Machine Learning, 5, 383–406 (1990)

CSM: A Computational Model of Cumulative Learning

HAYONG HARRY ZHOU (zhou@midget.towson.edu)
Department of Computer and Information Sciences, College of Natural and Mathematical Sciences, Towson State University, Towson, MD 21204-7079

Abstract. This paper presents a description and an empirical evaluation of a rule-based, cumulative learning system called CSM (classifier system with memory), tested in the robot navigation domain. The significance of this research is to augment the current model of classifier systems with analogical problem solving capabilities and chunking mechanisms. The present investigation focuses on knowledge acquisition, learning by analogy, and knowledge retention. Experimental results are presented that exhibit forms of intelligent behavior not yet observed in classified systems and expert systems.

Keywords. Long-term memory, cumulative learning, classifier systems, learning by analogy

1. Introduction

The application of expert systems in a variety of domains has achieved considerable success since the mid-1960s. Although impressive and practical, the expert system approach has its own weaknesses and limitations, which make it unsuitable as an adaptive problem solving approach. Generally speaking, expert systems cannot acquire knowledge automatically, they cannot update their knowledge dynamically, and they cannot function intelligently when they reach the edge of their knowledge.

The classifier system model provides a promising approach toward the development of general purpose learning systems (Holland, 1986). By way of comparison, classifier systems have two characteristics which are in sharp contrast with conventional expert systems: automated knowledge acquisition and adaptation. However, the problem of forgetfulness existing in the current classifier systems severely jeopardizes their ability to incrementally improve their performance in an extended period of time.

In response to this problem, we have designed and tested a classifier system with memory (CSM). Given a problem, CSM identifies the problem by searching its memory and attempts to find a solution. If no solution is available due to incomplete knowledge, CSM falls back on an analogical problem solving approach in which similar experience can be recalled, reasoning can be guided, and solutions to new, but similar problems can be constructed. CSM is more robust than conventional expert systems which are unable to solve problems that are different, but similar to the ones for which they were originally programmed. CSM is superior to classical classifier systems in several respects: long-term learning, analogical learning, and the immunity to dramatic changes in environments.

One major motivation of this research is to explore a promising route for automated knowledge acquisition and adaptation. This study concentrates on analogical learning and problem solving, an important approach for the acquisition and effective use of knowledge in AI

systems. Unlike one-shot learning systems which only concentrate on solving a single problem, CSM is capable of preserving problem solving expertise and tailoring it to fit new situations. As a computational model of cumulative learning, CSM is concerned with the transfer of learned knowledge within a domain, and the improvements of its learning performance in the long run. To demonstrate its feasibility and effectiveness, CSM has been evaluated and tested in two distinct domains: robot navigation and letter extrapolation (Zhou, 1987). This paper describes some of the experiments conducted for robot navigation, along with the performance observed.

The main body of this paper is broken down into the following sections: Section 2 introduces classifier systems; Section 3 outlines the major components and organization of CSM; Section 4 describes the robot navigation domain, experiments, and results; and the final section contains conclusions.

2. The structure of classifier systems

Classifier systems are a special class of rule-based systems (Holland & Reitman, 1978; Holland, 1986; Wilson, 1987). Like conventional production systems, knowledge is stored in rules in the IF-THEN form. Unlike conventional production systems, each rule represents its performance by a real number, called its strength. The rules interact with each other through a message list and are stored in a temporary knowledge base called a population, also known as short-term memory (STM). A set of detectors relay external information to the system in the form of messages. The detector messages may trigger eligible rules that in turn generate new messages. A set of effectors take the messages generated by rules and performs corresponding actions. Genetic algorithms which have been proven robust and efficient in a variety of domains (Holland, 1975; De Jong, 1975; Grefenstette, 1988; Zhou & Grefenstette, 1986) act as the main learning algorithm in classifier systems. The bucket brigade algorithm is responsible for assessing rules based on their performance. It is able to assign credits not only to ultimate goal-achieving rules but to stage-setting rules as well.

Taking all these parts together, classifier systems are *message-passing, rule-based, self-organized systems*. Overall, the main cycle of classifier systems can be described as follows:

```
Giving a task to classifier systems
    While (The task has not been solved) Do
    Begin
        Perception phase;
        Message Match phase;
        Competition phase;
        Message Generation phase;
        Response Generation phase;
        Credit Distribution phase;
        Rule Modification phase;
    End
```

In the process of learning, information exchange is accomplished by sending and receiving messages. Whenever the system generates desirable behavior, the involved rules are

rewarded by the bucket brigade algorithm. Otherwise, they are punished by giving negative rewards. As a result, the strengths of rules can be adjusted dynamically based on their performance in the past. If the current rules in the population prove inadequate, the genetic algorithm comes into play. It changes the system's behavior by deleting, modifying and creating rules. Since the size of the population is limited, only good rules can survive, while useless or incorrect rules are subject to replacement. By executing this loop repeatedly, the classifier system conducts an iterative trial-and-error interaction with its environment and learns incrementally.

In the past few years, interest has grown rapidly in classifier systems applied to a variety of domains. For example, Smith developed a classifier system, called LS-1, that demonstrated the ability to rapidly develop a set of strategies from scratch, and to achieve a level of performance comparable to the *level of experienced poker player* (Smith, 1980). Booker used a classifier system as a model of a hypothetical organism, investigating how knowledge could be acquired in a complex and uncertain environment (Booker, 1982). Goldberg designed a classifier system to gas pipeline control. Starting from randomly generated rules, the system constructed rules that were able to control the system under normal summer and winter conditions (Goldberg, 1983).

Although showing impressive performance in many learning domains, the current model of classifier systems would exhibit degraded performance when the environmental changes are dramatic. That is, classifier systems evolve in a Darwinian fashion, using the constraint of a fixed population of rules to cut out the less productive rules in favor of the currently best suited ones. Classifier systems tend to adapt themselves to new situations at the cost of losing inactive, useful rules. In the competition for space in a limited population, newly constructed rules replace existing rules that have been proven incorrect or inactive. The principle of *survival of the fittest* makes it inevitable to replace inactive rules even though they were well learned and may be useful in the future.

Several open questions related to tuning a one-shot learning system into a cumulative system are: how to retain knowledge even if it is unlikely to be accessed again for an extended period of time; how to recognize similar problems; and how to adapt the recalled problem solving expertise. In an attempt to answer these questions, we designed and tested CSM, which will be discussed in the next section.

3. Major components and organization of CSM

Two distinctive features of classifier systems are their ability to acquire knowledge from the outside world and to retain experience for future problem solving in dynamic environments. The environmental changes, however, have been assumed to be smooth and slight. These systems are still brittle to the extent that they are unable to benefit from the obtained problem solving skills when changes are *dramatic*. By dynamically modifying a small set of rules in short term memory (the population), the problem of rapidly forgetting inactive information over time is unavoidable. From the point of view of cumulative learning, this is intolerable. In an effort to lessen this problem, we developed a classifier system with long term memory, called CSM which consists of a Matcher, an Initializer, a Generalizer, and of course, a standard classifier system (Riolo, 1986). The structure of CSM is schematically presented in Figure 1.

