Acquisition and Maintenance of Text-based Plans

Qiang Yang and **Kirsti Racine*** and **Zhong Zhang†**
School of Computing Science
Simon Fraser University
Burnaby, BC V5A 1S6
http://www.cs.sfu.ca/cbr, (qyang@cs.sfu.ca)

Abstract

Text-based plans are plans whose steps and goals are described in a textual format. In contrast to logic-based plans – plans that are represented based on variations of a first-order logic format – such plans are much easier for a domain expert to describe, and is getting much easier to find given the tremendous growth of the Internet. In this paper we assume that the user is interested in an interactive plan retrieval system that is based on a library of text-based plans. We focus on two important issues on the maintenance of this library: the updating of the text-based plans themselves and the maintenance of the indexes with which to retrieve these plans. We present techniques that address the life-time maintenance of such plan libraries with techniques drawing from information retrieval, neural network learning and case based reasoning.

Introduction

Text-based plans are plans whose steps and goals are described in a textual format. In today's business and scientific environment, text-based plans are abound; with the explosive interest in the Internet, text-based plans can be easily found ranging from equipment troubleshooting and software installation procedures to cook-book recipes. Text-based plans are also found to be more natural than logic-based plans for non-computer experts to articulate. In many industrial applications, domain experts find it easy to describe the steps in which to accomplish a certain goal and objective. Such step-by-step instructions on how to accomplishing something, in audio form or written form, can be converted easily to a textual format.

In this paper, we focus on how to acquire and maintain a library of text-based plans for achieving user-specified goals. Planning using the text-based plan library is inherently interactive in nature, with the user incrementally specifying the plans to be retrieved and reused. We will point out that the text-based planning process builds on existing technologies of case based reasoning or planning (BHK95; Ham90; Kol93c; Lea96; Wat97).

Text-based planning is inherently interactive in nature. In a nutshell, a user of the system starts out by entering a goal description in natural language text. The planning system then extracts out the important key words for matching with the plans in the library. The system will then enter a dialog with the user by querying about index values. These values help identify a context for the correct plan to be selected. The system ranks the candidate plans based on a similarity based retrieval algorithm. Once a plan is found, the system can recursively answer the user's queries on the details of any steps, taking these steps as subgoals much like those in STRIPS or task network planning. The subgoal matching is performed using search engines and text-based matching algorithms.

In this paper, we will concentrate on the maintenance issues regarding how to maintain a library of text-based plans so that the plans are current and up to date. We contend that maintenance for the text-based plan library is also interactive in nature. The objective is to lessen the burden on the human user such that the resulting maintenance activities can be carried out throughout the lifetime of a plan library.

We distinguish between three different types of maintenance activities in text-based planning. First, plans may arrive from different data sources. These plans must be normalized before they are used by the users. Second, a comprehensive maintenance facility are needed to decide when new plans added to a plan library is redundant. Such a facility must have the ability to highlight the differences between two or more text-based plans and minimize redundancy. Third, a learning mechanism for updating indexes for retrieving text-based plans based on user input must be designed. Such a mechanism should evolve with the ever-changing needs and preferences of the user with time.

Text-based Plans

Text-based plans can be in many different formats. For brevity, we assume that a text-based plan is described as a pair: ⟨ Goal, Solution ⟩. The first element of the pair, the goal, is described in natural language sentences. The second element is a solution in a step-by-step format, where every step is also described in a natural language sentence or a collection of key words. An example is shown in Table 1, representing a text-

Question 1	Is the subscriber in pay status?
Question 2	What type of problem is being experienced?
Goal	**Solution**
Solve "no reception on low band"	1. Check no splitter on cable, fine tune TV channels.
	2. If problem continues, unplug TV for 30 seconds, replug.
	3. If problem continues, generate trouble ticket.

Table 1: A text-based plan in a Cable-TV domain

based plan in a Cable-TV troubleshooting domain.

In contrast, a logic based plan must describe more details in the representation of actions and goals, and such descriptions must be consistent and clean. In a STRIPS representation, for example, effects and preconditions of actions must be explicitly given, and goals must be explicitly encoded as literals. Such strict requirements place a high burden on users who are not well versed in logic or programming languages. It can also be argued that a logic based representation results in knowledge bases that are hard to maintain and update.

With a library of text-based plans, text-based planning is aimed at achieving the same objectives as logic-based plans, namely goal attainment. The difference here is that instead of conducting plan generation with algorithms based on state-space search, text-based planning relies on the retrieval from an available library or libraries of plans. Upon receiving a goal to be achieved, the main activities are to:

- retrieve candidate plans from text-based plan libraries or by invoking search engines;
- compare retrieved plans to highlight differences and relevance to the problem at hand;
- if necessary, splice or merge candidate plans to arrive at useful plans for solving the problem at hand;
- perform goal and subgoal analysis to allow planning queries to be answered hierarchically;
- record action logs and plan execution transcripts to maintain feedback from the user, and to mine such logs to obtain useful information;
- maintain and update the plan library and its indexing mechanism in the lifetime of the plan libraries, using the action logs and other data sources.

In this paper, we will focus on this last issue regarding the maintenance of a library of text-based plans.

Text-based Planning and Case Based Reasoning

One way to accomplish text-based planning is to retrieve plans that closely match the input goal descriptions by a certain similiarity measurement. The retrieval process resembles a case-based reasoning framework of retrieve-reuse-revise cycle. In this section, we briefly review some elements of this process.

