Implementing Case-Based reasoning in SICStus Prolog

by
Nils Hagner and Anders Tunevi
Implementing Case-Based Reasoning in SICStus Prolog

Nils Hagner, Anders Tunevi
Infologics AB
P.O. Box 91, S-191 22 Sollentuna, SWEDEN
Phn: +46 8 92 20 00, Fax: + 46 8 96 08 46
Electronic mail: NILSH@INFOLOG.SE, ANDERS@INFOLOG.SE

November 25, 1990

Abstract

Case-based reasoning (CBR) is a novel technique which can, for some domains, serve as an alternative to "traditional" rule-based expert systems techniques. Systems that are based on CBR have many interesting properties. Moreover, CBR have many important advantages over rule-based systems.

In this paper, we give a presentation of the basic ideas behind CBR. A comparision is made between rule-based systems and CBR systems. We also discuss how CBR systems can be implemented, in particular using SICStus Prolog. Finally, we present a simple CBR system for the domain of photography.
Contents

1 Introduction ........................................ 5

2 Knowledge-Based Systems .......................... 5
   2.1 Architecture of KBS Systems .................. 5
   2.2 Problems with Rule-Based Systems ............ 7
      2.2.1 Knowledge Acquisition .................... 7
      2.2.2 Intermediate parameters .................. 7
      2.2.3 Consistency of knowledge bases .......... 8
      2.2.4 Maintaining a knowledge base .......... 8
      2.2.5 The size of knowledge bases ............ 8
      2.2.6 Integration ................................ 8
      2.2.7 Efficiency ................................ 9
   2.3 What needs to be done? ...................... 9

3 Case-Based Reasoning ............................... 9
   3.1 Advantages of CBR ............................. 10
      3.1.1 Knowledge representation and acquisition 10
      3.1.2 Maintaining and expanding the system .. 10
      3.1.3 Integration ................................ 10
      3.1.4 Efficiency ................................ 11
   3.2 The General Principle ....................... 11
   3.3 The Architecture of a CBR System ............ 11
   3.4 Inference in a CBR System .................. 12
   3.5 Combining CBR and Other Techniques ........ 15

4 Constructing a CBR System ....................... 16
   4.1 Appropriate problem domains ............... 16
      4.1.1 Completeness of problem descriptions .... 16
      4.1.2 Retaining the organization .......... 17
      4.1.3 Routine problem solving ............. 17
      4.1.4 Hardware: storage capacity .......... 17
   4.2 Knowledge acquisition ..................... 17
   4.3 Generalizing a Case Base .................. 18
   4.4 Automation and Incremental Improvement .... 19
5 Using Prolog for Implementing CBR 20
  5.1 Unification as Matching 20
  5.2 Indexing for Large Case Bases 22
  5.3 Using Parallelism to Improve Efficiency 22
  5.4 Using Constraints in Matching 23

6 PHOTO — A Simple CBR Application 23
  6.1 Representation 24
  6.2 System Overview 25
  6.3 A Sample Session with PHOTO 26

7 Conclusions 28

A Listing of the PHOTO program 30
1 Introduction

Although a large number of applications based on knowledge-based systems (KBS) technology exist today, the KBS business has not grown as much as the previous expectations. One of the problems is that knowledge-based systems using today’s technology are very expensive. As a result, many companies and organization are unwilling or unable to invest in knowledge-based systems.

One way of getting around this problem, at least for some application areas, is to use an alternative technology. As a part of the ISP project at SICS, we study one such technique, called case-based reasoning (CBR), and its relationship to Prolog. Knowledge-based systems based on CBR have many advantages over the “traditional” KBS technology. These advantages make it possible to develop a number of new, interesting applications, simply because the cost/benefit situation has changed.

CBR systems can be implemented in a both natural and efficient way using Prolog. In this paper we present a simple application of CBR, written in SICStus Prolog. We discuss how e.g., the database system in the industrial version of SICStus Prolog can be used. Furthermore, we discuss how parallelism and constraints can be used in the implementation of CBR systems.

In order to illustrate the simplicity of the technique and its natural mapping to Prolog, we present a simple application of case-based reasoning. The program is called PHOTO, and is used for diagnosing failed photographs.

2 Knowledge-Based Systems

Although a large number of knowledge-based systems have been developed since the early 1970's, the basic principles have, by and large, remained unchanged over the years. Each system have the general architecture displayed in Figure 1.

2.1 Architecture of KBS Systems

The knowledge-based systems are based around an inference engine, which can be seen as the “heart” of the system. The system communicates with
the user or the mother application through a user interface. A database is used for storage of parameter values. The problem-solving knowledge in the system is represented as a set of rules in the rule base. Finally, most knowledge-based systems provide an explanation facility, which is used for explaining conclusions, providing motivation for requested user input, etc.

In these systems, problem-solving corresponds to a manipulation of the parameters in the database. The rules are, in general, divided into two parts: a precondition and a conclusion. If the preconditions of a rule, expressed in terms of the parameters in the database, are satisfied, then the rule can be applied. In general, this means that new parameter values are asserted in the database. When certain parameter values have been derived, then the problem at hand is solved.

In most systems, the parameters can be divided into three groups:

1. parameters that characterize the problem at hand,
2. parameters that represent the solution of the problem, and
3. parameters that are used as intermediate results during a problem-solving session.
2.2 Problems with Rule-Based Systems

2.2.1 Knowledge Acquisition

The main difficulty in the development of a KBS system is the acquisition and representation of knowledge. In fact, acquiring and encoding knowledge often is the major factor in the development of knowledge-based systems. Due to this problem, the term “knowledge acquisition bottleneck” as emerged. A lot of research effort has been spent on methods to automate the process, but few practically useful results have been presented.