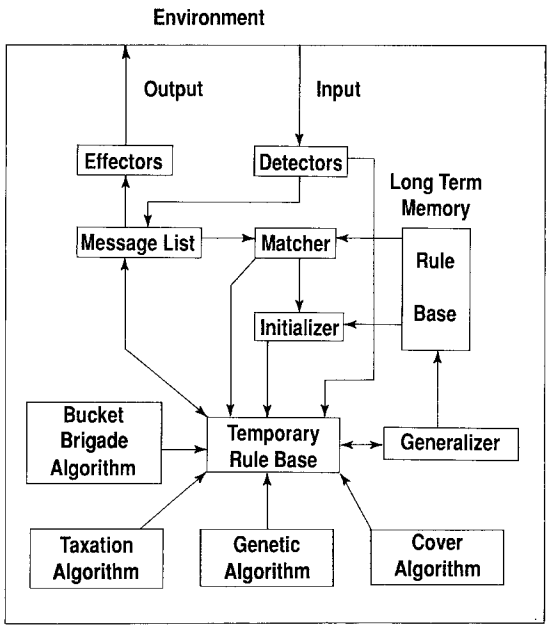

Figure 1. CSM organization.

When given a new problem, CSM tries to recognize the task and recall similar solutions by searching its long term memory for remembered experience. If no relevant past problem solving expertise is applicable, the learning system uses the standard classifier system to solve the problem. After solving the problem, the successful rules leading to final desirable states are generalized, and then transferred from short-term memory to long-term memory as a potentially useful source for future problem solving. By executing this cycle repeatedly, CSM is expected to improve its problem solving ability through experience.

In what follows, we describe four important mechanisms of CSM: the long-term memory, the Matcher, the Initializer, and the Generalizer.

3.1. Long-term memory

A classifier system which stores all of its knowledge in the population can suffer from rapidly forgetting valuable experience. With a limited size, the population is simply unable to retain useful, inactive information. There are two possible ways to lessen the problem of the forgetfulness in classifier systems.

First, we may consider using a much larger population to store knowledge. Unfortunately, this method severely increases the problem solving time. In each cycle, every rule in the population must be scanned and evaluated. Thus, increasing the population size will have the immediate effect of slowing down problem solving. In addition, the population size will still be finite. Each rule in a finite sized population has a nonzero probability of being

displaced. Hence, there is no guarantee that useful information will be retained intact in a dynamically changing environment.

The second way to inhibit forgetfulness is to maintain a separate memory of inactive, well-trained rules. The learning system might retain and preserve previously accumulated knowledge even if it may not be applied or accessed for some time. The trained rules are given a special status in which no deletion or modification is permitted. The approach of grouping relevant rules as chunks and applying them whenever appropriate is similar to the chunking mechanism in Soar (Laird, Rosenbloom & Newell, 1986).

In order to implement this approach, knowledge is stored into two different memories, a short-term memory and a long-term memory. The short-term memory contains temporary information, called active knowledge, which is closely relevant to the current problem. This type of knowledge is subject to evaluation, modification, and even deletion. The long-term memory contains permanent knowledge which differs from active knowledge both in duration and accessibility. The permanent knowledge is a record of successful experiences accumulated over a long period of time. The knowledge in the long-term memory makes it possible for the learning system to transfer its problem solving expertise into a solution for the problem of interest. Usually, the permanent knowledge is not currently accessible to the problem solving process. It acts like a remotely accessible knowledge base whose size and quality determine the degree of intelligence of a learning system. A learning system would have a significant advantage over its competitors if it could preserve and utilize its past problem solving expertise.

Whenever a given task has been solved, a set of rules which contribute to the system's success in problem solving are generalized, transferred, and stored in the long-term memory as a chunk. The basic units in the long-term memory are these chunks, and they are organized into a hierarchy. A chunk is nothing more than a set of related rules representing a solution to a problem, also known as *problem solving expertise*. In such a hierarchy, components of a lower order are abstracted to form a component of a higher order, which may in turn be abstracted to form a component of a still higher order. Thus, the most general concepts are at the top and the most specific concepts are at the bottom.

More specifically, the knowledge in the long-term memory is indexed and organized into domains on the basis of the nature of tasks. A domain contains a set of rules capable of handling a class of tasks. Within each domain, knowledge is further divided into different levels based on their generality.

For the simplicity of discussion, we assume that knowledge is divided into two levels within each domain: task-independent rules and task-specific rules. The differences between these levels are the applicability and duration. Task-independent knowledge can be applied to every task from the same domain, and is preserved in the long-term memory for the longest duration. They are usually obtained by generalizing task-specific knowledge, and serve as building blocks for constructing new solutions. In contrast, task-specific knowledge is applicable only to specific tasks and may be deleted if the long-term memory is loaded beyond its capacity. Task-specific knowledge includes detailed information about a specific task's solution as opposed to the general knowledge of a domain.

Knowledge update is another concern in the design of long-term memory. In the current implementation it only addresses the issue of adding and removing knowledge. Although the capacity of the long-term memory is assumed to be larger, it is still finite and limited.

Whenever the long term memory is loaded beyond its capacity, infrequently accessed knowledge will be replaced by incoming knowledge. The more often a problem solving expertise is retrieved, the less likely it is to be displaced. Recency of access seems to have much less importance than frequency.

In light of this phenomenon, the replacement algorithm is designed to be a combination of least-frequently-used strategy and least-recently-used strategy. When there is no more room in the long-term memory, the knowledge chunk that is least frequently referenced is chosen to be replaced. Each knowledge chunk has an associated frequency counter for the number of times it has been retrieved within a certain period of time. The frequency counter indicates the reference intensity for a knowledge chunk.

Tie-breaking is resolved by a least-recently-used strategy. The chunk that has not been used for the longest period of time is replaced. Thus, an age parameter is also needed for each knowledge chunk. The value of the age parameter increases over time, but decreases whenever the associated chunk is accessed.

3.2. Matcher

The function of the Matcher mechanism is to select from long-term memory the solution of a past problem which is similar to the one at hand. Recognizing the similarity shared by two problems is the most critical aspect of learning by analogy.

The Matcher provides a computer implementation of the *reminding* process (Hofstadter, 1985). Reminding involves judging the similarity or dissimilarity of events and locating the most relevant problem solving expertise from memory. As mentioned earlier, detector messages representing the current state of the external world are sent to the message list every time step. It follows that comparing the messages on the message list with chunks stored in memory would provide clues for the set of rules most suited to the current situation. Because executing the genetic algorithm consumes a major portion of the time needed to solve a problem, the cost of searching for similar problems can be easily justified.