In case based reasoning (Lea96; Kol93c; Wat97), the process of case based reasoning is described as a cycle. The most effective mnemonic used to described this process is the four **Res** (AP93):

1. **Retrieve** : Given the user's query, retrieve the most similar case(s) in the case base.
2. **Reuse** : Reuse the appropriate case to try to solve the problem.
3. **Revise** : Revise the current solution, if it is inadequate to solve the current problem.
4. **Retain** : Save the revised case as a new case in the case base, assuming the new case does not cause performance decrease in the future.

Simplified, the process works as follows. The user formulates a query for the system; many systems use free form text queries, but the query structure is system dependent. The system uses the query to **retrieve** the appropriate case(s) that exist in the case base. Often, the system returns a list of cases that are given relative scores according to their similarity to the query. Either the system **reuses** the case with the highest score or the user is given the chance to select a case for **reuse** from a list of similar cases. If the current case will not appropriately solve the current situation, either the user or the system **revises** the case to fit the current problem. This newly generated case is **retained** in the case base, so that it will be accessible to the next user of the system.

Once the cases are obtained, they have to be properly indexed to allow for efficient retrieval. To this end, feature indexing is an important task in authoring and maintaining the case base. Feature indexing involves determining which features of a case will be used to facilitate its retrieval. The features associated with a case are combinations of its important descriptors, which distinguish one case from the others.

After the features for all the cases in a case base have been decided to be used as indexes, the next task in a CBR system design is to decide on the weighting values assigned to feature-value-case triples.

It should be noted that features and their values are sometimes presented to the user in the form of questions and answers. This is the case in the Cable-TV case base in Table 1 which we designed in the Cable-TV troubleshooting domain. In this table, the system takes an interactive mode of operation, taking answers

directly from the customer service representatives in the Cable company, or from a database management system. Based on the answers to the questions and the weights attached to the question-answer-case triple, a ranking score will be produced for the user to assess the similarity of the case to the current situation.

After the similarity to the input query is computed for each case, a set of highly relevant candidate plans in a library are scored. Cases with high similarity scores are presented to the user.

Maintenance Issues

In this section we consider three important issues on the maintenance of a text-based plans. These issues are how to normalize text-based plans, how to detect redundant plans, and how to update the indexes used for retrieving plans. In addressing this last issue, we will also briefly describe how to retrieve a text-based plan using similarity based retrieval similar to that done in a case based reasoning system.

The Normalization Problem

Formulating a plan into a structured, standard format may require extensive knowledge engineering. To facilitate plan retrieval, each plan in a plan library must be indexed by a set of common indexes that are predetermined. Such an activity is called normalization. For a given domain, the user has to first determine the important attributes to use to represent each plan. Then a decision has to be made regarding the range of legal values each attribute may have. The process of authoring knowledge in this attribute value format requires extensive maintenance when a new attribute is discovered and inserted, or when an existing attribute becomes irrelevant. In addition, unstructured documents which are used as the basis of text-based plans rarely break down into obvious attribute value pairs. By reducing each plan to this structure, the meaning or the purpose of the plan can be lost in the translation. From this observation, we conclude that we must supply a plan-library maintenance system with more than the standard relational structures used in case-based reasoning.

In industrial practice, a majority of the plan library come directly from either unstructured text documents or end-users' verbal description. These plans may have generic attributes such as *problem description* and *Solution*, but each of these attributes probably will not be further partitioned down to a relational level. For instance, consider the following example of a free form plan used in a Cable-TV repair domain. This plan actually exists in a plan library used to diagnose Cable-TV related failures in a Canadian Cable-TV company:

Problem Description: There is no cable picture, only black screen. This may be a problem with your TV or the cable system.
Solution:

1. Check all electric connections to see if they are secure.
2. Once you checked the electrical connections, tune the TV channel to 333 and disconnect the cable.
3. If there's still no reception, the problem is most likely in the TV set. Call the TV vendor for a check up.

Text-based planning can be effectively applied to poorly understood domains provided that some type of legacy data source exists. Many of these poorly understood legacy data are in free form text format. We call plan libraries of this type as consisting of *unstructured* plans. To normalize the plans in a plan library one can apply the information retrieval technology for extracting key words and phrases. These key words and phrases can then form the basis for comparing and analyze the plans.

The Redundant Plan Problem

In a large plan library, redundancy identification requires the ability to detect two equal plans, if one plan subsumes another, or if two plans can be merged. At the rapid rate that industry is changing, it is possible that two previously distinct plan libraries will need to be merged. When this happens, it is critical to develop a mechanism that can collapse the redundant plans into a representative plan for the class of problems that the plans can solve. This mechanism must have the ability to explain why the plans were identified as redundant, so that the user can make an informed decision to resolve the problem. An example of redundancy in the Cable-TV repairing domain is displayed in Table 2. It demonstrates the difficulty of identifying redundant plans when the plans are unstructured. As we will demonstrate in the next section, our approach relies on an information retrieval method for detecting redundant plan information.