Knowledge acquisition, in general, consists of a few different steps:

1. Identify the problem that is to be solved.
2. Choose an appropriate representation of facts (parameters) and rules.
3. Identify the parameters that are needed for solving the problem.
4. Extract problem-solving knowledge from one or more experts and encode this knowledge in terms of rules.

The above process is very difficult and requires a lot of time. In many cases it is rather simple to identify the problem to be solved, the parameters needed, and to choose an appropriate representation formalism. The last part, on the other hand, is very complicated. One reason for this is that the knowledge of experts, which in many cases is of an intuitive form, is quite difficult to express in terms of general rules. Therefore, in general, the process of writing rules must be made in an iterative form, where rules are continuously constructed, tested, and debugged.

2.2.2 Intermediate parameters

It is, in general, very difficult to decide the intermediate parameters needed in a system, and how these should be related with each other and the other
parameters. In practice, the set of intermediate parameters grow (and sometimes shrink) during the development of the system.

2.2.3 Consistency of knowledge bases

The development of a rule base also has some other problems. One of these is that it is very difficult to keep a knowledge base consistent. This means, in effect, that a rule base may contain directly or indirectly contradicting rules.

2.2.4 Maintaining a knowledge base

Another problem in connection with knowledge bases is that it is very difficult to maintain a rule base. This is often noticed when existing knowledge-based systems are to be adopted to changes in the organization. Sometimes the adaption, which often means that the rule base has to be changed, results in a discouraging conclusion: it is cheaper and easier to start all over again — from scratch.

2.2.5 The size of knowledge bases

Finally, a more subtle problem associated with rule-based system is that such a system will not be useful before a rather large rule base has been constructed. It is very difficult to develop "incremental" systems, that is, systems that can easily be put to work, and that can be subsequently expanded into an operational system.

2.2.6 Integration

In the earlier days, knowledge-based systems were constructed as stand-alone applications. Today, the market situation has radically changed. It can be expected that almost every knowledge-based system in the future will be an integrated part of a larger application. Due to the KBS technology, it is possible to extend the functionality of existing systems. For instance, a higher degree of automation can be achieved by using KBS technology in a large system centered around a database.

Present KBS technology is, in general, very hard to integrate into other systems. As a response to the stronger integration requirements, the KBS
2.3 What needs to be done?

Tools market is slowly changing into an “integrational view” of KBS technology. There are a few expert systems development tools that directly support interfacing to databases, window systems, etc. Still, however, there is a rather wide gap between present KBS technology and e.g., relational database technology.

2.2.7 Efficiency

Finally, traditional KBS technology suffers from efficiency problems. This means that knowledge-based systems, in general, are too slow. Therefore, KBS technology is hard to use in e.g., real-time domains. Although the KBS tools available today are better at producing efficient systems (e.g., by compiling rules into machine code), further achievements in efficiency needs to be made.

2.3 What needs to be done?

In order to “revitalize” and expand the KBS market, a change in situation must be achieved. It is necessary the future knowledge-based systems are much cheaper than today. Otherwise, the investments in knowledge-based systems will be kept at a low level. The knowledge-based systems of the future must also be much easier to integrate into existing systems, as e.g., large database systems. Knowledge acquisition must be made more simple than today. Automatic and semi-automatic methods will shorten the development time radically. Finally, the knowledge-based systems of tomorrow must be more efficient than today.

3 Case-Based Reasoning

One way out of the problems described in the previous section, at least for a large number of applications, is to use an alternative KBS technique. One such technique, which we find very promising, is case-based reasoning (CBR). Although this technique is not as general as rule-based systems, it offers a number of important advantages for many problem domains.

Case-based reasoning is a relatively new technique and only a few applications have been developed using CBR. However, it can be expected that this technique becomes more and more important in the near future.
3.1 Advantages of CBR

The main disadvantage of case-base reasoning is that it is not as general as rule-based techniques. However, CBR offers many important advantages that in many cases outweighs this lack of generality.

3.1.1 Knowledge representation and acquisition

As mentioned before, it is very difficult to express problem-solving knowledge in terms of general rules that manipulates a set of parameters. In a CBR system, the knowledge has the form of cases, that is, clear descriptions of previously solved problems. This means, in effect, that old solutions are directly used as a "knowledge base". Collecting such a set of previously solved cases and rewriting them into a form which can be handled by the computer is an easy and a rather natural process. In many situations, a set of previously solved problems is already present in the documentation of the organization. In many situations it is also possible to automate the knowledge acquisition process to a large extent.

3.1.2 Maintaining and expanding the system

Rule-based systems are difficult to maintain and to extend. Systems based on CBR, on the other hand, can rather easily be expanded both in scope and in accuracy. Moreover, this can often be made automatically, which means that a knowledge-based system improves its performance over time. Although research on learning techniques has been intensive, it is very difficult to achieve a similar behaviour for rule-based systems.

3.1.3 Integration

CBR systems, in general, has a more natural correspondence with e.g., database systems than rule-based systems. In many situations, data structures similar to the ones used in database systems can be used. In fact, CBR can be integrated with database systems in a way where the knowledge-based subsystem is directly based on the database system. This, of course, has important consequences for integration. For many applications, CBR can easily be integrated with existing systems. It is even possible to write CBR applications using database languages as e.g., extended versions of SQL.
3.2 The General Principle

3.1.4 Efficiency

Finally, an important advantage of CBR systems over rule-based systems is that they are more efficient. This is due to the relative simplicity of CBR techniques. This makes it possible, for example, to use CBR for real-time monitoring tasks.

3.2 The General Principle

In theory, case-based reasoning is not really a “distinct” concept. CBR systems can have many different forms. There is, however, an underlying basic principle which is common to all CBR systems: Previously solved problems are used to solve new problems. Instead of abstracting information from previously formed problems and putting them into rules, previously solved problems are used in a more direct way — as they are — for solving new problems.