A partial match algorithm was described by Booker (Booker, 1985), but here a new match score computation algorithm is needed for the multi-level knowledge representation scheme employed in CSM. To present the Matcher algorithm in a precise and understandable way, we define the following terms. First, let *LTM* stand for *long-term memory*, *ML* for *message list*, *C* for *conditions* and *M* for *messages*. Then the following expressions hold:

$$
\begin{aligned}
LTM &:= \langle\ \text{Chunk}_1\ \rangle\ \langle\ \text{Chunk}_2\ \rangle\ \ldots\ldots\ \langle\ \text{Chunk}_x\ \rangle \\
ML &:= \langle\ M_1\ \rangle\ \langle\ M_2\ \rangle\ \ldots\ldots\ \langle\ M_y\ \rangle \\
\text{Chunk} &:= \langle\ \text{rule}_1\ \rangle\ \langle\ \text{rule}_2\ \rangle\ \ldots\ldots\ \langle\ \text{rule}_k\ \rangle \\
\text{rule} &:= \langle\ C_1\ \rangle\ \langle\ C_2\ \rangle\ \ldots\ldots\ \langle\ C_z\ \rangle\ \Rightarrow\ \langle\ \text{action}\ \rangle \\
C &:= (c_1\ c_2\ \ldots\ c_i\ \ldots\ c_n) \\
M &:= (m_1\ m_2\ \ldots\ m_i\ \ldots\ m_n)
\end{aligned}
$$

where

x is the number of chunks in the long-term memory
y is the number of messages on the message list

k is the number of rules in a chunk
z is the number of conditions in a rule
n is the length of conditions (messages)
c_i is defined over {1,0,#}
m_i is defined over {1,0}

A knowledge chunk consists of a set of rules representing a problem solving strategy. Rules are composed of a condition part and an action part. Messages are specified over the alphabet {0,1}, while conditions are specified over the alphabet {0,1,#}, where # is a *don't care* symbol that can match either 0 or 1. A condition C is said to match a message M if all non-# symbols in C are equal to the values of corresponding positions in M. Formally, a match of messages and conditions is defined as follows:

$$MATCH(C, M) = \forall_i\{(c_i \in C, m_i \in M) \rightarrow [(c_i = m_i) \vee (c_i = \#)]\}$$

Thus, a nonmatch operation is given as:

$$Non\text{-}MATCH(C, M) = \exists_i\{(c_i \in C, m_i \in M) \wedge [(c_i \neq m_i) \wedge (c_i \neq \#)]\}$$

The condition part of a rule usually consists of several conditions. Rules with more than one condition are called *multiple-condition* rules. To compare a multiple-condition rule, the match operation must be applied to all conditions. A multiple-condition rule is matched if all of its conditions are satisfied by at least one of the messages, M, on the message list, ML.

$$MATCH(rule) = \forall C_i\{(C_i \in rule) \rightarrow \exists M_j\,[(M_j \in ML) \wedge MATCH(C_i\ M_j)]\}$$

Similarly, a rule is not matched if one or more conditions are not satisfied.

$$Non\text{-}MATCH(rule) = \exists C_i\{(C_i \in rule) \wedge \forall M_j\,[(M_j \notin ML) \vee Non\text{-}MATCH(C_i\ M_j)]\}$$

If a condition, C, matches one of the messages on the message list, the best match score, $S_{bm}(C)$, is computed as follows:

$$S_{bm}(C) = \sum_{i=1}^{n} \begin{cases} 1 \text{ if } c_i \neq \# \\ 0 \text{ if } c_i = \# \end{cases}$$

where $c_i \in C$.

If a condition, C, fails to match any message on the message list, the partial match score, $S_{pm}(C)$, can be calculated as follows. First, for each message on the message list, compute the number of mismatched bits, Mis-Bit(C, M), which is defined as:

$$Mis\text{-}Bit(C, M) = \sum_{i=1}^{n} \begin{cases} 1 & c_i \neq m_i \wedge c_i \neq \# \\ 0 & otherwise \end{cases}$$

where $c_i \in C$, $m_i \in M$.

Let $k = \min\{\text{Mis-Bit}(C, M_i)\}$ where $M_i \in ML$. Then, the partial match score of C is:

$$S_{pm}(C) = \frac{n - k}{n}$$

where n is the length of conditions.

The partial match scores provide a reasonable determination of similarity without sacrificing generality. The basic idea behind this match score computation algorithm is the prototypicality which assumes the problems in the same class would approximately share the same features. Conditions are ranked according to their relevancy with respect to the messages on the current message list, where relevancy is proportional to the number of matched non-#s in a condition (Booker, 1985). The partial match score of a condition is 1 if it perfectly matches one of the messages on the message list. Otherwise, the partial match score is always less than 1. With these terms defined, we can now proceed to discuss the match scores of rules. The match score of a rule, S(rule), is defined as the product of the nonzero match scores of each of its conditions. Recall that the best match score of a condition would be zero if the condition is full of #s. By excluding zero scores from computation, it implies that these overgeneralized conditions have been dropped.

$$S(rule) = \begin{cases} S_{bm}(rule) = \prod_{i=1}^{z} S_{bm}(C_i) & \text{if } MATCH(rule) \\ S_{pm}(rule) = \prod_{i=1}^{z} S_{pm}(C_i) & otherwise \end{cases}$$

where z is the number of conditions in a rule.

If all the conditions of a rule are satisfied, the match score is represented by $S_{bm}(rule)$. Otherwise, the match score is represented by $S_{pm}(rule)$. Since the value of $S_{pm}(C)$ is in the range [0,1] and $S_{bm}(C)$ is in the range (1,n], the best match score of a rule $S_{bm}(rule)$ will be no less than any of the partial scores. The match score of a knowledge chunk, $S(Chunk)$, consisting of a set of rules is determined by:

$$S(Chunk) = \begin{cases} max[S_{bm}(rule)] & \exists\ rule \in Chunk\ [MATCH(rule)] \\ \sum_{i=1}^{k} S_{pm}(rule_i)/K & otherwise \end{cases}$$

where k is the number of rules in the chunk.

A sequence of rules in a chunk usually have some difference in their conditions, reflecting different states of problem solving. For some messages on the message list, however, it is possible to have more than one rule triggered, which is resolved by their specificity and relative strengths. In order to measure similarity, the following approach is employed.

for the new problem at hand. However, recall that a best match occurs if there is a rule in a chunk that matches a detector message exactly. Since the messages on the message list only reflect one aspect of the problem, the choice is actually made based on the similarity of one observation. In order to select the most relevant knowledge, we may have to take a further step.

The second approach is to measure similarities using the criterion of specificities. As discussed in the previous section, the score of best matches is determined by the number of non-# symbols found in the matched rules. In other words, a rule with a high best match score has a high specificity value and the fact that it has a best match score at all means that it can match a current message exactly. The specificity of a rule actually reflects its relevance to the messages on the message list, and thereby, the task at the hand. We conclude that the magnitude of best match scores estimates the degree of relevance to the nature and property of the current situation. For this reason, the second approach chooses the chunk of rules with highest best match score to seed the initial population. Since the accumulated experience only matches one aspect of the problem (the current message list), it is possible that the past solution cannot be directly applied to the new problem and some transformation is still required, which is accomplished by the genetic algorithm.

