Multiple Cases-Based Reasoning Including Justification

Toshio Morohashi and **Kuniaki Uehara***
Department of Computer and Systems Engineering
Kobe University
Nada, Kobe 657–8501, Japan
moro@jedi.cs.kobe-u.ac.jp uehara@kobe-u.ac.jp

Abstract

Case-based reasoning is a promising approach to develop a tutoring system. However, there still remains some problems to be solved. First, the system should explain the inference procedure upon the user's requests. Second, the explanation structure in the case may be inconsistent during the adaptation process. Third, when the system finds a gap between the new problem situation and the retrieved case, more complete matching should be performed by using not only the case but also multiple cases.

In this paper, the authors will introduce the explanation structure so as to explain the reasoning process. Then, the authors will introduce a justification process for justifying the explanation structure. Finally, the authors will study the adaptability of multiple cases to create a solution which explains a complex reasoning process.

Introduction

Currently, many researchers concentrate on a technique of storing text as case representations (*e.g.* legal documents and some kinds of manual). The technique which stores raw textual cases and retrieves some useful information from the case base is called "textual case-based reasoning."

One of the important features of case-based reasoning is adaptation between old and new situations. Adaptation of cases is needed for problem solving, because a retrieved case may not be the same as the given situation but partially matched with the situation. However, it is difficult to adapt raw textual cases since they are not composed structurally. For example, some kinds of manual contain information which tells the user how to solve a problem. In spite of this fact, human experts know many empirical techniques which are not described in the manual. In order to adapt such cases, the knowledge of empirical techniques should be integrated into the case so as to provide the user with helpful information.

Furthermore, the structured case can also be easily used to explain the reasoning process. Additionally, in order to explain the reasoning process, justification procedure will be needed. Justification procedure is to investigate its adaptability to the explanation structure and to enhance the reliability of reasoning. However, few researchers have studied the justification procedure.

The authors have been developing the system **Assist** that helps beginners' learning on UNIX environment and offers necessary UNIX commands to the user. The system needs not only to explain how to use the command but also to explain why the system arrives at the solution upon the user's request. To satisfy such a requirement, it is necessary for **Assist** to include an explanation structure in each case.

In **Assist**, each case corresponds to a specified knowledge of each command. Usually, it is enough for one command to make one operation (*e.g.* the **rm** command to remove file). However, a combination of commands is used to make two or more cooperating precesses. For example, the commands **cut**, **uniq**, and so on are usually used together with other commands such as **grep** and **sort** rather than used alone. As a result, the idea of combining multiple cases should be required for generating an adequate explanation.

In this situation, that is, adaptation of multiple cases, analyzing and modifying structured representation of a case is needed. However, adaptation of multiple cases may cause an inconsistency of the explanation structure after adaptation, because it may damage explanation structure during combination of multiple cases. Therefore, the authors will introduce the idea of justification to assure the explanation structure in a given case.

Example: Dialogue with Assist

In the authors' earlier work (Uehara and Ogino 1994), **Assist** was developed to be able to have the following dialogue with the user, where the dialogue between **Assist** and the user is written in italic, and we define "%" as shell prompt:

```
                % cp file.txt newfile.txt
User:           How can I change a filename?
System:         You should type the following:
                mv file.txt newfile.txt
```

*Research Center for Urban Safety and Security, Kobe University

Assist receives the lines followed by % as the user's command histories. Command histories are a sequence of UNIX commands which was typed just before the user's utterance. It provides information on the user's intentional behavior that cannot be obtained from the user's utterance. For example:

```
        % ps -aux
        % kill pid
User:   I can't kill the process.
```

Assume that **Assist** does not consider the command histories, then the answer may be the following:

```
System: You should type the following:
        kill (its process number)
```

It is a meaningless solution, because the user does not obtain any new problem-solving situation. However, **Assist** can obtain information from the command histories. That is to say, **Assist** can infer that the user knows the **kill** command and its process number. On the basis of this information, **Assist** can answer the question as follows:

```
System: You should type the following:
        kill -9 (its process number)
```