In case based reasoning, redundancy testing involves submitting the new plan as a query to the reasoner. If a high-ranking solution is returned, the plan is not entered. However, in practice, plans are often entered by a module separate from the problem resolution module. Plan authors enter a set of plans at the same time and then test the system. The iterative type of testing described above may not be feasible for a large plan library. A further problem is that redundancy is not always obvious. Plan authors may not be domain experts, and thus, not familiar with the domain jargon. In addition, the range of problems that a planner can solve may be wide. Manual, iterative testing for redundancy may be very time consuming. Finally, companies may already have their data available in a different format, where redundancy may not be obvious. Some companies already have their data collated in decision trees, where there may be great overlap. Redundancy within free form text is not always obvious. There are differences in vocabulary, depth of detail, and even punctua-

Plan 1
Planning Problem: There is no cable picture, only black screen. This may be a problem with your TV or the cable system.
Solution:
Step 1. Check all electric connections to see if they are secure.
Step 2. Once you checked the electrical connections, tune the TV channel to 3.
Step 3. connect and disconnect the cable.
Step 4. Call the TV vendor for a check up.

Plan 2
Planning Problem: No cable picture and screen is black. Faulty TV or faulty the cable system.

Solution:
Step 1: check electric connections to see if they are secure.
Step 2: tune the TV channel to 3.
Step 3: disconnect and then reconnect the cable;
Step 4: If there's still no reception, the problem is most likely in the TV set. Call the TV vendor for a check up if problem is with TV set.

Table 2: Redundant plans

tion. Therefore, a mechanism to detect redundancy and offer an explanation for that identification is critical.

The iterative nature of plan library construction means that the size of the plan library increases monotonically if no plans are removed. This can cause serious problems for storage spaces and retrieval efficiency. Therefore, a plan library management system should have the ability to detect whether two plans can be merged. Two plans may be candidates for merging if they are similar enough to share common attribute values, but also have a small number of critical differences. Merging is suggested by the application if two plans share some percentage of common key words and a common field, yet have at least one significant difference. Obviously, if the two plans do not share one significant difference, the application will suggest that the two plans are equivalent and one should be deleted.

The Plan Indexing Problem

In this subproblem, the issue we address is how to maintain indexes (sometimes called feature/values) and their weights in a plan library in a multi-user and changing environment. Furthermore, the environment is complex in the sense that the same solution may serve to solve different problems under different contexts, and the same problem may be provided with different, alternative solutions.

We can be more specific about the maintenance problem. We assume that our desired plan library maintenance system is given a set of features where each feature has a set of potential values. Some subset of the features and values may be relevant to a particular problem or solution at hand at any given time, but there is no prior knowledge on which ones are actually useful to the planner currently. For any given set of weights attached to the feature-value-plan combinations, the users of the system can provide feedback on the outcome of the solutions provided through a feedback process.

Our tasks are:

1. to acquire the feature-weights after a user has used the system for a certain period of time;
2. to adapt the feature-weights to a user's preferences with time, and to allow different users to have different weights;
3. to continuously track a user's changing preferences for the plans in the plan library and to update the weights correspondingly to reflect the change;
4. to allow the influence a feature-value on the selection of a plan to be dependent on the values of other features in the plan library; In other words, the feature weights are context dependent.

The above tasks are directly motivated by our fielded application in a Cable-TV troubleshooting application, in which we have over-seen the entire process of plan library creation, the application of a CBR-style system for real-time plan retrieval, and the critical problem of plan library maintenance. In this domain, the plan library creators are chosen customer service representatives from the Cable-TV company. To assign feature weights to the plan library, the users have to manually change the weights through a plan editor. The maintenance process is very long and tedious, such that it can potentially prohibit the end user from adopting the technology entirely. To make the problem more complex, the weights assigned to the initial plan library is changing with time. For example, with improvement in technology, the VCR-recording problem features may become secondary in importance. This feature weight should decrease correspondingly. Similarly, a feature's weight may be different depending on the difference in geographical regions; a remote area may experience more in one type of problem than a urban area.

Solving the Three Problems

Solving the Normalization Problem

We apply information retrieval to the problem of normalizing text-based plans. The steps in the information retrieval algorithm used are as follows:

1. Remove the stop words.
2. Collapse words using a domain thesaurus.
3. Remove the suffixes and prefixes from each string.
4. Build an inverted index.
5. Build a key word index.
6. Build a key phrase index.

The output of this algorithm is an internal, normalized array of plans. If the redundancy and inconsistency detection modules have not been activated by the user, the application builds a plan library from this array. As well, three flat files representing the inverted index, the key word index, and the key phrase index are generated. These files are represented in binary form to reduce the storage space required.

Removing the Stop Words The first step in the information retrieval algorithm is to remove the stop words. Stop words are those words proven to be poor indexers, such as "the" and "of". These words do not add any meaning to the plan. Stop words typically comprise between 40% - 60% of the words within a document (SM83). The application uses a general list of stop words generated for the English language used by the SMART system designed by Salton (SM83). This list of stop words can be edited by the user in order to specialize it for a particular domain.

Domain Thesaurus This function collapses words using a domain thesaurus. In this application, the thesaurus is used to standardize terms. The thesaurus can be edited iteratively as users become more familiar with the domain specific language. The user may choose not to use a thesaurus at all.

The Stemming Algorithm The stemming algorithm removes the suffixes and prefixes from each word in the plan library. Stemming is used to reduce the number of distinct terms and to improve retrieval. There are a number of available stemming algorithms varying from removing almost all possible prefixes and suffixes, to removing only those suffixes that pluralize a word.

The advantage of using a stemming algorithm is to further reduce the number of distinct words for consideration. A stemming algorithm will reduce the words "hook", "hooked" and "hooking" to the word "hook". This should increase the number of key words and phrases identified by the algorithm.