Partial Match

A partial match occurs if no rule in the long-term memory can match detector messages exactly. A high partial match score indicates there may be some useful building blocks from which to construct a solution. In the case of partial match, there is no direct applicable knowledge to the problem at hand in the long-term memory. However, by including these plausible hypotheses in the initial population, a new task may be solved more efficiently. Any partial match operation usually produces a large number of different solutions of varying plausibility. The question becomes how to determine which of several partially matched solutions is the most similar to the problem of interest. The objective here is to inject the most closely related solution sequence into the population.

There are several ways to choose more relevant knowledge chunks. First, a threshold on the match scores can be used to determine the similarity. The chunks with match scores above the threshold are considered relevant to the task at hand and are likely to be chosen. Second, one could choose the best **n** chunks from the long-term memory based on their match scores.

In practice, both approaches are constrained by the population size. This research combines these two strategies. There is a threshold, **t**, set in advance. If there are many matched chunks with match scores above **t**, only the first **n** chunks will be chosen to initialize the population. If there are fewer than **n** chunks which have match scores above **t**, we inject all of them into the population. Thereafter, the genetic algorithm is invoked to conduct a heuristic search. However, because the number of rules in a chunk varies from problem to problem, while the size of the populaton is fixed, it is possible to have fewer chunks injected. Basically, this approach provides some heuristically generated building blocks for the system to start with. It is hoped that the search space could be pruned and the computation effort would thereby be reduced.

That is, if there is at least one best match in a chunk, the match score of this chunk, *S*(*chunk*), will take the largest best match score. The idea is to give preference to a chunk with one or more rules which precisely match the current situation since it would likely provide problem solving expertise necessary to solve a given problem. It is reasonable to assume that a chain of rules, beginning at the matched rule, may be able to lead to a solution to the given problem.

If there is no rule that can exactly match the messages on the message list, *S*(*chunk*) is the average of the partial scores of all the rules in this chunk. Each chunk in the long-term memory specifies a situation in which its actions are especially appropriate. To find the chunk of knowledge most suitable in the current situation, the Matcher compares chunks to the detector messages on the message list. The more a chunk matches the new situation, the more likely it is to be useful. By choosing a chunk of rules with a high match score, one hopes to reduce the amount of required work to be done overall by sharply pruning the search size, and to conduct learning by analogy in classifier systems.

3.3. Initializer

A genetic algorithm with a randomly generated population may initially conduct a blind, inefficient search for difficult problems. By choosing and injecting some plausible hypotheses, learning can be guided in promising directions and become more efficient. The function of the Initializer is to form such initial populations for CSM. The obvious source of plausible rules is long-term memory. However, the difficulty the Initializer must face is selecting a small set of useful seed rules.

In order to achieve this, the Initializer must make a compromise in deciding how to identify and select problems from a large number of candidates. In the following paragraphs we discuss how the Initializer works. The discussion is divided into three parts based on the outcomes of the Matcher: best match, partial match, and no match.

Best Match

In this section we consider how to choose the most relevant chunk of rules if there are many chunks with *best* match scores. In the process of heuristic initialization, we do not want to heavily bias the population by inserting too many rules from long-term memory, for the search might be led into wrong directions. On the other hand, we do not want to start the learning process from scratch every time. Therefore, only a small portion of the population is used to store heuristically injected rules and the rest of the population is generated randomly. Thus, if there is more than one chunk which can match the current message list, the learning system must carefully select the most relevant chunk(s) of rules to initialize the population. A choice often must be made between matched chunks. For simplicity, we will consider in this paper how to select only one chunk of rules from the matched chunks to initialize the population. For solving this problem there are two alternatives.

The first approach is to randomly pick a chunk of rules from the matched chunks. If several matched chunks have best match scores and can recognize the current problem, it seems reasonable to assume that any of them can bring in helpful problem solving expertise

It is worth pointing out that the injected rules only compose a small portion of the population. The Initializer will randomly generate rules to fill the remainder. The diversity of the population is thereby maintained.

No Match

The low partial match scores of all the chunks stored in the long-term memory indicate that there is little applicable experience available. If no matches are found, indicated by very low match scores, CSM randomly generates an entire population and starts the search from scratch. Even if there is no relevant problem solving expertise to draw upon, the performance of CSM should be still comparable to the current classifier systems.

When sufficient knowledge has been accumulated and generalized, a set of *task-independent* rules will be constructed and stored in the long-term memory. Thus, the Initializer is able to seed the population with these building blocks. As a result, CSM will eventually outperform the current classifier systems even in the absence of similar experience.

3.4. Generalizer

Storing all solutions and problem descriptions in the long-term memory is impractical. Hence, how to store accumulated knowledge efficiently and economically is vital to the design of long-term memories. It is suggested that individual rules that share common elements should be generalized, and the rules that are totally subsumed by the more general rules should be permanently masked or deleted (Carbonell, 1983).

In this section, we discuss a mechanism by which general knowledge can be constructed. The Generalizer is a knowledge generalization mechanism extracting the common features from a set of relevant rules. It creates rules in more general forms. From the viewpoint of the genetic search, the Generalizer provides a simple yet effective way of locating essential building blocks from which new, hopefully better, rules can be constructed.

In order to construct general knowledge from specific instances, there are two alternatives for designing a Generalizer. The first approach is to extract features shared by every rule from a chunk. The idea is to locate building blocks common to this chunk of knowledge. It seems reasonable to assume that each strategy should have its own characteristics.

By extracting these features, general rules can be constructed. The argument against this approach is that many rules are not linearly separable in a complex domain. In other words, in the same i position, different rules may have different values. For example, rule 1 may have a "1" in the first position while rule 2 may require a "0" in the same position. By simply extracting common features, it would end up having many *do not care* symbols in generalized rules.

In many applications only information in certain fields counts for similarity. Following this idea, certain fields of rules are considered key fields. We can then place rules into different categories according to their key values. By grouping rules with same key values, general rules can be constructed for each category.

In this study, the second approach is the one implemented and tested. Generalized rules are created for each category, not for an entire chunk. In brief, and very roughly, the Generalizer algorithm can be outlined as follows.

1. *Collection Phase.* Identify and collect the rules which led the system to goal states.
2. *Generalization Phase.* Perform the intersection operation (discussed below) on each category of the rules with the same pattern in the key fields to construct generalized rules.
3. *Confirmation Phase.* Confirm the correctness of the generalized rules through competition and identify redundant rules.
4. *Retention Phase.* Store the winning rules in the long-term memory and discard incorrectly generalized rules and redundant rules.

The Generalizer is invoked after a given task has been solved. The purpose of the collection phase is to identify the rules coupled into solution sequences leading to positive rewards. In this phase, the genetic algorithm is turned off. The problem is rerun from the beginning and the bucket brigade algorithm rapidly rewards, and thereby identifies, the rules that led to system goals. Once these rules are found, the generalization phase starts. The intersection operator discussed below is applied to rules with same value in a key field. By choosing a set of rules, called the category G, with the same pattern in their key fields, a generalized rule can be constructed by the intersection operation.