It is the simplicity of this general rule that gives CBR systems their power. CBR systems are easy to construct because information about previously solved problems can easily be obtained. CBR systems can incrementally improve their performance by continuously adding new solved cases to the systems. This, too, is simple, since new solved problems are constantly produced by the system.

3.3 The Architecture of a CBR System

A CBR system, in general, consists of four distinct parts, as is illustrated in Figure 2.

1. An interface provides means of communication between the CBR system and the user or the “mother application.”

2. A case base is used to store a potentially large set of previously solved cases.

3. A matching mechanism is used for retrieval of old cases which are similar to the problem at hand.

4. An inference module is used to find a solution to the problem at hand, given some previously solved, similar cases retrieved from the case base.
The relative simplicity of the architecture makes CBR an appropriate technique. More importantly, the architecture allows a simple and natural integration with e.g., database systems.

3.4 Inference in a CBR System

In rule-based systems, a problem-solving session usually have the form of successively filling in parameter values, either by using rules, or by asking the user. This means, in effect, that a large portion of the problem solving process is spent on giving a further detailed description of the problem at hand. When this description has been achieved, then the solution to the problem can be deduced by the rules.

Case-based reasoning, on the other hand, assumes that a fairly complete classification of the problem at hand is present before the inference mechanism is engaged. We argue that this holds for many practical problems, and is further discussed later in the paper.

Inference in a CBR system consists of two major steps:
1. Given the specification of the problem at hand, retrieve a set of old cases from the case base which are most similar to the problem at hand.

2. Use information from the cases in the retrieved cases for solving the problem at hand.

The main difficulty lies in the first step, that is, the retrieval of similar cases. In some situations, a case base may contain thousands of cases, and it is the task of the retrieval mechanism to select ten or twenty cases which, in some sense, are "similar" to the problem at hand. The key issues are: how do we know whether one case is more similar to the present problem than another, and how can we efficiently retrieve the most similar cases?

It is more or less impossible to give a general description of the notion of similarity. Indeed, this notion is clearly very application-specific. The technique used in most CBR systems is to identify relevant features, which are used to discriminate between different cases. Speaking in terms of parameters, this means that some parameters are identified being particularly important for the problem solving process. Different features, or parameters, bear different importance, which results in an ordering between the features. In many systems, a simple division have been made into:

- very important features, which are used to make the coarse discrimination,
- less important features, which are used to make a more fine-grained discrimination, and
- features that are not needed to discriminate between cases in the retrieval process.

In order to achieve an effective retrieval of the old cases, it is necessary to use efficient indexing techniques. The indices used are linked to the features which discriminate between cases. Using the three-part division above, a primary index would be used on the most important features, and a secondary index on the less important features. This indexing scheme reduces the necessity to scan through a large set of cases, so that only those selected by the indices need to be scanned. With a good selection of features, and thus, indices, retrieval becomes very efficient.
When a set of similar cases has been retrieved, the cases are further analyzed and are ordered by the "degree of similarity" to the problem at hand. The next step is to use each case in the retrieved set in trying to solve the problem at hand. If the old case is an exact match of the problem at hand, then a solution can immediately be presented — the same solution as for the old case.

In other situations, there is not an exact match between the problem at hand and the retrieved cases. In such a situation, there are some different ways in which the process can proceed. The simplest one is to present the solution in case which is "most similar" as the most likely solution to the problem at hand. However, more sophisticated methods can be used:

1. Deductive mechanisms, together with a (possibly simple) model of the domain can be used to deduce further information about the problem at hand. It is, for instance, possible to include rules which describe how certain features (i.e., parameter values) can be deduced.

2. An interactive mechanism can be used to ask the user for more information, which can be used to further specify the problem at hand.

In both of the above approaches, a further specification of the problem at hand may cause some of the retrieved cases to be rejected. By filling in more information, by deductive mechanisms or by consulting a user, results in a continuous decrease in "distance" between the problem at hand and the retrieved cases. At some point, where an exact match have been made, or when the problem at hand cannot be further specified, the solution of the most similar case is presented as the solution to the problem at hand.

Some researchers have pointed out that it might be possible to use information from several old cases to solve the problem at hand [5]. Although the information in each of the retrieved cases may not alone be sufficient for solving the problem, the combination of information from several cases might just do the job.

It is also possible, for instance in a decision support system, to present a number of possible solutions. There are many applications where problemsolving actually comes down to finding similar earlier cases. In such a system, a solution corresponds to presenting one or more of the most similar cases, together with a detailed description of the cases themselves.
3.5 Combining CBR and Other Techniques

If the given problem specification is too general, then it might be possible that a large number of cases are retrieved from the case base. This is clearly an undesirable situation. However, this problem can in many situations be handled by using more sophisticated case representation schemes [6].

3.5 Combining CBR and Other Techniques

A knowledge-based system need not be based on CBR only. Indeed, CBR can be used as a complement in both rule-based systems and model-based systems. Some researchers have constructed systems which actually use all three methods in a "hybrid" system.

One interesting use of CBR in connection with rule-based system is in the form of a rule cache. In such a system, problems can be normally solved using ordinary rule technology. However, each solved problem is stored in a case base, together with the path (i.e., the sequence of rules used) that led to a solution. When a new problem is encountered, the case base is consulted for retrieval of similar cases. Using the paths associated with the cases, several levels of the search tree can be traversed directly. When a difference between the old case and the present problem is detected, the stored path is abandoned and another direction is used instead.

This rule cache has several advantages. First, it decreases the amount of search needed; the path of the old case is followed as far as possible. Second, the system will incrementally improve its performance. The main drawback is that such a system requires a larger working memory than an ordinary rule-based system.

It is also possible to go the other way, that is, to include rule-based technology in a CBR system. In such a system, the major part of problem solving is taken care of by the case retrieval mechanism. Then, rules that manipulate parameter values, or feature values, can be used to deduce more information about the problem at hand. This results in a further specified problem, which in turn results in a more precise overall conclusion.

