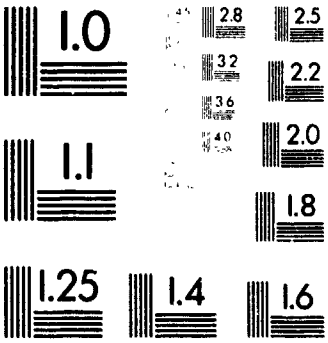


NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

EXTENDING CASE-BASED REASONING USING RULE-BASED AND GENETIC TECHNIQUES

by

Dwight L. Deugo, B.C.S.

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

Dec 13,1989

ISBN 0-315-59017-3

The undersigned hereby recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis,

Extending Case-Based Reasoning Using Rule-Based and Genetic Techniques

submitted by

Dwight D. Deugo. B.C.S.

in partial fulfillment of the requirements for

the degree of Master Of Computer Science

Thesis Supervisor

Chairman, School of Computer Science

Carleton University

Dec 13,1989

ABSTRACT

Human problem solving uses past experiences to solve current problems. When faced with a problem, one often locates experiences (cases) in memory that are similar and adapts them to meet the current situation. The reasoning process that combines these tasks for problem solving is called Case-Based Reasoning. Case-Based Planning is the process of applying Case-Based Reasoning to planning.

This thesis makes three main extensions to Case-Based Reasoning and Case-Based Planning. First, using rule-based techniques, a new model is proposed for the adaptation and repair of cases *during execution,* not only just *before and after* execution. Second, using genetic techniques, a method of learning cases in a noisy environment and controlling the integrity of case memory is proposed; and third, also using genetic techniques, a method for generating novel cases is developed that does not rely on the failures of other cases for the construction of a new case .

The last two methods are combined and illustrated in the domain of a 4x4 checker game. Favorable empirical results, as predicted by genetic theory, have been achieved and are described.

For Sue and Ryan

ACKNOWLEDGEMENTS

Since reminding forms the basis for much of the research presented in this thesis, it is only fitting that an example of reminding is mentioned here. The start, middle, and completion of this thesis reminds me of the arduous tasks that an Olympic long distance runner goes through. To be successful, a runner must have a good trainer, training, and facilities. Fortunately for me, these items were not difficult to find.

At the head of this list was my trainer, advisor, and friend Professor Franz Oppacher. Franz's repeated ideas, discussions, suggestions, instructions, and insights provided me with the focus, determination, and knowledge that every graduate student needs. Both on a personal and professional level, I have accumulated a debt which I am sure I will never be able to repay.

Course work is another integral part of a students' training. Dave Thomas and John Oommen helped provide me with the tools from which part of this work was constructed, and it was from taking their courses as a special student that the initial thought of returning full-time to graduate school was conceived. Their challenging courses and interesting conversations always helped to motivate me and increase my interest in research, for which I am grateful.

The facilities here at Carleton have always been excellent and have never been a roadblock to any of my research. It is both the faculty and staff of the School of Computer Science who I can thank for this. As well as the physical facilities there is also the support staff. For the many times they have helped me in my moments of panic I am always thankful.

Finally I wish to thank my biggest supporter, my wife Sue. Without her patience none of this would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

1 INTRODUCTION

The ability to be reminded of previous experiences allows one to make sense of the world in terms of what one has already experienced. Constantly one is faced with new situations or events - starting graduate school is one such event - and must produce evaluations based on those events; for example, the task of organizing one's time for completing a Master's Degree, months and even years ahead. This is a problem solving activity. Attempting to solve the problem of organizing one's time from scratch - counting the number of days, finding the amount of work required, factoring in sleep, etc - is impossible. However, looking at the time and work required to complete one's Bachelor Degree, gives one a good approximation to the solution of the current problem. One uses past experiences and solutions to problems to make predictions about how events will unfold and how to solve new problems.

The form of reasoning that applies the process of reminding to problem solving is called Case-Based Reasoning (CBR) [Kolodner87, Kolodner88a]. CBR solves problems by looking at solutions to past problems and adapting them to meet the current conditions, and retrieving similar problems to help formulate a better understanding of the problem at hand.

A simplified view of CBR, show in Figure 1.1, contains four components: retrieval, adaptation, evaluation, and storage, interacting with a central structure which is called a case. A case stores related features and actions representing a problem's description and solution. Case memory is a library of cases. The retrieval component locates the most appropriate case in case memory to adapt and solve a given problem. The most appropriate case to retrieve is decided b[illegible]hing features of the current problem against the features stored in individual cases, i[illegible] in case memory, and returning the best matching case. The adaptation component alters the differences between the best case and the current problem to return the best possible solution.

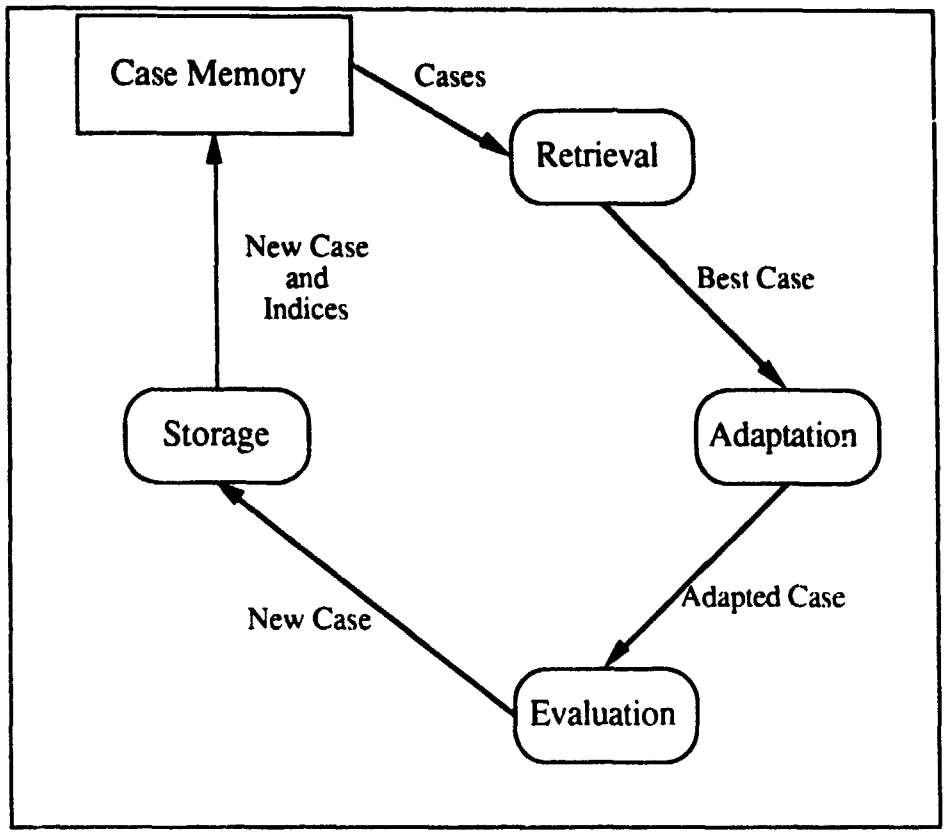


Figure 1.1 Simplified Case-Based Reasoning Components

The evaluation component looks at the adapted case's outcome and performance - was the case's solution good or bad - for the purpose of updating or creating a new case in case memory. The storage component adds the new case with the appropriate indices, made from possible features, so the case can be retrieved in subsequent sessions. Riesbeck's McMOPS system [Riesbeck88, Riesbeck89] captures this simplified view of CBR in a system that has been used to implement two other CBR systems: McCHEF [Hammond86b], a recipe generator; and McDMAP [Riesbeck 89], a natural language analyzer.

Closely related to CBR is Case-Based Planning (CBP). CBP [Hammond86a] is simply planning based on the paradigm of CBR. Since problem solving and planning are closely

related, the goals of CBR and CBP are very similar: both attempt to learn from experience and to reuse what has been learned instead of constructing new solutions from scratch. Just as a CBR system uses its past experiences to interpret new situations, a CBP system uses its past plans, together with information about their failures and fixes, in developing new plans. Repair of a plan is a major concern of CBP. When a plan fails, it is essential that the failure is noted in the plan so that it can be fixed or avoided when using the plan in the future.

The processes of CBR and CBP both use a cycle of case retrieval, adaptation, evaluation, and storage, for learning and adapting of knowledge, in the form of a case or plan, to make it better for subsequent executions of the cycle. Learning and the modification of knowledge in CBR are performed two ways: by adapting a case before its evaluation; or, after its evaluation, repairing a case due to failures, and then storing the case back in case memory.

The primary concern of this thesis is extending the learning capabilities of CBR and CBP. In particular, this thesis presents:

1) A new model for the adaptation and repair of cases *during* execution, not only just *before and after*;
2) A method of learning cases in a noisy environment and controlling the integrity of case memory;
3) A method of generating novel cases that does not rely on the failures of other cases for the construction of a new case.

To learn one must adapt. Adaptation is not something that one does at the end of a day, it is an ongoing process. When an expectation of a event fails, it is explained. When a plan fails, the execution of it should temporarily stop, the failure noted, the plan fixed, and

execution continued. To continue execution is not what is normally done by CBR and CBP systems. Rather than give up when a failure occurs and start the process of case retrieval and adaptation over, as in [Hammond89b], we immediately fix the failure and continue processing. Further evaluation of the planning-case is performed after the plan has finished execution. To do this, the process of adaptation is altered. We view a plan as a collection of expectations about how we are to achieve the goal. Failures of the expectations are also noted in the plan for the purpose of reminding, but they are also made available for the process of learning. By recording exceptions to the expectation, learning occurs because we can use the information about exceptions to try and avoid similar failures in the future and build repair strategies to deal with situations in which similar failures occur.

To learn one must be able to forget. Good cases should remain in case memory for future use. Bad cases should be removed from case memory to prevent their future use. What this means is that CBR must be able to handle noise. Cases are formed and adapted as a result of their interactions with the environment. Noise is likely to appear in cases that interact with a noisy environment. Noise can be a source of harmful, invalid, and redundant cases. The effect of noise is to increase the memory requirements by storing more cases in memory than needed, with the possibility of housing cases that are damaging to the Case-Based Reasoner. Handling the noise and its effects reduces the memory requirements and improves the overall case memory quality by removing the rubbish.

To learn one must be able to introduce novelty. Novelty is a product of a creative imagination, often as a result of a minor change. To produce novel cases, a Case-Based Reasoner must be able to make changes to cases that it already has that, more often than not, produce even better cases. However, there is a price for novelty: it is the creation of bad cases. If the Case-Based Reasoner is able to forget the bad cases, novelty becomes possible. Novelty, noise, and forgetting thus form a partnership that must be tackled as a whole to solve all of the issues.

To perform the extensions noted, this thesis presents methods of combining other paradigms with CBR. In particular, we combine rule-base and CBR techniques to perform the adaptation and repair of planning-cases, and perform a unique unification of CBR and Genetic Algorithms [Goldberg88] to handle noise and produce novel cases. The unification of CBR and Genetic techniques uses the metaphor of *a case is a chromosome string* to produce a remarkably successful method of controlling the integrity of case memory, as well as a directed method of introducing novel cases that are strongly related to existing cases but different enough to enable the possibility of creating even better cases. The marriage of CBR and Genetic Algorithms is a significant result of this thesis.

1.1 ORGANIZATION

This section describes the organization of the thesis and presents its major contributions.

Chapter 2 describes the background of CBR on which the bulk of the thesis rests. The first section of the chapter begins with a discussion of reminding based on Schank's Theory of Dynamic Memory and Reminding [Schank82]. Reminding is a memory process that is used extensively by humans to process and reuse information. Reminding enables one to process and solve problems based on past experiences without full knowledge of how the solution was reached - one just remembers that an answer is a solution to a problem and uses it.

CBR relies heavily on the notion of reminding based on events, plans, and goals. The next section of Chapter 2 elaborates on the differences between the three different forms of reminding, to give an indication of the types of memory structures required for reminding. This leads naturally into a discussion of the different memory structures proposed by Dynamic Memory Theory. The basic memory structures are Scripts, Scenes, Memory Organization Packets, Episodic Memory Organization Packets, and Thematic Organization Points. These structures group memory into a generalization hierarchy of events and goals.

At the top level of the hierarchy are general events that are identified only by an event's goal. As one moves down the hierarchy more and more specifics of the organized events are provided; until the bottom of the hierarchy is reached where full descriptions of events are found. The organization provides a method of relating similar cases; both by features, goals, and by generality. When a specific event can be remembered, it can be used to process the same current event; however, if a specific event cannot be remembered, the remembering of a general or related event can often be adapted to the current situation and used.

The next two sections provide short reviews of CBR and CBP. It is these two methods of reasoning that this thesis is extending, therefore, a general understanding of both is required to fully comprehend the significance of the extensions proposed. CBR and CBP are both reasoning methods based on the ideas of Dynamic Memory Theory, CBP being the specific form of CBR involving planning. To help understand each method, the general algorithm is provided. The basic script like structure of CBR is a case, analogous to a plan in CBP, and is described in detail. Of particular interest to CBR research are the areas of retrieval, indexing, adaptation, and learning, and each is described with examples from the literature. Of particular interest to CBP research are the areas of retrieval, adaptation, learning, repair, and anticipation, which are also described. To complete the sections on CBR and CBP, a summary is given which can be paraphrased as follows:

> Case-Based Reasoning and Case-Based Planning take the obvious notion of learning from experience seriously, and, like their common precursor, Dynamic Memory, they do not view learning as an independent activity but view both reasoning (the application of knowledge) and learning of new knowledge as inseparable aspects of memory processing.

Chapter 3 begins with a general introduction to the six fundamental issues of CBR: indexing, memory organization, retrieval algorithms, case selection (the best case),

matching, and adaptation. Within these six fundamental issues are many open issues, three of which are of concern to this thesis: the mixing of CBR with other paradigms to provide better adaptations of cases, forgetting and validating cases in the case library in a noisy environment for improved learning, and the construction of novel cases. In the remaining sections of Chapter 3, the problems of these open issues, and summaries of proposed methods of solving the problems, are described in more detail.

Chapter 4 introduces our first results on the issue of adaptation and repair of cases and the integration of execution into planning, also described in [Oppacher88, Deugo89a]. The chapter integrates CBP and rule-based techniques using the best of each technique to produce an overall planning approach and structure that can not only acquire new plans and replanning rules by itself and from an expert, but can also induce higher level learned replanning actions that are not immediately apparent to the user or a typical CBP system that is only looking at failures one at a time. The rules are placed in a knowledge base, and, for greater power, inductive methods are applied to them later to increase their overall potential. The cases from which the rules were extracted serve as the precedent situations for the rules. From the knowledge base's point of view, the rules can index the cases. We can search for a special case (or situation) by the rule-based directory. The approach acquires repair actions that can fix the whole plan, not a specific piece of it, so it can be later reused. The result is better adaptations and repairs of cases in an environment where planning and execution are not separate but combined processes.

Included in the chapter are sections on the approach, architectures, planning structures, knowledge base and heuristic structures, and a simple example of the form of replanning rules that can be learned. Of particular interest is the section on *Plan Modification and Repair* which supplies the algorithm for the integration of the techniques described.

Chapter 5 introduces our second major result of learning cases in a noisy environment, and controlling the integrity of the case library, also described in [Deugo89b]. The chapter

uses genetic techniques combined with CRR for producing the result. The majority of the sections in the chapter are dedicated to explaining and using the metaphor of *a case is analogous to a chromosome string*. Before attempting the explanation of the metaphor, a brief introduction on genetic algorithms is given to provide the reader with a base knowledge of genetic techniques. Next, the thesis proposes how the genetic concepts of reproduction and credit apportionment can be achieved with cases, giving rise to the Conservative Reproduction Algorithm; a method of generating a new case library, removing bad cases, and keeping good ones. Using the algorithm, detection of harmful, invalid, and redundant cases is possible, permitting one to learn cases in a noisy environment because unwanted cases can be detected and removed from the case library.

The next part of the chapter describes methods of extending the genetic operators of mutation, crossover, division, and connection to cases. The operators, when applied to cases, lead to the creation of novel cases, described in Chapter 6. Chapter 5 ends with a discussion of why genetic algorithms work, using the proof described in [Goldberg89]. The proof uses genetic strings not cases; however, it gives an indication of what is to be expected if genetic techniques are carried from strings to cases. These results are shown in Chapter 6.

In Chapter 6, by way of an example, the complete genetic process for learning and generation of strong novel cases is molded together. The process has two major steps: case reinforcement and the application of the Conservative Reproduction Algorithm. Case reinforcement - the process of increasing or decreasing a case's strength - is performed by a modified Bucket Brigade Algorithm [Goldberg89] applied to cases. The algorithm uses the success or failure of the application of a case in its judgement to reward or penalize a case's strength. After a period of case reinforcement, the Conservative Reproduction Algorithm, using the genetic case operators of mutation, crossover, division, and connection, reproduces the strong cases into the next generation of the case library and

removes the weak, invalid, and harmful cases. The learning process is thus defined as an infinite cycle of using the cases, reproducing the best ones, and adding new cases to the case library formed from genetic adaptations of existing strong cases. The end result is a library of strong non-redundant cases. The initial sections of the chapter present the structures and approach used for combining genetic techniques and CBR into a learning model. The approach is described by way of example - a 4x4 checker game.

The goal is to produce a system that plays a game of checkers with a human and improves over time. The only piece of domain information the system begins with is what a legal move in a checker game is. This gives plenty of room for the system to learn for itself. Cases are used to represent checker move strategies. The final goal is to show that using only a very small amount of domain knowledge, the system can generate and keep strong new cases for application in a checker game. As a measure of this, we show that the average strength of a case increases as the number of generations of the case library increases. This result, as predicted, is shown graphically in the final section of the chapter. Overall, this thesis presents a combined genetic/CBR model that adds novel cases to its library and removes the bad ones; the library and, thus, the reasoner are getting better - the model is learning.

The thesis concludes with a brief summary of the results and a discussion of some of the open problems in the area in which we have done our research.

2 CASE-BASED REASONING BACKGROUND

In order for any system to claim that it is intelligent, it must be able to perform two operations: one is to acquire new information and store it in a quick, efficient manner; and the second is to be able to quickly retrieve that particular information when it is needed. We are continually overwhelmed with new sensory information, and yet, we are able to process it in an orderly fashion. We determine what the relevant parts of the incoming information are, update our memory with them, and throw the remainder away. Therefore, our memories are dynamic - knowledge structures are continuously modified - not static. Knowledge is not stored and made ready for formatting and insertion at a later time, it is immediately processed - one could say compiled - and our memory structures altered. For this reason, we are not the same person we were one minute ago. We have evolved, at least from a cognitive point of view [Schank82].

Even if one has the most efficient algorithm for processing information into one's memory, the knowledge is of little help if it can not be retrieved (remembered) and used to make inferences from in the current situation. If one could not remember where one's keys are each morning, nor how to get to work, and had to solve these same problems every day, one would never get the chance to solve any new problems because of the overhead of solving the same set of day-to-day problems one is faced with. Some people can only remember where their keys should be, on a key rack perhaps. As long as they put their keys on the rack, they have no trouble finding them. It is when they put their keys somewhere else that a problem occurs. This short example illustrates episodic memory and its use. Episodic memory is memory that stores events based on personal experience. One puts one's keys on the rack at night and picks them up from there in the morning. If one wants to find one's keys, one should go to the rack because that is where the keys were retrieved from before. What one is looking for is an episode whose outcome closely resembles the current situation - finding the keys. To explore episodic memory further, we

must define what the terms *closely* and *similar* mean in the context of retrieval, and what processes use them.

The theory of Dynamic Memory and Reminding [Schank82] provides insight into the meaning of similarity, and the process of reminding which relies on looking for memory structures that are similar to an input stimulus. Reminding is a memory process that is used extensively by humans to process and reuse information. One can be reminded of peoples' names and faces, similar situations, old plans, math problems, and chess strategies. In more general terms, Schank notes that reminding can occur in at least the following different ways.

1. Physical objects can remind you of other physical objects; as a football helmet can remind one of a hockey helmet.
2. Physical objects can remind you of events; as a football can remind you of a particular football game.
3. Events can remind you of physical objects; as a football game can remind you of hotdogs.
4. Events can remind you of events in the same domain; as a game between two teams can remind you of another game between two different teams.
5. Events can remind you of events in different domains; as a football game may remind you of a hockey game.
6. Problems can remind you of solutions.
7. Solutions can remind you of problems.

The types of reminding we are concerned with are 4, 5, 6, and 7. These forms of reminding are concerned with complete events or episodes; for example, a plan, a design, or a checker strategy. Each complete event can be recalled when presented with the identical

tasks to be planned for, the same characteristics to design for, or the same checker board layout and past move history. Reminding is not, however, constrained to finding episodes that match identically. Similar episodes, as well as identical episodes, are all recalled. Sometimes the reminding does not immediately occur. One works on forming a plan and after a while is reminded of a similar plan. One has already begun some processing before the reminding occurs. Remembering one episode while thinking about another indicates that reminding is not a secondary process but is a central process of our thinking. One is reminded of an episode in order to begin processing it using the context of the reminded episode, but in the task of processing that episode one can be reminded of another episode, which one may choose to begin processing or continue with the initially remembered episode.

Human memory can be viewed as a recursive memory reminding machine - an infinite empty loop at birth. Starting with a single reminding, we record new events and how they occur. As time passes, and we record more events, the memory process is increasingly reminded of other events, which cause us to process those remindings rather than encode the new event experience. This in turn causes us to be reminded of other events which we may or may not process, and so on. Schank names this type of reminding: *processing-based reminding*.

A second form of reminding Schank calls *visually-based reminding*. This form of reminding occurs when one recognizes a person's face. Information is bundled into perceptual cues rather than events. Visual information triggers some, if not all, of a single cluster of perceptual cues. If the person's face matches all of the cues, one recognizes the person. If most of the cues are matched, one might say that 'the person looks like another'. If none of the cues match, we ignore the person.

Another form of reminding Schank calls *dictionary-based reminding*. This type of reminding is similar to *processing-based reminding* but is more general in nature. A word

or an event can be used to look into a memory dictionary that has different remindings ready at the various look-up points. For example, the word *mother* brings to mind many different remindings of mothers, just as we would find many different definitions for it in the dictionary under the word *mother* - one is forming a mental lexicon. *Dictionary-based reminding* does have its place for helping one process information, but if one had to rely on constantly looking up information in a dictionary, one would never be able to process the information fast enough. If one was trying to understand English and used only a dictionary, one would be forever parsing words to get the meaning of a sentence. It is not until you can start processing the English words using the context of the sentence that you become fluent in the language. The meaning of the sentence is formed from the current context and expectations that words make about other words following them in a sentence.

This is one reason why the *processing-based reminding* is so important. It is this type of reminding that is the essential part of our human memory process. Without it, we would neither be fast, nor efficient enough in order to process the information that we are given and use from day to day.

2.1 REMINDING BASED ON EVENTS, PLANS, AND GOALS

Before we can talk about the organization and structures used in *processing-based reminding*, it is necessary to look at the different types of remindings[1] that can occur. By doing this, the structures and organization strategies used for reminding may already be brought to mind.

The first type of reminding to look at is reminding based on events and event expectations. One makes assumptions about how events are going to unfold. In young children, assumption making is even more apparent. When a child *goes to bed*, he has a set of expectations about what is going to happen: night clothes are put on, teeth are

[1]Reminding from this point on shall refer to processing-based reminding.

brushed, a kiss given, he lies down, the lights turned off, and the bedroom door shut. Therefore, after the child is laid to rest, the lights are expected to be turned off. If the lights are left on and the door closed, this is a failure in one of the child's expectations (turning off the lights) of the *going to bed* event. In young children, this failure can be quite disturbing - he cries until you come back and turn the light off - but in adults the failure of turning their child's light off is of little concern. Why is this?

In young children, when a failure occurs it is likely the failure has never occurred before. It is not expected. In adults, forgetting to turn the light off is still a failure, but it is an expected failure, meaning that they probably forgot to turn the lights off with another one of their children. The failure has been noted in their structures, but only as an exception to the normal expectation. Children have not built up their exceptions (failures) of an event. From this example, we see that event expectations and exceptions are active and indexed from the same event being processed. However, one is not reminded of all failures of an event because events are processed a step at a time, only reminding of the next expectation and its exceptions is possible unless one forces a look ahead at the expectations of the event. When a failure occurs, we are reminded of similar failures that have happened at the same place in the event. If there are none, the current failure is recorded as an exception to the expectation in order that it can be referenced if it happens again in the future processing of the event.

A second type of reminding is reminding based on goals. In reminding based on events, the reminding helped us to process what was happening now, and what will happen next. Reminding based on goals helps one to be reminded of why an event happened. An event's goal could include its motivation or its purpose. Events that are not similar in expectations, but are, however, similar in goal, can now be used to help one interpret the current event in terms of a related event. The two events are not similar in physical structure, but are similar in a higher level organizational characteristic called a

goal. For example, the reminding of a child who cries when not permitted to play in the mud, and the reminding of an adult who sulks when not given a promotion, are similar remindings, not in event expectations but in goals. Both actions, crying and sulking, are a result of not *obtaining a desire*, one to play in the mud and the other a promotion. Often the phrase, *you are acting like a child*, is heard. The reason for this is that the desired goal of an event and its outcome reminds a person of an event that happened to him as a child with the same goal and outcome. The events are different (obviously because one is not a child now), but the goals are similar and the reminding occurs.

A third type of reminding is reminding based on plans. If goals are used for reminding, then so too must the plans that help obtain the goals. A plan can be considered as an ordered sequence of steps that are taken in order to achieve a goal. A plan, like an event, has expectations about what will occur next and what type of exceptions have occurred with the plan. We view the plan as a collection of expectations about how we are to achieve the goal. Failures about the expectations are also noted in the plan for reminding, but they are also available for another process. That is the process of learning. By recording exceptions to the expectation, learning is occurring because we can now use the information about exceptions to try and avoid them in the future. The fact that we are reminded of them makes this possible.

Two other types of remindings are reminding based on morals and intentional reminding. Reminding based on morals is quite similar to reminding based on goals in both concept and purpose. It is more a difference of generality. Morals are at a more general level than goals, morals dealing with behavior and goals dealing with purpose. Rather than being reminded by processing an event, plan, goal, etc, intentional reminding forces one to recall a relevant past experience. When the normal mechanisms of processing-based reminding fail, one can force the issue by exploring memory for other events that could not be initially indexed. This is done by changing the indexing scheme or indices, possibly by

faking an anomaly, in order to trigger a reminding. For example, if one is asked to remember the purchase of their first new car, intentional reminding is used to return the event. However, when one is asked to remember the purchase of their second car, one is often reminded of one's first car purchase. The reminding of the first car purchase is a result of normal *processing-based reminding*. *Intentional reminding* directs one to search and be reminded of a specific event.

Using the fact that events and plans can have goals associated with them, and that generalization of events and plans is also possible, reminding across different contexts is possible. It is easy to see that different situations can have the same goals and general expectations. Take for example playing the stock market and gambling at a casino. The way one gambles at a casino may be useful when one plays the stock market. These are different contexts, but the underlying features, such as *going for it all*, are the same; therefore, reminding across contexts is possible.

We begin to see a piece of the organization of memory. Events can index other types of events, as seen in event-based reminding. Goals are used to index events and plans, as seen in goal-based and plan-based reminding. Plans can index goals, other plans, and other events. Memory is a connected web of remindings. While we are processing a current reminding, it in turn can remind us of others, depending on what objects, events, goals, plans, expectations, exceptions, or morals are currently in the context of the processing. Generalizations from these objects are also possible. The fact that we have had dinner at a friend's house could help us predict what having dinner at another friend's house would involve. Similarly, if we are at the other friend's house, we may be reminded of having dinner at the first friend's house.

Reminding comes in many forms, but to be effective a common strategy is required to organize and structure the information. Dynamic Memory Theory uses *Thematic Organization Points*, *Memory Organization Packets*, *Episodic Memory Organization*

Packets, *Scenes*, and *Scripts* as the central structures of its organization of memory. These structures help organize memory into multiple hierarchies of events and plans, indexed using goals as the indices. Generalization of the plans and events in the hierarchies is performed in order that when faced with a situation that does not bring about an exact reminding, a more general one[2] is still available for use.

2.2 MEMORY STRUCTURES

In order for memory structures to enable us to learn and understand, they must provide the ability to encode one's experiences in such a manner that they may be retrieved when presented with the same or similar situation in the future. Understanding can be defined as finding the closest structure that enables us to continue processing input information, and learning can be defined as encoding information into the structures. Learning, understanding, and reminding, thus, form the three essential ingredients that any memory structure must account for.

One memory operator that all humans exhibit is the *copy* operator. The *copy* operator provides the ability for one to take an event and record an exact replica in memory. It may not be as apparent in adults, but we often see it in young children. They are always copying their parent's expressions, actions, and movements. To a child, driving a car means getting in the car, turning on the radio, and radically turning the steering wheel This is how the child is interpreting the event *going for a car ride*. It is not the exact way that an adult interprets the event because an adult has undoubtedly more expectations about how one drives a car. The difference is that the adult's version of driving has had more details added by experience. As well as having detailed expectations about driving, the failures of driving (no gas, engine will not start, flat tire) have also been encoded. These exceptions may be remembered by the adult if one occurs, but to the child this is a new experience. The key

2 General in that it does not contain all of the specifics of the current situation, but is close enough so that it could be adapted to meet the current situation.

word here is *experience*. The adult has had previous experiences and added more details, both in expectations and exceptions to the event of *going for a car ride*.

Driving a tractor is not that much different from driving a car. However, this event is a new one to a child. Why is driving a tractor a new event to a child and to the adult it is just a specialization of the general event of driving? The answer is that the adult has the ability to specialize and generalize events. This implies that our memory structures are hierarchical in nature. We encode general events that point to their more specific instantiations. The general event is mucn more useful because it can be used in many different situations, and it points to detailed events with more specific information that one can be reminded of if needed.