In the robot navigation domain, for example, the intersection operation is applied among rules with the same vision pattern. The heuristic is that the rules matching the same vision information may lead to the same action, and therefore, may have common patterns in their structures. The intersection operation is performed as follows: if all the rules in the category G have the same value at a given position, the corresponding position of a generalized rule is set to that value. Otherwise, the generalized rule gets a #. Notice that when checking for consistent values, #s are not considered. Formally, we describe the intersection operation as follows:

Let g_i = value of position i in a generalized rule
a_i^j = value of position i in rule j of the category G,

then

$$g_i = 1 \textit{ iff } \forall\, j(a_i^j = 1 \vee a_i^j = \#) \wedge \exists\, j(a_i^j = 1)$$
$$g_i = 0 \textit{ iff } \forall\, j(a_i^j = 0 \vee a_i^j = \#) \wedge \exists\, j(a_i^j = 0)$$
$$g_i = \# \quad \textit{otherwise}$$

The intersection operation is capable of locating essential building blocks. Theoretically speaking, the genetic algorithm could eventually construct such building blocks. However, empirical results indicate that the intersection operation significantly reduces the search time necessary to locate these bit patterns. As an operator for generalizing rules, the effect of the intersection operation is to drop some conditions of a rule, and to turn some constants into variables, which are the common approaches used in rule-based learning systems to generalize knowledge (Kodratoff & Canascia, 1986).

More specifically, if all symbols in a condition are changed to #s by means of the intersection operation, we claim that the number of conditions in a rule has been reduced by

is to discover the important building blocks and essential components necessary to characterize problem solving expertise. The confirmation phase identifies erroneously generalized rules and redundant rules. The retention phase picks up all the rules with a strength above a certain threshold from the population and stores them in the long-term memory. Meanwhile, the incorrectly generalized rules and redundant rules are discarded.

Overall, the heuristic intersection operation and the criterion of *Survival of the fittest* fit the general framework and theory of classifier systems quite well.

3.5. Execution cycle

CSM is an extension of the classifier system model that includes mechanisms for analogical and cumulative learning. Overall, the execution cycle of CSM is outlined as follows.

```
Run forever
begin
    Giving a task to CSM
        While (The task has not been solved) Do
        begin
            Perception phase;
            if (cycle = 1) or (no rules can match messages)
            begin
                Analogical Match phase;
                STM (population) initialization phase;
            end
            Message Match phase;
            Competition phase;
            Message Generation phase;
            Response Generation phase;
            Credit Distribution phase;
            Rule Modification phase;
        end
        Knowledge generalization;
        Knowledge transfer from STM to LTM;
        update LTM;
end.
```

In the perception phase, learning tasks are presented to CSM in the form of detector messages. The Matcher determines similarities between an incoming task and previously solved tasks. The Initializer heuristically seeds the population using the most relevant knowledge. By injecting previously learned and relevant rules into the population, the recalled experience is hoped to be transformed to fit the problem at hand. Some messages generated by rules may trigger the effectors which in turn perform external actions. Upon receiving feedback from the environment, the bucket-brigade algorithm adjusts the strengths of the rules accordingly. During the rule modification phase, the genetic algorithm modifies the rules in the population and constructs new rules if necessary.

one, for the condition full of #s can match any message on the message list. Similarly, if a symbol is changed to a #, it can match any value appearing at the corresponding position in a message, which has the same effect as turning a constant into a variable.

In addition to knowledge generalization, the intersection operation also filters unnecessary #s (variables). To illustrate, consider the following example.

string 1	0 0 # 1 # 0
string 2	# 1 # 0 1 #
new string	0 # # # 1 0

Notice that the #s in the first and last positions of String 2 are discarded in favor of the constant 0. The new string constructed by the intersection operation does not inherit some overgeneralized values. As a result, the intersection operation also filters unnecessary #s. Unless contradicted by some other strings, 1s and 0s are prefereable to #s, since they can capture the characteristics of the problem more accurately. Overall, the intersection operation plays an important role in constructing essential building blocks, and locating key components which can then be used in constructing new rules in the process of learning.

The following question needs to be addressed: what happens if the intersection operator produces wrongly generalized rules? It is true that the generalized rules are only likely, not guaranteed, to function well for the problem at hand. Confirmation is needed to discover and remove incorrectly generalized rules.

Although in many other AI systems recovery from bad generalizations is difficult and complex, it is relatively easy in CSM. These new rules constructed by the Generalizer are simply placed in competition with rules already in the population. Notice that this competition takes place only after a given task has been solved. Thus, the newly generalized rules would compete with the rules constructed during the process of learning. With the learning algorithms being turned off, the bucket brigade algorithm rapidly identifies and rewards the rules which make contributions in attaining system goals. The rules coupled into sequences leading to positive rewards will increase their strengths, while irrelevant rules and incorrectly generalized rules will get weaker.

Besides discarding incorrectly generalized rules, this confirmation phase also removes redundant rules which were created by the genetic algorithm in the course of problem solving and are subsumed by newly generalized rules. Since they are able to handle many situations, the generalized rules gradually accumulate high strengths despite the fact that they are overbid initially by more specific rules. Recall that bids are computed *probabilistically* based on the product of specificity and strength. The correctly generalized rules have a non-zero probability of being selected and rewarded. It has been observed that, by involving in many biding competitions, the generalized rules eventually accumulate high strengths and thereby, outperform more specific rules. To replace redundant rules with a generalized rule provides an economic way to manage accumulated knowledge in the long-term memory, and an effective way to locate useful building blocks.

In sum, the collection phase collects rules which made contributions in problem solving. The generalization phase applies the intersection operation in order to tentatively construct general rules. During generalization, some constants may be turned into variables, some conditions may be dropped, and some unnecessary #s may be replaced. The main idea

The above procedure is repeated until the genetic algorithm and the bucket-brigade algorithm adjust the population rule base to the task at hand. After a predetermined level of performance has been achieved on the task, the Generalizer processes the rules learned, and transfers them from the population to LTM. As CSM solves more tasks within the same domain, the knowledge base (LTM) grows incrementally, and CSM improves both its problem solving and learning abilities through experience.

4. An empirical study

This section describes an empirical study of CSM on a simplified robot navigation domain. As discussed by Carbonell (Carbonell, 1983), analogical problem solving consists of recognizing similar tasks, and using accumulated knowledge to construct new solutions to new problems. Carbonell distinguishes between two broad classes of problem solving by analogy: (a) transformational analogy, in which the solution of a previous task is recognized and transformed to apply to later, similar tasks; and (b) derivational analogy, in which the methods or deriving the solution to the earlier tasks are adapted to derive a solution to the later task.

This work concerns transformational analogy, and explores the possibility of adding such a capability to the classifier system model. We extend the current model with the abilities to store solutions as chunks in long-term memory, to match the stored chunks against new problems as they arise, and to solve them analogically.

4.1. The task domain

A two dimensional area in which the simulated robot navigates can be viewed as an undirected graph, where vertices represent intersections and edges represent paths. The detectors allow the robot to sense: its current position (x and y coordinates), its current direction (e.g., North, South-East, etc.), the presence of obstacles or a clear path in each of five directions (-90 to $+90$ degrees, in 45 degree increments). The effectors allow the robot to turn up to 180 degrees and to take a single step forward, as a single action. It is assumed that the robot can only see objects in front of it within one single step. In order to see objects behind it, the robot must make a 180-degree turn. The vision messages representing the nature of adjacent objects use 1s for legal paths and 0s for obstacles.