The Inverted Index After the preprocessing steps have been completed, the application generates an inverted index for the entire plan library. The index is simply a listing of all terms that still remain in the set of plans, their weight within each document and the document number in which they appear. The weight of a term within a document is simply a measure of the frequency that the term appears within that plan. This measure provides information regarding the statistical importance of a term. Inverted indices may also contain information reflecting the position of the term within the plan.

The Key Word Index After the inverted index is created, the next step in the algorithm is to build the key word index. Using the inverted index, this function identifies significant terms through statistical measures. Key words are those words which appear frequently within a small set of plans and infrequently across all other plans (FBY92; SM83). This application uses the inverse document frequency measure to identify key words (SM83).

The Key Phrase Index The application also identifies key phrases using the inverted index. Phrases are groups of more than one word which have high inter-plan cohesion (SM83); if one word appears in a plan, then the other words have a very high probability of also appearing. The phrases and their corresponding weight are retained. Identified phrases must appear in $> T$ plans, where T is a standard, or user specified threshold. Phrases can be more powerful than key words as they add some context to the statistical approach to information retrieval. To reduce the number of phrases identified by the algorithm and to increase their relative importance, there is an additional constraint that at least one word in the phrase must be a key word.

Normalization After the foregoing steps, a text-based plan is now converted to a set of three distinct fields. The solution field S consists of a set of steps where each step is represented by a set of important key phrases. The qualification field Q consists of a set of feature-value pairs which results from the assigned indexes for this plan, and for any additional requirements that are extracted by the information retrieval process to result in the form of key words and phrases. Finally, the goal or problem description field P consists of key words and phrases that represent the goals to be achieved by this plan. In all, a plan can be represented as a triple $\langle(P), (Q), (S)\rangle$.

Detecting Redundant Text-based Plans

Plans can be redundant because they are subsumed by other plans. In this situition, the subsumed plans can be removed from the plan base without affecting the overall competence of the plan retrieval system.

We adopt a uniform notation for representing a plan. Let $\texttt{Plan} = \langle (P), (Q), (S) \rangle$ be a plan. In the definition, P represents a set of normalized key words or key phrases denoting a goal to be achieved for the plan. Each element p_i of P represents a distinct problem feature such as "screen black" in a Cable-TV troubleshooting plan. Q represents a set of key word or attributes used for qualifying the solution S in the plan. In the cable-TV plan, Q might be a key word "Sony" denoting the brand of TV experiencing the problem. In the same example, S, the solution for the plan, might be "Call Sony at 1-800-...". In general, a solution S for a plan is a sequence of steps such that upon complete execution of S, the problems described in P can be solved. A precondition for the success in the execution of S is that every condition stated in Q must be satisfied.

This plan notation has an equivalent logical representation. Let $\texttt{Plan} = \langle (P), (Q), (S) \rangle$ be a plan, where P is a set of problem description key words (or phrases), Q is the set of key words qualifying a solution, and S are steps in a solution. Its equivalent logical representation is

$$\texttt{Plan} : Q \wedge \text{do}(S) \Rightarrow \text{solved}(P)$$

That is, if Q is satisfied and the steps in S are followed, P will be successfully solved.

We can now state our plan-subsumption rule informally as follows: given two plans $\texttt{Plan}_1 = \langle (P_1), (Q_1), (S_1) \rangle$ and $\texttt{Plan}_2 = \langle (P_2), (Q_2), (S_2) \rangle$, $\texttt{Plan}_1$ subsumes $\texttt{Plan}_2$ if $\texttt{Plan}_1$ solves more problems than $\texttt{Plan}_2$ (P_1 is a superset of P_2), the solution of $\texttt{Plan}_1$ requires fewer qualifications than $\texttt{Plan}_2$ (Q_1 is a subset of Q_2), and the solution of $\texttt{Plan}_1$ is requires fewer steps than that of $\texttt{Plan}_2$. In this situation S_1 is a subset of S_2[1].

More formally, $\texttt{Plan}_1$ *subsumes* $\texttt{Plan}_2$ if

1. $P_1 \supseteq P_2$,
2. $Q_1 \subseteq Q_2$, and
3. $S_1 \subseteq S_2$.

If $\texttt{Plan}_1$ subsumes $\texttt{Plan}_2$ then removing $\texttt{Plan}_2$ will not affect the coverage of the plan base. In this case, we say that $\texttt{Plan}_2$ is redundant.

In the formalization above we have divided the keywords in a plan into two classes, those that govern the qualification of a solution, and those that describe the problem statements. In practice, the subsumption rules can be simplified when only one type of keyword is present, giving rise to specializations of the subsumption rules.

Consider a situation where two plans in a plan base consist of identical solutions and problems. The only difference between the two plans are their keywords and phrases used to qualify the solutions. In this plan, we describe the plans as $\texttt{Plan}_1 = \langle Q_1, S_1 \rangle$ and $\texttt{Plan}_2 = \langle Q_2, S_2 \rangle$ where Q_1 and Q_2 are qualifying key words and phrases for the solution $S_1 = S_2$.

Specialization Rule 1:

$\texttt{Plan}_1$ subsumes $\texttt{Plan}_2$ if $Q_1 \subseteq Q_2$ and $S_1 = S_2$.

It is a direct corollary that removing $\texttt{Plan}_2$ when Rule 1 is satisfied will not affect the coverage of the plan base.