In Dynamic Memory Theory, the structure in which one encodes events, either general or specific, is called a script. It is a structure that encodes the expectations and the exceptions of an event. The structures used to organize the scripts into a hierarchy of generalizations and specializations are called Memory Organization Packets (MOPs) and Episodic Memory Organization Packets (EMOPs). A final structure, called Thematic Organization Points (TOPs), is used to organize similar scripts about a goal. We describe each in more detail in the following sections.

2.2.1 Scripts

The basic unit of encoding an event's expectations and exceptions is called a script. Its role is to order sequences of smaller events to form an expectation list of the event. For example, a child's *driving a car* script could be to get in the car, turn on the radio, and continually turn the steering wheel. An adult's *driving a car* script is obviously different. The main points here are that scripts are active structures in memory and they differ from person to person. When we want to drive a car, we are reminded of how we did it before and start doing it again. We are living the script, yet every one drives differently.

A script is of little use if one can not use it to help process different, yet similar, events. If one had to experience every possible event before performing tasks for oneself, one would die before ever having done anything - one can never experience everything in a life time. It is our ability to be reminded of different situations and events and be able to use those experiences, both favorable and unfavorable, to our benefit in the current context. In order to do this, we generalize our scripts so that we can apply and be reminded of them more easily.

Having generalized scripts, one loses some of the specifics of the original script, but its generality and usage have been increased. If the specifics are required, they can still be obtained from the specific instance of the script. However, the specific script may no longer be in memory and we must rely on the general scripts. It is still better to use a general script because its generalization is based on other specific scripts, making it more the norm than the specific instance. The script's expectations and exceptions have occurred over a range of specific scripts making it more stable, consistent, and therefore more useful.

Scripts have the nice property that they continually organize themselves. If a script has seen many exceptions of the same form for a given expectation, most likely the expectation is wrong. By noting the exceptions in the script, the detection of a failure is possible. The script can now take the exception, which is not really an exception any more but rather an expectation, and use it to cause the script to be updated or invalidated. For example, if one wanted to go upstairs in one's house and always turned on the third light switch on the left, but the basement li turned on, one would certainly change one's script so that one found the correct light switch to turn the upstairs' lights on. If one wanted to go upstairs in a friend's house, one might try using the same script to turn the upstairs' lights on.

The important fact here is that there must be some memory structures to connect the general scripts to the specific scripts. It is this function that Schank's scene, MOP, and Kolodner's EMOP structures are used for.

2.2.2 Scenes, MOPs and EMOPs

Scripts provide the mechanism for representing specific memories. Reminding using only scripts alone would not provide us with very much assistance in processing new situations because they are too specific. It is unlikely that the same event will ever occur again. For this reason, a mechanism to provide general scripts is required that can recall the specific scripts. For example, if one had been to a medical doctor's office and never been to a dentist's office, a generalized scene for a doctor's office would allow one to process a visit to the dentist - when you arrive you check in and after a while you see the doctor - even though one had not been to the dentist before. In [Schank77, Schank82] the notion of a scene is presented.

A scene is a general script that organizes specific scripts about it. It contains a generalization of the expectations and exceptions found in the scripts that it organizes. Scenes, likes scripts, are memories. They have expectations and exceptions; the difference is that a scene it is based on a collection of scripts. Scenes are processed in the same manner as scripts. In general, a script is an instantiation of a scene, and a scene a generalization of its collection of scripts.

Scenes organize scripts, but what is used to organize scenes? This task is left for the MOP. The MOP's function is to order scripts and scenes that have a common goal. For example, a person may have acquired scripts and scenes for getting a child up in the morning, feeding a child, playing with a child, putting the child down for a nap, going to the park with the child, having dinner with the child, and putting the child to bed. Each one of the scripts and scenes may be general or specific depending on the amount of experience the person has had with children. If we collect these memories together in the order

presented, we have produced an example of a day-in-the-life of *child rearing*. This is exactly what a MOP does. It collects and orders the scenes with the goal *child-rearing*.

MOPs are different from scripts and scenes in that they are not memories, they are pointers to memories. The memories can be found by looking in the scenes and scripts of the MOP. If one wanted to know about child rearing, one would only have to be reminded of the list of different scenes that are in one's *child-rearing* MOP. If more detail about a particular scene is required, its scripts are used to help fill in the details.

MOPs come in three different types: physical, societal, and personal. Physical MOPs contain mostly scenes that are physical in nature and have a physical setting. Societal MOPs contain scenes that are mostly societal in nature. A societal scene is a scene that has a common social setting. For example, going to dinner with a friend would likely be stored as a societal MOP. The scenes organized by the MOP involve the interactions of going to dinner. The common goal of the MOP might be *having a nice dinner with a friend*. Personal MOPs contain physical, societal, and personal scenes. Personal scenes are similar to societal scenes except the goal of the scene belongs to the person whose scene it is.

We have seen that scenes help to organize scripts into more general structures. At the MOP level, this is also achieved using the structure called a META-MOP. A META-MOP organizes similar MOPs into more general structures. The META-MOP contains only the general structures, but provides pointers to the more specific MOPs. Therefore, a MOP is an instantiation of a META-MOP, in the same way as a script is an instantiation of a scene.

Another structure called an Event-MOP (EMOP) [Kolodner84] is similar to a MOP. The difference between Schank's MOPs and Kolodner's EMOPs is a matter of degree. Schank's MOPs are general structures for organizing scripts and scenes. Kolodner's EMOPs process specific events, rather than scripts and scenes. The purpose of a MOP and a EMOP is the same: to organize related information.

2.2.3 TOPs

The organization hierarchy is not complete without organizing MOPs. This is the TOP's purpose. Using the same argument as for scenes and MOPs, each of these structures is too specific. MOPs are themselves general structures, META-MOPs even more so, however, all of the memories of the scripts and scenes organized are of a common domain. It is in one's ability to reason across domains, apply one solution technique to a different problem, or handling one situation like another, that provides the largest benefits. To do this, more than the sequencing of events, expectations, scenes, or MOPs is required. To be able to recognize that situations are similar, other variables dealing with the MOPs need to be tracked - goals and conditions.

A TOP organizes the MOPs it contains using two extra variables: a goal and a condition. It is the abstract notion of a goal combined with a condition that allows the TOP to organize memories from different domains. For example, suppose I am trying to work on my terminal and a thunder storm stops me from continuing. I could be reminded of the time I was playing golf and the course was too muddy to play on. Both of these events have the goal-condition pair of *attempting an activity: stopped by nature*. The events, scripts, scenes, MOPs are most certainly different, yet I am reminded of both - the solution in each case was to stop the activity. Here lies the power of the TOP. Using the goal and condition pairings (GCP), cross-domain remindings are possible.

Even if a TOP with a GCP cannot be found, locating a TOP whose goal or condition match the current situation could be of benefit. Exploring remindings of similar conditions, but with different goals may still be able to point to memories that have not been organized under the TOP we are currently looking for. The fact that we have been reminded may lead to this reorganization - learning is adapting. The same holds true for finding goals with different conditions. Tracking only the goal may still provide the reminding we are looking for. For example, a TOP whose goal was *attempting an activity* and whose condition was

short of time, could still help me remember possible steps in handling a power failure because the MOPs organized under the TOP (*attempting an activity:short of time*) deal with the consequences of a power failure - I am short of time for completing my work.

2.2.4 Putting the Structures Together

It is the ability to be reminded of past events, situations, and experience that enables humans to make intelligent decisions about their current environment. If we were not able to do this, we would waste much time and energy making, again and again, those decisions that are needed in our current environment. Having the ability to be reminded requires that we use very loose matching criteria for retrieving past experiences, yet only relevant past experiences must be retrieved. Therefore, the conditions of loose, but not too loose retrieval must be met. How is this possible? The answer is using context to aid retrieval.

Context in this case includes the goals, intentions, conditions, other more general properties of the situation, and the normal expectations about the event. Having the context defined in this manner provides two levels of indexing information for reminding. At the first level, general features are considered when reminding takes place. Because these are the most general features, they can be quickly used to reduce the memory space to a limited set of remindings. From there, the second level features, the specifics about the event, can then be used to further instantiate memory, reducing the number of best remindings.

Scripts, scenes, MOPs, META-MOPs, and TOPs form a structure that is based on the human memory abilities of reminding and generalization. The hierarchical structure is shown in Figure 2.1. Specific memories in the hierarchy include scripts and scenes, and generalized structure include MOPs and TOPs. Their expectations and adaptations are stored and provide pointers to other scripts and scenes for further remindings. Although scenes are generalized scripts, these two types of structures deal with more determinate forms of information. Organizers of memories include TOPs, MOPs and META-MOPs.

These structures do not contain any specific memories, but rather bundle similar types of information - may it be another MOP, scene, or script - and form a hierarchy of this knowledge. The hierarchy has more specific structures at each deeper level, each structure organizing its content, forming a generalization hierarchy for reminding.

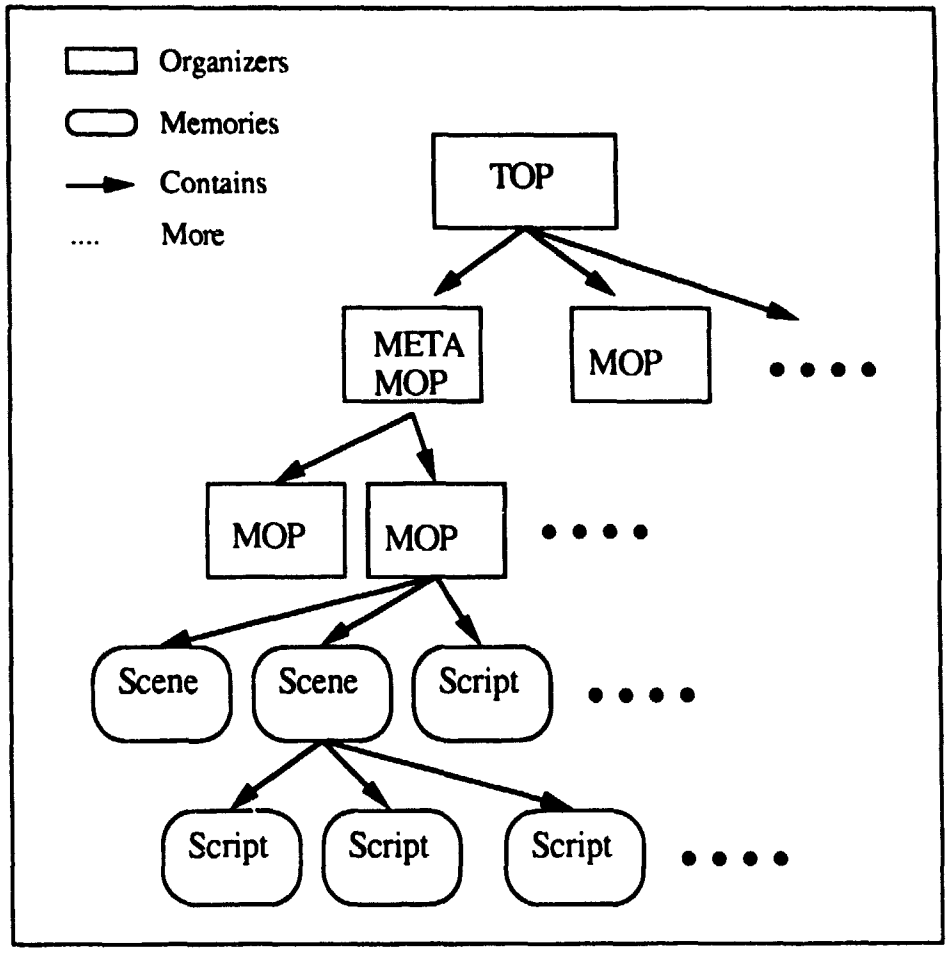


Figure 2.1 Dynamic Memory Structures

This hierarchy is not passive, but active; the key for reminding is context. By continually processing information in the hierarchy, one is led to different levels and structures within it, these in turn lead to other structures, and so on. It is processing from within the structures that give the situations and remindings context. The fact that one is in

some scene dictates what goals are active depending on what TOP was processed to get to the scene. If one has just experienced an expectation, the processing of memory informs one about what will or will not happen next.

Seifert has shown some psychological evidence that TOPs and MOPs affect the process of retrieving cases from memory [Seifert88] using two experiments. Seifert's first experiment involved showing subjects sets of two stories and asking them to classify the stories as similar or different. The results of the experiment showed that similar stories took longer for the subjects to classify as being similar. In the second experiment, subjects were asked to think about the theme of the stories as they were reading them, and then to perform the classification. The results of the experiment showed that similar stories took less time for the subjects to classify as being similar. Seifert concluded that 'reminding depends not so much on similarity-based memory as on the task context'. Reminding depends on the context that is active at the time of retrieval. Goals and generalizations permit one to define context; therefore, as in TOPs and MOPs, are useful for organizing memory.

In Ross's outline [Ross89] of his recent psychological results, the notion that reminding depends on context is reiterated: 'remindings that occur during learning, are influenced by superficial aspects of the task'. Ross goes further to outline the four processes of reminding that are important: noticing, reconstruction, mapping, and generalization. CBR involves feature extraction, retrieval, adaptation, but not necessarily generalization. We see that for the most part, CBR is concerned with the same mechanisms of reminding; thus, it forms a good model of reminding. Although generalization is not included in the mainstream of CBR, this thesis is concerned with it and introduces a method of using it for plan repair.

2.3 CASE-BASED REASONING

Human problem solving is a process that is based on past experiences. When faced with a situation or problem, we locate experiences in memory that are exact matches. That is to say we have faced the situation before and have solved it. By doing this the current problem does not have to be solved from first principles to create an answer. The answer has already been deduced and can be used immediately. This constitutes a large savings in time and allows us to solve problems relatively quickly, providing that we have seen them before.

Take for example the casting designer. Casting design is concerned with many features: general design rules, strain, pattern making, molding, coremaking, physical properties, size, weight, machining allowance, and dimensional stability [Meehanite65]. For people not familiar with the casting designer's mental process, his production of a casting design is similar to black magic. However, for him to reason from first principles would be an incredible, if not impossible task, as it would be for any type of designer; the complexity of the relationships between the different features is boundless. What the designer is actually doing is combining pieces of old designs to meet the constraints and conditions of the new casting. He avoids the problem of going to first principles by using designs that have already been proven to work. His task is to make the modifications of these old designs to meet the current constraints and conditions, still a complex task, but not as complex as going back to first principles; much of the calculations have already been performed.

When we cannot locate a past experience, we can retrieve past situations that are similar. These similar situations are reviewed based on the salient features that match the current situation. The greater the number of matching features, the better the match. The parts of the past solution that rely on those salient features and match the current situation can then be used as part of the solution to the current problem. The remainder of the solution can be found from other past solutions of situations whose feature subsets match

those that we are considering in the current situation. This is a process of adaptation. We take many different past experiences and their solutions and merge them into one that can provide a solution to our current situation. The form of reasoning that combines the above tasks to perform problem solving is called Case-Based Reasoning [Kolodner87, Kolodner88a].

Engineering design of fasteners [Ulrich88] provides a good domain for an example of the CBR process. Fasteners[3] are the connectors of different types of metals, wood, and plastic. When a designer wants to connect two materials, his task is to provide a fastener that will do the job. Once the problem is defined; what the materials are, the dimensions, the strength, and the environment; the designer must complete the design of the fastener under those constraints. He may be able to locate a fastener in his inventory that can perform the job. If he can find one, the design was simple - use a previous fastener - and his task is complete. However, if a exact previous fastener cannot be located, he must design a new one. To do this the current design requirements are looked at. As an example, the requirements are: the fastener must be strong, it must be three inches long, it must not rust, it must have a nut, and it should have thread throughout the entire body of the fastener. The designer locates a past similar fastener from his inventory; a metal hex bolt, four inches long, and has a head throughout the entire body of the fastener; that provides some, but not all of the requirements of his new fastener. This fastener is not an exact fit of the requirements, the material will rust and it is too long, but is close enough to begin designing from. The designer remembers another fastener that was too long by an inch and had its body shortened by an inch to meet its design requirements. So his design is shortened by an inch. He also remembers another fastener design that used a plastic material that was as strong as metal, but did not rust. This new material is also incorporated into his design and completes it. The new fastener is a three inch long plastic hex bolt with

[3] Fasteners include screws, bolts, and rivets.

a full thread. This example shows the process of creating a new design using one similar old design as the main starting point and two other designs that aid in patching the original one in order to meet its deficiencies.

2.3.1 Algorithm

The algorithm for the generic form of CBR is quite simple. The features of the problem - the requirements of a design, the tasks of a plan, or the past moves of a checker game - are given as input to the algorithm. From them, relevant past cases, which we will describe in more detail later, are retrieved that match the feature set. If the feature set of a case and the problem are identical, the solution to the case is returned as the solution to the current problem. If the feature set of a case and the problem are not identical, the case is adapted and modified so that it meets the current requirements. The adaptation is aided by other past cases whose feature sets did not provide a match to the current input set, but whose feature subsets do provide a solution to the partial problem of the input problem. After the case has been adapted and modified, its new solution is returned as the answer to the current problem. Figure 2.2 shows this algorithm in more detail.

The main three components of the algorithm are retrieval, adaptation, and learning, with learning being further decomposed into evaluation and repair. These three components form the backbone of any CBR system and provide the largest challenges and problems for it. The literature has not converged on what mechanism each should perform, but some concepts are starting to emerge. Each is summarized in the following sections, but we now focus on what a case is and what it is used for in CBR.

2.3.2 Cases

A case is the basic unit of storage for a CBR system. It contains a collection of different types of knowledge about an observable event. In its general form, a case could contain the goals, expectations, failures, and features that uniquely distinguish an observable event.

There is no clear cut opinion on exactly what knowledge belongs in a case. The different task requirements of a CBR system place different functional requirements on the cases.

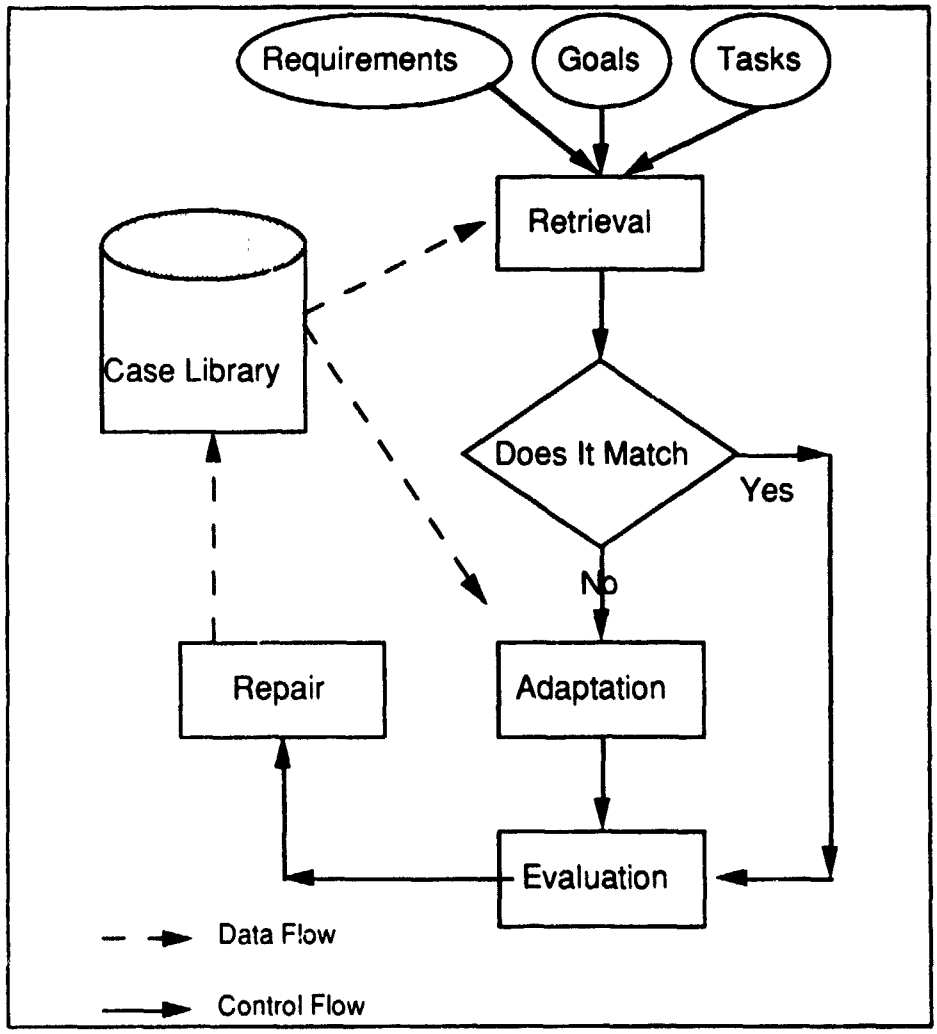


Figure 2.2 Case-Based Reasoning Algorithm

In the fault recovery domain of [Barletta88], cases are used to describe diagnostics of robotic machines. A case stores an initial problem that can be detected by an observer and entered into the system. Once the case is retrieved, it provides a list of diagnostic actions and their expected results that should be performed to help further instantiate the case. If, after the user has performed the diagnostic actions, the reported results match the predicted

results of the case, the problem that is stored in the case is returned as the problem of the reported symptoms. To summarize, this case has a primary observable symptom, a list of actions to perform and their expected results, a problem that is the cause of the observable symptoms, and the results of the diagnostic actions. A case takes the form of a flat list structure, as does the case library.

In the game playing domain of [Bradtke88], cases are used to describe move sequences in the 8-puzzle game. The first part of a case contains the start configuration of an 8-puzzle game; the last part of a case contains the goal configuration of a game; and the intermediate parts of a case contain ordered game moves for proceeding from the start configuration to the goal configuration. Cases in this example form examples of plans with a start, a goal, and intermediate conditions which implicitly describe moves sequences to go from the start to the goal state. Cases of this form contain many subcases[4], and because of this a case or portion of one can be part of other cases in the library. Therefore, cases are formed into a discrimination net so relationships (similar subcases) between the cases can be identified.

In the legal domain of [Branting88], cases are used to describe legal reasoning. Cases consist of two elements. The first element is a set of facts, which define a narrative history of related, sequenced events and any contextual or background information in order to understand the event. The second element is an analysis goal, which specifies the analytical task of the reasoner and consists of a set of relationships whose applicability to some subset of the actors in the narrative is to be defined. Thus, cases are a collection of information about a particular legal case and a method of arguing the case using the information to fill in the explanation.

In legal reasoning it is beneficial to have more than one precedent that can be argued, therefore, cases are organized into exemplars. An exemplar is a heuristically formed case,

[4] Smaller sequences of moves.

compiled from similar cases - a generalized case. The strongest case from which the exemplar was generalized forms the strongest specialization of the exemplar.

In the political analysis domain of [Alterman89], cases are simple structures describing the sequence of a political event. However, rather than adding explanations of the event as part of the case, explanations are used to form a concept coherence network (CCN). The CCN organizes cases into a causal network of relationships about the events of the cases. In this example, not all the information about a case is stored as part of the case. Explanations are used to organize the cases and provide additional information about them.

From these examples it is easy to see that there is no uniform approach for a case's construction or the organization of cases. It is the usage of the information that dictates the organization.

2.3.3 Retrieval and Indexing

With cases having different forms depending on the application, it is not hard to explain why the retrieval methods of cases differ from one CBR system to another. Different forms of cases require different retrieval methods. Putting this issue aside, the real question of retrieval is what constitutes the *best matching* case? Any retrieval algorithm, regardless of the information that it is attempting to retrieve, must attempt to locate the best case.

To compare retrieval methods, Waltz [Waltz89] asks the following questions:

1. Does the algorithm depend on a certain class of features?
2. Does the algorithm generalize across domains? Does it generalize Tasks?
3. Does the algorithm make strong hardware assumptions?
4. Is the algorithm efficient? Does is scale up to large numbers of cases?
5. Is the algorithm cognitively plausible?

Different researchers have proposed different retrieval methods. Hammond's approach to [Hammond88a] case retrieval is based on syntactic features - a common approach to retrieval. Memory is searched to locate the plan[5] that matches as many of the current goals as possible, while avoiding some problems that have been earlier anticipated. The anticipation of these problems could result from the recognition of combinations of goals that produce undesirable interactions. Wall [Wall88] uses concepts that are semantically similar to the current situation as the features to match during retrieval. Although concepts represent domain semantics, retrieval is still based on the syntactic matching of these concepts; therefore, the general approach is the same as Hammond's.

Rather than matching cases organized into a sequential, flat library and matching them against the features provided, Kolodner [Kolodner88b] searches a hierarchical organization of cases. This is a process of concept refinement. Initially, features are broadcast into memory. When a feature activates a portion of a case, it in turn activates other cases deeper in the hierarchy. Cases with high activation levels - those with many activated (matched) features - are returned. The benefit here is that rather than match against every case, a form of marker passing [Hendler88a] is performed with only the highly activated cases returned for further processing

Retrieving cases from memory is a search problem. It gets harder when partial matching is used to retrieve cases that are similar, but not exact. Case refinement and parallel approaches are possible solutions, but using a good indexing strategy will also benefit the retrieval strategy.

2.3.4 Adaptation

When constraints are placed on memory to restrict the number of cases, when the number of physical cases is large, or when case memory is sparse, it is very unlikely that a

[5] A case is known as a plan when CBR is applied to planning.

CBR system (CBRS) will find a case that matches the current input conditions. There is not enough memory or time to find that exact case. Finding a similar case within the memory and time restrictions is, however, possible. It just might be the fact that the case is the best one the CBRS could find. Adaptation of the similar case to the current input conditions is now required. Good adaptation techniques provide a broader coverage of the domain problem space because current cases in the case library can be transformed into new cases not found in the case library. It is a combination of cases and adaptation techniques that determines how good a coverage of the actual problem space CBR provides.

Hammond notes [Hammond89a] three different factors that adaptation techniques depend on. The first factor is the type of task a CBRS processes. CBRS can be usually classified into three classes: planning, explanation, and design. Planning tasks require an execution phase, explanation tasks require a context, and design tasks are more for inspection: each requires and produces different types of results; and use different adaptation methods. A second factor is the the type of knowledge used. Sometimes syntactic adaptations of knowledge are required, while other systems rely on strong domain causal knowledge in their cases to perform the adaptations. The third factor is the level of domain independence. Adaptations of fastener designs are certainly different from adaptations of car designs. One would expect that across domains, little or no adaptation techniques are similar. But this is not the case; for example, to make a car stronger the material could be changed, this could certainly be applied to a fastener. As the following examples demonstrate, domain independence and the other factors play important roles in adaptation schemes .

In the domain of legal reasoning [Sycara88], Sycara uses both heuristics and previous cases to adapt her plans to resolve labor conflicts. Cases are made of labor bargaining plans. Adaptation of a case occurs as a result of three different situations: a case is missing some of the features required in the plan, one of the plan's features has been rejected, or

one of the plan's features needs to be more acceptable. CBR is used to retrieve the initial similar case. The failing features in the returned case are modified by looking at other cases with solutions to the same failing feature. The case is then evaluated by heuristics to decide on its acceptability. If it is not acceptable, the process is repeated until an acceptable plan is found.

Analogy, as described in the Structure-Mapping Engine proposed by [Gentner83], is also used to adapt a case. Analogical reasoning is a method of grasping the abstract common features of two problems in order that one can transfer or map those features from one problem to the other. The process is used in [Shinn88]. Shinn's Abstraction Analogy model uses the mapping of knowledge via common abstractions. To perform transfer, the problem and source cases are formed into an abstraction hierarchy of common features. This is done using generalization techniques, making use of an ISA hierarchy. The solution to the source case is generated by using the common abstractions to fill in the source case's solution. The pieces of the source case's solution that can not be adapted are left for an expert or another analogical process to fill-in.

Analogy differs from similarity. Analogical reasoning deals with the relationships between the structures of the cases, the semantic type information. Similarity reasoning on the other hand deals with the shared attributes of the cases, the syntactic type information, not the abstract relational structure. Analogy also deals with generalization structures, where similarity deals with the cases themselves. A good CBRS will make use of both techniques. By using similarity reasoning, the number of cases that the CBRS is looking at can be reduced. Then the analogical reasoner can use the smaller set of cases to make its mappings.

Building a new case or plan from scratch can be a costly process. By using either parts of existing cases, analogy, or by transformation rules, a good adaptation method can increase the overall potential and coverage of one's case base.

2.3.5 Learning

Learning is the process of adding, removing, and adapting cases in the case library. Old cases that are invalid need to be removed from the library. Cases that have small errors should be updated and placed back into the case library. New cases are added to increase the overall potential of the case library. To learn, a CBRS must perform all three operations.

Adding new cases to the case library is an easy task for the majority of CBRS. The correct index or position is decided upon and the case added. However, determining the correct index can be difficult. A case should be retrieved only at the correct moment. A case should not be retrieved in every situation, nor should it never be retrieved. Therefore, some effort must be put into determining the correct index by thinking about how the case is to be retrieved. However, if the case's content forms the overall index, for example the tasks of the case, the case can be simply added to the case library.

Removing a redundant or invalid case is a difficult task for a CBR system. Kibler's modification to the nearest neighbor algorithm [Kibler88] is one of few methods of discarding bad cases from the library. Computer memory is not infinite. Cases cannot just be put into or left in the case library without thinking of their validity. Retrieval time is affected by the number of cases in the case library; therefore, a good case library has a small number of cases that are primarily good rather than a large number of cases that are half good and half bad.

Modifying a case can be done in a number of ways. The first is by annotations. Annotations add any extra, useful information to the case, like tacking a note to it. The second way is by patching. Patching modifies existing case structures. The final way is by adding failure information - a specific form of annotation. This provides a way of measuring how good the case is performing and to predict what conditions can lead to its

failure in order that the failure can be avoided in the future. We describe these with examples from the literature.

