs to fetch. The actual mechanics of fetching the corresponding web pages are straightforward, and well documented elsewhere (see, for example, SideClick search results for HTTP and RFC [SideClick 1998]).

Preprocessing the Data

Preprocessing the data consists of several reasoning steps. These steps include, 1). learning a set of filtering rules for URL canonicalization, 2). parsing web pages into cases composed of snippets, and 3). canonicalizing section headers into SideClick topics.

Learning URL Filtering Rules. URL filtering rules are a set of regular expression patterns that map URLs into corresponding URLs that refer to the same document. For example, a filtering rule might specify that if a URL is of the form "http://*/index.html" and there is another URL that is of the form "http://*/" and the two URLs differ only in that one contains the "index.html" at the end and the other doesn't, then the two URLs probably refer to the same document. Another rule might specify that "www." in the host name of a URL can usually be stripped out if there is another known URL that differs only in that part of the host name.

Such rules are learned in a two-step process. First, an index of page similarity is created for all of the pair-wise combinations of documents in the set of web pages. Note that determining whether two documents are the same is, itself, a difficult problem. On the one hand, many documents are script generated and differ in the inclusion of banner ads, dates and times, number of page views, etc. even on subsequent fetches of the same document. Such documents will appear to differ, incorrectly, unless suitable fuzzy matching techniques are used with appropriate similarity thresholds. Similarly, pages change over time. Since the spider (the component that fetches the web pages) might take several days to fetch the millions of pages that comprise the set, it is quite possible that some pages will have changed between subsequent fetches. Hence, determining whether two pages are distinct often requires modification based on the time those pages were fetched. On the other hand, many documents from the same site are identical with respect to navigation content, layout, headers, and footers and differ only a small amount on the actual content of the web page. Such pages will appear to be similar if matching thresholds are set too low.

After the index of similarity is generated, a heuristic pattern learning algorithm is applied to generate the filtering rules. For a particular pair of similar pages, the algorithm creates a set of regular expressions of varying generality that describe how one URL can be mapped to another. These candidate rules are scored by applying them to the entire body of URLs, and counts are kept of the number of times a URL is incorrectly mapped into a differing URL, the number of times a URL is correctly mapped into a differing URL, and the number of times a URL is mapped into a URL that appears to differ, but might be the result of a document changing over time. These values are combined heuristically, and the most successful candidate rule is chosen (success is based on the most general rule that doesn't introduce too many false mappings). The process repeats until all of the URL matches have been accounted for.

Parsing Web Pages into Cases and Snippets. Some organizational and scoping information for a web page is explicit in the (possibly broken) markup for that web page. For example, a delimited list within a delimited list represents that one snippet is a child of another snippet, and the scope of each snippet is defined by the scope of the delimited list. Other organizational information is implicit in the markup. For example, a sequence of markup tags and strings of the form: "string <a> string
 <a> string
 <a> string <p> string <a> string
 <a> <string> <p>," implicitly defines two groups of anchors, and could be represented by the fuzzy regular expression:

(string (<a> string
)* <p>)*

where the first string in each occurrence of the regular expression probably denotes the section heading (the expression is fuzzy because it allows the last "<a> string " of each subsequence to match the subexpression "<a> string
").

Parsing a web page, therefore, consists of two steps. First, a fault-tolerant HTML grammar is used to organize the tags and strings in the web page into a set of scoped subexpressions. Next, for each sequence of tokens and strings within a subexpression, a pattern detector reduces the sequence of tokens into a set of scoped subsequences based on increasingly complex regular expressions. The result of this analysis is a set of fully scoped tokens. "Interesting" scopes are detected and output as scoped snippets, and likely section headers for each snippet are identified and output.

Canonicalizing Section Headers. As previously mentioned, the raw organizational information present in web pages is not sufficient to generate an accurate taxonomy of topics. As such, we have knowledge engineered a taxonomy of over 3000 topics, by hand, with much suffering and loss of life. The maintenance and extension of this taxonomy is an ongoing process and consumes much of the bulk of human labor in the system.