Finally, some researchers have suggested a combination between both case-based reasoning, rule-based techniques and model-based techniques [1].
4 Constructing a CBR System

Due to the simplicity, it is fairly easy to construct a CBR system for many domains. It is, however, important that an appropriate domain is used. Although there are restrictive criteria on the appropriate domains, many practical problems are feasible for CBR solutions.

4.1 Appropriate problem domains

CBR systems can be used in a number of different “modes of operation.” First, a clear distinction should be made between autonomous and interactive systems. Autonomous systems solve problems without any interaction with a user. A CBR system operating in this mode is particularly appropriate when the system is designed to efficiently handle a large number of “normal” problems, and where more complicated problems are handled manually by human specialists. Examples of such applications is the classification of alarm signals, insurance underwriting and “forms analysis.”

CBR systems can also be used for interactive applications. It is, however, important that the problem to be solved has been given a fairly complete specification before retrieving cases from the case base. Examples of such applications are software debugging systems, document retrieval and diagnostic systems.

CBR systems should be used for solving problems, where the routines of the organization are retained intact. The reason behind this is that already available data and documentation can be used directly in constructing the system. Also, as explained below, it is possible to use “wire-tapping” techniques in the construction process.

The above remarks are concluded into a “checklist”, which is given by the subsections below.

4.1.1 Completeness of problem descriptions

The problems to solve should be more or less completely specified. This means, in rule-based terminology, that the number of intermediate parameters should be very small.

As an example, a typical insurance form contains a number of “boxes” and “fields,” and is more or less completely filled out by the insurance customer.
Here, the problem specification is fairly complete when given to the system.

4.1.2 Retaining the organization

The problem-solving system should retain the organization of the problem solving process. This makes it possible to use knowledge from earlier problem solving directly in creating the case base in the CBR system. A reorganization generally means that this knowledge takes another form, which then cannot be used directly in the CBR system.

4.1.3 Routine problem solving

Case-based reasoning systems operate most efficiently when their purpose is to solve the "routine" problems only. This means, in effect, that frequently occurring problems are managed by the CBR system, while more complicated and less frequent problems are handled manually by human experts.

One thing that can be observed is that a CBR system equipped with automatic mechanisms for extending the case base will successively be able to handle more and more complicated problems. This means that "nonroutine" problems gradually become "routine" problems.

4.1.4 Hardware: storage capacity

In order to achieve high performance in a CBR system, it is necessary for the underlying hardware to meet some requirements. First, the accuracy of conclusions of a CBR system is mainly dependent on the number of cases in the case base. In some domains it is necessary to include tens of thousands of cases, which in turn might require a rather large secondary storage space. Moreover, since each case might be of a considerable size, it is in general necessary to have a large internal memory space. Moreover, the performance of the case retrieval mechanism is dependent on the performance of the secondary storage.

4.2 Knowledge acquisition

In case-based reasoning, the process of knowledge acquisition can be divided into a number of separate tasks:
1. determining whether the problem domain at hand actually is suitable for CBR solution,

2. collecting a (possibly large) set of previously solved problems — a case set,

3. selecting an appropriate representation of cases,

4. converting the case set into a case base using the selected representation scheme, and

5. determining what features should be used as indices.

Since the cases in the case base is a rather natural and “direct” representation of actually solved problems, knowledge acquisition becomes much simpler than for rule-based system. In the latter case, the knowledge in the previously solved problems has to be generalized, partitioned and represented in relatively small rules. In CBR systems, the previously solved problems are maintained as units.

If a CBR system is developed to solve a problem that is well documented, for example in terms of fault and corresponding service reports or processed insurance forms, then it is possible to use the documentation more or less directly in the system. Some amount of knowledge engineering, in cooperation with domain experts, may be needed if explanations are included in the cases, or if the general rules are used in derivation of parameter values. Knowledge engineers can also increase the performance of a system by e.g., generalizing the case base.

An interesting observation that can be made is that it is often possible to construct a case base by “wire-tapping” an ongoing problem-solving process. This means that a case base can be incrementally constructed with very little effort. In such a situation, building the case base corresponds to “recording” the problem solving process.

### 4.3 Generalizing a Case Base

Cases in the case base can be organized in a hierarchy corresponding to their relative generality. Such a hierarchy forms a tree, where the most general cases are in the top of the tree, while the most specific cases are the leaves. Retrieving a case, then, corresponds to a traversal of the tree, and to find the
most specific case matching the problem at hand. If the deductive process
later rejects this case, then a more general case in the hierarchy is used.

It is, of course, possible to build a large case base which basically consists
of very specific cases. Unfortunately, the number of cases needed to make up
such a base quickly becomes very large. One way to avoid this problem is to
generalize the case base. Then, some of the leaves can be deleted, resulting
in a less complex tree structure.

In many situations, this generalization can be made without loss of in-
formation. For example, There may be two leaves in a tree, representing
two specific cases, where one parameter discriminates between the cases, but
where the solutions given in both cases are the same. Such a situation in-
dicates that the discriminating parameter is really of no significance for the
solution. This means that the two cases can be generalized into a single,
Somewhat less specific, case which covers both of the previous cases.

It should be noted, however, that the accuracy of the conclusions of a
CBR system is directly dependent on the “quality” of the case base. This
means, in practice, that generalization in which information is lost can result
in a less accurate system. Also, for efficiency, it is important that the most
occurring problems are solved by cases that are very specific.

4.4 Automation and Incremental Improvement

In many problem domains, both the acquisition of cases and the generaliza-
tion of a case base can be automated to a large extent. It is easy to maintain
a case base consistent, that is, to assure that two cases with the same features
hold the same conclusion.

Altogether, the relative simplicity of construction and generalization of
case bases makes it possible to use a high degree of automation in the con-
struction of CBR systems.