Trucker [Hammond88b], a route scheduling system, uses the method of adding failure information to a case in the form of suspended goals to aid in making opportunistic memory decisions. Trucker is a planner that attempts to avoid failures and recognize opportunities by looking at active and suspended goals of a case. When a plan is executing, its current task or goal as well as others that are to be later executed are reviewed. If the current goal cannot be satisfied, it is suspended. If suspended or future goals can be currently satisfied they are invoked. The suspension and subsequent activation of goals - goal failures and opportunities - in a case are noted by their position in the plan. On subsequent sessions when the case is invoked, the new opportunities are used and the past failures noted by looking into the record of the case's history. This makes for much better planning because the system learns what its cases can and cannot do.

TA [Williams88], a system that learns how to program, uses the method of patching existing case structures. Cases provide examples of different programs that the system has encountered. When a program is executed and an error occurs, a bug is noted. A patch is created by asking the user to identify the offending code piece and the patch for this piece of code. The case in memory is adapted to include the new patch and the patch is also added to a patch library. A patch includes the new and old code. If a bug occurs again, patches might be found that match the offending code and can be used to fix it. Over time, cases improve as they become bug free, and a patch library is built up that can be used to fix new programs.

Hendler's planner [Hendler88b] uses the annotation method for learning. The planner attaches information relevant to the reuse of a plan to it. The annotations provide justifications for the plan. The annotations are used when the plan is reused, aiding in mapping the plan onto different problem situations and contrasting it with other stored

annotations. This is a process of annotation verification. Using the annotations, plan repair and opportunism are possible. Although similar to the failure approach, annotations are not always failure information. An annotation is any information that the planner deems useful in subsequent planning sessions that use the plan.

2.3.6 Summary

CBR takes the obvious notion of learning from experience seriously, and, like its precursor, Dynamic Memory, does not view learning as an independent activity, but views both reasoning (the application of knowledge) and learning of new knowledge as inseparable aspects of memory processing. Instead of treating each problem as if it were brand-new, CBR uses the description of a problem to look for a case in its memory that matches the input description and returns the solution portion of that case. When no exactly matching case can be found, methods such as analogical reasoning [Winston80] are applied in order to return cases that are similar to the input description. These cases are ranked according to their similarity, and the winning case's solution is adapted to meet the current problem. Adapted cases are not thrown away, but are stored as new cases in memory. If the required adaptations are relatively minor, the 'exception information' is added to the existing case.

This form of learning has several advantages. First, the reasoner does not have to reason from first principles every time it has a new problem. Knowledge is implicitly stored in a case; including sequences, failure information, and other causal information relating to the case. This allows a CBRS to avoid past errors and anticipate problems before they occur. The CBRS also takes a short cut in producing the answer to a problem. The answer has already been computed, the CBRS must just find the case with the answer. CBR changes the problem solver's task from deduction to search. Second, the learning is uncomplicated. CBR does not require a deep causal model of the domain it is working with. Causal knowledge is encoded in the cases. Third, a case can easily be used to form

an explanation. A case's slots form explanation patterns, it is trivial to form the explanation from the patterns. Fourth, knowledge acquisition is much easier in CBR. Knowledge is often in the form of a case; therefore, unlike rule-based systems, little debugging is required to determine the interactions between cases. Experts rely heavily on solutions to past cases and can easily recall them. However, it is more difficult to recall the general cases. With these advantages, CBR is a valuable reasoning method in domains where past experience is essential.

2.4 CASE-BASED PLANNING

Case-Based Planning [Hammond86a] is simply planning based on the paradigm of CBR. Since problem solving and planning are closely related, the goals of CBR and CBP are very similar: both attempt to learn from experience and to reuse what has been learned, instead of constructing new solutions from scratch. Just as a CBR system uses its past experiences to interpret new situations, a CBP system uses its past plans, together with information about their failures and fixes, in developing new plans.

A case for a CBP is a plan. A plan is an ordered sequence of actions that when taken will lead the planner from a start state to a goal state. The start state may not be needed by the planner, however, the goal state is the correct outcome of the plan; for example, a correct method to assemble a device or a complete recipe for making a cake.

Plan failure information is incorporated in a case to aid in three forms of learning. The first is learning new plans to solve problems. A plan may experience a failure because the plan is incorrect. Recording failure information can lead to the formation of a new plan that will not produce the same failure. The second is learning features that predict problems. Recording failure information permits the planner to avoid making the same planning mistake by noticing conditions that bring about the failure. The third is learning the repairs that have to be made if those problems arise again. A planner cannot fix a plan unless it knows what is broken. Each form of learning enables plans to improve with experience.

2.4.1 Model

Hammond's view of CBP postulates six components: *The Anticipator*, *The Retriever*, *The Modifier*, *The Storer*, *The Assigner*, and *The Repairer*, illustrated in figure 2.3. Each component is described in more detail in the following sections.

2.4.2 Retrieval and Modification

The *Retriever* functions as an indexing system. It uses goals and abstractions of goals to differentiate plans in memory and retrieve the best one(s). To perform the retrieval, CBP requires three different types of knowledge:

1. A memory of plans indexed by the goals they satisfy.
2. A similarity metric for judging the similarity of goals required for determining partial matches.
3. A value hierarchy of goals to judge the relative utility of plans with respect to to the set of goals.

CBP takes as input a set of goals and locates a plan in memory which satisfies most of the important goals, using the similarity metric and value hierarchy to retrieve the plan with the maximum utility.

A retrieved plan is not likely to be a perfect match of the input goals. The *Modifier* modifies plans whose goals do not fully match the input ones. To do this, CBP requires three different types of knowledge:

1. A set of modification rules.
2. Critics with knowledge of goal specific requirements.
3. The plan's purpose.

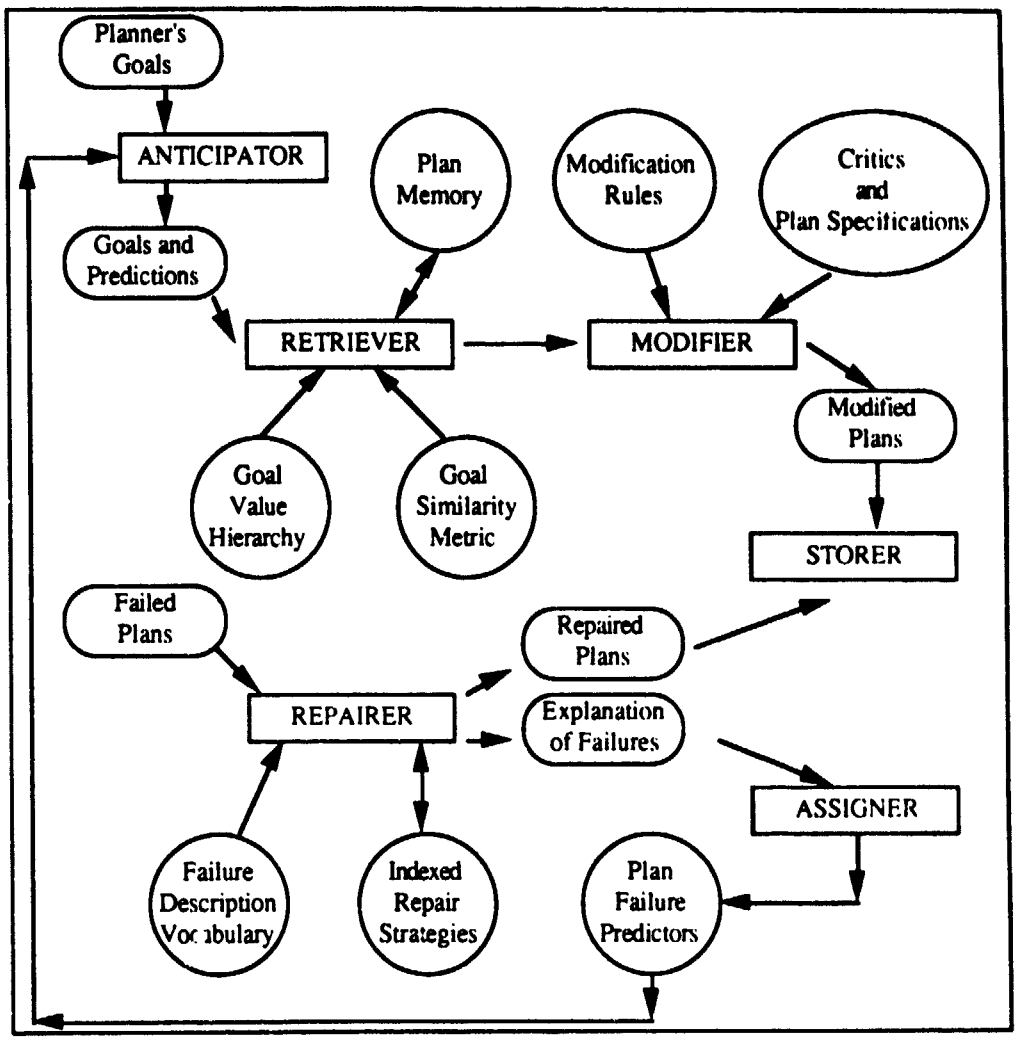


Figure 2.3 Case-Based Planning Architecture

A library of modification rules are used by the CBP planner to make changes to the plan to achieve its missing goals. These rules provide the steps of the changes. Critics provide information about items in the domain that tell the modifier how to change those items to meet the goals. Finally, the *Modifier* requires a plan to modify, and its purpose to direct the modifications.

2.4.3 Learning and Repair

After a plan is modified it is ready for execution and storage. The *Storer* takes the built plan and inserts it into memory for future reuse. The *Storer's* task is not to modify the plan further, but rather determine the appropriate indices in order that the plan can be stored in the correct place in memory for retrieval under the proper conditions in the future. The indices include the goals that are satisfied by the plan and the situations in which they are appropriate.

No matter how good a plan is, it is always subject to failures. The failures can be a result of a mistake in the plan; for example, the plan has become invalid due to a change in its operating environment, or the environment in which the plan is executing has produced an abnormal condition that causes the plan to fail. Whatever the situation, the plan needs to be fixed. This is the task of the *Repairer* component.

To repair a plan, the CBP requires three different types of knowledge:

1. The failed plan.
2. A failure description vocabulary.
3. Indexed repair strategies.

The input to the *Repairer* is the failing plan and a description of the fault. Using the fault description, a description vocabulary is used to deduce possible ways the fault could have occurred. The richer the vocabulary, the greater the number of possible explanations that can be generated. These explanations are used as indexes into repair strategies that have been recorded in other plans or are located in failure rules. After the plan is fixed, it - including its failure explanation and repair method - is returned to the *Storer* for incorporation into the plan library.

2.4.4 Anticipation

Anticipation is the ability to learn from failure. The *Anticipator* figures out when a problem with a plan is likely to occur again and tries to avoid the problem. To do this the *Anticipator* must be able to figure out why the problem happened before in order to alter the plan to avoid those conditions in the future. This task is performed by the *Anticipator* before the plan is executed using the results of the *Assigner*. The *Assigner's* task is to look at a failed plan after its execution and decide what caused the failure and what conditions will produce the failure again. The *Assigner* uses the explanations that were provided by the *Repairer* to predict what the cause of the failure was. The *Anticipator*, using these predictions, determines what failures of the plan are likely to happen under the current conditions and attempts to alter the plan so the failures can be avoided.

2.4.5 Summary

CBP is an effective means of planning when the relationship between the tasks can not always be explicitly declared. The ordering and scheduling of the tasks is implicitly declared in the plans that are built up in Case Memory. When one turns the key in one's car it starts; however, most people are not absolutely sure why. The fact remains, when the key is turned the car starts; and for the average person that is enough. CBP provides a means of generating a plan that meets the requirements of an initial set of tasks and constraints that are to be planned for. Case Memory provides a means of storing failure and adaptation information that is essential for the successful repair of a plan. Since the modification of a retrieved plan to meet the current tasks, and the replanning of a task due to a failure during execution are similar operations, CBP facilitates the integration of replanning into the planning process.

CBP has many advantages over traditional planners. The first advantage deals with Knowledge Acquisition. In many domains for which planners are required, only a small

proportion of the knowledge is in a form that lends itself to the traditional knowledge-engineering exercise of extracting general rules from experts. Much of such knowledge is episodic, i.e. it is easier to solve problems by relating them to cases that embody complete previous episodes than to attempt to formulate general rules. For some domains, which lack causal theories, the expertise may be available only in the form of cases. The knowledge representation of CBP is in the form of episodes rather than of rules, making the process of mapping acquired knowledge from experts to the knowledge representation much easier.

Another advantage of the CBP approach, from the point of view of knowledge acquisition, concerns the cost involved in developing deep causal models or elaborate hypothesis hierarchies. Such theory-based expert systems or planners are only useful for applications with a long, stable life spans because they are expensive to develop and maintain. For domains involving rapidly changing technologies, a CBP approach seems preferable: it enables a gradual adaptation of the system's responses to the changes in its domain, and it facilitates the incremental development of a case library.

The next advantage deals with plan indexing and multiple goals. Traditional planners like Hacker [Sussman75] address the problem of multiple interacting goals by first developing a plan indexed under a single goal and then debugging it with 'critics' when undesirable interactions with other goals occur. For example, Noah may develop several independent plans to fetch a hammer, some nails, a screwdriver, etc. Because of Noah's policy of least commitment, these several plans are simultaneously available to inspection by a critic that may notice that they can be subsumed under one plan. Even Pandora [Wilensky83], which is based on a classification of different types of goal interactions, starts with plans for individual goals. When a simulation of the proposed actions shows possible problems, a meta-goal to remove this type of problem is invoked.

CBP, by contrast, indexes plans by features of the situation that are predictive of goal interactions. As Hammond points out, this has several advantages over the usual 'propose and debug' approach: 1) often the best plan for achieving a single goal is less than optimal when this goal is one of several; 2) past replanning episodes, when retrieved initially, may help anticipate and avoid problems due to goal interactions; 3) features present in the input may predict goal interactions that a planner which indexes plans exclusively by goals could only detect through simulation; and 4) plans indexed by abstract descriptions of goal interactions, such as TOPs, can be used in analogous situations, i.e. in situations that differ from previous ones in the objects and even in the particular goals involved, but share similar goal interactions.

CBP, as described, has many advantages over traditional planners. In addition to these, CBP takes the approach of planning as viewed in humans. We are creatures of habit, but this habit can also cause some problems.

3 CASE-BASED REASONING ISSUES

CBR research is concerned with six fundamental issues: indexing, memory organization, retrieval algorithms, case selection (the best case), matching, and adaptation; each closely related to one or more of the others and all contributing to the predominant issue of learning. As a starting point we begin with case retrieval. Case retrieval is a search problem made even more difficult because the exact identity of the case is unknown. There may be a case in memory that supplies the exact solution to the current problem. Then again, there may be a case that can provide a solution to a similar problem, but not the exact problem of current concern. A retrieval algorithm must be able to find both types of these solutions, preferably the first, but more likely the second. In this case, the retrieval algorithm must make sure that only the relevant similar cases, not the contents of the entire case library, are returned.

The two main approaches to retrieval are concept refinement search methods and parallel retrieval methods. Conceptual refinement [Lebowitz83] relies on memory being organized into a hierarchy, starting with generalized cases at the top of the hierarchy and proceeding down the branches with more specific cases. Search begins at the top of of the hierarchy and proceeds down the branches when the current features of the problem match those features of the case represented by the branch. This process is repeated at each level until no more branches of the hierarchy are available. The deepest node reached represents the best matched case, and intermediate nodes reached represent partially matched cases.

Parallel methods such as [Stanfill86] permit every case in memory to be matched against the input conditions at the same time. Cases with a high degree of match are returned as source cases. This method, however, is just throwing more processing power at a hard problem, it does not really answer the problem.

In order to retrieve cases successfully and efficiently, indices must be created for the retrieval algorithm to work with. The normal approach is to create indices using the relevant

features of the case. For example, JUDGE [Bain85], a program that determines criminal sentences, uses indices derived from a set of features that include the statute of law that was violated, who started the fight, the violated actions and results, and the interpretation assigned to those actions. This approach does not easily lead to the formation of general concepts; it is too detailed in its domain specific environment. A method around this is to use an inductive learning method such as UNIMEM [Lebowitz87] to identify features in the case that are predictive. These features are then used as the indices because they are more predictive of the case.

Indexing and retrieval both rely on a memory organization. This is an organization of the case library and the cases themselves. Often the case library and the cases are flat structures - an ordered list of cases, each case with an ordered list of frames. The process of retrieving a relevant case is a matter of taking the cases one by one and matching each against the current input specifications, retrieving those cases with a good match. Case frames are also treated in a similar manner. A case contains all information about an event; the information is not found in any other place. Each frame of a case represents a distinctive aspect of the case's knowledge. To match a case, each one of its frames is matched one by one.

Choosing the best case and matching are two closely related processes. To choose the best case one must find a case that matches the input conditions. If the match is not exact, a counting method could be used, as in [Oppacher88], to determine the best case. The Dynamic Planning System (DPS) counts the number of features matched and those missing to determine the degree of match between a case and the input conditions. Other methods include the use of heuristics, while still others use analogy. The important concept here is for the method to determine what is important, relevant, and similar, and use it.

Within these six fundamental issues are many open issues including those noted by [Rissland89]:

1. *Matching Metrics:* finding schemes for matching cases in the case library;
2. *Selective Matching:* determining relevant features for matching, thus reducing the cost of matching;
3. *Selecting Indices:* selecting and generating indices for matching;
4. *Memory Organization:* determining standard memory organization schemes that ,as well as structuring knowledge, permits dynamical modifications of it;
5. *Dealing With Sparse Memory:* finding methods of interpolating a case to a problem description for use in sparse case libraries;
6. *Connection And Adaptations of Cases:* finding schemes for connecting different cases and their components for the purpose of adaptation;
7. *Forgetting and Validating the Case Library:* finding methods for controlling the contents of the case library, keeping it from being cluttered;
8. *Novel Case Construction:* generating methods of constructing novel cases not constrained by the normal adaptation routines;
9. *Noisy Environments:* generating methods of reducing the overfitting effect, too many similar cases;
10. *Mixing CBR With Other Paradigms:* understanding methods of combining different reasoning paradigms, when appropriate, to increase the overall strength of the combined system.

in this thesis we present methods of approaching the last five issues: the mixing of CBR with other paradigms to provide better adaptations of a case, forgetting and validating cases in the case library in a noisy environment for improved learning, and the construction of novel cases. In the following sections, the problems of these issues and summaries of the proposed methods that solve them are described in more detail.

3.1 ADAPTATION OF A CASE

We begin by describing some of the critical problems in CBP and rule-based systems, before suggesting why combining and connecting the two is beneficial for case modification and repair. This is to provide the motivation for our approach that integrates and adapts CBP and rule-based techniques to produce a flexible planner, that in addition to replanning during execution, also has a main goal of acquiring new cases and rules to make it a more powerful planner after each planning iteration.

Rule-based systems that are used for planning suffer from two weaknesses: brittleness, and the inability to change with experience. These weaknesses critically affect a system's ability to learn. In order for a system to learn it must have the ability to cope with new information in a manner that causes some adjustments in the internal structures of the system. A system that relies strictly on the information provided by so called experts is bound to fail. Only when a system can notice its lack of knowledge or the incorrectness of its knowledge and adapt, will it ever learn.

CBR overcomes the problems of brittleness and inability to change with experience; however, it too has problems in the way knowledge is encoded. Adaptation and replanning heuristics are often generated in an ad-hoc fashion and are imbedded into the problem solving architecture with little or no validation of their usefulness. So called 'tweaking rules' [Kass88] are used in this manner in order to adapt and fix plans. However, if these are to be the only heuristics, how does one determine if they are sufficient to cover all possible adaptations of a case? How does one know if they are even correct? These are useful questions to ask of any systems that claims to be efficient and flexible.

The problem with these specific heuristics is that for them to be useful they need to be of a very general nature. If the person developing the heuristics does not have an intimate knowledge of the problem space and of the case library, it is a difficult task for the developer to compose the general heuristics. This is an even harder task when one's

problem space is dynamic. The constant changing of the environment will soon make the initial heuristics obsolete.

Although an initial set of heuristics is required, they should not be the definitive set for any system. Over time, more general tweaking rules should be acquired from detailed cases that have been formed from the execution of past plans. The heuristics are derived by analyzing what adaptations do and do not work, and then, if possible, generalizing the heuristics to make them applicable in a wider range of situations. Also using knowledge acquisition techniques and expert's advice, new primitive heuristics can be introduced into the system for evaluation. We describe methods for doing this later.

While we feel that the features of brittleness and inability to change strongly recommend the use of some form of CBP for 'realistic' domains, we also believe that the current CBP paradigm has to be extended in certain respects. First, tweaking strategies must be learned as well as expert-supplied. Generalized, learned tweaking rules have the benefit that they are more concern with the adaptation of the plan in a global sense, not in pinpoint recovery from a specific failure. For example, if one of a plan's tasks fails to execute in four different positions and finally executes in a fifth position, and some other task also failed to execute in its allocated position but succeeded in another, it would be better to have one tweaking rule that notices that the first task failed, and move the first and the second task to their new locations, than proceeding to go through six different tweakings in order to get the adapted plan. This more general heuristic is also beneficial to other plans that contain the same combination of tasks and have a similar failure. The build-up of advanced heuristics based on experience is more beneficial to the system than having a minimal set of low level heuristics to cover all adaptations.

Secondly, related to one, if cases fail similarly, and are similarly fixed, one does not just store the failure and repair information in only those cases, but abstracts the failure and repair information from the cases into a more accessible, general form - rules! The cases

themselves are still around to provide possibly needed information that was 'left behind' during the abstraction; cases generally contain more information and detail than the indices used to find them and the rules abstracted from them.

Thirdly, TOPs themselves should be learned; other CBPs don't do this. TOPs can be generalized from initial planning episodes. Putting what has been learned from cases into rule form has the advantage that it is potentially more efficient to access a knowledge base of replanning rules than to index into a case library and abstract what amounts to, on each occasion of needed replanning - again and again - essentially, a single replanning rule.

More powerful than a rule-based system and more efficient than a case-based system, a case-based and rule-based system provides a definite advantage. It's not good enough to have many cases available, one must also learn to condense them into a more convenient format such as rules. Rules thus abstracted can help determine where new cases should be stored, i.e. they provide an approach to the indexing problem.

3.2 LEARNING CASES IN A NOISY ENVIRONMENT

Any system that is actively acquiring new information with limited memory, such as CBR, will find three problems emerging. The first problem is concerned with interaction and relationships between cases. Using CBP, new cases are acquired and added to the case library over time ; however, one cannot continually add more and more cases to the library without looking at how these new cases are affecting the old cases and visa-versa. For example, a new case could be in complete conflict with an old one, possibly because it suggests a different solution to a problem. However, it could be only suggesting a widely different approach to solving a problem, but still resulting in the same solution. One now must ask the question: which case is better and under what circumstances or are both cases acceptable but just proposing different methods of a solution?

Cases may be only partially correct. For example, the first part of a case - say half of its features - gives rise to the case's solution, but the remainder of the features in it serve no

purpose or are incorrect. Similarly, another case with the same solution may have only the second half of its feature set correct. Are both cases wrong or should a new case be formed from the union of the correct parts of the original cases? Do we then remove the original cases?

These questions lead to a second problem of discovering cases that are no longer considered acceptable for use by the Case-Based Reasoner. Either over time or after excessive failures, cases need to be reconsidered as to their current validity and effectiveness. A case that was developed a long time ago may be of little use in an evolving environment. A case that is consistently providing the wrong solution to the initial conditions must be dealt with. A method of noticing and removing the invalid cases is certainly required to provide a more flexible and adaptive system. What about a case is there to notice? Goals, failures, outcomes can be used as observable values; however, a more systematic method is required. A method that looks at the strength of a case, not one that relies on heuristics or simple outcomes. Such a method must answer the question: how is the strength of a case calculated?

The final problem deals with size. Memory often comes in a fixed amount - all computers have a maximum available memory capacity. Whatever the number, a good goal for any CBRS to keep in mind is to have a strong library of cases with little redundancy, rather than many cases with considerable redundancy. The reason is that few strong cases, providing a good representation of the entire domain problem space, along with good adaptation methods provide much better coverage of the problem space than many redundant similar cases. In the experimental results of Bradtke [Bradtke88], this observation is also noted. The results compared how good a CBR system performed using cases that were either generated at random or provided by a human expert. The random cases covered a random area of the problem space. The human cases covered less of the problem space, but provided greater detail of the solution and initial conditions. In general,

the random generated cases solved more problems from a sample set, and required less search to find them than the cases generated by the human.

Forgetting is thus an essential process of the case library. Cases cannot be continually added into memory, nor can harmful, redundant cases be left in it.

Genetic algorithms (GA) [Goldberg89, Odetayo89, DeJong87] have properties that if combined with CBR could allow us to tackle forgetting. First of all, GAs search a space of possible solutions and remove the weak solutions, keeping only a smaller set of stronger solutions. This method would allow us to control the quantity and quality of the case library, if you consider the search space as the set of all possible cases. We can obviously not keep the entire set, but we are looking for the best reduced set possible, and then leave it up to the adaptation methods to extend that space. Since GAs rely on a strength value, a method of calculating a case's strength must be provided. The goal of using GAs is to apply the subsymbolic techniques of genetic algorithms to cases and CBR.

3.3 NOVEL CASES

One of the shortcomings of many CBR systems is their weak ability to discover new cases that deviate strongly from those already in their libraries. This thesis presents a new method of formulating new cases, such as strategic plans for playing board games like checkers, that is designed to overcome this deficiency by strengthening the CBR technique with genetic algorithms.

When one is solving problems using cases of past solutions, the solutions can tend to become repetitive - stuck in a rut. Even the strongest of adaptive methods can make only similar adaptations of a case, time and time again, using the standard methods of adaptation and analogy. What is more desirable is a method of altering a case in a novel way so as to, more often than not, produce a case that is not one of the normal adaptations of the case. We are looking for novel cases, loosely based on the original but not an adaptation that is consistently performed using other traditional methods. If the problem of removing poor

cases were solved, the system would be able to garbage collect them, removing unacceptable novel cases that were created and added to the case library. Good novel cases, however, will remain in the case library and improve the overall quality of it.

GAs again have properties that will allow us to tackle this problem. GAs are based on natural selection, mutation, and crossover. Applying genetic techniques to cases can lead to methods of creating novel cases in a controlled manner. GAs use processes that are random in nature and are not constrained to the extent that typical adaptation methods are. However, they are controlled in that they are directed to a general purpose - finding the best possible solution. Taking GAs and applying them to cases, rather than chromosome strings, is one goal of this thesis.

4 USING RULES FOR THE ADAPTATION OF A CASE

The aim of this chapter is to integrate CBP and rule-based techniques, using the best of each technique. By combining rules with cases, we create an overall planning approach and structure that can not only acquire new plans and replanning rules by itself and from an expert, but can also induce higher level learned replanning actions that are not immediately apparent to the user or a typical CBP system that is only looking at single point failures[6]. The rules, located in a knowledge base, constitute general concepts. For greater power, inductive methods can be applied to the knowledge base later to increase the overall potential of it. The cases serve as the precedent situations allowing the system to produce rules at different levels for use by case-based and analogical reasoning processes. From the knowledge base's point of view, the rules are the directory structure for cases. We can search for a special case (or situation) by the rule-based directory. What we are concerned with is the acquisition of repair actions that can fix the entire plan, not a specific piece of it, in order that it can be later reused.

4.1 INTRODUCTION

One form of intelligent behavior that is important to capture in machines is planning. Planning is often thought of as the process of transforming an initial state to a goal state [Genesereth87]. Early planners such as GPS [Ernst69] or Strips [Nilsson80] search through a space of situations in order to find a sequence of operators for transforming an initial state to a goal state. At each step, such a system would measure the difference between the current and the goal state, and select an operator permanently indexed under the difference it is capable of reducing. Since these systems attempts to reduce differences

[6]This is a failure involving a single task.

without regard for their importance to the overall problem, making low-level decisions before high-level differences are reduced, they have to backtrack excessively.

Noah [Sacerdoti74], Nonlin [Tate77], and their descendants (which include the majority of existing planners) eliminate some backtracking by following a policy of Least Commitment: partially ordered plans are hierarchically expanded and ordering decisions are only made when necessary. Whereas Strips-like planners create each plan from scratch and tackle problems due to unfavorable interactions of operators in the same way in which they handle any planning problems, i.e. by backtracking, Noah-like planners attempt to anticipate unfavorable interactions among subplans, but are helpless when there is no appropriate plan in their library. It seems, however, that they can only detect and cope with that type of interaction in which one task's effects completely undo another task's precondition. Neither of these types of planners can deal adequately with feedback imposed constraints which are similar to being told of changed goals while planning, nor can they recognize conflicts among tasks in domains having to do with deadlines, time, and other continuous resources.

Planning is not, however, the mere act of determining the transformations. People, as intelligent creatures, reuse old plans and tend to fix them 'on the fly' as needed. Realistic planners should be able to adapt old plans to meet the current situation, and extend their plan library by learning. A robust and efficient planner should neither be forced to give up if there is no appropriate, ready-made plan in its library, nor have to replan always from scratch. A realistic planner should anticipate some planning failures and be prepared to handle unforeseen deviations from the usual course of events by carrying on, without backtracking, from the point of interruption; be able to use past plans that worked in situations similar to the current planning environment rather than always planning from scratch; and should also know when it requires additional help in planning due to its lack of knowledge.

Planning, as generally understood, is a problem-solving activity whose output is a plan, i.e. a tentative sequence of operations for subsequently performing some task. The apparently plausible view that planning precedes and is clearly distinguished from action, i.e. from the execution of the plan, has led many researchers in this area to concentrate almost exclusively on the process of plan construction and to down-play the importance of updating plans with information gained during execution. This emphasis, in turn, has tended to produce planning systems that create each plan from first principles and discard plans after having used them once. After all, when plans are not systematically updated with information about their successes and failures, there is little reason to keep them around.