However, consider the following example:

```
        %rm dir
User:   Why? I can't remove the directory dir.
System: You should type the following:
        rm -r dir
```

In this case, in spite of obtaining the command histories according to the user's request, **Assist** cannot provide an adequate explanation of what the user desires. In other words, the user's request is *"why" isn't this command correct?*, and it means that what the user wants to know is the reason why the command was not executed correctly and how **Assist** generated the answer. In this situation, **Assist** should explain the reason why the user failed to use the command and the way to recover from the failure.

Needs of Cases Including Justification

It is important to answer the user's request and to explain the process of case-based reasoning, because the user may ask why the system generated such a solution. The system needs to store the process in order to explain the system's capability. In other words, representing the inference process indicates the range of the system's capability and reliability of its solution.

To solve this problem, we will introduce "explanation structure." It is possible to reuse this explanation structure if the system solves the problem and stores the process of problem solving. In addition, it is also possible to confirm the adaptability of cases by justifying each explanation structure in a given case. It means that "justification" is the ability to confirm the adaptability of the matching case to the problem situation.

Explanation Structure Including Justification

In the authors' earlier research, the system tried to retrieve the exact single case indexed by both user utterance and command histories. However, even if the system gives the user an adequate command to satisfy the user's request, the user may complain why the system arrives at such a solution. Therefore, the authors will introduce the explanation structure which is used to trace the procedure of how the system led to the solution. As a result, the user can ask the system for details when the user is not satisfied with the answer. For example, consider the following dialogue:

```
User:   How can I browse only the tenth line from
        a file?
System: You should type the following:
        head -10 foo.txt | tail -1
User:   Thank you. But ....
        Why do you use "head -10"?
System: I use the command to display the first 10
        lines of the foo.txt.
```

In this case, the system has an explanation structure (Figure 1). The system can indicate the process of why the **head** command was used before the **tail** command. The explanation process is as follows: First, the system points to the nodes **head** and **-10**, and searches for links with respect to the two nodes. As a result, the system reaches the node **line 1-10** and finally creates the sentence, "*I use the command to display the first 10 lines of the foo.txt .*"

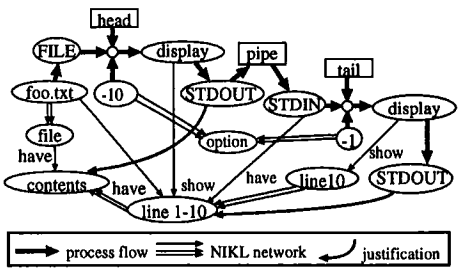

Figure 1: Explanation structure of "head -10 | tail -1."

At this point, each node associated with the command procedure is justified. Justifying the explanation is performed by tracing down a semantic network. That is to say, the network allows the system to create an explanation and assures the result. We will call such assurance "justification."

Figure 1 shows an explanation structure to explain the command, **head** and **tail**, where each node is linked by some arrows: a process flow, the NIKL network, and justification link. A process flow is represented by bold arrows. The other arrows represent semantic network for justifying explanation structure, where justifying an

explanation structure is defined as follows: Given a successful process of one UNIX command, a justification link is drawn to justify the reasoning process with respect to the command.

The authors have utilized the idea of the NIKL network (Neches, Swartout, and Moore 1985) to describe the semantic network. The justification link is associated with the NIKL network. NIKL has the usual complement of representational devices, such as concepts, roles on concepts, and links between concepts indicating subsumption. It is used to investigate whether the justification link is pointing the command procedure to its corresponding result after adapting cases with respect to its semantics.

Figure 2 shows a simple NIKL network, where double arrows indicate a-kind-of relations, while single arrows indicate attributes of concepts. Thus, in Figure 2, a file is defined to be a file system with a has-contents role that must be filled by contents, while text-file has its has-contents role filled by written contents.