Due to the simple form, it is possible to continuously enter new cases into
the case base as new problems are solved. This means, in effect, that a CBR
system can be constructed in a way where its performance is continuously
improved. This is very difficult to achieve in a rule-based system.

In a CBR system intended to solve only the “routine problems”, those
problems that are solved manually can later be entered into the case base.
This means that more and more problems actually become “routine,” which
in turn means that the scope of the CBR system becomes wider.
5 Using Prolog for Implementing CBR

CBR systems can, in principle, be written in any programming language. It is, for instance, possible to construct a CBR application using SQL and utilizing a database to store the cases. As can be expected, however, some languages are more appropriate for the implementation of CBR than others.

One language which we argue is indeed appropriate for the implementation of CBR, is Prolog. Prolog offers many features which makes the construction, management and operation of CBR systems easy and yet efficient. Some of these advantages are:

- The data structures of Prolog makes the representation of cases simple and natural.
- The powerful unification mechanism can be directly used in case retrieval.
- The indexing scheme of Prolog improves the efficiency of case retrieval.
- Logical variables can directly be used as 'unknown' and 'unimportant' values.
- The high level of the language makes the CBR system intuitive, clear and short.

In particular, the industrial version of SICStus Prolog [3] that has been used in this study, offer several features that support the construction of CBR systems.

5.1 Unification as Matching

In many domains, it is possible to use the unification mechanism directly as a matching operation, that is, to retrieve similar cases. This means that one of the more difficult parts of a CBR system is moved down to a level below the programming language. Using unification as matching, however, poses some constraints on the representation of cases.

In the simplest form, a case consists of two parts:

- a set of features, or parameters,
• a solution.

This means that the case base can be simply represented as a set of binary unit clauses:

\[ \text{case}(p(F_1, F_2, \ldots, F_n), \text{Solution}). \]

The parameters \( F_1, \ldots, F_n \) are given a strict order, that is, the parameters are identified by their position in the term. Parameters that are irrelevant for the conclusion in a specific case can be represented by an uninstantiated variable. The problem at hand is a term of the form

\[ p(P_1, P_2, \ldots, P_n) \]

where each \( P_i \) is a parameter, using the same ordering as in the case base. A parameter is either a value, represented by an atom or an integer, or is unknown, which is represented by a variable.

Now, as a simple measure of similarity, we say that cases, whose set of parameters unify with the parameters of the problem at hand are similar. If they are not unifiable, then one or more of the parameters are conflicting, which means that the case considered is not appropriate for the solution of the present problem. It is important to point out that unification in this situation should be used as a test — neither the variables in the present problem nor the variables in the old case should be bound.

This simple matching mechanism has one drawback: it is not possible to distinguish between more and less similar cases. This, however, can be solved in a rather straightforward way. A more special matching mechanism can be constructed, totally based on unification and equality tests. A similarity measure, or rank, can be computed in the following way:

1. Create a list of those cases that are unifiable with the problem at hand.
2. For each of the selected cases, do the following:
   (a) Compare each pair of parameters between the present problem and the examined case.
   (b) If the parameters match directly, e.g., are equal, then give the parameter a higher rank. Two variables are in this case considered equal, but two directly matching (atomic) values should be given a higher rank than two matching variables.
(c) If the parameters match, but one of them is a variable, then give the parameter a lower rank.

(d) Multiply the computed rank by a number which represents the significance of the parameter.

(e) Sum the ranks of each parameter into a rank for the case.

3. Sort the list of cases according their ranks.

As a result, a list is produced in which the retrieved cases are organized according to their "similarity" to the problem at hand. This list can then be passed to the inference module for the rest of the processing.

Case retrieval, using the above principle, is further simplified by the `findall` predicate that is provided by SICStus Prolog. This means that there is no need to manually scan through the case base — it is all taken care of by the underlying Prolog system.

### 5.2 Indexing for Large Case Bases

One observation should be made at this point. The internal database provided in Prolog systems, including SICStus Prolog, is not optimized for a large number of clauses defining a single predicate. This means that, in applications where the case base contains thousands of cases, case retrieval will be rather inefficient. The reason behind this is that the indexing mechanism of Prolog is too simple. The available indexing is sufficient to support efficient execution of Prolog programs, but insufficient for the management of large data bases.

For large case bases, therefore, other means of storing and retrieving cases are needed. As a part of the industrial version of SICStus Prolog, a more powerful database system is being developed. This database system allows more complex forms of indexing, and thus support applications as large-size CBR systems.

### 5.3 Using Parallelism to Improve Efficiency

Adopting the above scheme, it is possible to achieve significant improvements in the efficiency of case retrieval by using parallel execution. Parallel Prolog
systems such as MUSE [2], which is based on SICStus Prolog, provide "transparent parallelism" for e.g., case retrieval. In effect, the findall predicate is executed in parallel, which is really what it needed.

The transparency of parallelism in case retrieval means that a Prolog implementation of a CBR system can directly be executed in e.g., MUSE. Indeed, applications based on CBR are particularly suitable for parallel execution since the major part of computation is a rather straightforward search.

Previous research has shown that parallelism can be used in a very efficient way for case-based reasoning. For instance, memory-based reasoning [7] is a form of case-based reasoning which is run on a massively parallel Connection Machine™.

5.4 Using Constraints in Matching

As a part of the development of SICStus Prolog, the introduction of constraints [4] is being studied. Basically, the relation between unification and constraints can be regarded as 'syntactic' versus 'semantic' equality. Constraints make it is easier to construct more general notions of similarity, and yet be able to achieve an efficient case retrieval mechanism.

Constraint logic programming makes it easier to construct advanced case-based reasoning systems. For instance, it is possible to express equality between different atomic values, or to numerically describe similarity. Again, much of the work that has to be made in the CBR system's program is moved to a level below the programming language.