In contrast with this view, we treat planning as a process that combines developing an initial plan, executing the plan, fixing failures as they arise during execution, evaluating, modifying, and storing the plan for later reuse.

We agree, of course, that some planning precedes plan execution, but we, as well as others [Ambrose88], believe that in many realistic domains planning and execution should be interleaved, i.e. that execution should start before a complete plan is achieved, that unforeseen plan failures should be handled by replanning on the spot, and, most importantly, that the results of evaluating the execution of plan fragments and plans should be stored with the plans for future use. In other words, it is this integration of planning and execution that enables a planner to improve its performance in future planning episodes by taking advantage of information gained during past executions and subsequent evaluations.

This approach seems particularly appropriate in what we call 'realistic' domains. A manufacturing plant is an example of such a domain. For a product to be assembled, one needs to have a fixed amount of man-power and all of the right components. Different components require different quantities and manpower to install depending on the product. Some components must be added before others, while others may be added at different

times. An assembly line tends to fix the order of subtasks in order that a single plan can produce a product. However, a breakdown in the assembly process can necessitate an on-the-spot reshuffling of the original plan in order to keep production moving. After all, workers and resources cost money, and one does not want them to be inactive.

Such domains - which include mundane tasks - have the following functional characteristics: the domain in which the planning takes place undergoes changes - due to other agents and laws of nature - that are unforeseeable in principle or practically unpredictable, i.e. trying to predict them would immobilize the planner. Such unforeseen changes can alter the planner's goals and, by disabling or enabling preconditions for plan steps, can constitute hazards as well as opportunities. The planner should be able not only to avoid hazards in time, but also to take advantage of opportunities. Due to the inevitable incompleteness of knowledge and the insufficiency of computational resources, it is simply not possible to plan in detail for all contingencies. On the contrary, in such a domain, it is important to plan 'loosely' so as to facilitate eventual revisions while anticipating past failures as early as possible i.e. the planner has to combine the three basic strategies of postponing decisions (as in hierarchical such as Abstrips [Nilsson80]), of planning early against presumed possible failures, and of assuming, for the time being, that the course of events will unfold in a way that is similar to what happened in the past.

The demands that 'realistic' domains place on planners seem best met by an approach in which the planner constantly monitors and learns from the execution of its plans. This emphasis on integrating planning and execution leads us to adopt a CBP approach [Hammond89b]: when presented with a new problem, our planner tries first to retrieve a past plan, annotated with failure and success information that has worked in a similar situation, and to adapt it to the current situation. A new plan is constructed from scratch only if no sufficiently similar past plan can be found. Since totally new and modified plans are constantly being added to the plan library, CBP is really a form of learning plans from

experience. The following sections describe a variant of CBP that is capable of meeting the outlined objectives in a scheduling domain.

4.2 APPROACH

Our approach uses a combination of both CBP and rule-based techniques, exploiting the best features of each. The central structure in the architecture is a case (plan). A case contains information such as the goal, the tasks, the resources, the constraints, and the past failures. In our approach, cases are indexed in two different ways: by configurations of goals and resources, and by rules storing past failures and their solutions. These fixing rules (FR) are learned by generalizing failing cases. The acquired FRs are placed in a hierarchy in such a way that the more general rule is put ahead, or higher, than a more specific rule. Since each such rule has a link to the cases from which it was learned, this rule hierarchy produces an alternative indexing method of the cases based on failures and fixes.

Initial FRs are acquired in three different ways: by extracting them from past failure solutions found in existing cases, by generalizing current fixing rules into more powerful abstract FRs, and from experts. Experts are better at replanning, making specific changes, than of forming a complete plan. Experts tend to miss the broader ramifications of their plans. A good generalization method solves this problem because it is not constrained with what it knows already or what is appropriate.

Our planning architecture is an extension of the CBP architecture. It relies on a feed back loop to provide information about the success or failure of the plan's tasks during execution. Subsequent planning sessions involving this or a similar plan can use this information and thereby gain from past experience. Thus, starting with an initial library of plans generated by an expert, the planner acquires new plans through a form of plan learning from past planning episodes.

4.2.1 Architecture

Our approach to the planning problem postulates five major components: the *PLANNER*, the plan *EXECUTOR*, the dynamic *REPLANNER*, the *RULE LEARNER*, and the plan *EVALUATOR*. Figure 4.1 shows how these components fit together.

The PLANNER controls the planning process, from information input, past plan locating, to plan construction. Initially the planner is configured with its resource information, i.e. it is told by a (not necessarily human) OPERATOR which resources it can use over the plan execution period and which planning parameters, i.e. tasks and constraints, are to be met in the initial plan construction phase. The retrieval component of the *PLANNER*, called the *Locator* , uses a case-based approach to locate a past similar plan-goal-resources configuration. Using the supplied input information, the *Locator* indexes into a library of old plans, indexed by their goals (tasks), in an attempt to find a plan that matches the current planning parameters. If a matching plan can be found, it is passed to the plan *EXECUTOR* component. If no plan can be found, the *Locator* attempts to find a plan whose goals are a subset, superset, generalizations, or specializations of the current goals. The modification component of the *PLANNER*, called the *Constructor*, modifies the plan by using domain planning information or planning heuristics found in the knowledge base, or modified by the OPERATOR, to create a new plan to be executed by the *EXECUTOR*. The index to the knowledge about planning and plan modification is formed using the task and constraint information about the task the planner is currently considering. The modified plan is then verified using expected resource information to ensure the plan's integrity. A failure in a task's verification will cause the *Constructor* to alter the plan.

This approach enables the planner to continue planning even though it has no exact ready-made plan to deal with the current tasks.

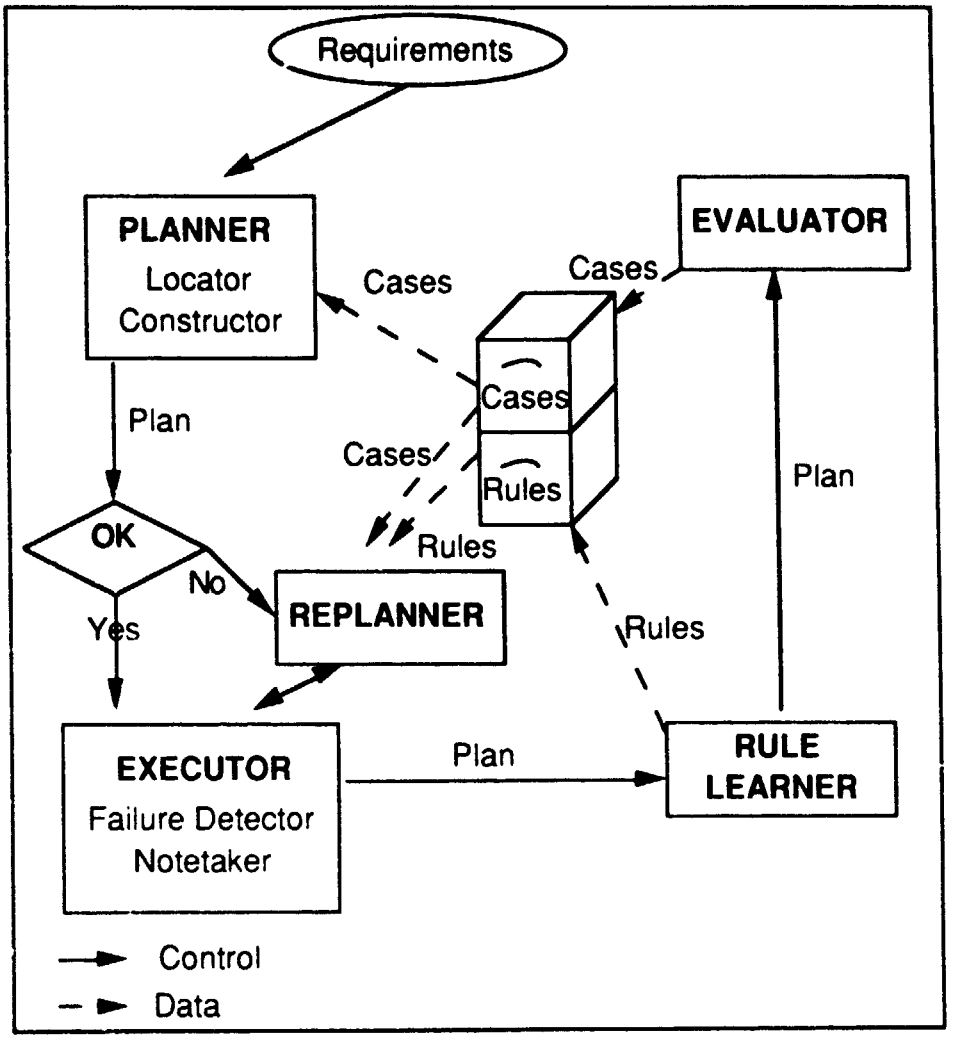


Figure 4.1 Planning Architecture

The plan *EXECUTOR* takes the plan and starts to execute it. If the *EXECUTOR's Failure Detector* component experiences a failure condition (lack of resource) that was unforeseeable at the time of plan construction and arose during the plan's execution, the exception is recorded as a failure, and the dynamic *REPLANNER* component is activated This component will attempt to reorder the plan, remove failing goals, or ask the *OPERATOR* for assistance in order to keep the *EXECUTOR's* plan execution continuing These replanning actions are found in the knowledge base or in that plan's failures, indexed like any other planning information. On-the-spot replanning is important, after all,

when a plan fails one does not want to stop the execution since resources have been allocated and are ready to use. The information about what replanning was done is noted in the failure for later use by the *EVALUATOR* component.

The dynamic *REPLANNER* is an important improvement that distinguishes our planner from other planners described in the literature. It prevents minor faults from stopping plan execution, and causes only moderate modifications of the plan. The *REPLANNER* uses the planning technique known as goal planning or reactive planning [Schoppers87] to keep a plan executing. The *REPLANNER* indexes its own failures, the knowledge base, or other plans to see if any replanning information is available for use in the current situation.

The *EVALUATOR* helps the planner acquire new plans and knowledge by learning from itself. This is achieved by adding new, successfully executed plans constructed by the *Constructor*, and by altering old plans due to planning failures. By recording the failures, the *EVALUATOR* also learns to fix and adapt plans in new environments over time.

The *RULE LEARNER* acquires rules for the *REPLANNER* to use when a failure occurs and a replanning action is required. Rules can be learned in one of three ways: from direct entry by the user, by looking at existing cases and using their existing replanning actions as new rules to be added to the knowledge base, or by analyzing the *RULE LEARNER's* current rules and producing new general and specific rules. The *RULE LEARNER'* purpose is two fold: it separates control knowledge out of the system and into a knowledge base where it can be more easily managed and manipulated; and, because of the separation, rule generalization and specialization can be performed. Without the separation, learning of any form is a difficult task because one must modify the entire system not the part it is using - the rules.

4.2.2 Structures

A good plan is one that wastes little time and resources and can adapt itself under these same constraints. Conditions that affect the execution of a plan, such as lack of a part or

manpower, are often symptoms of a plan execution failure in a real-world environment. Planning structures must, therefore, have the ability to incorporate new plan knowledge and repair information for plan modification as it becomes available, making it as dynamic as the environment around it. Typical components of a plan, in a domain where tasks[7] with resources are scheduled include, include: the tasks, the constraints placed upon each task, and the failures and corrective actions taken with the plan. The following are examples of the components in an assembly domain.

Tasks are treated as unit activities (assemblies) that cannot be further decomposed from an *EXECUTOR's* point of view. They are assumed to be directly executable by the execution component of the planner. A task's information aids the planner to efficiently position it among the other tasks in the plan. A task has associated data slots identifying the task's purpose, its constraints, and its failures. For example, a task could be to install the doors on a car, it requires two people, two doors (in the case of a two door car), four door hinges, and should only take five minutes to execute.

Task Constraints are requirements of the task in order for its execution to start or fully complete. For example, a task constraint could be the paint on a car must dry for at least 24 hours before any further work may proceed; therefore, the car painting task must start execution at 0800 hours one day and finish execution no earlier than 0800 hours the next day. This is a task completion constraint. Other task constraints can take the form of resources. **Resources** are any physical supply, such as manpower, light bulbs, or car

7 For example, connecting a car door to the frame, or attaching the motor to the car frame

doors, that are required for the execution of the task. With out the resources, a task can not begin its execution and would fail.

Plan Failures identify task failures that the plan previously encountered during execution, and what was done to correct them so the plan could continue execution. For example, if the task of putting on the two car doors took longer than five minutes, this would cause a failure in the plan for that task. The action taken in this case could be to take the extra time required to finish putting on the doors and shortening the next task's execution time in the hope that the next task could be done quicker. The failure and repair information is stored as a single instance of a plan failure. A plan may have no failures.

4.2.2.1 Plan Structure

Plans are an ordered sequence of tasks produced by the planner for execution by its execution component. Slots are allocated to contain the task information and other information required for the retrieval, execution, and replanning of the plan, as seen in Figure 4.2. The first slot, **plan-name-slot**, is used for distinguishing a plan from others. The next slot, **goal-slot**, contains the goal or the purpose of the plan. The **start-time-slot** and the **stop-time-slot** contain timing constraints for the start, stop and maximum execution time of the plan. The **alternative-slot** provides a list of other plans that have been retrieved along with this plan in similar situations. The **success-slot** is a counter used to measure the number of times the plan has executed successfully without failure. The **failure-n-slots** contain failure descriptions, replanning actions, and pointers to failure rules used to decide upon the replanning action taken. Finally, the **task-n-slots** contain information about the individual tasks that are to execute for the given plan. The following figure, is a physical description of a plan as it resides in memory

```
(plan-name-slot        ; the name of the plan.
goal-slot              ; the overall goal of the plan
start-time-slot        ; the start time of the plan
stop-time-slot         ; the stop time of the plan
alternatives- slot     ; a list of similar alternative plans
success-slot           ; a count of the number of times, initially zero, the plan
                       has executed successfully without having to be
                       replanned by the Replanner.
failure-1-slot         ; failure slots include information about how the plan
                       failed and what was done to correct it.
  "
failure-n-slot
task-1-slot            ; e.g., install doors.
  "
task-n-slot).
```

Figure 4.2 Plan Structure

The **Plan Library**, as the name suggests, contains plans that have either been created by hand, or created by the system and added to it. Plans are stored sequentially and are indexed by the *PLANNER* component of the architecture using the tasks as the indices

4.2.2.2 Task Structure

Tasks are units ready for execution by planner. They are not decomposable It is reasonable to associate tasks with executable units because our form of planning is for tasks at a specific level. Assemblies can by thought of as many subassemblies, and those of many other subassemblies, and so on. The plans here deal with only one level of assembly

and one level of subassemblies; there are only so many stations on an assembly line that are required to build a car, the engine station is not concerned with how the engine was built, but rather that the component is present for its assembly into the car. With this in mind, slots are allocated to contain the information required for execution and replanning, as seen in Figure 4.3. The first slot, **task-name-slot**, is used for distinguishing the task from others. The next slot, **goal-slot**, contains the goal or the purpose of the task. The next slot, **activity-slot**, contains the action the task is to achieve. The **start-time-slot** and the **stop-time-slot** contain timing information for the actual start and stop times of the task. The **constraint-n-slots** contain the constraints, resources, and timing, that are placed on the task for its execution. The following figure, is a physical description of a task as it resides in memory.

```
(task-name-slot        ; the name of the task; e.g., install doors.
 activity-slot         ; what the task is to do; e.g., install two doors on car
 start-time-slot       ; the start time of the task
 stop-time-slot        ; the stop time of the task
 constraint-1-slot     ; e.g., two men are required.
 constraint-2-slot     ; e.g., two doors are required.
 constraint-3-slot     ; e.g., four door hinges are required.
 constraint-4-slot     ; e.g., five minutes of execution time required.
   "
 constraint-n-slot)
```

Figure 4.3 Task Structure

4.2.2.3 Plan Failure Structure

Recorded past plan failures play an important role in the fixing of plans in the fastest, most efficient manner. A plan's task can fail for one of two reasons: a constraint, such as there is no manpower to perform the task, cannot be met; or there has been a time violation in the execution of the task. Using the constraint or time violation, and the task, an index is formed to look for a similar past failure in the plan. If none can be found, the knowledge base is consulted for a replanning rule. Once the rule is found, its is performed, and the task, constraint, and replanning action are formed into a failure and added to the plan. If the same constraint cannot be met for a task in a subsequent execution, the failure is immediately indexed and the replanning action done without going to the knowledge base for the replanning action. Storing the replanning rule in the plan's failure has two benefits. The first is that the replanning action is present for immediate replanning. The benefit of this is that it allows the *REPLANNER* to immediately locate other occurrences of similar failures. By doing this, the *REPLANNER* can check on the possible results of taking the replanning actions. The second benefit is that learning is now possible. Without recording the failure it would be a difficult task to remember what happened or was done in the past.

With the above goals in mind, slots are allocated to contain the information required for replanning and indexing, as shown in Figure 4.4 . The first slot, **task-name-slot**, is used for distinguishing a failure from others. The **constraint-slot** and **task-slot** identify the failing constraint and task. The **frequency-slot** counts the number of times the failure has occurred before, and the **index-slot** points to the replanning action to take for the defined failure.

```
(failure-name-slot      ; the name of the failure; e.g., doors not installed
 constraint-slot        ; insufficient man power; only one man.
 task-slot              ; install doors
```

```
frequency-slot      ; e.g., one occurrence.
index-slot)         ; delay rule
```

Figure 4.4 Failure Structure

4.2.2.4 Knowledge Base and Heuristic Structure

The *Knowledge Base* is essential for replanning when no existing replanning information is stored within the currently executing plan. Such information could be lacking for two reasons: the plan is a new one, or it has never encountered a similar failure in the past. The *Knowledge Base's* task is to decide what to do with the plan, specifically with the task that failed, to ready it for further execution.

Replanning occurs because of a failure in either a plan's or task's constraints. Such a failure could be due to an unexpected effect of an action on the environment, an unforeseeable environmental change, or a mistake in the plan. The *Knowledge Base* uses the task and failing constraint as indices to retrieve a replanning action to modify the current plan. These actions, for example, could consist of deleting the task, delaying it, repositioning it, stopping the plan's execution altogether, reapplying CBP to this small task and the tasks in the remainder of the plan, asking for human intervention, or using traditional methods to replan from scratch. It is important to store the actions taken on a plan, both to evaluate how good the actions themselves are (in context), and because they figure prominently in the evaluation of the plan after it has been executed. Figure 4.5 shows a typical rule that could be used by a inference engine to deduce such a replanning action. It states that if the failing constraint type is **Manpower**, then remove the task in the plan until the manpower is found for the task.

This type of knowledge is declarative. Initially it is a collection of heuristics that can be extracted from human experts during knowledge acquisition. Resolution planning, based on logic, attempts to use this form of domain knowledge to build all its plans. The interac-

tions between tasks must be derived at planning time. This is a formidable task in environments more complex than the blocks world. In our approach, this knowledge is only used as an aid in replanning. The success or failure of the replanning actions are shown during the future execution of the plan. This information is then used for modification or repair of the plan in subsequent planning iterations. Using the *Knowledge Base* in this manner leads to a less complicated set of rules because they are only used for intermediate planning steps, not for the complete planning process.

Rule1

```
Input: task, constraint, plan
Output: action

    If the type of the failing constraint is 'MANPOWER'
    Then
            action <-  delay the task
    Link
            case 2
```

Figure 4.5 Sample Replanning Rule

When looking for a replanning action, if a rule matches exactly it is used as in an expert system. If no rules matched, they are ordered according to partial match criteria (e.g. least number of unmatched parts). These partially matched rules index failures within existing plans (from which they were generalized in the first place); the retrieved failures and their plans are now reviewed for similar planning failures, thus enabling their replanning actions to be used in the current situation. This framework is shown in figure 4.6.

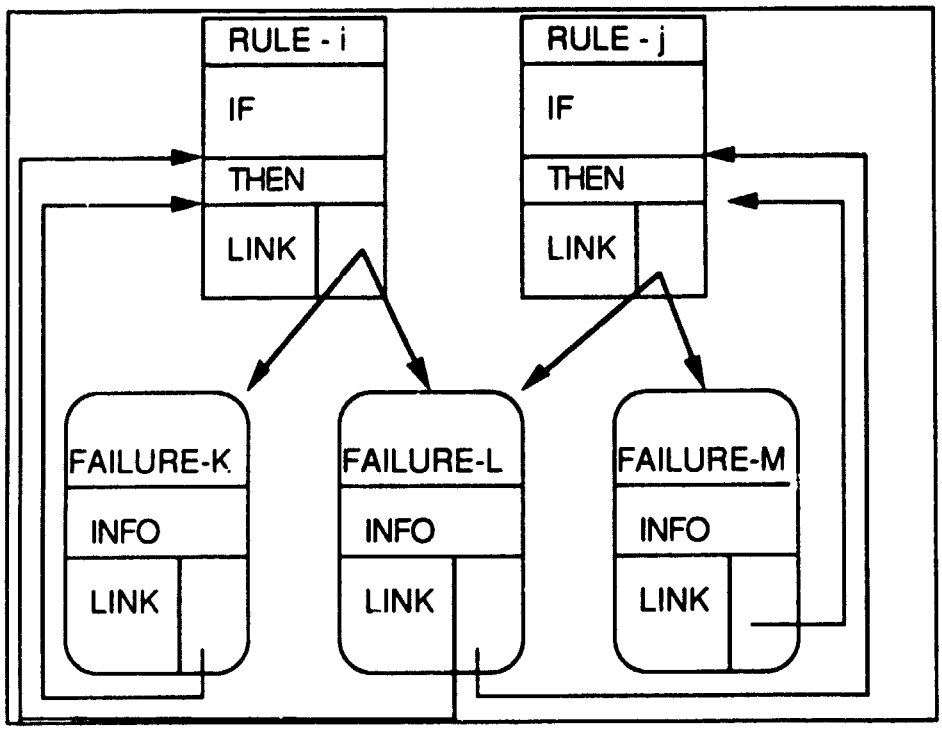


Figure 4.6 Case-Failure Rule Relationship

Primitive replanning rules are initially provided by an expert in the planning domain; however, the system generates new rules through generalization and combination of active replanning rules. For this reason, rules are sorted and stored in a classification hierarchy. If a rule is generated by rule generalization, the rule is more general than its ancestor and has fewer limitations than its ancestor when it is applied. It will be stored at the place just before its ancestor in order that it can be applied in more cases than the ancestor. If a rule is generated by rule specification, the rule is more specific than its ancestor and has more limitations than its ancestor when it is applied. It will be stored just after its ancestor. If a rule is generated without any ancestor, it will be stored at the beginning of the *Knowledge Base*. The following is an example of such a process.

GIVEN:

Initial **Rule R-i** and **Rule R-j**

RULE R-i

IF (A B G)

THEN (F S)

LINK (CA1 CA2)

RULE R-j

IF (D B G)

THEN (F W)

LINK (CA4)

The rule R-k is formed through generalization.

RULE R-k

IF (B G)

THEN (F)

LINK (CA4 CA1 CA2).

The resulting classification hierarchy with these three rules in the *Knowledge Base* is shown in figure 4.7. The following section describes the complete process.

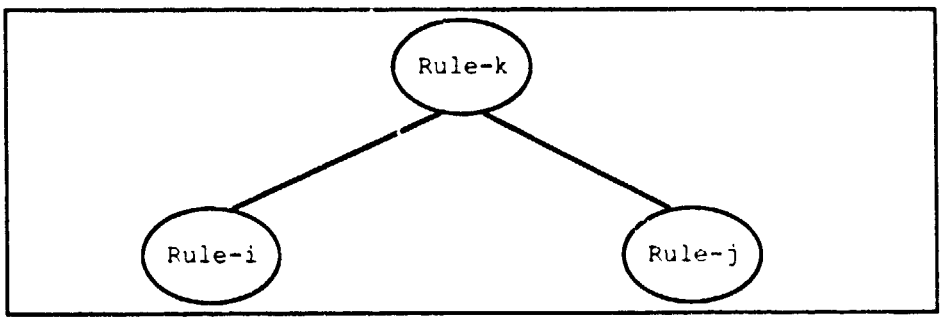


Figure 4.7 Replanning Rules Classification Hierarchy

Rules for the *Knowledge Base* come from one of three places: an expert user, case failures, or from a rule learner such as LEW [Matwin89] which, as well as learning new rules, maintains the integrity of the *Knowledge Base*. LEW relies on a type of conceptual

clustering of rules in order to achieve its functionality as a knowledge acquisition tool. It uses both generalization and specialization from expert-supplied sample solutions to sample problems to generate a *Knowledge Base* that can be subsequently used by an expert system.

4.2.2.4.1 LEW At Work: A Simple Example

As an example, we show how LEW could be used to acquire and learn new replanning rules. Terms used here such as cues, clusters, type, and 1-different, are not described, but can be found in [Matwin89]. The inputs to LEW - which are called *cues* - consist of an optional environmental component (omitted in our examples below) and pairs containing a specific problem and its specific solution. Unlike traditional forms of learning from examples [Dietterich83], in which the system uses the teacher's answer to improve or modify the result of a prior generalization of an example, LEW treats the problem-solution or question-answer instances, i.e. the cues themselves, as the basic units for generalization. LEW will accept as a cue any pair that can be parsed into a pair of conjunctive structures. In the cues below, Ti, Ci and Pi refer to $Task_i$, $Constraint_i$, and $Plan_i$.

cue1 = Task1 Constraint1 fails in Plan1; Take Action1 on Task1 in Plan1!

Cue1 is a cue in which the environment has been omitted. For notation, the problem is separated by a ';' from the solution. Cue1 is now in a *cluster c* by itself. Suppose now that after *cue1* was entered, the following *cue2* has been given to LEW.

cue2 = Task2 Constraint1 fails in Plan1; Take Action1 on Task2 in Plan1!

Since the problem solutions are identical, LEW treats the words that realize the single difference between them (Task1 vs Task2) as *synonyms*, and stores *cue1* and *cue2* in the same *cluster c*. If another cue is added:

cue3 = Task1 Constraint2 fails in Plan2; Take Action2 on Task1 in Plan2!

Cue3 is found to be *1-different* from the cluster c, and the words that realize that difference create a *type* **T** = (<Constraint1, Action1>, <Constraint2, Action2>). Any two cues that are identical up to a synonym or a *type* belong to the same cluster, therefore the cluster c will now contain cues *c1*, *c2* and *c3*. It should come as no surprise that LEW will now be able to answer the question

Task2 Constraint2 fails in Plan2 ?

Since Task1 and Task2, and Plan1 and Plan2 are synonyms and *cue3* is *1-different* from the problem, LEW can generate the cue

cue4 = Task2 Constraint2 fails in Plan2; Take Action2 on Task2 in Plan2!

LEW also performs *conceptual* learning, i.e. the kind of learning in which the system relies on additional rules in the background *Knowledge Base* to transform the original representation space. Even the simplistic example above shows that the generalizations themselves lead normally to a condensation of knowledge. Learning and *Knowledge Base* condensation occur whenever the assertion or retraction of a cue by the expert, i.e its presentation as a positive or negative example, triggers the process of *Knowledge Base*

reorganization. During that process, each expert-supplied cue can bring about a complete reorganization of memory.

This short example helps to illustrate that a rule learner, such as LEW, using a generalization mechanism can help provide a greater range of replanning rules. For more detailed information about the learning and knowledge updating algorithm refer to [Matwin89].

4.3 PLAN MODIFICATION AND REPAIR

Plan modification occurs when a complete plan, including its goals and organization, is altered to form a new plan. In an integrated planning approach such as ours, plan modification may occur at two different occasions in the planning process. Each is distinct in its purpose and the type of modification required.

Plan modification can first occur when a past plan, retrieved from *Plan Library*, does not match the current set of tasks to be planned for. In a complete environment, there exists a plan for every possible set of combinations of tasks. However, the power of CBP does not come exclusively from its ability to use past plans, but also from its ability to take a plan that does not exactly match the current situation and modify it to meet the current requirements. In our planner, the current requirements are the tasks to be planned for. The algorithm for plan modification has two stages; the first stage is the retrieval of the appropriate past plan that is similar to the input tasks and constraints.

Algorithm for Plan Retrieval:

Input:

Tasks { The tasks to be planned for}

Output:

Retrieved_Plan {The plan that closely matches the input task environment}

```
Variables
    Extra_Tasks      {Those tasks found in the Retrieved_Plan but not in the
                        input task environment}
    Missing_Tasks    {Those tasks found in the input task environment but not in
                        the Retrieved_Plan}
Begin
    {Find the best past plan}
    Retrieved_Plan := nil
    For past_plan := all plans in the library Do
    Begin
        Extra_Tasks := NumberOfExtraTasksIn(past_plan, Tasks)
        Missing_Tasks := NumberOfMissingTasksIn(past_plan, Tasks)
        past_plan.DistanceFromPerfectCase := (Extra_Tasks + Missing_Tasks) /
                                                    # of Tasks
        { save the plan that is minimally distant from the input tasks}
        If (Retrieved_Plan.DistanceFromPerfectCase >
                past_plan.DistanceFromPerfectCase)
        Then
            Retrieved_Plan := past_plan
    End

    { Remove the extra tasks in the retrieved plan }
    For extra_task := all Extra_Tasks Do
        RemoveFrom(Retrieved_Plan, extra_task)

    { Add missing tasks in the retrieved plan }
```

```
    For missing_task := all Missing_Tasks Do
        AddTo(Retrieved_Plan, missing_task)

    { return the modified retrieved plan}
    Return Retrieved_Plan
End
```

The *Locator* locates the plan that best matches the planning tasks. In our example, a best match means a plan that minimizes the sum of the number of tasks found in the case and not found in the planning tasks, plus the number of tasks not found in the case and found in the planning tasks, then normalized by the total number of tasks. An outcome of zero means the case has all and only the planning tasks. A strictly positive difference means that there are either superfluous or missing tasks, or a combination of both. The *Constructor* has the task of modifying the retrieved plan to meet the current situation. Superfluous tasks are simply removed from the plan. Such removals create slots in the existing plan, which may eventually be filled with missing tasks. The missing tasks are scheduled into the plan in slots that permit enough time for the tasks to start and stop execution.