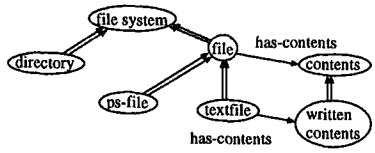

Figure 2: Simple NIKL network.

Using Multiple Cases

Components of Cases

In a tutoring system, pre-stored cases can be used as examples that can be offered to the user to help in solving a new problem. For the system itself, these cases can be used to analyze and to classify problem solutions. However, a different situation occurs when the system is supposed to monitor the problem solving process of a human expert user. Then, episodes from the user's own learning history consisting of solutions to problem solving tasks may serve as remindings (Weber 1994). On the basis of this idea, we will deal with two types of cases, that is, the **example case** and the **reminding case**.

The **example case** is a basic case to solve a problem. If the user treats **Assist** as the electric manual or on-line help system, it has enough capability. Figure 3 shows an example case which describes the **rm** command, where justification link is not described for the sake of simplicity. While the **reminding case** is an empirical case which is obtained from experts' techniques, they are not usually described in the manual.

For example, **Assist** can answer the following request by using the example case:

User: *How can I find "words" in this directory?*
System: *You should type the following:*
% grep *words* *

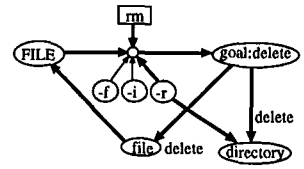

Figure 3: Example of example case.

This answer may satisfy the user's request; however, if *"words"* appears at so many places in all files in the working directory, the user can just see only the last part of the **grep** list. As a result, the user has to solve a new problem, that is, how to browse all the **grep** list. In this situation, **Assist** has the following reminding case:

% ls -al | more

This combination of commands, which utilizes "|", *i.e.* a pipe command, to create two or more cooperating command processes, enables the user to see one screenful at a time when the command returns a standard output larger than the screen on a terminal. This reminding case can be adapted to the all commands that output a result of the procedure on a terminal, *e.g.* **ps**, **nkf**, **zcat**. Then an example case with respect to the **grep** command should be merged with the reminding case. As a result, **Assist** obtains the following:

% grep *words* * **| more**

In this situation, if the **grep** list is short enough, the **more** command is an unnecessary procedure and the user may ask the reason why the system arrives at the solution. The system can explain the reason by using explanation structure as follows:

System: *Too long list unable to display on one screen,* **more** *command was recommended.*

Thus, combining reminding cases and the example cases enable the system to answer questions on more complex situations. It is important how the system combines and adapts these multiple cases.

Combination of Multiple Cases

Case-based reasoning is a method which uses stored cases as a guide to solving new problems. Given a new problem, the best case is retrieved from stored cases, and similarities between the retrieved case and the problem situation are identified. These similarities are used to adapt the retrieved case to fit the problem situation. While adapting the retrieved case, additional case is also used to cover a lack of knowledge (Sycara and Navinchandra 1991). On the basis of the idea, we will propose to use multiple cases for solving a problem.

In combining multiple cases, the adaptation process consists of three steps: *Step 1* is to select the best match case by comparing the index between the problem situation and each case. *Step 2* is to eliminate redundant

parts of the best match case. *Step 3* is to fill the gap between the problem situation and the best match case by using additional cases called cover cases. These steps are repeated until the index of the best match satisfies the problem situation.

In Figure 4, the procedure for *Step 1* retrieves the `Best Match Case` for `Problem Situation`, because it has the same indices (A, B, C, and D) except E. The procedure for *Step 2* eliminates superfluous composition of `Best Match Case`, that is, index F. The procedure for *Step 3* fills the gap index E between `Problem Situation` and `Best Match Case` by adding `Cover Case` with indices D and E. A new structural solution can be given by combining `Best Match Case` and `Cover Case`.