Similarly, consider a situation where the key word and phrase portion of the two plans are identical. In that plan there is a strong indication that the two plans are solving the same problems and that the qualifications for the solutions are identical too. This situation calls for *Specialization Rule 2:*

Specialization Rule 2: Let $\texttt{Plan}_1$ be $\langle K_1, S_1 \rangle$ and $\texttt{Plan}_2$ be $\langle K_2, S_2 \rangle$ where K_1 and K_2 are sets of key words and phrases.

$\texttt{Plan}_1$ subsumes $\texttt{Plan}_2$ if $K_1 = K_2$ and $S_1 \subseteq S_2$.

Similar to Rule 1, Rule 2 can be applied to raise alarms for redundant plans. In fact, Rule 2 can be explained informally as follows. For an incoming problem both $\texttt{Plan}_1$ and $\texttt{Plan}_2$ will be returned by the text-based plan retrieval system because their problem descriptions are identical. However, $\texttt{Plan}_2$ offers more solution steps than $\texttt{Plan}_1$ does. These extra steps are not necessary, since having $\texttt{Plan}_1$ in the plan base simply certifies that the problem can be solved successfully using just S_1. Thus, the extra solution steps offered by $\texttt{Plan}_2$ are redundant.

When deciding to delete subsumed plans, the plan-library maintenance system should allow the user to view both plans and highlights the unnecessary conditions. As it is possible that $\texttt{Plan}_1$ is an incorrect or outdated plan, the fact that it subsumes $\texttt{Plan}_2$ does not mean that $\texttt{Plan}_2$ should be summarily deleted from the plan base. Rather than simply deleting the plans identified as subsumed, the application presents these plans to the user together with reasons why they are believed subsumed. This is because a typical user of the application may not be familiar enough with the domain to delete the plan that offers more information. Perhaps the extra premise offers valuable information to the novice user that the plan that subsumes it does not.

A redundancy identification module should also be able to detect plans that are candidates for merging. For example, if two plans offer the same solution but slightly different problem descriptions, it is likely that the plans can be collapsed into one. Please note that if the differences within the problem description field are not considered significant by the application, then the system suggestion will state that the plans are essentially equivalent and the user may choose to keep either or both plans.

Relating to the work by Smyth and Keane (SK95) on competence-preserving methods for managing a case

[1] We have assumed that solution length is a measure of solution quality in this section. In some domains there are other measures of solution quality such as the cost of solution etc. Also it may be the plan that a solution is longer because it contains more explanatory data. Extensions of our subsumption rules in these areas can be done; however we will only focus on solution length here for simplicity.

base in case based reasoning, the subsumption rules defined above provide a significant operational advantage. In that work, all definitions of auxiliary, spanning and support cases are defined in terms of problem coverage and reachability. These definitions have the problem of computational inefficiency, since to compute the coverage and reachability of a case in terms of incoming problem queries is very expensive. Our subsumption rules, on the other hand, provides a problem-independent approach to finding redundant cases; instead of computing coverage and reachability for each case using all user problems, we compare the contents of the two cases directly. This direct comparison enables us to deduce whether one case makes another case redundant.

Solving the Plan Indexing Problem

We now model the plan library as a three-layer structure (see Figure 1). In the plan layer, we extract the solutions from each plan, and put them onto a third layer. This makes it possible for different goals to share a solution, and for a goal to have access to alternative solutions. An important motivation for this separation of a structure of a plan is to reduce the redundancy in the plan library. Given N problems and M solutions, a plan library of size $N \times M$ is now reduced to one with size $N + M$. This approach eases the scale-up problem and helps make the plan library maintenance problem easier, since when the need arises, each problem and solution need be revised only once.

In order to make this change possible, we introduce a second set of weights, which will be attached to the connections between goals and their possible solutions. This second set of weights represents how important a solution is to a particular plan if this solution is a potential candidate for this plan. In addition, it distinguishes a solution within several plans if the solution belongs to several plans at the same time. Initially, there is a weighted connection from every feature-value pair to every problem in the problem layer, and from every problem to every solution in the third layer. In the end, if the weight at a particular connection is zero, it is equivalent to disconnecting the two nodes at the two different layers.

With this architecture, a user select plans by entering values for the indexes and by specifying a verbal description of a goal. The plan retrieval system responds by providing a set of candidate goals and solutions to the user, and ranked by their relevance which are computed from the current weights. The system then prompts the user for feedback regarding the accuracy of the returned solutions and plans. From this feedback, the weight-learning system propagates the weights back through the network using a back-propagation algorithm. The weight learning continues throughout the lifetime of the plan library.

We introduce notations for different entities in Figure 1. There are two sets of weights, similar to the weights in a three-layer back-propagation neural network. Suppose that there are N features. For each feature F_i, there are m_i values, where $i = 1, 2, \cdots, N$. The plan library contains J problems and K solutions. For the structure shown in Figure 1, there is a total of $I = \sum_{i=1}^{N} m_i$ feature-value pairs, or nodes in the feature-value pair layer. We label these feature-value pairs as FV_i, where $i = 1, 2, 3, \cdots, I$. In the problem layer, we use P_j to represent each problem, where $j = 1, 2, 3, \cdots, J$. In the solution layer, we use S_k to represent each solution, where $k = 1, 2, 3, \cdots, K$.

The first set of weights $V_{j,i}$ is attached to the connection between problem P_j and a feature-value pair FV_i if there is an association between them. The second set of weights $W_{k,j}$ is attached to the connection between a solution S_k and a problem P_j if S_k is a solution to P_j.