The second stage verifies that the plan and its task's constraints, e.g. Start Time, Stop Time, Man Power, etc., are not violated, using expected values of these constraints at the given time. When a task's constraint fails to verify, the plan's failures are checked to see if such a failure has been handled previously. If one exists, the same action is taken with the current failure. If no such failure exists, the constraint, task, and plan are passed to the inference engine in the Knowledge Base to deduce an action to take with the failure. This replanning action is carried out on the plan, and the plan is re-verified from the last point in it that was modified. If the plan repeatedly fails to verify, human intervention will occur to

reposition tasks in the plan. This may look like a break-down in the planning process, but we prefer to view it as a method of acquiring knowledge from experts in the field; the replanning information is stored directly in the case and available for reuse. After the plan has been verified, it is ready for execution. The algorithm for this stage is as follows:

Algorithm for Plan Adaptation:

```
Input:
    Current_Plan        {The plan that matched the input task environment}
Output:
    Modified_Plan       {The verified plan that matches the input task environment}
Variables
Begin
    Modified_Plan := Current_Plan
    { Verify that all constraints in all tasks of the plan are ok }
    For aTask := all tasks in the Modified_Plan Do
        For aConstraint := all constraints in aTask Do
        Begin
            { If a failure is found then replan }
            While not Verify(Modified_Plan [ aTask, aConstraint ]) Do
            Begin
                If failure has occurred before Then
                    { Use failure information stored in the plan to determine the
                        replanning action }
                    failure_action := FailureFrom(Modified_Plan , aTask, aConstraint)
                Else
```

```
                { Use failure information found in the Knowledge Base
                    to determine the replanning action }
                failure_action := KB(Modified_Plan , aTask, aConstraint)

            { Update the replanning information in the plan. This could add a
                new failure to the plan or add information to an existing failure }
            UpdateFailureInPlan(Modified_Plan,failure_action, aTask, aConstraint)

            { Perform the replanning action on the plan }
            PerformRePlanningActionOn(Modified_Plan, failure_action)

            { The plan must be verified from the task closest to the start of the
                task in the plan that was modified by the replanning action.}
            If failure_action has modified tasks before the task aTask Then
            Begin
                aTask := FirstTaskInPlanThatWasModifiedByReplanning
                    aConstraint := FirstConstraintInTask(aTask)
            End
        End
    Return Modified_Plan
End
```

Plan modification can also occur after plan execution. After the plan has executed, it and the information provided by the *Notetaker* are given to the *EVALUATOR*. The *Notetaker* component of the *EXECUTOR* records the exceptions and replanning descriptions during the execution of a plan. If the plan was an old one that executed

successfully, this is recorded in its success slot. If the plan was newly created, i.e. there were enough modifications to the base plan to consider it a new plan, and it executed successfully, then it is added to the plan library along with the goals it satisfied. If the plan failed, the exceptions and replanning information are recorded in the plan along with the reasons why the goal, or goals, failed. If the plan has had a bad track record, it may be altered by the *EVALUATOR* using the recorded information to make it a 'better plan' in the future. An updated plan is 'better' than the original plan because either tasks with a proven history of failure have been removed from it, or it includes information which can be used in future planning sessions. Both failure and success information are valuable in determining the best plan for the current situation.

Plan repair, unlike plan modification, occurs when an individual component in the plan, such as a single goal, is altered to enable the plan to continue execution from its current execution point, after a failure has been encountered. During execution, failures in either the plan's constraints or its task's constraints can occur. Such failures may be due to an unpredictable change in the environment or to a failure of an expectation that was validated at initial planning time.

Once a failure occurs, the constraint and the failing task are used to generate a failure index. This is used to index into the existing failures, if any, found in the plan. If a previous failure exists, its associated replanning action is returned. If no previous failure exists, the *Knowledge Base* is consulted to return a replanning action, as in the second stage of Plan Modification. Finally, the replanning actions are performed on the plan to repair it for continued execution.

The part of the plan from the failure point to the end can also be thought of as a plan. This partial plan can be verified as in the second stage of Plan Modification. The result after verification and possible additional replanning is a partial plan that can continue execution. All replanning actions and failures are noted for plan modification after execution is

completed. The application of these techniques to partial plans allows execution to continue in many cases where other approaches would have to stop execution and to replan from the start. Once tasks have been executed and resources allocated, it is desirable to avoid replanning from scratch.

4.4 SUMMARY

It is important to outline the salient points that break out of the traditional CBP framework. The first point is the separation of the adaptation heuristics - tweakings - from the cases and the system. To imbed the heuristics into the system makes it difficult, if not impossible, to find or alter these heuristics once they are in place. Since cases have adaptation information encoded in them, it is important to get access to it, but it may be in different cases and occur in many different places. By having a knowledge base - in our case comprised of rules - this knowledge is separated from both the cases and the system where it can be easily accessed. By having a unitary representation for it, it can be manipulated, and because it can be manipulated we can use a learning method on it. Our method learns simple generalizations of rules. Other learning methods and knowledge representations could be used. The important fact is to get the knowledge into a common area with a common representation so one can make better use of it.

The knowledge representation that we are using is simple and straightforward. This makes it easier to manage. The rules alone have a simple structure, as do the cases, and the case library. The complexity comes when they are used together. This is still more manageable then an embedded system. Experts are usually good at coming up with simple cases and rules which are more specific to the problem presented. It is the general rules that experts find hard to produce. Our system asks only for the specific rules, and then tries to produce the more general rules itself. Cases also provide specific information. The important point is to get the knowledge that is available - the specific form - and use it to produce the knowledge that is not available - the general from.

The second point is the development of a model that views planning as a process that combines developing an initial plan, executing the plan, fixing failures as they arise during execution, evaluating, modifying, and storing the plan for later reuse. We agree, of course, that some planning precedes plan execution, but we believe that in many realistic domains planning and execution should be interleaved, i.e. that execution should start before a complete plan is achieved, that unforeseen plan failures should be handled by replanning on the spot, and, most importantly, that the results of evaluating the execution of plan fragments and plans should be stored with the plans for future use. In other words, it is this integration of planning and execution that enables a planner to improve its performance in future planning episodes by taking advantage of information gained during past executions and subsequent evaluations.

5 EXTENDING CBR WITH GENETIC TECHNIQUES

Genetic algorithms [Goldberg89] simulate mechanisms of biological evolution, and can benefit case-based approaches in a number of ways through the application of their genetic operations. The classic operators used by genetic algorithms are mutation, crossover, and reproduction. Applied to CBR, mutation replaces one or more action components of a case by another such component with a given probability; and crossover takes parts of two different cases and creates a new case with previously unencountered action sequences. The new case along with the old ones are now subjected to the reproduction operator. The reproduction operator selects cases from the library to form its next generation, controlling the size of the library by only keeping strong cases. In the following sections we develop these ideas further and combine them with CBR to form a general learning approach for a dynamic environment.

5.1 GENETIC ALGORITHMS FOR CASES

For a CBRS to learn, it must acquire and incorporate new cases into the existing case library. A case library may be organized into manageable structures as in Dynamic Memory Theory, or, as in our case, a flat structure. The important point is that new cases are placed in the appropriate place in the case library, whatever the organizational strategy. New cases are generated by three methods: hand-coded cases from experts in the domain, adapted cases produced by the reasoner through major modifications to already existing cases, or newly developed cases recognized or produced by the reasoner that reflect the current environment better than any existing case in the library. One might believe that experts are better at generating cases, that represent solutions in a problem space. However, as [Bradtke88] points out: 'the effectiveness of a case-base is the number of unique problem states underlying the case-based encoding', and humans tend to cluster cases about

different points in a problem space rather then providing unique points. Therefore, cases developed by experts are more redundant than might be initially expected. The reasoner too can produce the same effect, when it modifies a case to such an extent that it resembles another case, and the new case is then added to the case library. Two important questions should be asked before the addition is made: does this case already exist or is it similar to other cases in the library; and if so, is there a need to add the new case at all? If the reasoner cannot find a case to solve a current problem, it can easily ask a human expert for a solution and record it. The new case produced by the expert is different than the expert generated initial cases, because in this instance the expert is asked to solve a specific problem, not provide a general answer. It is much easier for the expert to produce the exact case rather than a general one because the focus the problem provides aids the expert in producing a solution.

Each method of case generation described produces a new case, but how will the addition of the case affect the existing cases in the library? Will some case not be retrieved any more? Are there two opposing solutions to a problem?

Before considering these questions there is another factor that can affect the relationships between the cases in the library. In a dynamic environment, the environmental conditions are continually changing. Solutions to past problems may no longer fix similar problem in the present. The case library in effect is growing old. Redundant and possibly harmful cases need to be removed. To perform this operation, the invalid cases must be detected; but how? One prerequisite for a method to aid in the detection of invalid cases is to record what aspects of the environment the case interacts with are changing. This is not the only prerequisite but a good starting point; one must notice changes before any actions can be done in response to them. A case must be able to handle three forms of environmental changes: an anomaly (freak occurrence), an unforeseen but lasting change in the environment, and a probabilistic failure (predictable).

Figure 5.1 helps to illustrate the different forms of cases that can arise in the case library as a result of environmental changes. Unique cases and those that are loosely similar are desirable in the case library because they provide solutions to different problems. Redundant and harmful cases are undesirable and need to be removed from the case library because they do not provide any new solutions and may provide an incorrect solution.

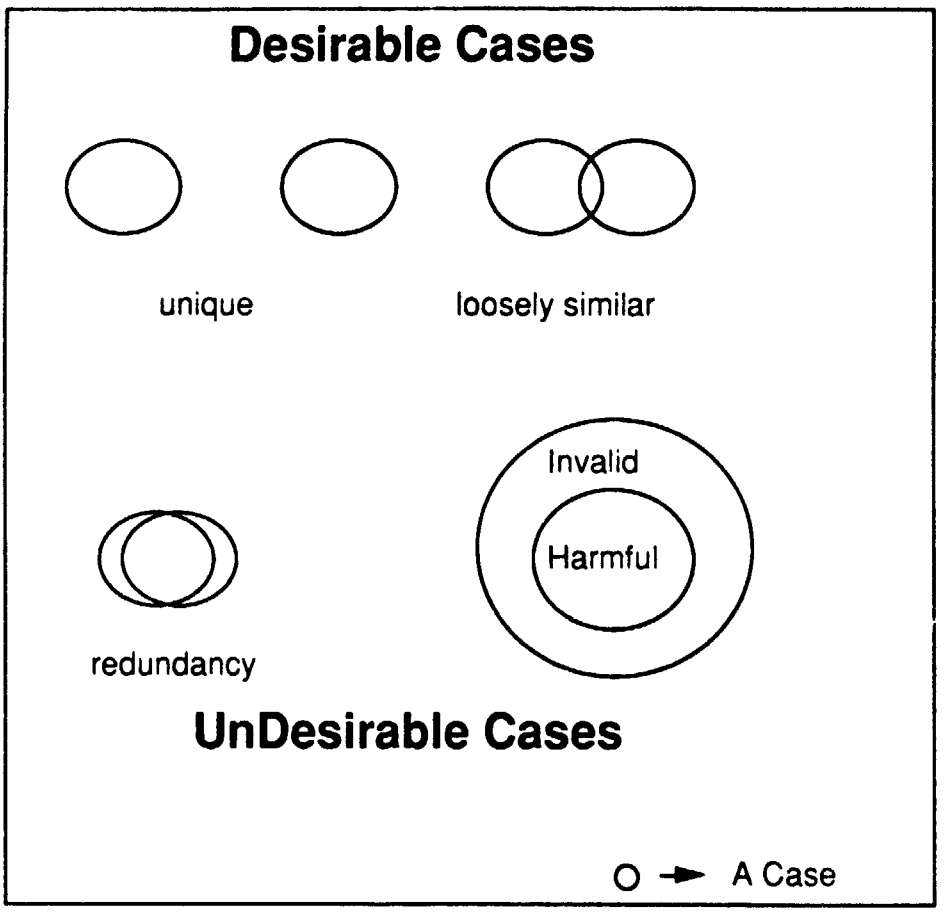


Figure 5.1 Case Relationships

To perform the detection and removal functions, and to handle the entropic graying of the case library, this thesis presents the use of genetic algorithms and specifically the application of the reproduction operator. Cases, like all resources, follow the second law of Thermodynamics. Cases move from being usable to unusable as the entropy of the system

increases with each addition of a case; genetic algorithms can determine and remove unusable cases while maintaining a strong set of cases for use in future generations.

5.1.1 Introduction to Genetic Algorithms

The father of genetic algorithms (search based on the mechanisms of natural selection and genetics) is John Holland. The goals of genetic algorithms (GAs) are quite simple, to perform search in an optimum, yet efficient, manner. Many methods of computation may produce the optimal performance to a problem but they are computationally expensive. There is a trade off between performance and optimality; efficiency is the right balance between the two.

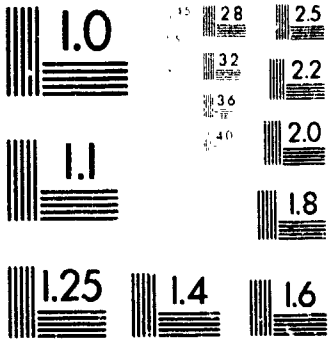
Goldberg notes [Goldberg89] that GAs differ from traditional search methods in the following ways:

1. GAs use a coding of the parameters, not the parameters themselves.
2. GAs search from a population of points, not a single point.
3. GAs use an objective payoff function not derived from auxiliary knowledge.
4. GAs use probabilistic not deterministic transition rules.

These observations are explained in later sections.

Before proceeding with an example, we define some of the terminology used by GAs One of the basic structures that GAs process is called a chromosome string, string for short. A chromosome string is composed of a fixed number of alleles; an allele represents a binary feature or detector that is built from a binary alphabet $V = \{0, 1\}$. For example, a sample string A of five alleles, or size five, is: $A = 01101$. An allele has a bit position in the string, the first bit being position one, the second bit position two, and so on. A population or generation of strings is a collection of n strings.

2 OF/DE 2



Another structure the GAs process is called a schema. A schema is used to describe a class or generalization of individual strings in a population. A schema is built from an alphabet V = {0, 1, *}. The symbol * is used as a wild card. Therefore, the schema H = *11*0, can be used to represent the strings 11100, 01100, 11110, and 01110 because each string matches H at all bit positions.

The following example taken from [Goldberg89], demonstrates the simple mechanism of GAs. The goal is to maximize the function $f(x) = x^2$ on the interval [0,31]. To apply GAs, one must first code the parameter x into a finite length string. For example, if x was 24, using a binary string representation, x would be encoded as 11000. The 1's and 0's are the different values each allele in the string can assume. Rather than use a single string to start the search with, GAs use a population of chromosomes. If the population size[8] was to be 4, one could start out with an initial chromosome population of 01101, 11000, 01000, and 10011 representing the numbers 13, 24, 8, and 21. The final item required by a GA is a fitness function in order to provide a strength measure of any string. For this example, the fitness function of a string is equal to the square of the string's value, e.g. if the string represented the number 13, its fitness value would be 169. Using these values we construct an initial population and its associated data. Table 5.1 shows the results of these calculations. The **% Total** represents what portion of the total population fitness, the fitness of an individual string contributes. The **Fitness** value in the table is the value of the binary string squared.

A basic GA is composed of three primary genetic operators: reproduction, crossover, and mutation. In this chapter we are mainly concerned with the reproduction operator. Reproduction is the process of selecting strings from the current population to be reproduced into its next generation. The bias for selecting strings is based on their fitness

[8] Population sizes are selected by the user. Often population sizes of 30-100 chromosomes are best [Grefenstette86].

values. The strong ones are more likely to be selected for the next generation than the weaker ones. This is a version of Darwin's survival of the fittest. To do the selection, a weighed roulette wheel is constructed out of the fitness values of the strings, shown in figure 5.2.

No.	String	Fitness	% Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Table 5.1 First String Population

The wheel is spun 4 times to select 4 chromosomes for the next population, and the winning chromosome at each spin is added to the next generation. Using the fact that chromosome-2 has 49.2% of the total fitness value of the entire population ($\frac{576}{1170}$), in four spins probability theory shows that we can expect[9] two or more chromosome-2's with probability ($\frac{11}{16}$); a very good chance it will reproduce twice. The effect demonstrated here is that strong chromosomes will be reproduced into the next generation. After a new generation is created the process of selecting 4 new chromosomes is repeated for another generation, again and again, generating new populations at each iteration until a final stable population of strong chromosomes is established. These chromosomes, usually different

[9] Calculation done as if chromosome-2 took up 50% of the total fitness sum.

but possibly the same, represent the approximate answer(s) to the search question of what chromosome maximizes the function $f(x) = x^2$?

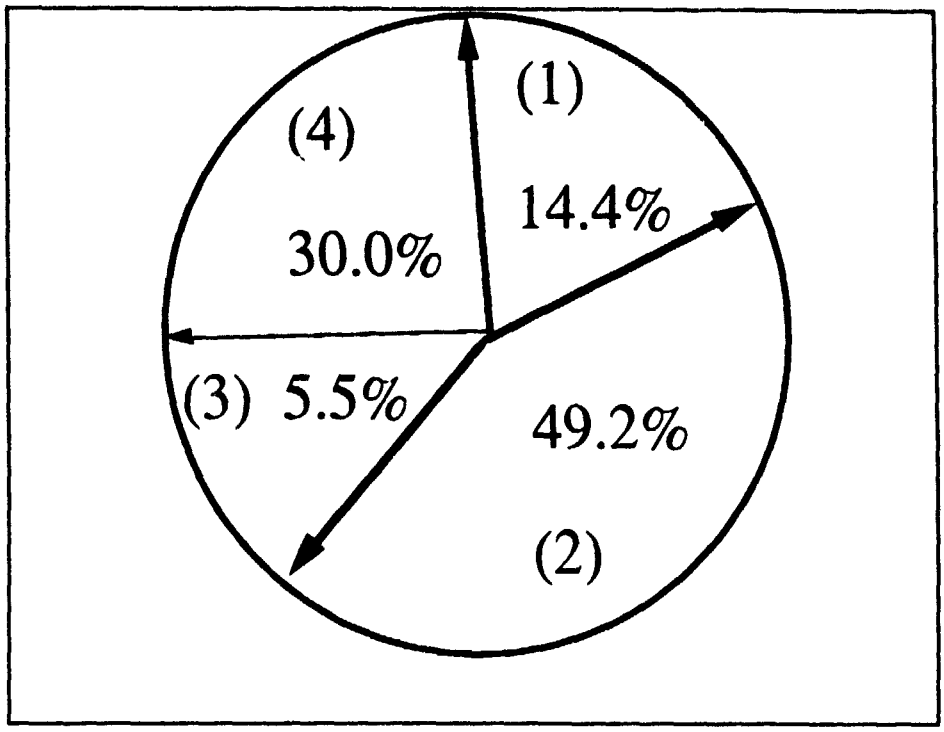


Figure 5.2 Reproduction Roulette Wheel

From this simple example, it is easy to see that the best achievable population is one with the strongest chromosome from the initial population duplicated 4 times, filling the entire population. If the strongest chromosome in the initial population is not close to the best answer, but is the strongest of the population, how can GAs be used to achieve a stronger result? GAs solve this problem by using other genetic operators to alter the alleles of the existing chromosomes to produce new, hopefully stronger, ones. There are two traditional genetic operators to perform chromosome modification in collaboration with reproduction: mutation and crossover.

Mutation is the process of randomly selecting an allele from a string and changing it to another legal value. In the example below, this is achieved by changing a zero to a one, or a one to a zero, in a string, shown as follows:

MUTATION: of 3rd bit

Chromosome A: fitness = 19

1	0	0	1	1

Chromosome A': fitness = 23

1	0	1	1	1

This example demonstrates that the effect of mutation is to alter the fitness value of the string; this is because, in this case, the fitness value is a function of the number of ones found in the string. A problem one might ask is: how does this make the population stronger as a whole? The process can make an individual chromosome either stronger or weaker.

To combat this problem the reproduction operator is used. Reproduction is the process of selecting the strongest individual from the current population for the next. By selecting only the strongest chromosomes, the probability of getting a stronger schema as a result of the mutation is increased. The proof of this will be shown later. If the mutation produces a weaker chromosome, that chromosome will have a reduced probability of being reproduced in subsequent generations due to its weakness with respect to other chromosomes. On the other hand, if the mutation produces a stronger chromosome, it will have a better probability of being selected for future populations because of its increased strength.

Crossover is the process of randomly selecting an allele position from two strings and connecting the first half of each chromosome with the last half of the other, producing two new chromosomes. The process is a follows:

Crossover: at 3rd bit

Chromosome A: fitness = 21

1	0	0	1	1

Chromosome B: fitness = 26

1	1	0	1	0

Chromosome A': fitness = 18

1	0	0	1	0

Chromosome B': fitness = 27

1	1	0	1	1

This example shows that the effect of crossover is to also alter the fitness value of the string; this is because, in this example, the fitness function is based on the number of ones found in the string. Others GAs, such as [DeJong89], use a fitness function that relies on the relationships between the allele, while others, such as [Axelrod87], use environmental feedback as part of the fitness function. Again, the process of crossover can make individual chromosomes stronger or weaker, but how does this make the population stronger as a whole? By the same argument used in the example of mutation, the reproduction operator is the key. Selecting strong chromosomes ensures that crossover is only performed on the best chromosomes of the population. Selecting the strong

chromosomes permits chromosomes to get even stronger. If a weaker chromosome is generated, it dies out in the following generations because it has a reduced probability of being selected over the strong ones during reproduction.

The question to answer next is: how can a GA and its operators be used to help CBR?

5.1.1.1 Case vs Chromosome

The notion of using GAs with cases differs from the traditional use of GAs in three respects. GAs typically encode the data as fixed length strings using the symbols 0 and 1 as the language of the alleles. This makes for ease of use in applying genetic operators; however, the expressive power is not enough for symbolic problem solving used by many expert systems today. Control systems that monitor temperatures of devices need to consider absolute temperature values such as 100 C and fuzzy temperatures such as warm, hot, or cool. It is a difficult task to encode these values and all other parameters that contribute to a problem's description into a fixed length string of zeros and ones. A language of two symbols can only encode two possibilities for any one allele; a temperature could have one thousand different possibilities, therefore, a two symbol language is inadequate. Case structures, for example frames, use a rich symbol notation; thus, frames can represent a considerably larger number of values than that of a single allele. We increase the descriptive power of cases by increasing the alphabet, at the cost of increasing complexity.

This distinction is important; cases do not necessarily have to have a fixed number of frames. A checker strategy could be considered as a case where each frame provides the next game move of the strategy. As a strategy grows and shrinks, so too does the number of frames in the case. Strings have a fixed size, so expansion or contraction of the number of alleles is impossible. Having dynamic cases expands the number of representations the case can form;thus, increasing the potential use of GAs. Often a problem produces more and more recognizable symptoms as it is better understood. Having dynamic cases

provides a means of easily adding information to them. Case frames also require a strength field. Figure 5.3 outlines a general case frame structure to support a GA. Making the strength of a case a function of the strength of each frame, the addition of new frames causes no problem for the concept of strength .

The final distinction between a GA for cases rather than strings is a case is symbolic while a string is subsymbolic. It is easy to perform mutations and crossover on strings with 0 and 1, but the language is restrictive. As will be described later, genetic operators such as crossover and mutation can also be performed on cases. The process of using GAs with symbolic rules has been suggested [Bickel87, Antonisse87] and performed using bit strings to represent symbolic rules [Holland86b], but never have GAs been used with cases. The power of GAs makes them very attractive when applied to cases.

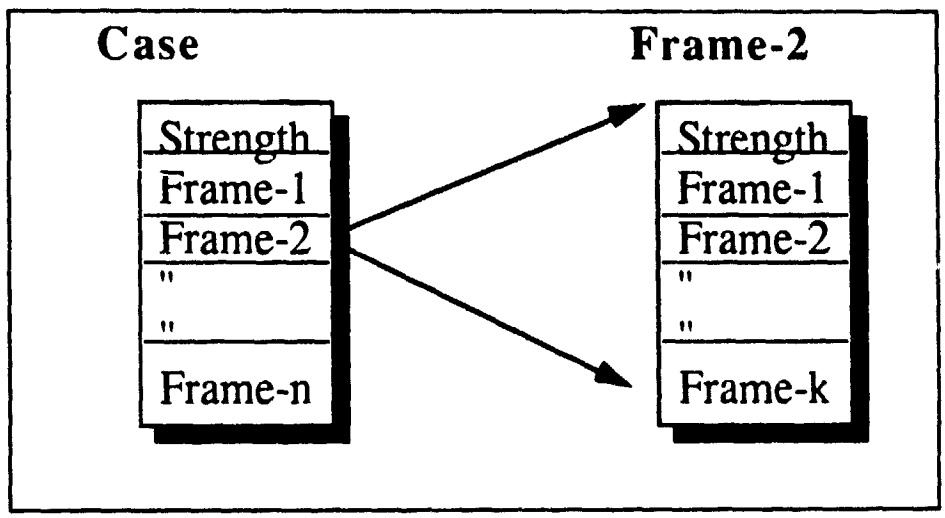


Figure 5.3 Case Structure Supporting Genetic Algorithms

5.1.1.1.1 Case Reproduction Operator

A case reproduction operator (CRO) must probabilistically select the strongest cases from the case library to build a new case library. This is equivalent to the traditional

reproduction operator selecting the strongest chromosomes to be placed in the next population. A case is thus analogous to a chromosome; a case has a collection of frames, a chromosome has a collection of alleles; a case's frames are ordered, a chromosome's alleles are ordered; a case is valid or invalid, a chromosome is strong or weak. A chromosome has a strength associated with it, so too must a case. The validity or strength (fitness value) of a case refers to its usefulness[10] in the current environment. Thus, a method of computing a case's strength must be available. In addition to the normal case structure, a strength field must be added to the case and each of its internal structures. A function to compute the strength of the case is described later. The important point here is that we compute the strength of a case as function of its internal structures (frames), similar to the method of computing the strength of a chromosome as a function of its alleles.

5.1.1.2 Credit Apportionment for Cases

The fact that strength values can be computed is not enough to ensure the best possible library of cases. Only when the environment can react to the strength of a case, either to increase or decrease it, can a strong library of cases be obtained using the genetic techniques. The desired interaction can be achieved with a modified version of an apportionment of credit algorithm called the bucket brigade algorithm [Holland86a, Goldberg89]. The apportionment of credit problem is to determine which cases should be chosen for the case library, out of many available cases at a given step, say time t, in order to obtain a desirable case library at time $t + n$.

Two algorithms, shown below, are used to adapt the bucket brigade algorithm for cases. The *Adapted Bucket Brigade Auction Algorithm for Cases* is used for the retrieval of the best case, using an auction approach on retrieved cases from the case library that match the current environment. The *Adapted Bucket Brigade Clearinghouse Algorithm for*

[10] Usefulness is a function of the case's validity; does the case provide an acceptable solution or does it cause failures due to returning the wrong solution.

Cases is used for the awarding of credit, using a clearinghouse approach on the best case retrieved based on its performance in the environment.

Adapted Bucket Brigade Auction Algorithm for Cases

```
Input:    Case-Library, Current-Environment, Existence-Tax
Output:   Best-Case, Bid-Total
Begin
    {Find the best matching cases using any normal CBR retrieval method}
    candidate-cases := Find-Matching-Cases( Case-Library, Current-Environment)
    Best-Case := nil
    Bid-Total := 0
    {The bid of the case is a function of its strength. Use the bid to determine the best
            case}
    For each candidate-case Do
            If (Bid( Best-Case) < Bid(current-case) Then
                    Best-Case := current-case
            Bid-Total := Bid-Total + Bid(current-case)
    {Reduce each case's strength, that was returned as a candidate case, by its bid.}
    For each candidate-case Do
            current-case.Strength := current-case.Strength - Bid(current-case)
    { Collect an existence Tax from the Cases }
    For those cases in the Case-Library but not in the candidate-cases Do
            current-case.Strength := current-case.Strength - Existence-Tax
End
```

Adapted Bucket Brigade Clearinghouse Algorithm for Cases

```
Input:      Best-Case, Bid-Total, Action, Reward-Constant, Penalty-Constant
Output:     Best-Case (updated)
Begin
   Case Evaluation of
            { The case was used successfully so add a reward and the sum of
                  all other bids to its strength}
      success:      Best-Case.Strength := Best-Case.Strength + Bid-Total +
                                                Reward-Constant
            { The case failed so its strength is penalized  }
      failure:      Best-Case.Strength := Best-Case.Strength -
                                                Penalty-Constant
            { The case was not an out right success but it was not a failure so
                  do not reward it but add the  other bids to its strength}
      near-success: Best-Case.Strength := Best-Case.Strength + Bid-Total
End
```

The *Auction* algorithm contains four parts: candidate case selection, best case selection, bid decrease, and existence tax collection. First the algorithm selects the candidate cases that match the input environment from the case library using traditional CBR case retrieval. Next, the best case is selected from the candidate cases by finding the candidate case with the largest bid. The bid is a function of the current strength (fitness) of the case. The total sum of all bids is recorded for the *Clearinghouse* algorithm. Next, all candidate cases have their bid amount subtracted from their strengths, and the best case and bid total are returned. The effect of this algorithm is that although all retrieved similar cases will have their strength reduced, it is only the best case that has any possibility of regaining that

strength, and more, in the *Clearinghouse* algorithm. Finally an existence tax is taken from all cases not selected as candidates. The existence tax is typically a very small number. The reason for this subtraction is that over time cases that are never used will have their strength reduced to zero, or below some threshold. These cases can then be removed from the library because they are never used - they can be forgotten. In a sea of cases, some cases will have an increase in strength while the others will have a decrease over time.