Based on detector messages and learned navigation strategies, CSM instructs the robot to make moves via the effectors. The robot receives a positive reward for moving to an unexplored position, and a negative reward for attempting to move into an obstacle. The magnitude of rewards is reduced if the robot moves to a previously explored location. Furthermore, the robot is given a much larger reward for reaching a goal. Notice that this reward scheme is a fairly low level of reinforcement. It does not have any idea about the optimal path.

A cycle of robot activities contains three steps: perception, planning and action. The CSM system which controls the robot uses its detectors to sense, its knowledge to plan, and its effectors to make moves. The goal of a navigation task is to directly reach a destination

from a starting place without committing any wrong moves. For each navigation task, the robot is placed at a starting place and requested to move to a destination. Having completed a journey, call a *run*, the robot is positioned back to the starting place to repeat this process until no more wrong moves are committed. The performance is measured by the total number of cycles needed to reach a destination without any wrong moves.

In this research, we tested CSM on a series of robot navigation tasks, evaluating the feasibility of conducting learning by analogy in classifier systems, demonstrating the usefulness of long-term memory, and assessing the improvement of learning efficiency.

4.2. Constructing task-independent rules

The robot navigation tasks, as described in the previous section, would be trivial if CSM knew in advance the basic rules, the criteria and the legal actions of navigation, such as the operators for moving forward and making turns, and the meanings of different vision patterns. To concentrate on the effectiveness of knowledge transfer, and the comparison of CSM and the current model of classifier systems, we did not build in any domain-specific mechanisms, nor did we provide any knowledge. CSM must learn from very simple concepts and basic navigation skills.

The objective of this experiment was to test whether CSM could extract essential components from specific rules and then construct task-independent knowledge. To act appropriately without any human assistance, CSM must first learn how to distinguish paths from obstacles, how to issue appropriate commands to control the robot's moves, and how to walk along straight paths and avoid obstacles. In light of the above, CSM was given 20 navigation tasks which did not have any intersections and dead ends. These tasks had different starting places, different destinations, and different orientations. The average length of these learning tasks was about 30 steps.

In our simulation the robot could take one of the following six actions: left turn, forward-left turn, forward, forward-right turn, right turn, and a 180-degree turn. CSM used six action commands to control the robot's move. Since the robot had a direction account remembering the global direction it was facing, such as North or South, it needed to know how to update the account accordingly for every move it made. For example, if the robot was facing North, and the next move was forward-right, the direction information stored in the account should be updated to North-East. The robot was expected to learn how to adjust its direction account appropriately as it moved around.

None of this was known initially. CSM spent a great deal of time learning the coding of the action commands, accumulating successful experience, and eventually establishing connections between perception and action. The general format of the rules in this study is given as follows:

IF ⟨ direction ⟩
⟨ vision pattern ⟩
⟨ x-coordinate ⟩
⟨ y-coordinate ⟩
THEN ⟨ action command ⟩

Having collected several sets of rules from successfully solved navigation tasks, the Generalizer was applied and was able to turn the fields ⟨direction⟩, ⟨x-coordinate⟩, and ⟨y-coordinate⟩ into variables, and thereby, construct task-independent knowledge. For example, a rule such as

IF (direction is ⟨x⟩)
(there exists a path ahead)
(location is ⟨y, z⟩)
THEN (go forward)

appeared several times in STM, with various values for x, y, and z. Applications of the intersection operator during the generalization phase resulted in dropping the conditions on the location field and the direction field. The result was a task-independent rule such as

IF (there exists a path ahead)
THEN (go forward)

By doing so, a set of general rules (task-independent rules) could be constructed which were applicable to many navigation tasks in this domain. In what follows, we convert some of these general rules to English statements and present them below.

1. IF (a path on the right)
   THEN (make a right turn).
2. IF (a path on the left)
   THEN (make a left turn).
3. IF (a path on the forward-right)
   THEN (make a 45-degree turn and go forward).

The experimental results demonstrated that CSM was able to capture the essential characteristics and extract important components from task-dependent problem solving expertise. The six action commands, such as the action commands *make a left turn* and *go forward*, were examples of *building blocks* which were needed in the construction of many rules.

Given the same 20 tasks to CFS-C, a standard classifier system (Riolo, 1986), however, it was unable to construct these general rules since it kept losing some inactive yet useful rules due to the competition for space in a population with limited size (Zhou, 1987).

4.3. Specialization of task-independent rules

The objective of this experiment was to show whether CSM could transfer the essential building blocks and specialize task-independent rules for the task at hand. In order to conduct cumulative learning, we did not reinitialize the long-term memory for each experiment. Instead, CSM retained the problem solving expertise after a given task had been solved. Prior to the beginning of the experiment described below, CSM contained chunks of knowledge from the result of previous learning, some of which were task-independent rules. By

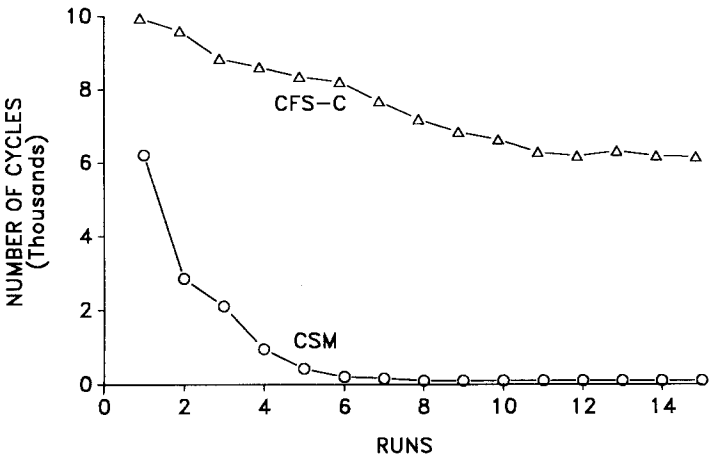

Figure 2. Benefiting from general rules.

replacing some variables and adding more conditions, a general rule could be tailored to fit a particular situation.

The navigation task used in this experiment, called *Task A*, contained 50 intersections and 20 dead ends. On the average, 100 steps were needed to reach a destination from a starting place. The hypothesis to be tested was CSM's ability to benefit from task-independent rules, and to construct new and more specific rules based on these building blocks. The performance curves of CSM and CFS-C were plotted in Figure 2. A cycle represented an iteration in which the system attempted to issue an action command. Since many cycles ended up with unrecognizable commands, the total number of cycles was much larger than the number of moves needed to complete a journey, which was known as a *run*.

The improvement shown when solving *Task A* was attributed to task-independent knowledge which already appeared in LTM as the result of previous experiences (Zhou, 1987). Once such a rule occurred in a LTM chunk, that chunk was more likely to be matched, given the way match scores were computed. This in turn meant that the task-independent knowledge was more likely to be inserted in subsequent STMs. Some of task-independent rules represented the basic skills necessary to navigate in a 2-dimension area. For example, the rule *If there exists a path ahead, then go forward* was used to instruct the robot to walk along a path. By including direction and location information, the general rules could be specialized to guide the robot from a starting place to a destination in a complex environment.