We now show how the weights attached to the problem-to-solution layer is adjusted based on user input. The weights attached to the feature-to-problem layer is similarly adjusted and we omit the description here.

Given the input feature-value pairs, the corresponding first layer nodes are turned on (set to one). A problem's score is computed first based on those feature-value pairs as follows. For each problem P_j, its score is computed using the following formula:

$$S_{P_j} = \frac{2}{1 + e^{-\lambda * \sum_{i=1}^{I} (V_{j,i} * X_i)}} - 1 \qquad (1)$$

where $j = 1, 2, 3, \cdots, J$, S_{P_j} is the score of the problem P_j, and X_i is 1, if there is connection between problem P_j and feature-value pair FV_i **and** FV_i is selected. Otherwise X_i is 0.

After the problem scores are computed, the problems and their associated scores will be presented to a user for selection and confirmation. For the confirmed problem, the user may next select their corresponding solutions which are associated with the current **selected and confirmed** problem. The computation of a solution's score is again similar to the computation of an output in a back-propagation neural network.

$$S_{S_k} = \frac{2}{1 + e^{-\lambda * \sum_{j=1}^{J} (W_{k,j} * S_{P_j} * \alpha)}} - 1 \qquad (2)$$

where S_{S_k} is the score of solution S_k, and S_{P_j} is the score of problem P_j. If there is no connection between solution S_k and problem P_j, then we do not include it in $\sum_{j=1}^{J} (W_{k,j} * S_{P_j} * \alpha)$ In the formula α is the *bias factor.*

As soon as the score of a solution is computed, it will be presented to a user for his judgment. If the user thinks that this solution is the right one and it has an appropriate score, he can confirm this by claiming success. Otherwise, a failure can be registered by the system. In both situations, a user can have the option to specify what the desired score of the solution is. This information will be captured by the learning algorithm, and will be used in the computation of the errors. After computing the learning delta values for weight adjustments, next we need to adjust the weights from the

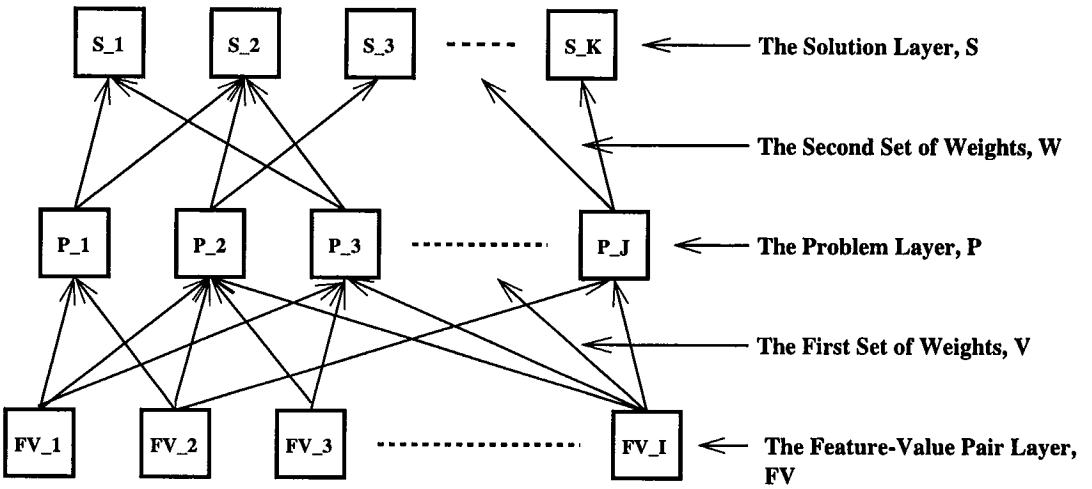

Figure 1: Layered structure of a plan library

solution layer to the problem layer, and then from the problem layer to the feature-value pair layer. We will adjust the weights attached to the solutions which are associated with the current **selected and confirmed** problem. The weights attached to the connections between the problems and the solutions will be adjusted first using the learning delta values computed above, and the problem scores computed in Formula 1. The formula for this adjustment is:

$$W_{k,j}^{new} = W_{k,j}^{old} + \eta * delta_{S_k} * S_{P_j} \quad (3)$$

where $W_{k,j}^{new}$ is the new weight to be computed, and $W_{k,j}^{old}$ is the old weight attached to the connection between solution S_k and problem P_j. In this formula, η is the learning rate.

Evaluation

In this section we present some preliminary test results to validate our techniques. We aim to show two properties of our maintenance systems, namely, speed and quality in the result. Below, we discuss the tests of our maintenance system called CaseAdvisor, implemented in Visual C++ on PC/NT and Unix, in each of the three problems we presented.

Testing the Information Retrieval Module

Testing was completed on large test files to illustrate how the information retrieval module scales up. Figure 2 demonstrates that even the one time cost of normalizing a text library is not that expensive. The time displayed is the CPU time required to remove the stop words from all of the texts, stem all of the terms, apply the user defined thesaurus, to extract key words and phrases from the texts and to build the inverted file structure. The information retrieval module was applied to a number of different text libraries containing different types of data. Each text was on average 0.3 kilobytes in size. The Sheffield LISA collection is a