The *Clearinghouse* algorithm is used to update the best case that was returned by the *Auction* algorithm, but only after is has been used in the environment and evaluated as to its usefulness. Three possible actions can be performed by the algorithm as a result of the case's evaluation: reward, penalty, and compensation. A case is rewarded - it gets its bid, all other candidate cases' bids, and a reward value added to its strength - if its use in the environment is evaluated as being successful. A case is penalized - it gets a penalty value subtracted from its strength - if its use in the environment is evaluated as being a failure. A case is compensated - it gets its bid and all other candidate cases' bids but no reward value added to its strength - if its use in the environment is evaluated as being acceptable, but not outstanding. This algorithm drives the case's strength up or down depending on its evaluation in the environment. Now an algorithm is needed to make use of the strength of a case to maintain a strong, healthy case library - the Conservative Reproduction Operator.

5.2 CONSERVATIVE REPRODUCTION

Two of the pieces are now available to help satisfy the goals of detecting redundant, invalid, and harmful cases within the case library: associating strength with cases, and a method of assigning credit to the cases. The solution to the main goal of maintaining a minimal set of strong cases in a fixed size case library can now be achieved with a Conservative Reproduction Operator.

The traditional reproduction operator, as we have seen, selects a new population of chromosomes from the strongest chromosomes of the previous population; however, this

leads to the possibility of the same strong chromosome being selected more than once for the next population. As was seen in the short example of four chromosomes, the chromosomes 11000 had a 0.6875 probability of occurring two or more times in the next generation. This would lead to case redundancy if the same reproduction approach were used for a population of cases. A conservative approach, insuring a minimum redundancy of cases, is required.

An adapted reproduction operator, called the *Conservative Reproduction Operator for Cases* (CROC), ensures that the next generation of the case library contains a nonredundant majority of strong cases from the past generation, but still allows for the introduction of new cases that have yet to be proven or disproven.

Conservative Reproduction Operator For Cases

```
Input:    Case-Library, Minimum-Acceptable-Strength,
          Maximum-Number-Of-Cases, New-Case-Percentage
Output    New-Case-Library
Begin
    number-of-cases := 0
    New-Case-Library := Empty-Case-Library
    { Sort the cases into descending order based on their strength}
    ordered-case-library := Order-Case-Descending-In-Strength(Case-Library)
    { Select the maximum number of cases allowable for the next generation whose
          strength is above an acceptable value }
    current-case := Next( ordered-case-library)
    While ((current-case.Strength >= Minimum-Acceptable-Strength) AND
          (number-of-cases <= Maximum-Number-Of-Cases)) Do
```

```
            Add( current-case, New-Case-Library)
            current-case := Next( ordered-case-library)
            number-of-cases := number-of-cases + 1
        End
        { Normal CBR adaptation methods could be performed here if so desired}
        {Apply other genetic operators to probabilistically selected (roulette wheel) cases,
            adding a given percentage of new cases into the case library - see the next
            section for the details.  }
        For (New-Case-Library * New-Case-Percentage) Times Do
            new-case := Probabilistically-Select-Case(ordered-case-library)
            new-case := Apply-Other-Genetic-Operators-To( new-case)
            Add( new-case, New-Case-Library)
        End
    End
```

The CROC contains three parts: case strength ordering, case selection, and new case generation. Memory is always a fixed resource; therefore, as input to the algorithm an upper bound on the number of cases in the library is provided. This is not an absolute upper bound because a percentage of that number in new cases may be added to the library, but for now it is reasonable to think of it as an upper bound. Another parameter provided to the algorithm is a minimum acceptable strength threshold value. This determines the minimum strength of a case that is permissible in the new case library. Initially the case library is ordered by descending case strength. From this list, the maximum allowable (say n) strongest cases are selected and added to the new case library. One might ask how the number of cases grows beyond n? The answer is that new cases are added to the case library through the application of other genetic operators (see the next section) or through normal CBR case adaptation and repair. This conservative approach assures that from one

generation to another, a constant number of strong cases is always reproduced. It does not, however, select the same cases from one generation to another. It is the performance of a case that indicates its acceptability for reproduction into the next generation. After initial case selection, a percentage, *x of n*, new cases are added to the new case library. This process is discussed in detail in the next section. Finally, the new case library is returned. From an old case library a new one is produced with the weak cases weeded out, a small portion of new cases introduced, and a solid base of strong cases remaining. The library can now be considered a self cleaning entity rather than a closed box susceptible to the forces of entropy. When memory is fixed, this capability is even more necessary and beneficial, ensuring that the case library integrity is maintained.

The conservative approach ensures that only the strong cases are kept from one library generation to another, yet it avoids duplication and guarantees the consistency of the library between generations. The operator may also be considered a genetic algorithm; it uses the other operators for the generation of new cases. Chapter 6 provides an example of using the algorithm, along with other genetic operators for producing new cases, in the domain of checkers.

5.3 GENERATING CASES USING GENETIC OPERATORS

By having a method of removing inappropriate cases, an automated method of generating cases is possible without the requirement of an inordinate amount of verification by a human expert. However, new cases should not be exceedingly novel, yet they should deviate enough from existing cases to potentially provide new coverage of the problem space. When one is solving problems using cases of past solutions, the solutions can tend to become repetitive - stuck in a rut. Even the strongest of adaptive methods can only get so far from the original case using standard methods of adaptation and analogy. What is desired is a [illegible] of altering a case in a novel way so as to, more often than not, produce a case that is [illegible] of the normal adaptations of the case. We are looking for novel cases,

loosely based on the originals, but not as restrained as using other traditional methods. If the problem of removing poor cases is solved and a novel case is not acceptable, the system would be able to garbage collect it. Good novel cases, however, will remain in the case library and improve its overall quality.

Edward De Bono [DeBono87] describes this notion when he talks about creative thinking, especially in children:

> 'There is no doubt that children can come up with very original solutions to a problem. It is partly because they are uninhibited and unfettered by too much knowledge of how things work; and partly because they tend to think in function concepts. The main reason, however, is that if you do not know the established approach to a situation you have a very good chance of coming up with an original approach.'

In adults, or computer models, this notion can be achieved by attempting to solve a problem by not following the normal flow of thinking. In figure 5.4 there are two pictures. The first picture symbolizes thinking as a flow down an existing path; not noticing that there are secondary paths. The second picture symbolizes thinking taking one of the tributaries of the normal flow. This could be an excellent diversion in hindsight, but it is extremely hard to notice when one's thinking is constrained by what is known. Novel thinking is the ability to notice and jump into those tributaries when originality is desired.

Creative thinking deviates from the normal sterile thinking flow, yet it still has a connection to it, making this form of thinking highly plausible, but sometimes not practical. Traditional case adaptation methods follow the flow of sterile thinking because once the adaptation method is formed it is very difficult to alter. What is required is a methodology that can push the adaptation method into the creative thinking tributary; but only push into a flow that is logically possible from the current thinking flow. Applying genetic operators to cases is an excellent method for producing new cases that have been energized to step into the tributaries of adaptation.

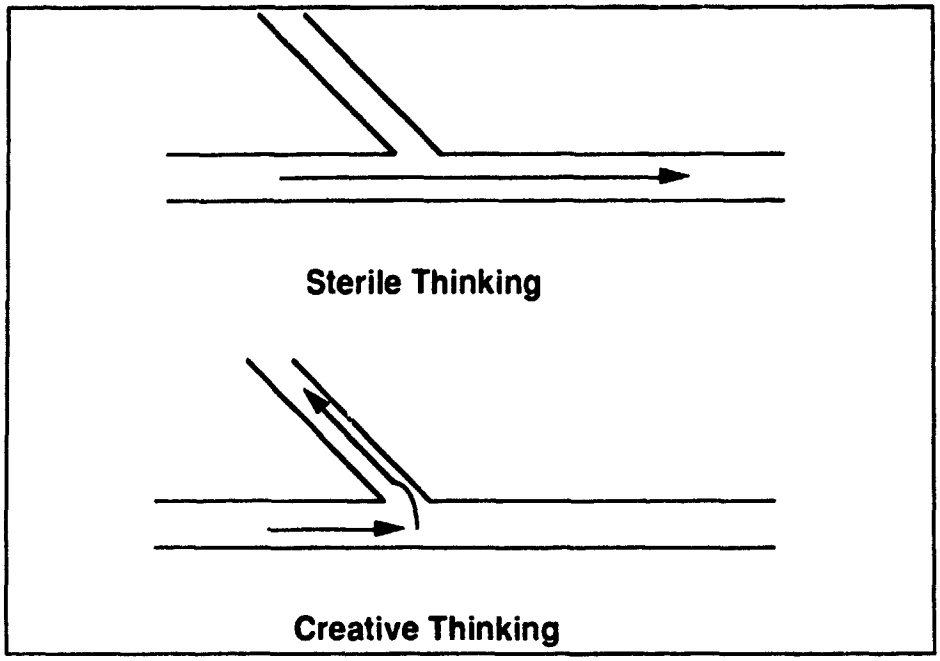


Figure 5.4 Thinking Flows

5.3.1 Genetic Case Operators

For the adaptation of cases using a genetic algorithm, this thesis presents five genetic operators:

1. Reproduction
2. Mutation
3. Crossover
4. Division
5. Connection

The reproduction operator selects strong cases from the current case library for reproduction into the next generation of the library. The selection criterion is based on a fitness value (strength) of a case. Strong cases have a high fitness value. The reproduction operator was described in the last section. However, the reproduction operator is only the first piece of a genetic algorithm; after selecting strong cases for the new library generation, the remaining genetic operators are performed on selected strong cases providing a set of

possibly novel cases for the next generation of the case library. The remaining genetic operators for cases are described in detail in the following sections.

5.3.1.1 Mutation

Mutation of a case is the process of changing the contents of one or more of the case's frames from its current value to another legal value. The input and output of the operator is a single case. Associated with the decision to mutate a case are two probabilities:

Pc = Probability of mutating the case

Pf = Probability of mutating a frame in the case

To make the operator more selective in the mutation it performs, mutations have a greater probability of occurring on frames of weak strength. Remembering that a case's strength is a function of the individual strengths of its frames, a case can be considered strong but have weak parts. The mutation operator exploits this knowledge in an attempt to mutate the frames that are in most need of change; if the frame is already weak, a mutation can not make it much weaker and has a good chance of making it stronger.

The mutation algorithm is as follows:

Mutation Algorithm For Cases

```
Input:    Case, Pc, Pf
Output:   Mutated-Case
Begin
    { Set up an empty return Case}
    Mutated-case := NewEmptyCase
    { Decide if one should mutate the case}
    random-number := RandomNumberBetween(0.0, 1.0)
```

```
If (random-number <= Pc) Then
Begin
        For all frames in Case Do
          random-number := RandomNumberBetween(0.0,1.0)
          { Decide if one should mutate frame, repeat for each frame}
          { Weak frames increase the chance of deciding to be mutated}
          If (( random-number + Case.currentFrame.Strength) <= Pf) Then
            { Mutate frame with a new legal value for it }
            Mutated-Case.currentFrame := OtherLegalValue(Case.currentFrame)
        FndFor
Else
Begin
        { Return the original case, no mutation performed}
        Mutated-Case := Case
End
End
```

For the above algorithm the strength of each frame is assumed to be in the range of 0.0 to 1.0. Initially, a decision is made if the case, in general, is to be mutated. This decision is made probabilistically by randomly selecting a number between 0.0 and 1.0 and comparing it to the number Pc. If the random number is less than or equal to Pc the case now has the possibility of having its frames mutated, if not, the original case is returned. Each frame is then considered for mutation. Again the decision to mutate a frame is made probabilistically by randomly selecting a number between 0.0 and 1.0, increasing that number by the case's strength, and comparing the result to the number Pf. If the random number is less than or equal to Pf, the frame is mutated to another legal value , if not, the original frame is added to the mutated case. The addition of the frame's strength to the

random number will decrease the probability of mutation greatly for strong frames and make the mutation of weak frames more probable. The process continues for all frames, producing a mutated case consisting of both original frames from the old case as well a mutated ones.

As an example, consider a case that encodes a move strategy (for white) from a 4x4 game of checkers, each frame in the case consisting of a legal white move and the black opponent's move.

Original Case:

White A1 - B2
Black D4 - C3
White B2 - D4 Black gets jumped
Black D2 - C3 (Removing white's piece)
White D4 - B2 Black gets jumped and loses the game
"

Mutated Case:

White A1 - B2
Black D4 - C3
White B2 - D4 Black gets jumped
Black D2 - C1*
White D4 - C3 White can not jump so only moves to square c3
"

The case mutation (marked with *) stops white from jumping a second black piece and lets the black piece move in for a king. The mutated case is a better strategy than the original case both in attack and defence; less pieces are lost and black's position is improved. A mutated case frame may cause repercussions on other frames - as seen by the example, where white's next move must be altered to make it legal - after the frame is mutated; however, this can be handled, and in cases where frames have no relationship between one another this is not a problem. Mutation of a case may cause a reduction in the case's potential, but reproduction of the case library using the reproduction operator will remove weaker cases over time and keep the library strong.

5.3.1.2 Crossover

Crossover of cases is the process of merging a collection of the first half of a case's frames with that of the second half of another case's frames. The crossover point in the two cases is not necessarily the middle frame of the cases, but a randomly selected frame number between 1 and the maximum number of frames in the cases. The inputs to the operator are two cases. The outputs are also two cases which are the crossover of the two input cases. Associated with the decision to crossover cases are two probabilities:

Pc = Probability of crossover occurring on the cases

Pf = Probability of crossing over on a particular frame

To make the operator more selective in the crossover it performs, crossover has a greater probability of occurring on a frame that has weak strength. Remembering that a case's strength is a function of the individual strength of its frames, a case can be considered strong but have weak parts to it. The crossover operator exploits this knowledge in an attempt to crossover at a frame in a case that is in most need of change; if the frame is already weak, a crossover at that point can not make it much weaker and has a good chance at making the case stronger.

The crossover algorithm is as follows:

Crossover Algorithm For Cases

Input: Case1, Case2, Pc, Pf

Output: Crossover-Case1, Crossover-Case2

Begin

{ Set up an empty return Case}

Crossover-Case1 := NewEmptyCase

```
Crossover-Case2 := NewEmptyCase
{ Decide if one should mutate the case}
random-number := RandomNumberBetween(0.0, 1.0)
If (random-number <= Pc) Then
Begin
        crossover-point := 1
        crossover-point-found := false
        { Decide if one should crossover at a  frame, repeat for each frame}
        { Weak frames do not reduce the chance of deciding to be the crossover
                point as much as strong frames do}
        While (Not ((crossover-point-found)  or
                (crossover-point = Size(case)))) Do
        Begin
                random-number := RandomNumberBetween(0.0,1.0)
                If (( random-number + Case.frame[crossover-point].Strength) <=
                                Pf) Then
                        crossover-point-found := true
                Else
                        crossover-point := crossover-point + 1
        End {While}
        Crossover-Case1 := Concatenated (Frames (Case1, 1, crossover-point),
                                Frames (Case2, crossover-point + 1, Size(Case2)))
        Crossover-Case2 := Concatenated (Frames (Case2, 1, crossover-point),
                                Frames (Case1, crossover-point + 1, Size(Case1)))
End
Else
```

```
    Begin
        { Return the original cases, no crossover performed}
        Crossover-Case1 := Case1
        Crossover-Case2 := Case2
    End
End
```

For the above algorithm the strength of each frame is assumed to be in the range of 0.0 to 1.0. Initially, a decision is made if the cases, in general, are to be crossed over. This decision is made probabilistically by randomly selecting a number between 0.0 and 1.0 and comparing it to the number Pc. If the random number is less than or equal to Pc the cases will have a chance at crossing over, if not, the original cases are returned. Each frame in the first case is considered as the crossover point. Again the decision that a frame is to be the crossover point is made probabilistically by randomly selecting a number between 0.0 and 1.0, increasing that number by the frame's strength, and comparing the result to the number Pf. If th random number is less than or equal to Pf the frame is chosen as the crossover point, if not, the process is repeated for subsequent frames or until the last frame of the case is reached, in this case the last point is considered the crossover point[11]. The addition of the frame's strength to the random number will decrease the probability of crossover at a given point greatly for strong frames and make the weak frames more probable as a crossover point. After the crossover point is determined, the original cases, case1 and case2, are recombined at the crossover point forming two new cases which are returned as the results by the operator.

As an example, consider the cases that encode two move strategies from a 4x4 game of checkers, each frame in the case consisting of a legal white move and black opponent's move.

[11] When the crossover point is the last frame in a case the crossover of the cases returns the original cases.

Original Case1:

(White A3-B4)
(Black D2-C1)
(White A1-B2)*
(Black D4-C3) Bad move, black will be jumped(Should jump)
(White B4-D2)
(Black C1-A3) White loses a man
"
(Black goes on to lose)

Original Case2:

(White A1-B2)
(Black D2-C1)
(White A3-B4)*
(Black C1-A3) White is jumped
(White B4-C3)
(Black D4-B2) White is jumped, game over

Crossover Case1':

(White A3-B4)
(Black D2-C1)
(White A1-B2)
(Black C1-A3) White is jumped
(White B4-C3)
(Black D4-B2) White is jumped, game over

Crossing over the first part of case1 with the second part of case2 at the crossover point[12] (marked with *) stops black from losing a piece, as in case1, and in fact, the new case goes on to win the game as in case2. The crossover case is a better strategy than the original case1 both in attack and defence; no pieces are lost and black wins the game. Crossover of cases may cause a decrease in the case's potential, but reproduction of the case library using the reproduction operator will remove weaker cases over time and keep the library strong.

[12] The crossover point for a checker strategy is chosen at a point that is weak in the cases and the board layout is also the same at the crossover point.

5.3.1.3 Division

Division of a case is the process of splitting a case at a frame producing two smaller cases. The input to the operator is a single case and the output are two subcases of the original case. Associated with the decision to divide a case are two probabilities:

Pc = Probability of dividing the case

Pf = Probability of dividing at a frame in the case

To make the operator more selective in the division it performs, divisions have a greater probability of occurring on frames that have weak strength. Remembering that a case's strength is a function of the individual strength of its frames, a case can be considered strong but have weak parts to it. The division operator exploits this knowledge in an attempt to divide a case at a frame that is in most need of change; if the frame is already weak, a division can not make it much weaker and has a good chance at making it stronger.

The division algorithm is as follows:

Division Algorithm For Cases

```
Input:    Case, Pc, Pf
Output:   Case1, Case2
Begin
    { Set up empty return Cases}
    Case1 := NewEmptyCase
    Case2 := NewEmptyCase
    { Decide if one should divide the case}
    random-number := RandomNumberBetween(0.0, 1.0)
    If (random-number <= Pc) Then
```

```
Begin
        division-point := 1
        division-point-found := false
        { Decide if one should divide at a  frame, repeat for each frame}
        { Weak frames do not reduce the chance of deciding to be divided as much
                as strong cases do}
        While (Not ((division-point-found)  or
                (division-point = Size(case)))) Do
        Begin
                random-number := RandomNumberBetween(0.0,1.0)
                If (( random-number + Case.frame[division-point].Strength) <= Pf)
                Then
                        division-point-found := true
                Else
                        division-point := crossover-point + 1
        End
        Case1 := Frames (Case, 1, division-point - 1)
        Case2 := Frames (Case, division-point + 1, Size(Case) )
    End
    Else
    Begin
        { Return the original case, no mutation performed}
        Case1 := Case
        Case2 := Case
    End
End
```

For the above algorithm, the strength of each frame is assumed to be in the range of 0.0 to 1.0. Initially, a decision is made if the case, in general, is to be divided. This decision is made probabilistically by randomly selecting a number between 0.0 and 1.0 and comparing it to the number Pc. If the random number is less than or equal to Pc the case will have a chance at being divided, if not, the original case is returned. Each frame in the case is then considered as a division point. As in previous operators, the decision to divide at a frame is made probabilistically by randomly selecting a number between 0.0 and 1.0, increasing that number by the case's strength, and comparing the result to the number Pf. If the random number is less than or equal to Pf the frame is chosen as the division point, if not, the process is repeated for the remainder of the frames or until the last frame in the case is reached. The addition of the frame's strength to the random number will decrease the probability of division greatly for strong frames and makes the division at weak frames more probable.

As an example, consider a case that encodes a move strategy (for white) from a 4x4 game of checkers, each frame in the case consisting of a legal white move and black opponent's move.

Original Case1:
(White A1 - B2)
(Black D4 - C3)*
(White B2 - D4) Black is jumped
(Black D2 - C1)
(White A3 - B2)
(Black C1-A3) White loses a man
"
(Black goes on to lose)

Division Case1:
(White A1 - B2)
"

Division Case2:
(Black D2 - C1)
(White A3 - B2)
(Black C1-A3) White loses a man
"

Division occurs at the weak part of the original case (marked with *) where black moves into a white jump situation. Splitting the case at this point forms two new shorter cases both without the bad move. Although smaller cases, each is considered a strong strategy, especially division case2, because it has no poor black moves in it. The divided cases are better strategies than the original case, no black pieces are lost. Division of a case may cause a decrease in the case's potential, but reproduction of the case library using the reproduction operator will remove weaker cases over time and keep the library strong.

5.3.1.4 Connection

Connection of cases is the process of joining two smaller cases producing a larger single case. Used in coordination with the division operator, the pair have the ability to produce strong new cases built from smaller subcases. The input to the operator is two cases and the output is one case (possibly empty). Associated with the decision to connect the cases are two probabilities:

Pc = Probability of joining the cases

Pf = Probability of joining the cases at a frame

To make the operator more selective in the connection it performs, connections have a greater probability of occurring at frames that have strong strength. Remember that a case's strength is a function of the individual strength of its frames. A case can be considered strong but have weak parts to it. The connection operator exploits this knowledge in an attempt to connect the cases at frames that have good strength; if the frame is already weak, a connection can not make it much stronger and has a good chance at making it worse.

The connection algorithm is as follows:

Connection Algorithm For Cases

```
Input:    Case1, Case2, Pc, Pf
Output:   Connected-Case
Begin
   { Set up an empty return Case}
   Connected-Case := NewEmptyCase
   { Decide if one should attempt connecting the cases}
   random-number := RandomNumberBetween(0.0, 1.0)
   If (random-number <= Pc) Then
   Begin
         { Strong frames increase the chance of deciding to connect the cases more
               than weak cases do}
         random-number := RandomNumberBetween(0.0,1.0)
         If (( random-number - Case1.frame[Size(Case1)].Strength) <= Pf)  And
         (( random-number - Case2.frame[1].Strength) <= Pf)
         Then
               Connected-Case:= Concatenated (Frames (Case1, 1, Size(Case1)),
                                    Frames (Case2, 1, Size(Case2)))
   End
```

For the above algorithm, the strength of each frame is assumed to be in the range of 0.0 to 1.0. Initially, a decision is made if the cases, in general, are to be connected This decision is made probabilistically by randomly selecting a number between 0.0 and 1.0 and comparing it to the number Pc. If the random number is less than or equal to Pc the cases will have a chance of being connected, if not, an empty case is returned. Only the last frame of case1 and the first frame of case2 are considered as a connection point. As in other operators, the decision to connect there is made probabilistically by randomly selecting a

number between 0.0 and 1.0, decreasing that number by the case's strength, and comparing the result to the number Pf. If the random number is less than or equal to Pf, the frame is chosen as the connection point, if not, the process is not permitted to proceed. The decrease of the frame's strength from the random number will increase the probability of connection greatly for strong frames and makes the connection at weak frames less probable.

As an example, consider a case that encodes a move strategy (for white) from a 4x4 game of checkers, each frame in the case consisting of a legal white move and black opponent's move.

Case1:

(White A1 - B2)
(Black D2 - C1)
(White B2 - C3)
(Black D4 - B2) White gets jumped
(White A3 - B4)
(Black B2 - A3)

Case2:

(White B4 - C3)
(Black A3 - B4)
(White C3 - D4)
(Black C1 - B2)
(White D4 - C3)
(Black B4 - D2) Wins game

Connected Case:

(White A1 - B2)
(Black D2 - C1)
(White B2 - C3)
(Black D4 - B2) White gets jumped
(White A3 - B4)
(Black B2 - A3)
(White B4 - C3)
(Black A3 - B4)
(White C3 - D4)
(Black C1 - B2)
(White D4 - C3)
(Black B4 - D2) Black wins game

Connection occurs when the end of one strong case can be attached to the start of another. As seen in the example, two smaller cases - possible created by the division operator - are combined into one strong complete case. The connection of cases forms a better strategy than the original cases. Connecting cases may cause a decrease in the case's potential, but reproduction of the case library using the reproduction operator will remove weaker cases over time and keep the library strong.

Connection must make certain assumptions about the domain. It must be able to know that it is allowable to connect two pieces at a given point. Although not as general as the other operators, it complements the division operator and produces novel cases.

5.4 WHY GENETIC ALGORITHMS WORK

The reason that a genetic algorithm works is a result of two classes of operators: the first class is a collection of the general operators such as mutation, crossover, division, and connection; and the second, is the reproduction operator. Using only the reproduction operator, the strength of the case library is determined by the initial set of cases placed in it. The library can never become stronger than the best case initial placed in it. However, with the addition of other operators, as outlined earlier, the library has the possibility of improving its strength with the generation and addition of new cases. The effect of this is to stir up the case library, and with the aid of the reproduction operator, strong cases remain while weak ones are removed. The case library's strength is always gradually increasing with an upper bound approximately equal to:

$$\text{Max}\left(\sum_{i=n-c}^{n} \text{CaseStrength}(i) \right);$$

where n is total number of cases allowed in the library, and c is the number of cases reproduced from the last the case library.

To understand how a GA works, one must look at how the chromosome strings (in our case, cases) are processed. To do this we must look at the Fundamental Theorem of Genetic Algorithms [Goldberg89] (FTGA) and how this brings about the concept of *Implicit Parallelism* [Goldberg89, Grefenstette89].

First, three objects need to be defined for use by the FTGA: a string, a population, and a schema. Strings (cases) are the fundamental objects that genetic algorithms manipulate. A string of length five constructed over a binary alphabet V = {0,1}, for example A_1 = 10101, can be represented symbolically as follows:

$A = a_1,a_2,a_3,a_4,a_5$

A population of strings is represented by A(t); the population of strings at generation t. A_j, j= 1, 2, ...,n, represents the n strings found in A(t).

A schema is similar to a string, but has a vocabulary of V = {0, 1, *} and is used to represent a class of strings. For example, the schema H = *0*01 can be used to represent the strings A_1 = 10101 and A_2 = 00001. This is because H matches the strings A_1 and A_2 at positions 2, 4, and 5, and matches the wild card symbol, *, at the other positions.

Given that three possible values (0, 1, and *) can occur at any position in a schema, there can be 3^l different schemata associated with a string of size l. In general, for a vocabulary of cardinality k there are $(k + 1)^l$ schemata. Put another way, a fixed string of size l can be part of 2^l schemata. If there are n strings in the population, then there are at most $n2^l$ possible schemata. These observations give one a feel of the magnitude of the schemata processing of a genetic algorithm. When one reproduces a string into the next generation, it is not just the string that is reproduced but the entire range of schemata the string can be generalize to.

Two properties are used to further describe a schema: schema order and schema defining length.

The schema order of schema H, o(H), is the number of positions in a schema that do not have the wild card symbol (*). The schema H = *0*01 has order o(H) = 3.

The schema defining length δ(H) is the number of positions between the first and last specific bit position. The schema H = *0*01 has defining length δ(H) = 3. The first specific string position is two and the last is at position five, therefore, δ(H) = 5 - 2 = 3. A schema H = *0*** has its defining length defined as zero since the first and last specific string positions are the same.

At time *t*, let there be *m* examples of a schema H in the population A(*t*); this is written as *m* = m(H,*t*). During reproduction, a string A_i is selected based on its fitness value f_i. The probability p_i of string A_i being selected is represented as:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

for a population of size *n*. Using this formula for p_i, the value *m* = m(H, *t*+1) can be calculated as:

$$m(H,t+1) = m(H,t) \bullet n \bullet \frac{f(H)}{\sum_{i=1}^{n} f_j}$$

where f(H) is the average fitness of the strings represented by schema H.

Letting the average fitness value of the population be F where:

$$F = \frac{\sum_{i=1}^{n} f_j}{n}$$

m(H,*t*+1) can be written as:

$$m(H,t+1) = m(H,t) \bullet \frac{f(H)}{F} \qquad (5.1)$$

A schema's growth rate is as a ratio of the fitness value of the schema and the average fitness value of the population. It is easy to see that above average fitness schemata will increase their presence in the population while below average fitness schemata will die off.

Assuming that schema H stays above the average fitness of the population by an amount cF for a constant c, the equation (5.1) can be written as:

$$m(H,t+1) = m(H,t) \bullet \frac{(F + cF)}{F} = (1 + c) \bullet m(H,t)$$

Assuming at $t = 0$, m(H,0) >= 1, the simple recurrence equation has the solution:

$$m(H,t+1) = m(H,0) \bullet (1 + c)^t$$

This assumption clearly shows that the effect of reproduction is to exponentially increase the number of above average schemata, while reducing the below average ones.

A question to ask at this point is: how many schemata are processed usefully at any given point? The answer to this, proposed by [Goldberg85], is $O(n^3)$. This means that there are n^3 schemata manipulated when n real structures (strings) are processed. This is an outstanding result. Even though the algorithm is processing only n strings, the processing of n^3 schemata is actually performed. The result is called *Implicit Parallelism*.

The proof of this result is as follows.

Let a population of strings be of size n, each string of length l. To count the number of schemata of length $s<l$ we show the following counting argument for $l = 10$ and $s = 5$. Consider the string A = 1010101010. To calculate all possible schemata of length 5 we generate a cell of size 5 about that string A.