As an example, consider a general rule:

if (a path on the left)
then (take a left turn)

which could be specialized to the following rule:

if (faces South) and
(positioned at (2,3)) and
(a path on the left)
then (make a left turn).

Thus, this newly constructed rule would be triggered only at a certain location when facing South. This kind of knowledge specialization was often used by CSM to guide the robot at intersections. Inspections of the curves showed that, with these building blocks available, it was much more efficient for CSM to construct new rules necessary to solve navigation tasks with intersections.

With more specific rules available, the task-independent rules would not be given preference, which was determined by relevance and specificity. Otherwise, the Matcher of CSM always recognized task-independent knowledge. If there was no specific problem solving expertise available in long-term memory, CSM fell back on general knowledge to tackle a new problem. An important piece of knowledge that got transferred to new situations was the basic navigation skills and background knowledge, which did not initially exist in the system. In the current implementation the THEN part of a rule consisted of 16 bits, forming a total of 2^{16} different patterns. There were only six patterns recognizable as legal operators. Without learning and retaining these operators, it would take a long time for the robot to make a single move. Some other required pieces of knowledge were: how to update the direction account accordingly; what to do when faced with a dead end; and how to move along a path. Once these pieces of knowledge had been acquired and remembered, CSM had ingredients needed to form new rules for more complex learning tasks.

One important observation from this experimental result was CSM's ability to perform *knowledge generalization* as well as *knowledge specialization*. Having solved a given problem, the Generalizer generalized the rules learned and transferred them to long-term memory. During generalization, some specific terms were replaced by variables. Given a new problem, the operator "Crossover" was applied, resulting in the replacement of some variables with task-dependent attributes. In doing so, general rules could be tailored to fit in a particular situation.

By means of knowledge generalization, knowledge could be stored economically and retrieved efficiently. By means of knowledge specialization, important building blocks could be utilized, and essential characteristics could be inherited in the construction of new problem solving strategies.

4.4. Learning increasingly complex tasks

The objective of this experiment was to demonstrate the ability of CSM to benefit from the solutions of simpler problems, and transfer them to solve more difficult problems. At the beginning of this experiment, the long-term memory contained a set of general rules along with about 20 chunks of rules obtained in solving navigation tasks with intersections and dead ends, one of which was Task A as described in the previous section.

The navigation area in this experiment consisted of two regions: Region 1 was similar, but not identical, to the one used in Task A; and Region 2 was totally unknown to CSM.

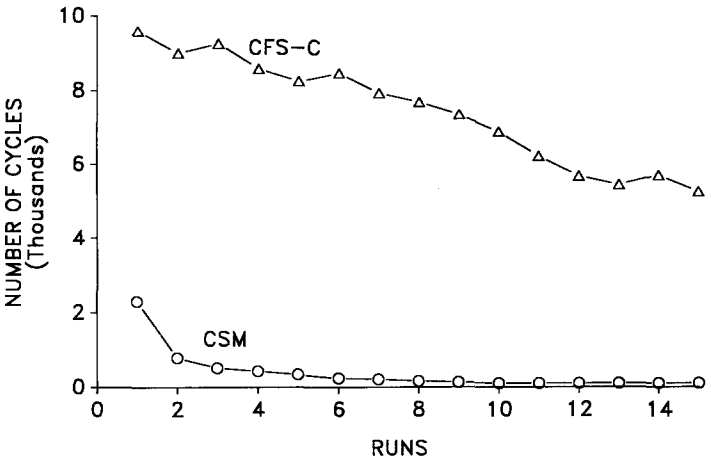

Figure 3. Benefiting from simpler tasks.

The robot was placed in Region 2 and expected to reach a destination in Region 1. This task was designed to show how CSM could solve a more complex task by benefiting from the experience it accumulated in solving simpler tasks, and how it could recall relevant problem solving expertise when the problem suddenly appeared familiar during problem solving. The performance curves of CSM and CFS-C were presented in Figure 3.

Inspections of the performance curves showed that CSM could use its analogical learning ability to solve new tasks that bore some similarity with previously solved tasks. At the beginning of the navigation, the robot was placed in the unknown area. With the accumulated building blocks, CSM still showed much improved performance when compared with the current model of classifier systems.

As described in Section 3.5, CSM invoked the Matcher either at the beginning of problem solving or when no relevant rules were found in short-term memory. This technique reduced the dependence in CSM on superficial similarities between initial problem states, and made it possible for CSM to search LTM from time to time for knowledge relevant to the *current state* of a given task. When the detector messages representing similar situations were posted on the message list, the Matcher matched the corresponding rules stored in LTM, and the Initializer injected them into STM. Thus, the robot could find the destination from that point on. Therefore, CSM needed to explore only the unknown portion of the navigation area. Soon after the robot reached the familiar area, the relevant chunk was matched in LTM and used to guide the robot to the destination. The ability to learn by analogy made it possible for CSM to solve difficult and complex tasks by benefiting from simpler problems, which was an important property of cumulative learning systems.

The experimental results showed that CSM could in fact modify the rules associated with a previously solved task in order to efficiently solve a new and more complex task. More importantly it could refine its knowledge accordingly and gracefully. An analysis of the resulting rules (Zhou, 1987) revealed that learning in CSM was driven by the genetic

algorithm that effectively identified partially correct rules in STM and used these rules as building blocks in the construction of rules applicable to the current task.

4.5. Reversal learning

Reversal learning was concerned with learning behaviors under sudden changes. Two tasks were given alternatively, so were the positive and negative rewards. Initially, learning was slow because of negative effects of reversals. Eventually, each new reversal was expected to be learned with only a single trial if two sets of problem solving expertise could be constructed and retained. The objective of this experiment was to demonstrate how knowledge could be preserved even if it had been proven wrong in a recent situation, and could be recalled when a familiar scenery occurred.

In animal discrimination learning, rats, dolphins, and other animals were subsequently trained on a repeated series of reversals, and showed the ability to solve each new reversal with only a single error (Mackintosh, 1974). Booker tested the classifier system approach in reversal learning without much success. He concluded that *the emphasis on recency and short-term memory in the system is too great* because *by the time the organism had reached criterion on a given reversal, the classifiers learned during the previous reversal were likely to have been deleted—that is, become* 'extinct' *due to the drastic changes in the environment* (Booker, 1982). In the following experiment CSM was evaluated and tested under the similar experimental conditions.

Reversal learning in the robot navigation domain was to go to different destinations alternatively. To make sudden changes, two navigation tasks with different destinations and the same starting place were given alternatively, causing dramatic changes in the environment. A goal in the current learning task would become a wrong destination in the next reversal, and vice versa. The learning subject was expected to realize it, and employed a different navigation strategy it developed in the past when receiving a negative reward. In this experiment, CFS-C and CSM were tested on a series of reversal learning tasks. The performance was measured by the number of wrong moves made from the starting place to the destination. The actual performance curves of CFS-C and CSM were presented in Figure 4.