Mapping section headers extracted during the previous processing stage consists of applying a large number of phrase canonicalization rules (which were constructed and are maintained by hand) to the section header, and performing a statistical analysis of how well the resulting section header matches each of the known topics. This analysis is based on morphological analysis of the words in the section header and topic, the number of matching words in the section header, the frequency of occurrence of these matching words in the set of documents as a whole,

and the total length of the section header. Section headers that match topics above a certain threshold are canonicalized into the corresponding SideClick topics. The remaining section headers are rejected, and a knowledge engineer periodically reviews frequently occurring rejected headers for possible inclusion as new topics within SideClick.

The result of these preprocessing steps is a set of relatively clean and well-organized snippets and cases, which are fed into the CBR component.

Synthesizing the Database

Primary functions supported by the run-time system include:

- **Links Related to Links:** Given a URL, retrieve all of the snippets containing that URL. Synthesize these snippets into a new snippet, as follows: 1). count the number of snippets each URL appears in, 2). compare this count to the base probability that the URL will appear in a random collection of snippets, 3). if the URL occurs significantly more frequently than random chance, include the URL in the synthesized snippet.
- **Topics Related to Links:** Given a URL, retrieve all of the snippets containing that URL. Synthesize these snippets into a new snippet, as follows: 1). count the number of snippets each under each topic, 2). compare this count to the base probability that a randomly selected snippet will appear under each topic, 3). if the topic occurs significantly more frequently than random chance, include the topic in the synthesized snippet.
- **Links Related to Topics:** Given a topic, retrieve all of the snippets under that topic. Synthesize these snippets into a new snippet, as follows: 1). count the number of snippets each URL appears in, 2). compare this count to the base probability that the URL will appear in a random collection of snippets, 3). if the URL occurs significantly more frequently than random chance, include the URL in the synthesized snippet.
- **Topics Related to Topics:** Consult the knowledge-engineered taxonomy for related topics.

Constructing a run-time database consists of iterating through all of the known URLs and topics, and generating lists of the most closely related URLs and topics along with the strength of the relationship, as described above, and saving these results into a database.

There is no theoretical reason why these functions couldn't be supported by a run-time CBR module. However, there are three practical reasons for using the CBR module to build an optimized run-time database and to respond to most queries using database lookup. The first reason is, of course, speed. Popular URLs, such as Yahoo [Yahoo 1998], occur in tens of thousands of snippets within the case base. Each snippet may, in turn, contain references to tens or hundreds of links. Synthesizing all of these snippets can take orders of magnitude longer than the maximum time allowed for responding to a query.

The second reason for having a run-time system distinct from the CBR module is code complexity. The CBR module requires code for loading cases, organizing case memory, retrieving snippets and synthesizing these snippets. Also, the internal data structures used to represent and index case memory are somewhat elaborate. It is a simple fact that a live system on the world-wide web is **not** allowed to crash (sometimes they do anyway, which is one of the reasons why large web services run two or three times as many servers in their server farms as they really need to handle capacity). The CBR module weighs in with six times as many lines of code as the run-time system. It is safe to assume that the run-time system is easier to modify and maintain.

Finally, the run-time database is actually *smaller* than the original case base. Instead of keeping around information about every link that appears in every snippet in every case that occurs in the case base, the run-time system only needs to know the relative strength of the relationship between a particular URL and its most closely related topics and URLs. In fact, the run-time database is small enough to fit within a gigabyte of RAM, and dual 200Mhz Pentium Pro servers with one gigabyte of RAM can be purchased for around $6000 (as of April, 1998). Avoiding any disk lookup whatsoever drastically increases the speed of the run-time system.

Using the Database

As described above, the run-time system consists of a large, precomputed database and a simple lookup mechanism. This run-time system is implemented as a TCP-based server that responds to requests from a set of front-ends. Each front-end is a web server that is responsible for processing web page requests, querying the back-end run-time system for link and topic referral information, and generating suitable HTML web pages. The back-end is capable of handling over 30 requests per second, and most of this time is spent in TCP socket setup and teardown. Perhaps surprisingly, it takes longer to query the back-end and format the web page under Microsoft's IIS web server, with C-language DLLs and Visual Basic Script web page generation under Windows NT than it does to process the back-end queries. Each front-end is only capable of processing around 11 requests per second.

What does this Say about CBR Integration?