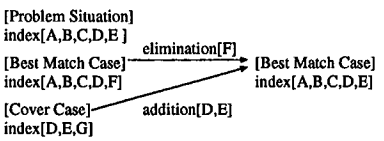

Figure 4: Combination of Multiple Cases.

Adaptation of Multiple Cases

In this section, we will consider adaptation of multiple cases. During the adaptation process, it is difficult to maintain explanation structure correctly, because adding and eliminating cases may create inconsistent explanation structures. Therefore, we will add and eliminate part of a case in consideration as follows:

Partial elimination of case If part of a reminding case is eliminated, its explanation structure may be broken. Therefore, the system has to investigate whether it is possible to eliminate the explanation structure which corresponds to part of the case.

Addition of case Some covering cases may be added to reminding cases. If the reminding cases are reconstructed as a result of addition, they are also justified according to their semantic networks.

Example 1

We will consider a situation where the system selects a reminding case which consists of **dvi2ps**, **psnup**, and **lpr** commands, and eliminates a portion of the **psnup** command from the case (Figure 5). The explanation structure of the reminding case may be broken by the lack of an intermediate node after elimination. Therefore, the system has to investigate its consistency of the explanation structure.

In this case, justification is still maintained for each command, **lpr** and **dvi2ps**, before and after elimination. In Figure 5, for example, consider justification link for the **lpr** command, the justification link points the **lpr** procedure to the **separated ps-file** node. After elimination, justification link points to the **ps-file** node

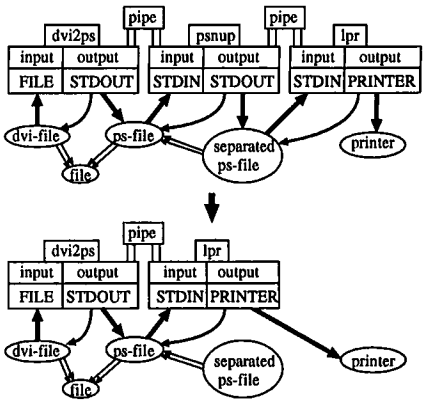

Figure 5: Example of eliminating **psnup** to **dvi2ps | lpr**.

which is a higher node with respect to the NIKL network. Therefore, its justification is maintained, and the system obtains a correct explanation structure. It is also certain that justification to the **dvi2ps** command is maintained, because the **dvi-file** node is still linked to the **dvi2ps** procedure after elimination.

Example 2

Consider another situation where the system retrieves a reminding case and inserts another cover case to satisfy the problem situation. Suppose that the user has a data file which has some kinds of words and wants to know the number of times that each word occurs in its file. As a result of retrieval, the system obtains the following cases, where BMC stands for the best match case and CCs are cover cases:

BMC: **cat** *foo.txt* | **uniq -c**
CCs: **psselect -p2,5** *foo.ps bar.ps*
CCs: **sort** *foo.txt*
CCs: **vi**

The system attempts to complement BMC, **cat | uniq**, with CCs, **psselect**, but the consistency of its explanation structure may be broken, because the justification link does not point the correct node. The result is shown in Figure 6. The two nodes, **contents** and **selected ps-file**, are not justified from the result of commands, **psselect** and **uniq**, because it is necessary for the input **contents** node to be pointed to the **uniq** procedure by justification link in order to justify a command procedure. We consider the node **contents** in Figure 6; the justification link of the **psselect** command does not point out the node **contents** but the node **ps-file** which is incorrect for justifying its command procedure. Therefore, this explanation structure from the procedure of the **psselect** command is judged to be inconsistent.