A = (10101)01010

We count all possible schemata of the form:

H_i = (^^^^1)*****

where ^ can be the value (0,1,*), and where * is the wild card. Clearly there are 2^4 different schemata H_i of the form: (1***1)*****, (0***1)*****, (*1**1)*****, etc. In general there are $2^{(l_s - 1)}$ schemata of this form. The process is then repeated for the next cell in the string A. For example:

A = 1(01010)1010

We count all possible schemata of the form:

H_i = 1(^^^^0)*****

Clearly again there are 2^4 different schemata H_i of the form: *(****0)****, *(0***0)****, *(*1**0)****, etc. This process is repeated another four times until the cell cannot be moved further left. In fact the process of schemata cell counting is repeated $l - l_s + 1$ times. Therefore, we can estimate the number of schemata of length l_s derived from A as $2^{(l_s - 1)} \bullet (l - l_s + 1)$

To overestimate the number of schemata in the population n_s, we multiple the number of schemata derived from A by n giving:

$$n_s >= n \bullet 2^{(l_s - 1)} \bullet (l - l_s + 1)$$

It is an overestimate because there are obviously duplicates of low-order schemata.

Let us pick a population size of $n = 2^{(l_s/2)}$. Because the number of schemata is binomially distributed, half should be of size greater than order $l_s/2$ and half of less than order $l_s/2$. Counting only the higher order ones, the estimate on a lower bound on the number of schemata is as follows:

$$n_s >= n \bullet \frac{2^{(l_s - 1)}}{2} \bullet (l - l_s + 1) = n \bullet 2^{(l_s - 2)} \bullet (l - l_s + 1)$$

Remembering that $n = 2^{(l_s/2)}$ we substitute $\frac{n^2}{4} = \frac{2^{l_s}}{2^2} = 2^{(l_s - 2)}$ in the equation, which gives the result:

$$n_s = \frac{(l - l_s + 1) \bullet n^3}{4} = C \bullet n^3$$

where $C = \frac{(l - l_s + 1)}{4}$

Therefore, the number of schemata in a population is $O(n^3)$.

The result verifies that there are more schemata processed than the actual number of strings during the application of genetic operators. The results of *Implicit Parallelism* show that an increase in processing can be achieved for a relatively small amount of effort, making the concepts of genetic algorithms quite attractive and powerful for application with cases.

These results will be later illustrated by applying genetic techniques, discussed in this chapter, to a sample application of the game of checkers.

6 LEARNING CASES USING GENETIC ALGORITHMS

In the last chapter, methods for controlling the integrity of the case library and generating novel cases have been presented. One method, the Conservative Reproduction Algorithm[13], combined genetic operators for use with cases, forming a genetic algorithm for use in the symbolic world of cases not in the traditional GA world of subsymbolic domains. The chapter made reference to the strength or fitness of a case; a case's reproduction and the effect of other genetic operators often depend on this value. Strong cases must have an increase in their fitness and weak cases must have a decrease in their fitness over time. How is this increase or decrease achieved? The answer to this question is presented in this chapter.

In this chapter, by way of an example, the complete genetic process for learning strong new cases is molded together. The process has two major steps: case reinforcement, and the application of the Conservative Reproduction Algorithm. Case reinforcement - the process of increasing or decreasing a case's strength - is performed by the modified Bucket Brigade Algorithm [Goldberg89] described in the last chapter. The algorithm uses the success or failure of the application of a case in its judgment to reward or penalize a case's strength. After a period of case reinforcement, the Conservative Reproduction Algorithm, using the genetic operators of mutation, crossover, division, and connection, reproduces the strong cases into the next generation of the case library and removes the weak, invalid, and harmful cases. The learning process is an infinite cycle of using the cases, reproducing the best ones, and adding new cases formed from genetic adaptations of existing strong cases. The end result is a library of strong, non-redundant cases.

---

[13] Known as the Conservative Reproduction Operator for Cases in its operator form.

The example domain we will use is that of checkers. The initial goal is to produce a system that plays a game of checkers with a human and improves over time. The only piece of domain information the system begins with is what a legal move in a checker game is. This gives plenty of room for the system to learn for itself. Cases are used to represent checker move strategies; the Conservative Reproduction Algorithm, genetic operators, and the modified Bucket Brigade Algorithm are used to introduce and control the knowledge in the case library.

The final goal is to show that using only a very small amount of domain knowledge, the system can generate and keep strong new cases for application in a checker game. As a measure of this, the average strength of a case should increase as the number of generations of the case library increases.

The chapter is divided into four sections: structure, reinforcement schema, integrated approach, and the results. The structure section describes the organization of the cases to support a checker strategy and the application of genetic operators. The reinforcement schema section describes how the modified Bucket Brigade Algorithm is used to strengthen or weaken a case. The integrated approach section describes how the Conservative Reproduction Algorithm, genetic operators, and the modified Bucket Brigade Algorithm are combined to form a complete case learning system. Finally, the results section demonstrates, by way of graphic results, that the case library is indeed improving and becoming stronger.

6.1 STRUCTURES

The basic and only structure is a case; a case defining a checker strategy. Reviewing the case structure described in Chapter 5, a case has a collection of frames and a strength value; and a frame has a strength value and a data item. A checker case frame has a strength value and a checker piece's move as its data item. A move consists of a board layout and a legal move for a player. A checker case consists of a strength value and a set of checker frames -

each frame a checker move - alternating between between legal moves of the white and black players in the game of checkers. A checker case - known as a strategy - is just a sequence of moves experienced during the progress of a checker game.

An important feature of a case is that its size is dynamic. A case strategy does not represent the moves taken in a full game of checkers, but rather that of a partial sequence of moves of different lengths. A good case can then be defined as a sequence of moves that leads the system player, black, to a better position.

New cases are formed in two manners: by genetic operators or by system addition. It has been described how genetic operators form new cases. Strong existing cases are taken and altered by the genetic operators: mutation, crossover, division, and connection. However, before these operators can be performed, cases have to exist. This is where the domain theory plays an important role. Not in forming strong cases, although it might, but in producing cases that can be manipulated by the genetic portion of the system to form strong cases.

To describe how the system produces cases we must first talk about the general operation of it. The game begins with the human making a move. The system must now produce its own move. If the system can find a case in memory that contains, in its frames, the exact game history[14], the next move in the case is returned as the system's move. However, if no such case can be found, the last white move and a legal move returned by the domain theory form a new case that is added to the case library, and the legal move is returned as the system's move. The new, two move case that was formed by the system can also be extended in the following manner. If it or any other case was used to return the last black move, but does not have a move in response to the next white move because the case is at its last frame, and no other cases in memory have a move in response to white's move, rather than using the last white move and a legal move returned by the domain

[14] The game history is the previous sequence of game moves to a selected depth.

theory to form a new case, these moves are added to the end of the last used case, and the legal move returned as the system's move. The effect of this is that rather than forming another smaller case, the case that was used in the last turn is extended by two more moves.

This is the general method of how the domain theory is used to build-up cases. The fact that the domain theory returns any legal move, not the best, leads to the fact that the strategies formed from the domain theory are not necessarily optimal; however, the system must detect this. A case library of strong strategies is the system objective.

6.2 REINFORCEMENT SCHEMA: BUCKET BRIDGADE

The Bucket Brigade Algorithm [Goldberg89, Westerdale89, Grefenstette88], named by Holland, is an apportionment of credit algorithm for classifiers[15]. The algorithm, simply put, is a basic execution cycle of locating the appropriate classifiers for the current environmental conditions, posting these classifiers' messages back to the environment or as input into the next execution of the loop, and rewarding or penalizing the posted classifiers' strength based on an evaluation criterion or function. This thesis is concerned with cases, not with classifiers, and therefore, the thesis proposes a modified and renamed algorithm for the apportionment of credit of cases called the Reinforcement Schema for Cases (RSC). Whether for a classifier or for a case the approach is similar; genetic operators require a method of ranking cases or classifiers. The rank, a case's strength or fitness value, is a measure of the performance of the case in its environment. Using a modified metaphor from [Goldberg89], the RSC can be viewed as an information economy where the selection of a case's solution to be used is bought and sold by the case. Cases are produced by the manufacturer (environment) and are sold to the game (consumer).

[15] A classifier is a special form of chromosome string that encodes a rule for use in a classifier system.

The economy of the RSC is composed of three components: the bidding house, the taxation department, and evaluation process. The structures these components use are a case, the case library, and the environment. A schematic sketching the RSC algorithm and its three processes is shown in figure 6.1.

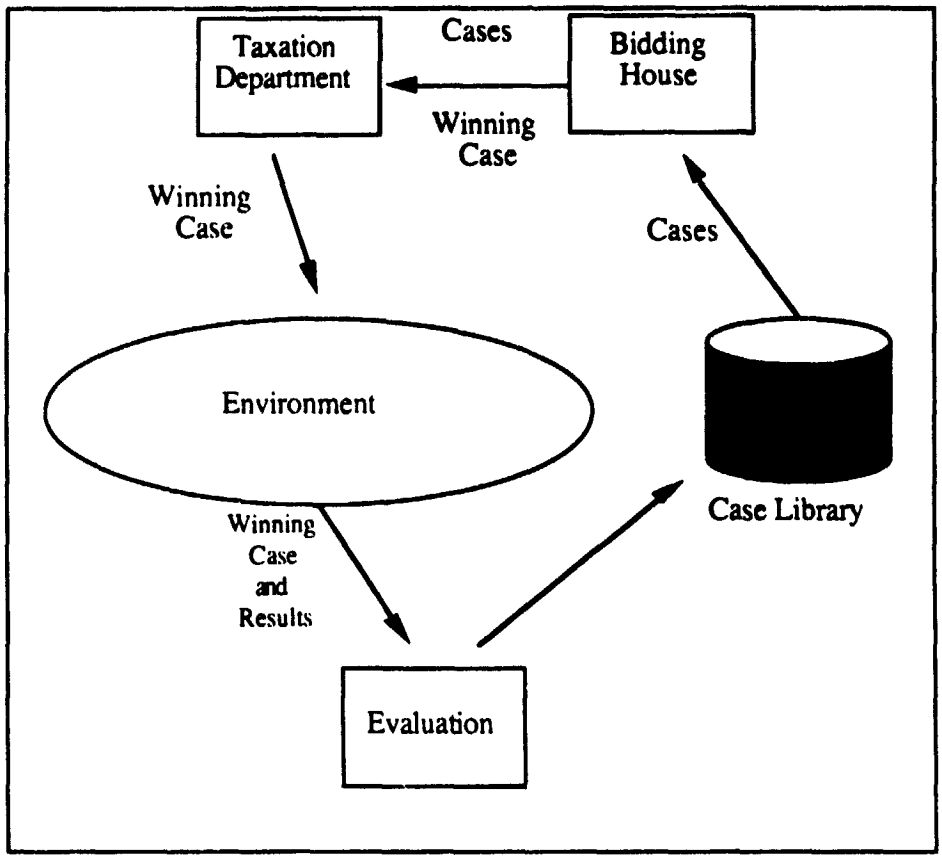


Figure 6.1 The Reinforcement Schema for Cases

Initially, all cases matching the current environmental features are located. The cases are said to be activated for duty (ready to return a solution). To perform a case's duty, an auction is held to determine the case that has the right to return its solution. When only one case is returned, it automatically wins the auction and returns its solution. When there is more than one case in the active case set, each case makes a bid for the right to return its solution. The bid of a case is proportional to its strength. Stronger cases make larger bids

than weaker ones. In the checker example, a simple bid function could be the average strength of a case's frames:

$$\text{Case Bid} = \frac{\sum_{i=1}^{\text{Size(Case)}} \text{FrameStrength}(i)}{\text{Size(Case)}}$$

The case with the largest bid wins the right to perform its duty and the other cases are deactivated.

All cases that bid for the right to perform their duty must now pay for that right. Each is taxed by the amount of its bid. The effect is that each case has a decrease in its strength. How does a case increase its strength? This depends on how well the case performs its duty in the environment. If a case performs adequately, its strength regains the amount of its bid; its strength is now at least as strong as it was before it performed its duty. If the case performed exceptionally well, its bid and a reward amount are added to its strength, making it a little stronger than is was before performing its duty. Finally, if the case performed poorly, the bid is not returned and a penalty amount is subtracted from its strength. Cases that perform well will have their strength increased, while cases that appear to be strong but perform poorly, will have their strength decreased more than the cases that lost the right to perform their duty. This gives these secondary cases a better chance at performing their duty on the next iteration, given the same conditions.

In the next section, the differences between the RSC method and the generic Bucket Brigade Algorithm are brought forward in a discussion of an overall approach, merging genetic algorithms, genetic operators, apportionment of credit, and CBR. The success of any genetic algorithm is closely coupled with how the strength's of its structures, in our example cases, are manipulated. Therefore, the RSC is an integral part of the overall framework.

6.3 APPROACH: SIMPLE CLASSIFIER SYSTEM FOR CASES

The approach taken to learning cases is based on Holland's classifier systems [Holland86a]. A classifier system learns strong, simple rules (called classifiers) to guide its performance in a given environment. A classifier system has three major components:

1. Rule and message system
2. Apportionment of credit system
3. Genetic Algorithm.

The rule and message system is a special type of production system that generally uses classifier (rules) of the form:

IF <condition> THEN <action>

For example:

IF 00#10 THEN 01010

The classifier is normally stored as a string; therefore, the example above would be encoded as 00#10:01010. The condition and action are simple string patterns of 0,1, and #, and are of fixed length. This restriction makes it easy to apply genetic operators to the classifier to increase the rule base. The apportionment of credit system (Bucket Brigade Algorithm) and GAs have been described in the previous chapter. For the remainder of this chapter, we will present an approach for using a classifier-like system, with cases as the classifiers, and will not present a formal description of a simple classifier system (SCS) [Goldberg89].

Our approach called the Simple Classifier System for Cases (SCSC), and its three components: CBR, genetic algorithms , and reinforcement schema, in a closed learning loop , are shown in figure 6.2.

We describe the architecture by way of a checker example. As a starting point for the SCSC, we begin with the case library. The case library contains a collection, possibly empty, of source cases. In the checker example, each case represents a checker strategy. For the example, we start with an empty case library, although initially we could seed the library with hand-coded cases. As long as there is a limited domain theory, and an initial learning period, the SCSC will develop its own cases.

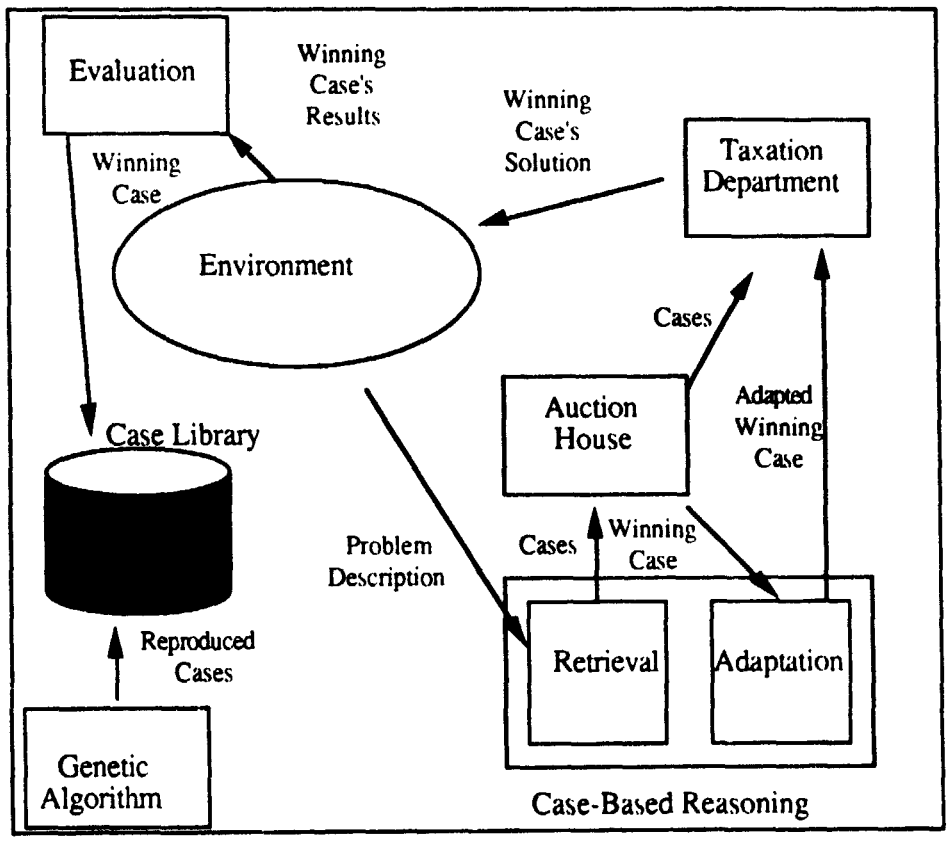


Figure 6.2 Simple Classifier System for Cases

Next, the environment produces an input problem in which an action or response is required. The input situation is taken and all appropriate cases are retrieved from the case

library that have a chance of providing a solution to the input problem. This is the normal process of CBR retrieval. These activated cases are then passed to the *Auction House*. Here the cases bid for the right to return their responses. As a result of the auction, the case with the largest bid is found.

In the checker example, the bid of a case is a function of the case's strength which is a function of the individual strengths of the case's frames. These functions are as follows:

$$\text{Case Strength} = C_S = \sum_{i=1}^{n} C_{f_i}$$

where C_{f_i} is the strength of frame *i*; and *n* is the number of frames in the case.

$$\text{Case Bid} = C_b = C_S \bullet \left(\frac{\text{number of frames matched}}{3 \bullet \text{total number of frames}}\right)$$

The strength of the case is the sum of the strengths of the individual frames. The bid is the normalized ratio of the number of frames that match the problem description in the retrieval process divided by the number of frames in the case, multiplied by the strength of the case, and then divided again by the two times the number of frames to get the average bid of a frame. The reason for the extra division by two times the number of frames is to generate a maximum bid of 1.0 when the maximum allowable strength of a frame is limited to 2. The reduction is shown later. The function causes cases that have a greater number of matching frames to be favored over smaller cases that match the same number of frames.

At this point, a limited domain theory is required. It is limited because it provides a minimal amount of information to enable the formation of initial cases. For example, if there are no cases retrieved by the CBR retrieval process, the domain theory can form a short simple case and return it as the winning case. In the checker example, this would be equivalent to starting a case with the last move of the opponent followed by any legal

system move. It is not difficult for a domain theory to compute a legal move as long as it knows the rules of the game. What the domain theory is missing is knowledge as to what constitutes a good move. This, as we will see, is where the genetic algorithm and the RSC form an important part of the approach.

The winning case is now passed back to the CBR adaptation process so final modifications can be made, ensuring that its response fits the input problem. This is a link back to the CBR process. This thesis does not present any such adaptations, but if so desired, further adaptations can be made using traditional CBR adaptation techniques. The adapted winning case and all other activated cases are passed to the *Taxation Department* for the decrease of their strengths by their bids.

The winning case's response is now passed back to the environment for use. All other activated cases have no chance of getting their strengths back, except on further iterations of the learning loop. It is only the winning case that has a chance of gaining its strength back and more. This all depends on how well the case performs in the environment. The goal is to select the best case for use in the environment. The environment provides feedback as to how well, or poorly, the case performs, in the form of three results, -1, 0, and 1. The feedback is used by the evaluation component of the approach. In the checker example the environmental evaluation is as follows:

If the move produces a king or jumps an opponent a 1 is returned
    Else If the move produced an opponent jump a -1 is returned
        Else a 0 is returned

The evaluation component takes the following actions based on these three results:

Case Evaluation of

0 :	add the case's bid to its strength;
1 :	add the case's bid and a reward value to its strength;
otherwise:	{the evaluation is a -1)
	penalize the case's strength further by a penalty amount

In the checker example, a frame's maximum allowable strength is 2, and the reward and penalty values 1. In the computation of a case's bid, it is also further divided by two times the total number of frames in the case. The reason for these restrictions, are to restrict the bid value of a case to between 0 and 1. Using these values, a case's bid can be shown to be less than or equal to 1.0.

$$C_b = C_s \bullet \left(\frac{\text{number of frames matched}}{3 \bullet \text{total number of frames}}\right)$$

$$C_b <= 2 \bullet \text{total number of frames} \bullet \left(\frac{\text{number of frames matched}}{3 \bullet \text{total number of frames}}\right)$$

$$C_b <= \left(\frac{\text{number of frames matched}}{\text{total number of frames}}\right)$$

$$C_b <= 1.0$$

In the checker example, cases initially have all their frames' strengths set to 0.2. After completing the evaluation, the active and winning cases, including the modification to strength, are updated in the case library.

The processes of case retrieval, adaptation, and evaluation are repeated for a fixed[16] number of iterations, after which the Conservative Reproduction Algorithm is performed on the case library, removing the bad cases, keeping the good ones, and adding new ones

[16] System parameter set upon initialization of the SCSC.

based on variations of existing strong cases. The result is that over time, a library of strong cases will be generated from a initial, possibly empty, set of original cases.

Before showing the result of the system running in the described checker domain, it is useful to describe the SCSC in comparison with Holland's SCS. Our approach to learning new strong cases is similar to Holland's description of a SCS; however, its adaptation for cases differs in four respects:

1. Only one case (classifier) can be active at any one time. In a SCS many classifiers can be active at any one time. The reason for this is that there is no conflict between the classifiers because classifiers can only send their messages to the global message list. CBR is a problem solving activity, and only the best answer to a problem is returned to the environment, not a collection of them.
2. Cases are of variable lengths. Classifiers in a SCS are fixed length strings, allowing for simple matching of rules. Having variable size cases increases the complexity of the matching method, but it is still possible to find matching cases. In CBR matching is not necessarily exact, and the best cases are returned and adapted to match the input from the environment. Because of the adaptation mechanism, cases of all forms and lengths can be matched.
3. Messages (case responses) are not posted to a message list. In a SCS, messages are posted to a message list which may either send them to the environment or to the detection part of the SCS for further processing (like a forward chaining production system). When a case is located and adapted, it is always sent to the environment, not back to the system again. A case can not affect the next action in the SCSC, only the environment which the case interacts with can produce another message (problem) for the SCSC.

4. The RSC algorithm rather than the Bucket Brigade Algorithm is used for the apportionment of credit. The reason for the modification concerns the way CBR processes cases. First, the evaluation criterion is based on the results of applying the case's results in the environment. Therefore, it is the environment that determines the success or failure of the case, not, as in the the Bucket Brigade, where the majority of a classifier's strength increase comes from the fact that its message was actually posted to the message list, and little or no environmental reward or punishment is added or subtracted from the classifier. The second difference concerns the fact that only one, not multiple cases, is posted to the environment. CBR returns one matched and adapted case - the best one possible - not a collection of possible cases. Therefore, the RSC is modified to meet this restriction.

The SCSC does, however, preserve the following characteristics of the SCS:

1. The basic algorithm of the SCSC is a simple loop. The environment produce a message (input case requirements) to the SCSC, the SCSC locates a classifier (case) that matches the input message. The action part of the classifier (case response) is sent back to the environment, and the SCSC waits for a new response from the environment.
2. Communications, input and output, are done using only messages: environmental requirements on input, and case responses on output.
3. The order in which classifiers (cases) are used is independent. Which case is used depends on what the environment sends as part of its requirement message. No previously returned case response can cause a different case response to be posted in the future.

Adaptations to the SCS, and the Bucket Brigade Credit Apportionment Algorithm, provide systematic methods for obtaining strong cases; genetic algorithms provide a novel

method of case adaptation, but still rely heavily on the past history of the case so as to not alter the case too drastically; and CBR is a method of reasoning from cases. The combination of the three leads to a complete approach to learning and reasoning in a dynamic environment. Each of the above components aids the others in performing their duty. The name of this thesis is EXTENDING CASE-BASED REASONING USING RULE-BASED AND GENETIC TECHNIQUES; however, with the discussion from the last two chapters it could of just as easily been called EXTENDING GENETIC ALGORITHMS USING CASE-BASED REASONING. Each of these areas can greatly benefit from the power of using the other's techniques.

6.3.1 Results

The results described in this section are produced from a human playing a 4x4 checker game against a SCSC as described in the last section. The last sections presented the SCSC and its goals: to produce new, strong cases using genetic operators, and to have the overall strength of the case library improve with each successive generation. Using a 4x4 checker game integrated with the approach, we demonstrate that these goals are met.

The following results show how the SCSC operates on two different sets of parameters. The first set is a volatile parameter set. With these parameter settings, genetic operators are applied with a probability of 1. That is, given the correct condition, the genetic operator will be performed. Also, the number of cases in the library is small compared to the number of possible cases in the domain. This means that there is a high turn-over of cases from generation to generation. The end result is that new cases are added and removed at a high rate.

The second parameter set is less volatile. Genetic operators are performed with probability 0.5. Also, the number of cases in the library is smaller than the first parameter set. Again there is a high turn-over of cases, but the rate of new cases added to the library as a result of the application of genetic operators is less.

Before describing the individual results, we describe the common factors affecting the SCSC. Initially the case library is empty, no seed cases are provided and the only domain theory is the simple rules of checkers; that is when called upon the domain theory will return a random legal move for black, the system player. This is an important point. Good case information is not added to the case library to begin with. Therefore, its initial strength is zero. Just relying on the domain theory to build our cases would result in cases built at random - since the domain theory only returns random legal moves. If the moves returned were poor, the cases would be constantly loosing strength, and with such cases the library would experience an overall decrease in strength, not an increase as desired. Each session consists of forty checker games where the human and system player alternate moves under the rules of checkers. The Conservative Reproduction Algorithm is performed every *n* games, where *n* is set as a parameter.

The results plot average case and case library strength against the number of games played.

The following list and results describes the volatile parameter set found in Appendix A-1.

Maximum Case Library Size = 10

Genetic Algorithm Performed Every 2 Games

Probability of Mutation = 1

Probability of Crossover = 1

Probability of Division = 1

Probability of Connection = 1

In figure 6.3, the average case strength is plotted against the number of games played. As is demonstrated in all graphs, there is a steep rise in the average strength of a case over the first few games because there are no cases in the library to begin with. However, after

the case library is filled up with the domain theory cases, it is easy to see that the overall average case strength continues to increase. There are many local minima and maxima, but this is the nature of using genetic operators. There are major advancements in strength followed by retreats; however, the number of advancements is larger than the number of retreats. From the graph in figure 6.3 it is easy to view that the overall strength of the cases is increasing with every game played.

Figure 6.4 plots the strength of the entire case library against the number of games played. The result is the same. After a fast jump in the total strength of the library, as result of the addition of new cases to the case library by the domain theory, the overall strength continues to increase as a result of the application of the genetic algorithm and the RSC.