Inspection of CFS-C's performance showed that the classifier system without LTM suffered from forgetfulness. After having presented different tasks, CFS-C had to construct a set of rules leading to the same destination all over again. The criterion of *survival of fittest*, the competition for space in the population, and low strength of previously learned rules resulting from wrong moves, made it inevitable for these rules to be replaced. Too much information, or information changing too suddenly, overloaded short-term memory beyond its capacity, and resulted in the permanent removal of unsuitable knowledge. CFS-C would have to spend a great deal ot time for solving the same task it experienced before.

In contrast, CSM was able to preserve its problem solving expertise even if it was not relevant to, or even incorrect in, the current situation. Turning its attention to different tasks, CSM simply put inactive knowledge into its LTM, and recalled it if similar vision patterns were posted on the message list. With an original copy of learned knowledge kept intact in long-term memory, CSM was immune to distinct changes in its environment, and robust in the long run.

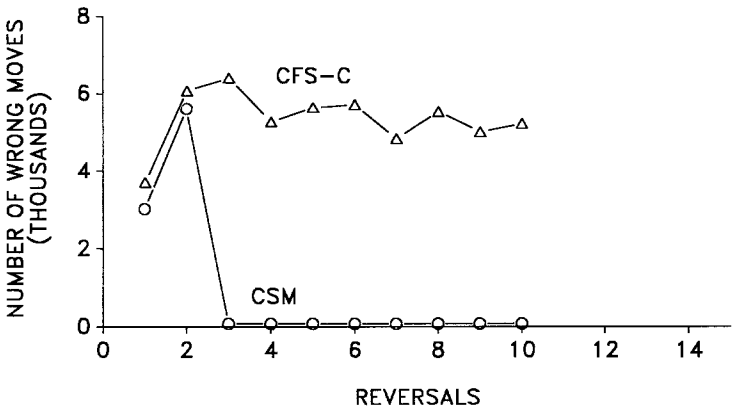

Figure 4. Reversal learning.

The lesson learned from this experiment was the importance of separation of long-term memory and short-term memory (working memory), which made it posible for CSM to preserve its inactive yet useful rules intact, while accessing, modifying, and even deleting tentative rules in short-term memory. Given a problem, CSM recalled its most similar experiences from long-term memory and injected them into the population, which were subject to evaluation, modification, and even deletion. Having solved a problem, CSM transferred the successful problem solving expertise from the population to long-term memory. Thus, CSM accumulated knowledge incrementally and was able to remember, match and apply solutions to recurring problems.

5. Conclusion

This research was motivated by the belief that long-lived AI systems *ought to be undertaken in the same sort of spirit as would a manned landing on Mars—to see if we could do it and learn by doing it* (Nilsson, 1983). This work has made it to the initial *proof-of-concept* stage. In an attempt to turn classifier systems to long-lived, cumulative learning systems, we designed and tested CSM.

This research shows the feasibility and usefulness of incorporating long-term memory into the design of classifier systems. We also augmented classifier systems with the Matcher, the Generalizer, and the Initializer. As a result, CSM is able to transform solutions from one problem to another, and solve subsequent, similar problems in an increasingly efficient and direct manner. With the benefit of prior experience and accumulation of problem solving expertise, CSM constructs its knowledge base incrementally through interaction with it environment and improves its problem solving ability over time.

Experimental evidence shows that CSM exhibits forms of intelligent behavior not yet observed from classical classifier systems. It is worth pointing out that the problem of forgetfulness is often seen in many other adaptive systems aside from classifier systems. Thus,

the question of how to adapt to new sitatuions while still preserving accumulated, inactive knowledge remains an important research topic in a much broader area.

In summary, the development of CSM has made a significant contribution to the field of classifier system research. It also sheds light on some basic AI research issues, such as adaptive expert systems, long-lived learning systems and analogical learning in general-purpose intelligent systems. The work reported in this paper has discussed several important problems in classifier learning systems, has proposed and implemented solutions to these problems, and has demonstrated the usefulness and feasibility of these solutions through the construction of CSM.

Acknowledgments

The author wishes to thank John J. Grefenstette for his advice and encouragement during the development of CSM, and to K.A. De Jong for his comments on an earlier version of this paper.

References

Booker, L.B. (1982). *Intelligent behavior as adaptation to the task environment.* Ph.D. thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI.

Booker, L.B. (1985). Improving the performance of genetic algorithms in classifier systems. *Proceedings of An International Conference on Genetic Algorithms and their Applications* (pp. 80–91). Pittsburgh, PA: Carnegie-Mellon University.

Carbonell, J.G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In *Machine Learning: An artificial intelligence approach* (Vol. 1). San Mateo, CA: Morgan Kaufmann.

De Jong, K.A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Ph.D. thesis. University of Michigan, Ann Arbor, MI.

Goldberg, D.E. (1983). *Computer-aided gas pipeline operation using genetic algorithms and rule learning.* Ph.D. thesis, University of Michigan, Ann Arbor, MI.

Grefenstette, J.J. (1988). Credit assignment in rule discovery system based on genetic algorithms. *Machine Learning, 3,* 225–245.

Hofstadter, D. (1985). Analogies and roles in human and machine thinking. *Metamagical themas.* New York: Basic Books.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: The University of Michigan Press.

Holland, J.H. & Reitman, J.S. (1978). Cognitive systems based on adaptive algorithms, *Pattern-Directed Inference Systems.* New York: Academic Press.

Holland, J.H. (1986). Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. *Machine Learning* (Vol. 2), San Mateo, CA: Morgan Kaufmann.

Kodratoff, Y. & Canascia, J. (1986). Improving the generalizing step in learning. *Machine Learning* (Vol. 2). San Mateo CA: Morgan Kaufmann.

Laird, J.E., Rosenbloom, P.S. & Newell, A. (1986). Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Mackintosh, N.J. (1974). *The psychology of animal learning.* New York: Academic Press.

Nilsson, N.J. (1983). Artificial intelligence prepares for 2001. *The AI Magazine* (winter), 7–14.

Riolo, R.L. (1986). *CFS-C: A package of domain independent subroutines for implementing classifier systems in arbitrary, user-defined environments* (Technical Report). Ann Arbor, MI: University of Michigan, Logic of computer group, Division of computer science and engineering.

Robertson, G.G. & Riolo, R.L. (1988). A tale of two classifier systems. *Machine Learning*, *3*, 139-159.

Smith, S.F. (1980). *A learning system based on genetic adaptive algorithms*. Ph.D. thesis, University of Pittsburgh, Pittsburgh, PA.

Wilson, S.W. (1987). Classifier systems and the animat problem. *Machine Learning*, *2*, 199-228.

Zhou, H.H. & Grefenstette, J.J. (1986). Induction of finite automata by genetic algorithms. *Proceedings of 1986 IEEE International Conference on System, Man and Cybernetics* (pp. 170-174), Atlanta, GA.

Zhou, H.H. (1987). *CSM: A genetic classifier system with memory for learning by analogy*. Ph.D. thesis. Computer Science Department, Vanderbilt University, Nashville, TN.