As a result of the judgment, the system attempts to complement the BMC with the next CCs, *i.e.* **sort** (Figure 7). In this case, it is possible to justify the two nodes **contents** and **sorted contents** by the result of their processes. In consequence, the final output is a

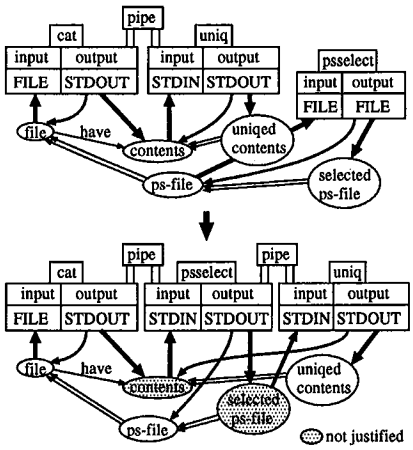

Figure 6: Example of inconsistency in adding case.

file that is affected by both **sorted** and **uniqed** contents, and higher-level concept **uniqed-sorted-contents** is created. This is enough to explain the process, and the system can answer the following request:

System: *You should type the following:*
cat *foo.txt* | **sort** | **uniq -c**
User: *Why is the sort command required?*
System: *As* **uniq** *removes only adjacent duplicate lines, a sorted data is necessary.*

In this case, although two different cases are combined, the system can explain the reason for the necessity of the **sort** command. It is because, after adaptation, explanation structure is connected by the semantic network, and it indicates consistency in explanation.

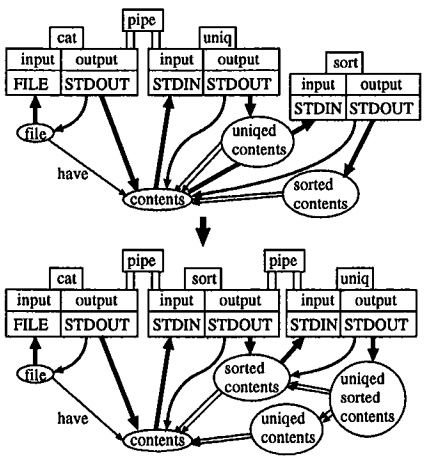

Figure 7: Example of adaptation of multiple cases.

Conclusions

In this paper, the authors have presented the necessity of including explanation structure and justification in the case base and have shown the effectiveness of utilizing these multiple cases. The authors have integrated them in the adaptation process of case-based reasoning and have shown that justification preserves the adaptability of multiple cases. Therefore, we can conclude that this integration works well for complex problem-solving cases.

At present, the authors are working on the user's utterance and command histories as case indices for retrieval. For another indexing candidate, the authors will consider a "standard error message" which is returned by a UNIX system. If such messages are treated as indices that indicate the reason for failure, we will be able to explain the user's failure more strictly.

Here, we know a similar approach that is called "derivational analogy." It proceeds through the reasoning steps in the construction of the past solution and considers whether they are still appropriate in the new situation or whether they should be reconsidered in light of significant differences between two situations (Carbonell 1986). In derivational analogy, Carbonell proposed a procedure called "failure-cause propagation" in its derivational trace. If we adopt this idea, the system will be permitted to explain the user's failure-cause.

Furthermore, in derivational analogy, to modify a plan accurately and efficiently, knowledge compilation and decompilation techniques are proposed. If we can adopt knowledge compilation, we will be able to control explosion in increasing the number of cases.

References

Uehara, K. and Ogino, H.: A Case-Based Approach to Text Planning, *Proc. of the IASTED International Conference on Artificial Intelligence, Expert Systems and Neural Networks*, pp.123–127 (1994).

Neches, R., Swartout, W. R. and Moore, J. D.: Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development, *IEEE* Transactions on software engineering, Vol.SE-11. No.11, pp.1334–1350 (1985).

Weber, G.: Methods and Tools: Examples and Reminding in a Case-based Help System, Advances in Case-Based Reasoning, Lecture Notes in Artificial Intelligence, Vol.984, pp.165–177 (1994).

Sycara, K. P. and Navichandra, D.: Index Transformation Techniques for Facilitating Creative use of Multiple Cases, *Proc. of 12th IJCAI*, pp.347–354 (1991).

Carbonell, J, G.: Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds.) *Machine Learning*, Volume II, pp.371–392, Morgan Kaufmann (1986).