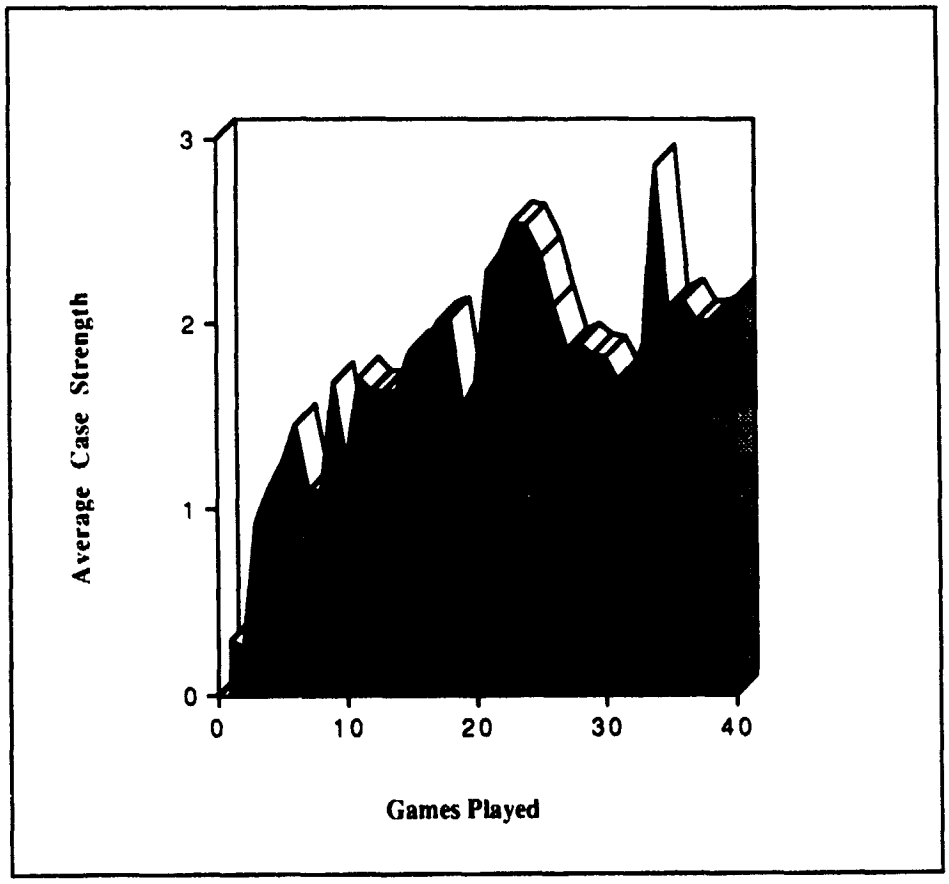


Figure 6.3 Average Case Strength in a Volatile Environment

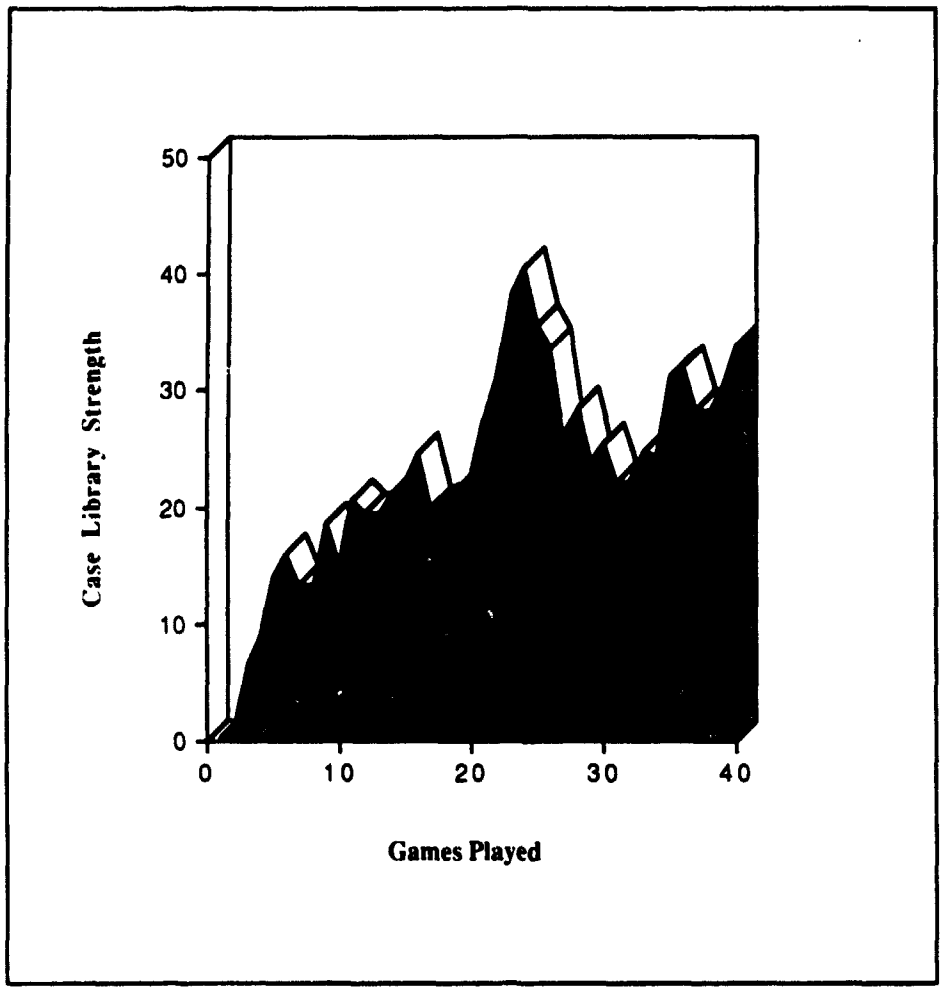


Figure 6.4 Case Library Strength in a Volatile Environment

Let us now look at a set of less volatile parameter settings. The following list and results describes the less volatile parameter set found in Appendix A-2:

Maximum Case Library Size = 5

Genetic Algorithm Performed Every 4 Games

Probability of Mutation = 0.5

Probability of Crossover = 0.5

Probability of Division = 0.5

Probability of Connection = 0.5

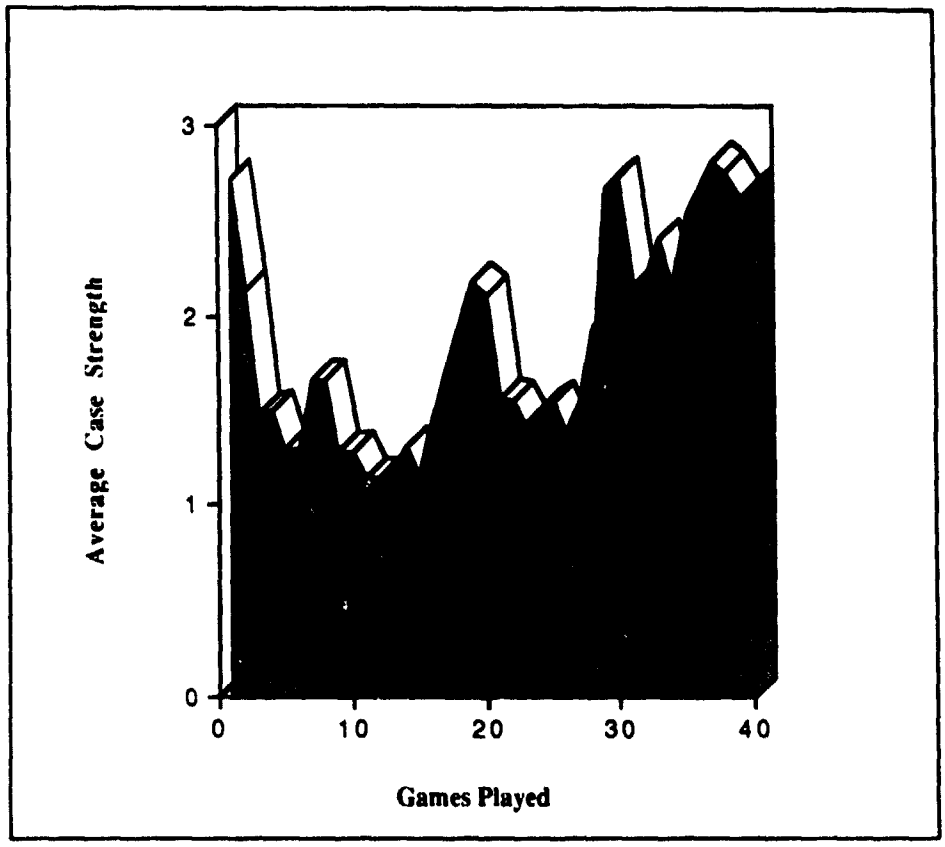


Figure 6.5 Average Case Strength in a Less Volatile Environment

Figure 6.5 shows the average case strength plotted against the number of games played. The results are less dramatic as in the volatile parameter set, but still the same; an overall increase in the average strength of a case is seen. The initial jump in strength is a result of the influx of introductory domain cases. After subsequent games are played the poor cases are weeded out and the genetic process takes effect. New, strong cases are produced systematically following the familiar pattern of strength growth. A smaller maximum case library is used in this example due to the decrease in the frequency of applying the genetic algorithm which requires a longer time to build up a maximum set of

cases. Reducing the maximum number of cases counter-acts this situation and ensures the proper application of the genetic operators.

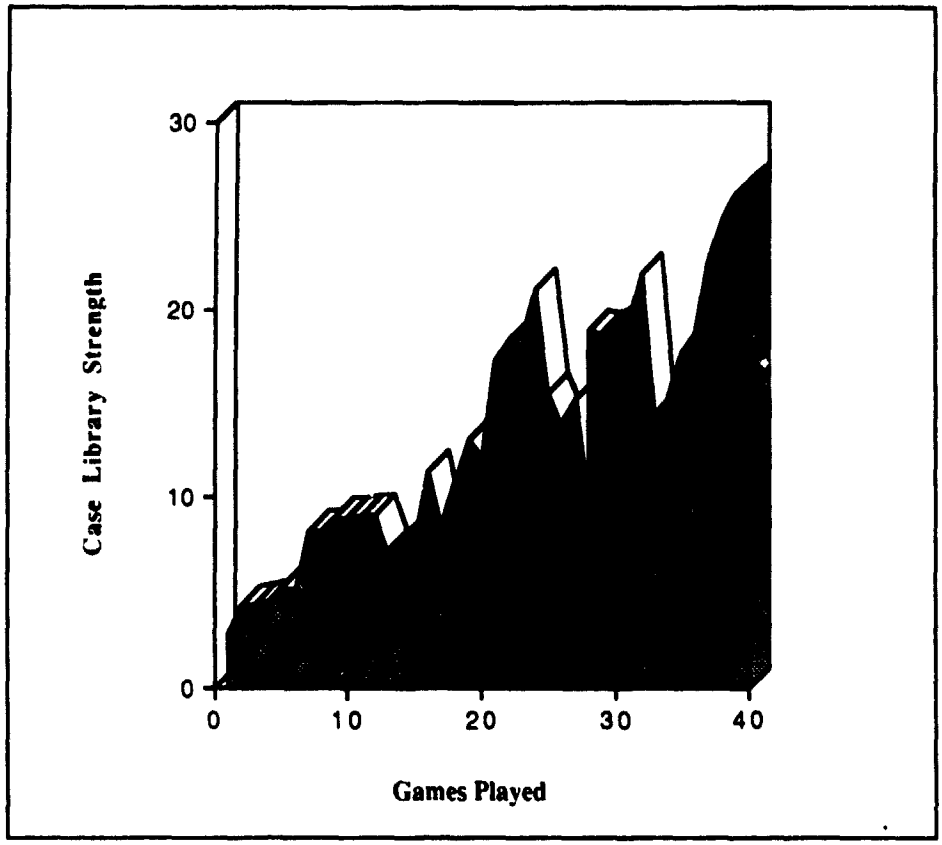


Figure 6.6 Case Library S'rength in a Less Volatile Environment

Figure 6.6 plots the strength of the entire case library against the number of games played using the less volatile parameter set. The result is the same. After a jump in the total strength of the library, as result of the addition of new cases to the case library by the domain theory, the overall strength continues to increases as a result of the application of the genetic algorithm and the RSC.

Since the case size is dynamic, one might ask: is the overall strength of a frame, in this instance a checker move, also increasing in strength? Figure 6.7 answers this question. In

this figure the average strength of a frame (move) is plotted against the number of games played.

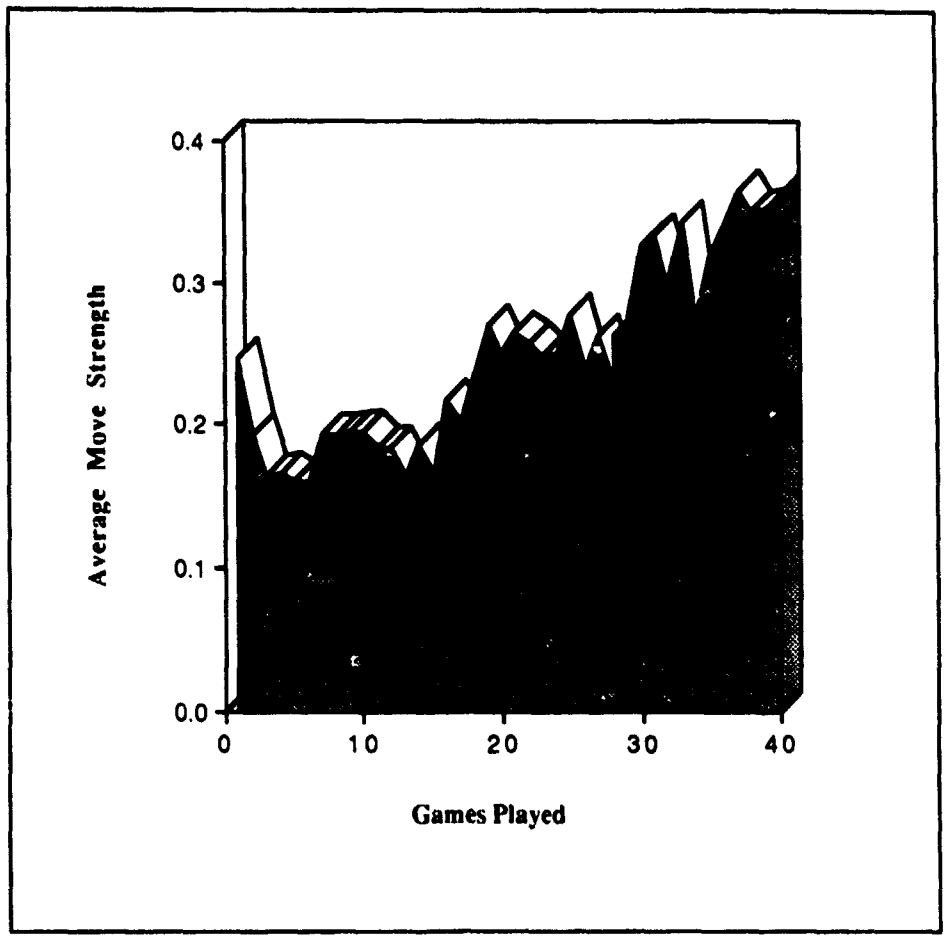


Figure 6.7 Average Move Strength in a Less Volatile Environment

The results are the same for the individual frames (moves), the average move strength is increasing as the number of games played increases. Its is not only the case that is getting stronger is, it is also the piece's moves as well.

In corroboration with the results and analysis, the presented approach demonstrates that a library of strong, sound cases can be produced, used, and maintained, starting from an empty case library in a dynamic environment. With this in mind, parameters to the graph were set for maximum agitation of the case library. Even with the constant turn-over of the

cases in the library, the approach shows a slow, gradual increase in the overall average case library strength, case strength, and frame strength, showing that the approach presented - a joint method of reasoning using CBR and genetic techniques - improves its knowledge over time.

7 CONCLUSION

CBR is a powerful model of reasoning based on the human ability of reminding. Although powerful, still many issues remain to be solved to make CBR a more practical and useable method of reasoning. The approach of this thesis has been to take other AI methods and integrate them, where appropriate, to extend CBR by providing solutions to some of the open issues. For example, we have adapted genetic techniques for use by CBR. Adaptation is in fact the central theme of CBR; adapt solutions when appropriate to provide, in this case, novel methods of extending CBR.

We will now recapitulate the contributions to CBR that this thesis has made. This thesis has:

- Described a method for the integration of rule-based, case-based, and generalization techniques for the modification and repair of plans.

- Proposed a planning approach and structure that can not only acquire new plans and replanning rules by itself and from an expert, but can also induce higher level learned replanning actions that are not immediately apparent to the user or a typical CBP system that is only looking at failures one at a time.

- Developed a planning model that views planning as a process that combines developing an initial plan, executing the plan, fixing failures as they arise during execution, evaluating, modifying, and storing the plan for later reuse We agree, of course, that some planning precedes plan execution, but we believe that in many realistic domains planning and execution should be interleaved, i.e. that execution should start before a complete plan is achieved, that unforeseen plan failures should be handled by replanning on the spot,

and, most importantly, that the results of evaluating the execution of plan fragments and plans should be stored with the plans for future use. In other words, it is this integration of planning and execution that enables a planner to improve its performance in future planning episodes by taking advantage of information gained during past executions and subsequent evaluations.

- Successfully demonstrated that the metaphor of a *case is chromosome* can be used to apply genetic techniques to cases.

- Proposed a genetic algorithm, called the *Conservative Reproduction Algorithm*, for the generation of a new case library that over time builds up a library of strong, valid cases, by selectively removing redundant, invalid, harmful, weak ones. Controlling the contents of the case library is an issue that, although ignored by many researchers, needs to be solved if any system using CBR can learn. You can not keep increasing the number of cases in the library without looking at what cases are already in the library and how the new cases added to the library affect them. This thesis proposes a method of handling this problem and demonstrates it to be an effective means of controlling the integrity of the case library.

- Developed four other operators: *Mutation, Crossover, Division, and Connection*, based on their genetic counterparts for application with cases.

- Combined the five genetic case operators, forming a genetic algorithm for the generation of novel cases, strongly based on cases found in the case library,

but not achievable through the application of traditional case adaptation techniques.

- Proposed and tested a novel learning approach that combines CBR and a modified classifier system for cases, called the *Simple Classifier System for Cases*, to develop cases solely using its own resources, and an extremely limited domain theory, to produce its own stronger cases over time.

This list of contributions easily provides answers to the initial three concerns presented in the introduction of this thesis: 1) developing a new model for the adaptation and repair of cases *during* execution, not only just *before and after*; 2) developing a method of learning cases in a noisy environment and controlling the contents of case memory; and 3) developing a method of generating novel cases that does not rely on the failures of other cases for the construction of a new case.

There are still many open issues left to pursue in CBR; CBR is a relatively new field of AI. Areas such as indexing, memory organization, retrieval, matching, and adaptation, are only the headings of large research areas with many issues to resolve. Research into each area is required for the further development of CBR.

7.1 DIRECTION OF FUTURE WORK

The use of genetic techniques has proven quite helpful in the formation of a self-perpetuating learning model. Although the genetic case structure was constructed as a general structure, not connected to any specific domain, the application of the model to another domain would help to further develop the characteristics that cases must have for the further use of genetic techniques. The current strength of a case, for instance, is a function of the strengths of individual frames (slots, game moves, etc) of a given case. If cases do not have a frame structure, what other methods of assessing case strength are

there? Different methods of calculating strength and fitness have been proposed by genetic algorithm researchers and would be of great interest to those wishing to alter the method of credit apportionment for cases.

Schemata processing is also of great interest to the genetic algorithm community. As described in this thesis, the number of schemata processed during one reproduction of a given population of size n yields n^3 operations on schemata. A schema is analogous to a generalization of a set of cases. Genetic algorithm research has shown that schemata of low order and low defining length are best. This would mean that for case reproduction, very general cases are preferred over more specific cases. Is this assumption true? This is one area that we feel warrants a further look.

REFERENCES

[Alterman89] Alterman, R., A Concept Space For Reasoning About Cases Involving Event Structures, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 16-19, 1989.

[Ambrose88] Ambrose-Ingerson, J.A., and Steel S., Integrating Planning, Execution, and Monitoring, Proceedings of the Seventh National Conference of Artificial Intelligence, Morgan Kaufmann, 83-88, 1988.

[Antonisse87] Antonisse, H. J., and Keller, K. S., Genetic Operators for High-Level Knowledge Representation, Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum, 69-76, 1987.

[Axelrod87] Axelrod, R., The Evolution of Strategies in the Iterated Prisoner's Dilemma, Genetic Algorithms and Simulated Annealing, London: Pitmann, 32-41, 1987.

[Bain85] Bain, W. M., A Case-Based Reasoning System For Subject Assessment, Seventh International Joint Conference on Artificial Intelligence, 523-527, 1985.

[Barletta88] Barletta, R. & Mark, W., Explanation-Based Indexing of Cases, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 50-60, 1988.

[Bickel87] Bickel, A. S., Bicker, R. W., Tree Structured Rules in Genetic Algorithms, Representation, Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum, 77-81, 1987.

[Bradtke88] Bradtke, S. & Lehnert, W. G., Some Experiments with Case-based Search, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 80-93, 1988.

[Branting88] Branting, L. K., The Role of Explanation in Reasoning from Legal Precedents, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 94-103, 1988.

[DeBono87] De Bono, E., Letters to Thinkers: Further Thoughts on Lateral Thinking, Penguin Books Ltd, Markham Ont., 1987.

[DeJong87] De Jong, K. A., On Using Genetic Algorithms to Search Program Spaces, Proceedings of the Second International Conference on Genetic Algorithms, 210-216, 1987.

[DeJong89] De Jong, K. A., and Spears, W. M., Using Genetic Algorithms to Solve NP-Complete Problems, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, 124-132, 1989.

[Deugo89a] Deugo, D. L., and Oppacher, F., Case-Based Techniques for Learning and Repairing Plans, Proceedings of The Second International Symposium on Artificial Intelligence, Monterrey, N.L. Mexico, Forthcoming.

[Deugo89b] Deugo, D. L., and Oppacher, F., Applications of Case-Based Reasoning Using Knowledge Base and Genetic Techniques, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 239-243, 1989.

[Dietterich83] Dietterich, T., and Michalski, R., A Comparative Review of Selected Methods for Learning from Examples. In [Michalski83], ch. 3, 1983.

[Ernst69] Ernst ,G.W., and Newell, A., GPS: A Case Study in Generality and Problem Solving, Academic Press, 1969.

[Genesereth87] Genesereth, M.R., and Nilsson, N.J., Logical Foundations of Artificial Intelligence, Morgan Kaufmann, 285-304, 1987.

[Gentner83] Gentner, D., Structure-mapping: A theoretical framework for analogy. Cognitive Science, 7(2),155-170, 1983.

[Goldberg89] Goldberg, D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[Goldberg85] Goldberg, D.E., Optimal initial population size for binary-coded genetic algorithms (TCGA Report No. 850001), Tuscaloosa: University of Alabama. The Clearing house for Genetic Algorithms, 1985.

[Grefenstette86] Grefenstette, J. J., Optimization of Control Parameters for Genetic Algorithms, IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-16, No. 1, January/February, 122-128, 1986.

[Grefenstette88] Grefenstette, J. J., Credit Assignment in Genetic Learning Systems, Proceedings of the Seventh National Conference on Artificial Intelligence,Morgan Kaufmann Publishers, 596-600, 1988.

[Grefenstette89] Grefenstette, J. J., and Baker, J. E., How Genetic Algorithms Work: A Critical Look at Implicit Parallelism, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 20-27, 1989.

[Hammond86a] Hammond, K.J., CHEF: A Model of Case-Based Planning, Proceedings of the fifth National Conference of Artificial Intelligence, Morgan Kaufmann, 267-271, 1986.

[Hammond86b] Hammond, K.J., Case-Based Planning: An integrated theory of planning, learning, and memory, Ph.D. thesis, Yale University, 1986.

[Hammond88a] Hammond, K. J., Case-Based Planning , Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 17-20, 1988.

[Hammond88b] Hammond, K. J., Opportunistic Memory: Storing and recalling suspended goals, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 154-168,1988.

[Hammond89a] Hammond, K. J., Adaptation of Cases , Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 88-89, 1989.

[Hammond89b] Hammond, K.J., Case-Based Planning: Viewing Planning as a Memory Task, Academic Press, 1989.

[Hendler88a] Hendler, J. A., Integrated Marker-Passing and Problem Solving: A Spreading Activation Approach to Improve Choice in Planning, Lawrence Erlbaum Associates, 1988.

[Hendler88b] Hendler, J. A., Kambampati, S., Refitting Plans for Case-Based Reasoning, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 179-181, 1988.

[Holland86a] Holland, J. H., Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems, In Machine Learning: An Artificial Intelligence Approach, Vol II, Michalski, Ryszard S., Carbonell, J. G., and Mitchell, T. M. (Eds), Morgan Kaufmann Publishers Inc, 1986.

[Holland86b] Holland, J. H., Holyoak, K.H., Nisbett, R.E., and Thagard, P.R., INDUCTION: Processes of Inference, Learning, and Discovery, The MIT Press, 1986.

[Kass88] Kass, A. M., Leake, D. B., Case-Based Reasoning Applied to Constructing Explanations, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 190-208, 1988.

[Kibler88] Kibler, D., Aha, D.W., Case-Based Classification, Proceedings: Case-Base Reasoning Workshop, AAAI-88, 62-67, 1988.

[Kolodner84] Kolodner, J. L., Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model, Lawrence Erlbaum Associates, 1984.

[Kolodner86] Kolodner, J. L., & Riesbeck C. K. Experience, Memory, and Reasoning, Lawrence Erlbaum Associates, 1986.

[Kolodner87] Kolodner, J.L., Simpson, K., and Sycara-Cryanski, K., A Process Model of Case-Based Reasoning in Problem Solving, Proceedings of the Sixth National Conference of Artificial Intelligence, Morgan Kaufmann, 284-290, 1987.

[Kolodner88a] Kolodner, J. L., Extending Problem Solver Capabilities Through Case-Base Inference, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 21-30, 1988.

[Kolodner88b] Kolodner, J. L., Retrieving Event from a Case Memory: A Parallel Implementation, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 233-249, 1988.

[Lebowitz83] Lebowitz, M., Generalization from Natural Language Text, Cognitive Science, V7:1, 1983.

[Lebowitz87] Lebowitz, M., Experiments with Incremental Concepts Formations: UNIMEM, Machine learning Vol 2, 130-138, 1987.

[Matwin89] Matwin, S., Oppacher, F., Constant, P., Knowledge Acquisition by Incremental Learning from Problem - Solution Pairs, Journal of Computational Intelligence, 5,2, 1989.

[Meehanite65] Meehanite Metal Corporation, CASTING DESIGN as influenced by FOUNDRY PRACTICE, Meehanite Metal Corporation, White Plains, N.Y. and St. Louis, MO. Bulletin No. 44, 1965.

[Michalski83] Michalski, R.S., Carbonell, J.G., Mitchell, T.M., Machine Learning: An Artificial Intelligence Approach, Morgan Kaufmann, Vol. 1, 1983.

[Nilsson80] Nilsson, N.J., Principles of Artificial Intelligence, Tioga Publishing Company, 298-306, 1980.

[Odetayo89] Odetayo, M. O., and McGregor, D. R., Genetic Algorithms For Inducing Control Rules For A Dynamic System, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, 177-182, 1989.

[Oppacher88] Oppacher, F., A Dynamic Case-Based Planning System for Space Station Applications, Fourth Conference on Artificial Intelligence for Space Applications, NASA Conference Publication 3013,431-440, 1988.

[Riesbeck88] Riesbeck, C.K., McMops: Case-Based Reasoning in Miniature, Proceeding: Case-Based Reasoning Workshop, AAAI-88 Workshop, 114-119, 1988.

[Riesbeck89] Riesbeck, C.K., Schank, R.C., Inside Case-Based Reasoning, Lawrence Erlbaum Associates, 1989.

[Rissland89] Rissland, E. L., Kolodner, J., Waltz, D., Case-Based Reasoning, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 1-13, 1989.

[Ross89] Ross, B. H., Some Psychological Results on Case-Based Reasoning, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 144-147,1989.

[Sacerdoti74] Sacerdoti, E.D., Planning in a Hierarchy of Abstraction Spaces, Artificial Intelligence, 5:115-135, 1974.

[Schank77] Schank, R. C. Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures, Lawrence Erlbaum Associates, 1977.

[Schank82] Schank, R. C. Dynamic Memory: A Theory of reminding and learning in computers and people, Cambridge University Press, 1982.

[Seifert88] Seifert, C. M., Goals in Reminding, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 357-369, 1988.

[Shinn88] Shinn, H.S., Abstractional Analogy: A Model of Analogical Reasoning, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 370-387, 1988.

[Stanfill86] Stanfill, C., Waltz, D., Toward Memory-Base Reasoning, Communications of the ACM, V29:12,1213-1228, 1986.

[Sussman75] Sussman, G.J., A Computer Model of Skill Acquisition, New York: Elsevier, 1975.

[Sycara88] Sycara, K., Using Case-Based Reasoning for Plan Adaptation and Repair, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 425- 434, 1988.

[Tate77] Tate A., Generating Project Networks, Proceedings of the Fifth International Joint conference on Artificial Intelligence, Morgan Kaufmann, 1977.

[Ulrich88] Ulrich, K., and Seering, W., Computation and Conceptual Design, Robotics & Computer-Integrated Manufacturing, V. 4, No. 3/4, 309-315, 1988.

[Wall88] Wall, R. S., Donahue, D., Hill, S., The Use of Domain Semantics for Retrieval and Explanation in Case-Based Reasoning, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 447- 462, 1988.

[Waltz89] Waltz, D. W., Panel Discussion on 'Indexing Algorithms', Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 25, 1989.

[Westerdale89] Westerdale, T. H., A Defense of the Bucket Brigade, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann , 282-290, 1989.

[Williams88] Williams, R. S., Learning to Program by Examining and Modifying Cases, Proceedings: Case-Based Reasoning Workshop, Morgan Kaufmann, 463-474, 1988.

[Wilensky83] Wilensky, R., Planning and Understanding, Addison Wesley, 1983.

[Winston80] Winston, P.H., Learning and Reasoning by Analogy, Communications of the ACM, V23:12, 689-703, 1980.

APPENDICES

Appendix A-1

Volatile Data Set

Maximum Case Library Size = 10

Genetic Algorithm Performed Every 2 Games

Probability of Mutation = 1

Probability of Crossover = 1

Probability of Division = 1

Probability of Connection = 1

Game Number	Case Library Strength	Average Case Strength Value
1	0.3	0.3
2	0.5	0.25
3	6.42	0.91
4	8.83	1.10
5	13.93	1.26
6	16.03	1.45
7	13.29	1.10
8	13.299	1.10
9	18.56	1.68
10	15.00	1.25
11	20.57	1.71

12	19.57	1.63
13	19.51	1.63
14	21.26	1.64
15	22.20	1.85
16	24.45	1.88
17	20.00	2.00
18	20.34	2.03
19	21.72	1.55
20	22.82	1.63
21	27.38	2.28
22	31.00	2.38
23	38.19	2.54
24	40.59	2.53
25	35.51	2.36
26	33.61	2.10
27	26.11	1.86
28	28.443	1.89
29	23.822	1.83
30	25.359	1.81
31	22.046	1.69
32	22.14	1.70
33	24.697	1.89
34	24.234	1.86

35	31.268	2.08
36	32.028	2.13
37	28.405	2.02
38	28.371	2.02
39	30.617	2.04
40	33.949	2.12

Appendix A-2

Less Volatile Data Set

Maximum Case Library Size = 5

Genetic Algorithm Performed Every 4 Games

Probability of Mutation = 0.5

Probability of Crossover = 0.5

Probability of Division = 0.5

Probability of Connection = 0.5

Game Number	Case Library Strength	Average Case Strength Value	Average Move Strength Value
1	2.72	2.720	0.247
2	4.258	2.129	0.193
3	4.458	1.486	0.165
4	4.499	1.499	0.166
5	5.174	1.293	0.161
6	5.174	1.293	0.161
7	8.302	1.66	0.193
8	8.302	1.66	0.193
9	8.972	1.281	0.195
10	8.972	1.281	0.195
11	9.039	1.136	0.185

12	9.094	1.136	0.185
13	7.092	1.182	0.164
14	7.815	1.302	0.187
15	8.042	1.148	0.167
16	11.375	1.421	0.218
17	8.562	1.712	0.203
18	9.679	1.935	0.23
19	13.011	2.168	0.271
20	12.064	1.63	0.251
21	17.207	2.28	0.264
22	18.507	2.38	0.257
23	18.607	2.54	0.248
24	21.091	2.53	0.251
25	15.525	2.36	0.277
26	13.9	2.10	0.239
27	15.297	1.86	0.263
28	18.874	1.89	0.263
29	18.686	1.83	0.269
30	19.133	1.81	0.327
31	19.433	1.69	0.335
32	21.935	1.70	0.3
33	14.438	1.89	0.343
34	15.111	1.86	0.274

35	17.727	2.08	0.322
36	18.735	2.13	0.34
37	22.457	2.02	0.366
38	24.857	2.02	0.35
39	26.157	2.04	0.353
40	26.76	2.12	0.361

END

19 02 91

FIN