The first observation is that while CBR seems to be an ideal technology for solving this problem, significant reasoning work is needed before the available data is in anything like a suitable format for processing. The system described here includes fuzzy page matching, a novel technique for inducing pattern matching rules, a fault tolerant grammar, pattern detection, some simple Natural Language pattern matching, statistical matching of patterns and phrases, and a hand-engineered taxonomy of over 3000 topics before the CBR can even begin. This is on top of

more "conventional" programming tasks such as creating a spider for fetching documents from the world-wide web, creating software for the efficient storage and retrieval of millions of web pages, etc.

The second observation is that even though a CBR module as "master" in a run-time system may be functionally adequate, it may be undesirable on practical grounds due to high-capacity requirements, code complexity and maintenance issues, and case base size.

For these reasons, we have ended up with a pipelined architecture of processing steps from raw data through a standalone database with CBR planted squarely in the middle.

Is this General?

While clearly an inappropriate architecture for some reasoning tasks (for example, the *Battle Planner* system where the ability to retrieve and examine cases forms an integral part of the decision support process [Goodman 1989]), this methodology has been applied to two other systems, *Fido the Shopping Doggie* [Goodman 1997], and *FutureDB*.

Fido is a web-based shopping service. As in SideClick, web pages are downloaded and preprocessed. In Fido, however, CBR is used to label parts of these web pages as product descriptions, product categories, vendors, prices, etc., based on a case library of pre-labeled web pages. These newly downloaded and labeled web pages are fed into a push-down automata that use the labels to construct a database of products and prices. The run-time system allows web users to perform keyword searches on this database to locate products of interest along with links back to the web page from which the product was extracted. As in SideClick, a variety of processing steps are needed to convert raw web pages into cases, and CBR is used as a component in a pipeline to synthesize an efficient run-time database.

In *FutureDB*, a product based on Projective Visualization [Goodman 1995], raw historical data is preprocessed and fused with external data sources, and CBR is used as a key component in constructing a simulator. This simulator is used to project historical data into the future, and the projected data is stored into a database in the same format as the historical database. This allows users to analyze the projected data using the same decision support systems and on-line analytical processing tools that they currently use to examine historical data. Once again, a variety of reasoning techniques are used to preprocess raw data into a form suitable for CBR, and CBR is used in a pipeline to produce a static run-time database.

Hence, while not universal, the architecture described here does support a variety of reasoning systems.

References

Adams, S. 1998. *Welcome to the Dilbert Zone.* <http://www.unitedmedia.com/comics/dilbert>.

AltaVista. 1997. *AltaVista: Main Page.* <http://www.altavista.digital.com>.

Brady, P. 1998. *The Official Rose is Rose Site.* <http://www.unitedmedia.com/comics/roseisrose>.

Fry, M. and Lewis, T. 1998.*The Official Over the Hedge Site.* <http://www.unitedmedia.com/comics/hedge>.

Goodman, M. 1989. CBR in Battle Planning. In Proceedings of the Second DARPA Workshop on Case-Based Reasoning, 312-326.

Goodman, M. 1995. *Projective Visualization: Learning to Simulate from Experience.* Ph.D. Thesis, Brandeis University, Waltham Mass.

Goodman, M. 1997. *Fido the Shopping Doggie.* <http://www.shopfido.com/>.

Goodman, M. 1998. *SideClick.* <http://www.links2go.com/>.

Kolodner, J. 1993. *Case-Based Reasoning*, Morgan Kaufmann Publishers, San Mateo, CA.

Redmond, M. 1992. *Learning by Observing and Explaining Expert Problem Solving.*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA.

Search Engine Watch. 1997. *How Big Are The Search Engines?* <http://searchenginewatch.com/size.htm>.

Shulz, C. 1998. *The Official Peanuts Home Page.* <http://www.unitedmedia.com/comics/peanuts>.

SideClick. 1998. *SideClick: +HTTP +RFC.* <http://links2go.com/search?search=%2BHTTP+%2BRFC>

Trudeau, G. 1998. *Doonesbury Electronic Town Hall.* <http://www.doonesbury.com/>.

United Media. 1998. *Welcome to the Comic Zone.* <http://www.unitedmedia.com/>.

Yahoo. 1998. *Yahoo.* <http://www.yahoo.com>.

Zito-Wolf, R. and Alterman, R. 1993. A FrameWork and an Analysis of Current Proposals for the Case-Based Organization and Representation of Procedural Knowledge. In Proceedings of the Eleventh National Conference on Artificial Intelligence, 73-78.