6 PHOTO — A Simple CBR Application

In order to illustrate the ideas behind case-based reasoning, and how it can be implemented in SICStus Prolog, we have written a simple application, called PHOTO. It is a diagnostic system, based on CBR, which operates in the domain of photography. The program interacts with the user and tries to find out the causes for a bad photograph.

The program is written in SICStus Prolog, and can be run in most available Prolog systems. A listing of the PHOTO program is given in Appendix A.
6.1 Representation

In the PHOTO program, thirteen parameters are used as distinguished features of photographs. The features used, and their possible values, are the following:

1. the available "type of light" (indoors,dark,daylight)
2. the position of the lightsource (behind,in front)
3. the shutter setting (fast,normal,slow)
4. the aperture setting (tight,normal,wide)
5. the use of flash (on,off)
6. the light on the motive (light,normal,dark)
7. the movement of the motive (still,moving)
8. the focus on the motive (sharp,fuzzy)
9. the light on the background (light,normal,dark)
10. the focus on the background (sharp,fuzzy)
11. the type of camera used (standard,af)
12. the type of lens used (wide,normal,tele)
13. the sensitivity of the film used (100,200,400)

In the PHOTO program, all parameters (features) are considered equally important. In a more sophisticated system, some parameters would normally be considered more important than others.

The parameters are represented as a term on the form

\[ t(P_1, P_2, \ldots, P_{13}) \]

In the case base, each parameter is either an atomic value or an uninstantiated variable, which indicates an "irrelevant" value. In the problem to solve, uninstantiated values represent "unknown" values.

The case base in this simple applications only contain 10 cases. In a more sophisticated system, many more cases would be present. Actually, using the above parameters and their ranges of values, a "complete" case base would
contain approximately 9 million cases. A practical case base for this domain would probably consist of around 100 cases.

The cases are represented as three-place unit clauses in SICStus Prolog. Each clause has the following form:

\[ \text{case(Number, Parameters, Conclusion)} \]

where \( \text{Number} \) is an integer that identifies a case, \( \text{Parameters} \) is a term containing the thirteen parameters, and \( \text{Conclusion} \) is a text which is the diagnose that holds for the case.

6.2 System Overview

The PHOTO program can be divided into four major parts:

1. prompting the user for specifying the problem, that is, describe the photo to be diagnosed,

2. retrieving a set of old cases that are similar to the current problem,

3. using the retrieved cases, along with user interaction, to find the most similar case (if this is not accomplished by an exact match), and

4. presenting the most likely diagnose to the user.

The session is started by calling the predicate \( \text{start/0} \). In the first dialog, the user is supposed to enter facts about the current problem. Here, the input should have the form \( \text{motive is dark. etc.} \). It is possible to get some advice by entering \( \text{help.} \). Finally, when the facts have been entered, \( \text{end.} \) is input to start the problem solving process.

The first thing that is done after the facts have been entered is that the case base is consulted and a list of similar cases is retrieved. Each case is given a rank, which signifies the measure of similarity. The list is then ordered so that the case with the highest rank is positioned first in the list.

After the cases have been retrieved, each case is considered in turn for further matching. If the most similar case contains a value for a parameter where the current problem has a variable, then a query is given to the user. The user responds with a value, which might be unknown. After this input has been made, the list of received cases is reconsidered and the ranks are
recomputed; it is possible that some of the retrieved cases are not longer applicable, and it is possible that their relative ordering has changed.

If the interaction results in a "perfect match" between a case and the problem at hand, then conclusion of the old case is given as the solution to the new problem. If the interaction does not result in a perfect match, then the conclusion of the case with the highest rank is given as the most likely solution. Finally, if the problem at hand does not match at all with any of the cases in the case base, then a solution could not be found.

6.3 A Sample Session with PHOTO

STCStus 1.6 #1 : Wed Sep 26 15:24:11 MET DST 1990
Copyright (C) 1988-1990 Swedish Institute of Computer Science.
All rights reserved.
| ?- load(photo).
{loading /tmp_mnt/home/osiris/hagner/photo.ql...}
{/tmp_mnt/home/osiris/hagner/photo.ql loaded, 230 msec 30342 bytes}
yes
| ?- start.

Welcome to the PHOTO Program. I will do my best
to figure out why your photograph did not come
out right.

Enter data>help.

Please describe some of the characteristics of your photograph. You should write as, for example, "motive is fuzzy." The characteristics that you may use and the possible values are the following:
FEATURE        VALUES
light           indoors, dark, daylight
lightsource     behind, infront
shutter         fast, normal, slow
aperture        tight, normal, wide
flash           on, off
motive          light, normal, dark, still, moving, sharp, fuzzy
6.3 A Sample Session with PHOTO

Enter the values of the features you know and terminate by writing "end."

Enter data>motive is fuzzy.

Enter data>motive is moving.

Enter data>camera is standard.

Enter data>end.

What was the type of light?
Available answers are: indoors,dark,daylight, or unknown.
|: daylight.

What was the shutter setting?
Available answers are: fast,normal,slow, or unknown.
|: normal.

What was the aperture setting?
Available answers are: tight,normal,wide, or unknown.
|: unknown.

Is the background sharp or fuzzy?
Available answers are: sharp,fuzzy, or unknown.
|: sharp.

The most likely cause of the trouble is:
Moving motive, use fast shutter.
7 Conclusions

Case-based reasoning (CBR), for some domains, is an important alternative to traditional rule-based techniques in knowledge-based systems. CBR systems are much simpler than rule-based systems, and hence, much easier and cheaper to develop. Moreover, it is possible to achieve incremental improvement in system performance, and to use automated methods in knowledge acquisition.

SICStus Prolog is a suitable language for implementation of knowledge-based systems based on CBR. It is possible to use unification in case retrieval, and to use the database system for management of the case base. Extending SICStus Prolog with parallelism and constraints makes SICStus Prolog even more suitable for CBR systems.