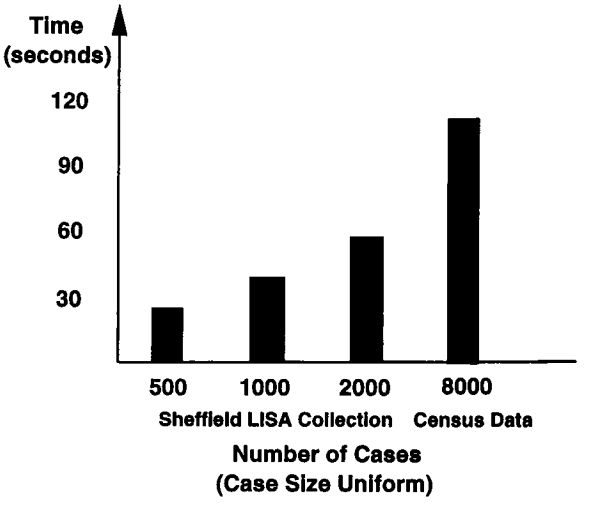

Figure 2: CPU Time To Apply Information Retrieval Techniques

database of abstracts and titles extracted from The Library and Information Science Abstracts database from Sheffield University.

We also tested the quality of the keywords generated by the maintenance system. Our test was done in a Cable-TV troubleshooting domain. Without tweaking, the key words generated by our system matched 87% of the key words generated by experts familiar with the cable-TV domain. The only list that the system did not match at least 80% of terms with was a whopping 116 words provided by a subject with limited domain expertise and computer experience.

Testing the Redundancy Detection Module

The redundancy module is responsible for testing an incoming text-based plan for possible redundancy. If

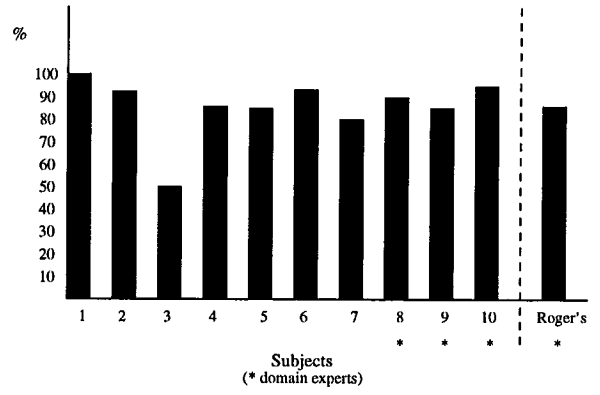

Figure 3: Percentage of Key Words Matching Those Found by our System

there is no possible redundancy, the plan is simply added to the existing plan library. If there is, the plan is presented to the user along with the plan causing the possible conflict. The user then determines which, if any, of the plans should be deleted from the plan library.

In our test of the redundancy detection module, 94% of the redundant plans were correctly identified by the application. Another encouraging statistic is that 83% of all plans identified as redundant were in fact redundant. Out of the 210 plans entered, 97 were correctly identified as redundant, 20 were falsely identified as redundant, 6 were falsely identified as not redundant and the remaining 87 plans were correctly classified as not redundant. This means that 88% of the plans were correctly classified. Using fuzzy string matching to determine redundancy allows for false positives. The threshold for identifying redundancy can be modified. However, this modification must be made at the expense of increasing the number of redundant plans that are not identified by the module. An additional area of improvement is that all of the plans involved in this experiment were derived from the same source. As part of the future work, it would be interesting to see how the above results generalize to plans authored by different plan authors at different times.

Testing the Plan Index Learning Module

In our next test, we used the plan library used in a local Cable-TV Company for troubleshooting equipment failures. This plan library is used by the technical representatives of the company to solve the customers' problems on the help desk. Up to now, this plan library contains 28 text-based plans and five features or questions. Within the five questions, there is a total of 30 question-answer pairs. We label each question-answer pair as QA.i, where $i = 1, 2, \cdots, 30$. The weights assigned initially to the individual question-answer pairs by the domain experts from the company.

In order to do our experiment, we set up two copies of the plan retrieval system using CaseAdvisor system with different sets of weights. The first copy of the plan library uses the weights specified by the domain experts from the company. The second copy has the weights initialized to 0.5. If we think the weights in the first copy represent a user's preference in the company, then we will learn these weights in the second copy using our dynamic learning method.

In our experiment, the whole learning process took four rounds, each of which is composed of query 1 to query 7. All the scores of the chosen plans produced by the second copy of the system converge to their desired scores produced by the first copy. We define the error of a plan produced by a valid set of question-answer pairs in the learning process as the absolute difference between its computed score and its desired score. In our test, the desired score of a plan is produced by the first copy of the plan library while the computed score is produced by the second copy. The error convergence chart for selected plans is graphed in Figure 4. In the figure, the X-axis represents the time line of the training process as the queries are entered. The Y-axis represents the errors in plan score. In the experiment, we can find that all the errors converge to 0, which means that all the plan scores converge to their desired scores. It can also be seen that the scores converge to zero at a fairly fast rate.

Conclusions

We maintain that text-based planning is an important form of planning. In the past, it has been more or less overshadowed by logic-based planning. One of the biggest advantages of text-based planning is its ability to convert text-based documents into plans which are useful for providing human users with expert advice. Because such documents are easily available through a number of data sources, particularly due to the rapid growth of the Internet, text-based planning will become more and more important in the future.

We also maintain that although text-based planning addresses the plan acquisition and maintenance problems to a certain extent, these problems do not simply disappear. Instead the problem of maintaining a library of plans with high quality becomes more apparent. To conduct knowledge maintenance and acquisition, we adapted techniques from information retrieval, machine learning using neural networks and case based reasoning. We have shown how to apply these techniques to normalize plans, to compare plans in order to find redundant ones, and to learn index weights in order to feedback the usage patterns into indexing mechanisms.