As an illustration, we have presented a small, simple program that uses CBR techniques. The program, called PHOTO, is written in SICStus Prolog. It can be used to diagnose causes behind bad photographs.

Acknowledgements

The authors would like to thank the Swedish Institute of Computer Science (SICS), and in particular Tomas Sjöland and Kent Boortz, for interesting and fruitful discussions. We also would like to thank our colleagues at Infologics, in particular Göran Hagert and Dick Ericson, for important remarks and suggestions. Finally, we express our gratitude to the Swedish Board on Technical Development (STU), Swedish Telecom, Ericsson, and Bofors for funding this research.

References


A  Listing of the PHOTO program

% PHOTO
% This is a simple program which is based on case-based reasoning.
% The program operates in the domain of photography and is able
% to diagnose the reasons why a photograph did not come out right.
% The program is written in SICStus Prolog, version 1.6. It can,
% however, be executed in most Prolog systems using the standard
% Edinburgh syntax.
%
% Written by: Anders Tunevi and Nils Hagner, Infologics AB
% Date: 1990-11-22

:- op(900,xfy,is). % A new operator for convenience

start :- % The main predicate
  greet,
  define_new_problem(NewProb),!,
  retrieve_cases(NewProb,Cases),
  process_problem(NewProb,Cases,Solution),
  present_solution(Solution).

greet :- % Prints out greeting header
  nl,write('Welcome to the PHOTO Program. I will do my best'),
  nl,write('to figure out why your photograph did not come'),
  nl,write('out right.'),nl.

% Definition of the new problem to solve

define_new_problem(Problem) :- % Request facts about the photograph
  EmptyProb = t(_,_,_,_,_,_,_,_,_), % start without params.
  prompt(OldPrompt,'Enter data'), % use a new prompt
  collect_facts(EmptyProb,Problem), % collect params. from user
  prompt(_,OldPrompt). % restore old prompt

collect_facts(PIn,POut) :- % Establish a read loop
  nl,read(Term), % let the user enter a term
  process(Term,PIn,POut). % process that term

process(help,PIn,POut) :- % The user requests help
  nl,write('Please describe some of the characteristics of your'),
  nl,write('photograph. You should write as, for example,'),
nl.write("motive is fuzzy.
The characteristics that you may,
use and the possible values are the following:
'FEATURE VALUES',
lwritelight 'light indoors,dark,daylight',
lwrite'lightsource behind,infront',
lwrite'shutter fast,normal,slow',
lwrite'aperture tight,normal,wide',
lwrite'flash on,off',
lwrite'motive light,normal,dark,still,moving,sharp,fuzzy',
lwrite'background light,normal,dark,sharp,fuzzy',
lwrite'camera standard,af',
lwrite'lens wide,normal,tele',
lwrite'film 100,200,400',
lwritelwritelwritelwriten
nterminate by writing "end."'),nl,
collect facts(PIN,POut).

process(end,PIN,POut).

The specification is complete!

process((P is V),PIN,POut) :- % A parameter has been given
valid_input(P,V,N), % check if input is valid
make_term(N,V,T), % convert it into a term
PIN = T, % unify it with the old term

collect facts(PIN,POut).

process(_,PIN,POut) :- % Catch-all, handles bad input

nlwrite('I am afraid that is not a valid input. Try again.'),

nl,collect_facts(PIN,POut).

valid_input(Par,Val,Pos) :- % check for valid input
parameter(Par,Par,Val,_)%. % should be in the param. list
member(Val,Val). % with an appropriate value

make_term(Par,Val,Term) :- % Construct a "bind term"
build_list(Par,Val,Term), % create "argument list"

Term =.. [Term]. % convert it to a term

build_list(0,_,[]). % Base case

build_list(N,Val,[Val|T]) :- N is 0,1,Val,[Val|T]). % The right position!

N1 is N - 1, % decrease loop counter

build_list(N1,0,..,T).

build_list(N,P,Val,[..|T]) :- % Default recursive case
N1 is N - 1,
P1 is P - 1,

build_list(N1,P1,Val,T).
% Case retrieval section

retrieve_cases(Problem,Cases) :- % Predicate retrieving cases
    findall(s(Case,Rank),match(Problem,Case,Rank),Clist),
    order(Clist,Cases).

match(Problem,Case,Rank) :- % Matching of single case
    case(Case,Params,_,) % retrieve one case
\+ \+ Problem = Params, % unifiable?
    Problem =.. [t|ProbParams], % listify problem parameters
    Params =.. [t|CaseParams], % likewise for the case
    match_parameters(ProbParams,CaseParams,Rank).

match_parameters([P1],[P2],Rank) :- % Base case, one par. left
    match_param(P1,P2,Rank).
match_parameters([P1|T1],[P2|T2],Rank) :- % Recursive case
    match_param(P1,P2,R1), % Match the first one
    match_parameters(T1,T2,R2), % Match the rest
    Rank is R1 + R2. % Sum the ranks

match_param(X,Y,2) :- var(X),var(Y),!.. % var/var match
match_param(X,Y,10) :- X = Y. % value/value match
match_param(X,Y,1) :- var(X),!.. % match with unknown
match_param(X,Y,5) :- var(Y),!.. % match with irrelevant

order([],[]). % order/2 is a "reversed" version
order([X],[X]). % of quicksort that uses the
order([H|T],L) :- % terms s(No,Rank) to create an
    part(H,T,B,S), % ordered list of the cases.
    order(B,R1),
    order(S,S1),
    append(B1,[H|S1],L).