Acknowledgment

The work is supported by grants from Natural Sciences and Engineering Research Council of Canada (NSERC), BC Advanced Systems Institute, ISM-BC and Canadian Cable Labs Fund.

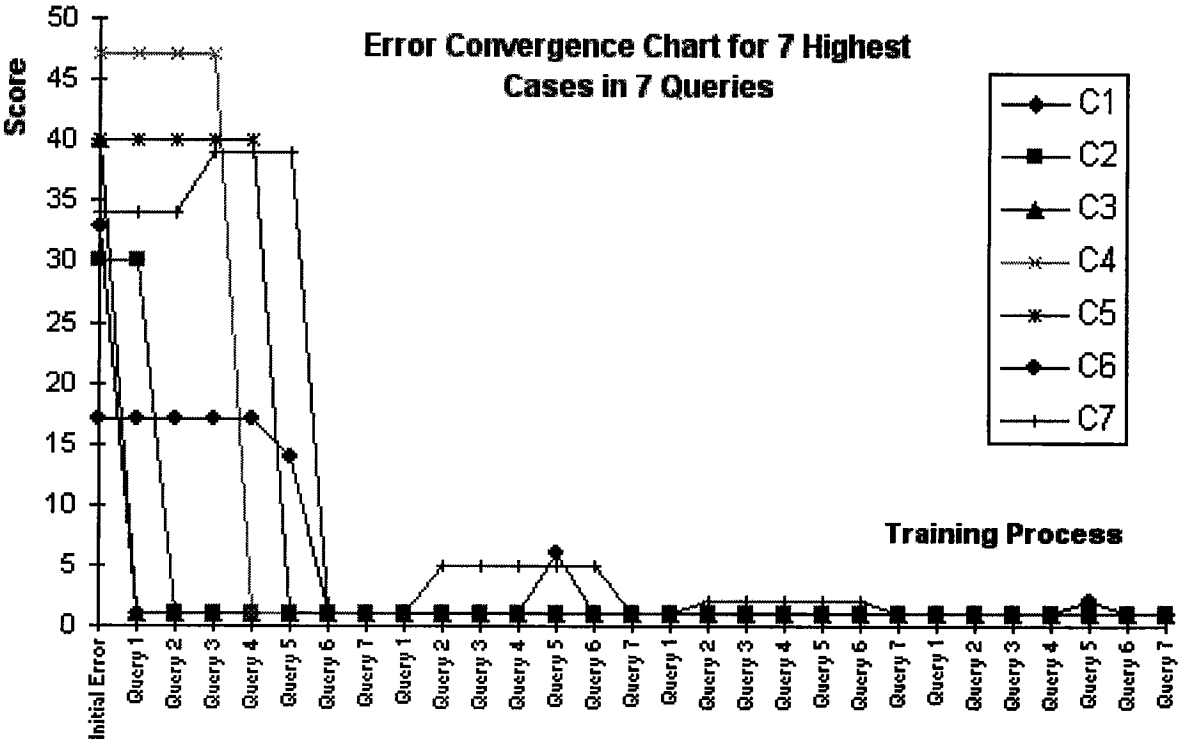

Figure 4: Error convergence chart for the seven highest ranked plans in the Cable-TV domain

References

A. Aamodt and E. Plaza. Foundational issues, methodological variations and system approaches. *Artificial Intelligence Communications*, 7(1), 1993.

R. Burke, K.J. Hammond, and J. Kozlovsky. Knowledge-based information retrieval from semi-structured text. In *in Working Notes from AAAI Fall Symposium on AI Applications in Knowledge Navigation and Retrieval.* American Association for Artificial Intelligence, 1995.

William B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-HALL, North Virginia, 1992.

Kristian Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45(1):173–228, 1990.

J. L. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publisers, Inc., 1993.

Janet Kolodner. *Case-based Reasoning.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

Janet Kolodner. *Case-based Reasoning.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

P Koton. Reasoning about evidence in causal explanation. In *Proceedings of the seventh National Conference on Artificial Intelligence*, pages 256–261, Menlo Park, CA.

David Leake. *Case-based Reasoning - Expriences, Lessons and Future Directions.* AAAI Press/ The MIT Press, 1996.

Kirsti Racine and Qiang Yang. Maintaining unstructured case bases. In David B. Leake and Enric Plaza, editors, *Case-Based Reasoning Research and Development*, volume 1266 of *Lecture Notes in Artificial Intelligence*, pages 553–564. Springer, Providence, RI, USA, July 1997. Second International Conference on Case-based Reasoning, ICCBR-97.

B. Smyth and M. Keane. Remembering to forget : A competence-preserving case deletion policy for case-based reasoning systems. *International Joint Conference on Artificial Intelligence*, 1:377–382, 1995.

G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* Computer Science Series McGraw Hill Publishing Company, New York, 1983.

D. Wettschereck and D.V. Aha. Weighting features. In *Proceedings of the First International Conference on Case-Based Reasoning, ICCBR-95*, pages 347–358, Lisbon, Portugal, 1995. Springer-Verlag.

I. Watson. Case-based reasoning tools: An overview. In *Proceedings of the Second UK Workshop on Case Based Reasoning*, pages 71–88, 1996.

Ian Watson. *Applying Case-Based Reasoning: Techniques for Enterprise Systems.* Morgan Kaufmann Publishers Inc., 1997.