% Inference module (From here on the code is a bit more "messy"...)

process_problem(Prob,\,[],'No solution was found...'). % No cases left!
process_problem(Prob,[Case|Cases],Solution) :- % Otherwise...
    try_case(Prob,Case,Result), % try the first!
    continue_process(Result,Prob,[Case|Cases],Solution).

try_case(Prob,Case,Result) :- % Let's check out this case!
    Prob =.. [t|Pargs], % Split problem into param. list
    Case = s(N,___), % Find the case's number
case(N,P,_) :- % Retrieve the case
P =.. [t|Cargs], % Get parameter list of case
process_parameters(1,Pargs,Cargs,initial,Result), % Check params.

process_parameters(N,[],[],X,X). % No parameters left
process_parameters(N,[H1|T1],[H2|T2],In,Out) :- % Try interaction...
var(H1),nonvar(H2),ask(N,H1), % ...if match says so
process_parameters_2(N,[H1|T1],[H2|T2],In,Out). % continue...

process_parameters(N,[H1|T1],[H2|T2],In,Out) :-
N1 is N + 1,
process_parameters(N1,T1,T2,hit,Out). % hit, process the remaining parameters

process_parameters_2(N,[H1|T1],[H2|T2],hit,Out) :- % A value parameter
nonvar(H1), % needs no further
N1 is N + 1, % processing
process_parameters(N1,T1,T2,hit,Out). % Do the rest!
process_parameters_2(N,[H1|T1],[H2|T2],initial,Out) :- % A (first) value
nonvar(H1), % parameter needs no
N1 is N + 1, % further processing
process_parameters(N1,T1,T2,hit,Out). % Do the rest!
process_parameters_2(N,[H1|T1],[H2|T2],match,Out) :- % Similar to above...
nonvar(H1), %
N1 is N + 1,
process_parameters(N1,T1,T2,match,Out).

process_parameters_2(N,[H1|T1],[H2|T2],_,Out) :- % A variable param
var(H1), % signifies an
N1 is N + 1, % (non-perfect)
process_parameters(N1,T1,T2,match,Out). % match

ask(N,Result) :- % Prompt user for value
parameter(N,_,Vals,Query), % Retrieve pre-stored query
nl,write(‘Available answers are: ’),
write_list(Vals),write(‘ or unknown.’),
nl,read(Reply),process_query(N,Reply,Vals,Result).

process_query(_,unknown,unknown). % Process user input!
process_query(N,A,V,A) :- % Use known

process_query(N,_,V,A) :- % Use unknown
nl,write(‘I am afraid that it is not a valid answer!’),
ask(N,R),
continues % return to the top of the loop

continue_process(hit,_,[Case|_],Solution) :- % Go on checking
Case = s(N, _), % out the cases
    case(N, _, Solution).
continue_process(initial, Prob, [Case|_], Solution) :-
    Case = s(N, _),
    case(N, _, Solution).
continue_process(_, Prob, Cases, Solution) :-
    rematch(Prob, Cases, RevisedCases),
    process_problem(Prob, RevisedCases, Solution).

rematch(Prob, OldC, NewC) :- % rematch is used to do a new
    reunify(Prob, OldC, MidC), % match between the current prob.
    order(MidC, NewC). % and the list of cases.

rematch(_, [], []). % In reunify, the old cases are
reunify(Prob, [H1|T], [H2|Z]) :- % reconsidered, their ranks are
    H = s(N, _), % recomputed and cases may be
    case(N, P, _), % discarded.
    \+ \+ P = Prob,
    Prob = .. [_[Pargs],
    P = .. [_[Cargs],
    match_parameters(Pargs, Cargs, Rank),
    H2 = s(N, Rank).
    reunify(X, [_[T], Z) :-
    reunify(X, T, Z).

% Presentation of solution

present_solution(Solution) :-
    nl, write('The most likely cause of the trouble is:')
    nl, write(Solution), nl.

% parameters are specified by clauses of the form
% parameter(Position, Parameter, Values, Query)

parameter(1, light, [indoors, dark, daylight], 'What was the type of light?').
parameter(2, lightsource, [behind, infront], 'Where did the light come from?').
parameter(3, shutter, [fast, normal, slow], 'What was the shutter setting?').
parameter(4, aperture, [tight, normal, wide], 'What was the aperture setting?').
parameter(5, flash, [on, off], 'Was the flash on or off?').
parameter(6, motive, [light, normal, dark], 'How is the light on the motive?').
parameter(7, motive, [still, moving], 'Was the motive still or moving?').
parameter(8, motive, [sharp, fuzzy], 'Is the motive sharp or fuzzy?').
parameter(9, background, [light, normal, dark], 'How is the background light?').
parameter(10, background, [sharp, fuzzy], 'Is the background sharp or fuzzy?').
parameter(11, camera, [standard, af], 'Is the camera standard or autofocus?').
parameter(12, lens, [wide, normal, tele], 'What type of lens was used?').
parameter(13, film, [100, 200, 400], 'What is the sensitivity of the film?').

% The case base with ten different cases.

case(1, t(indoors,...normal,normal,...fuzzy,...fuzzy,standard,...), 'Lens out of focus').

case(2, t(daylight,inf...off,dark,...sharp,...,...), 'Lightsource in front, use flash').

case(3, t(daylight,...normal,...moving,...fuzzy,...sharp,...standard,...), 'Moving motive, use fast shutter').

case(4, t(indoors,...normal,...on,...light,...,dark,...standard,...), 'Use tighter aperture').

case(5, t(daylight,...normal,...fuzzy,...fuzzy,standard,...tele,...), 'Use fast shutter for stability').

append([H|T], Y, [H|Z]) :- append(T, Y, Z).

member(X, [X|X]).

append([], Y, Y).

append(P, [H|T], [H|B]) :-

P = s(_, X), H = s(_, Y),
Y >= X, part(P, T, B, S).

append(P, [H|T], B, [H|S]) :-
P = s(_, X), H = s(_, Y),
Y < X, part(P, T, B, S).

% Various "standard" predicates

part(_, [], []).
write_list([]).
write_list([H|T]) :-
write(H),write(','),write_list(T).