INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A

SEQUENTIAL INSTANCE-BASED LEARNING FOR PLANNING IN THE CONTEXT OF AN IMPERFECT INFORMATION GAME

By

Jenngang Shih

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2000

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

April 25, 2000 — Susan L Epstein

Date — Professor Susan L. Epstein, Chair of Examining Committee

5/2000 — [signature]

Date — Professor Theodore Brown, Executive Officer

Dr. David W. Aha

Professor Cullen Schaffer

Professor Bon Sy

Supervisory Committee

The City University of New York

Abstract

SEQUENTIAL INSTANCE-BASED LEARNING FOR PLANNING
IN THE CONTEXT OF AN IMPERFECT INFORMATION GAME

By

Jenngang Shih

Advisor: Professor Susan L. Epstein

Finding sequential concepts, as in planning, is a complex task because of the exponential size of the search space. Empirical learning is an effective way to find sequential concepts from observations. Sequential Instance-Based Learning (SIBL), which is presented here, is an empirical learning approach, modeled after Instance-Based Learning (IBL), that learns sequential concepts, ordered sequences of state-action pairs to perform a synthesis task. SIBL is highly effective; it learns expert-level knowledge. SIBL demonstrates the feasibility of using an empirical learning approach to discover sequential concepts. In addition, this approach suggests a general framework that systematically extends empirical learning to learning sequential concepts. In this dissertation, SIBL is tested on the domain of bridge.

To My Parents

Chien Yuan Shih and Pi Lien Chen Shih

ACKNOWLEDGMENTS

Exploring a thesis and writing a successful dissertation are impossible without the help from many dedicated people. I have been very fortunate to have the assistance and support of such a distinct group of family, friends and colleagues in my quest to achieve my goal in this long journey. The following words can only express a fraction of my appreciation towards them.

First, I would like to thank my thesis advisor, Susan Epstein, who has been a stellar mentor, and who not only provided me with advice on numerous occasions, but also introduced me to the field of artificial intelligence, guided me through the search for research topics, and remained supportive during my entire discourse. The complexity of providing advice, counseling, encouragement and numerous reviews was compounded by the fact that I was on the west coast of California and she was on the east coast in New York. Throughout, Susan ensured her availability for the numerous discussions, endless clarifications, and many long distance phone calls that were a necessary aspect of undertaking this effort. It seemed her patience never wavered and even at the last critical moments, she continued to encourage and support me toward this end, for which I am, and will remain, deeply grateful.

Attempting to know the breadth and depth in a field of science is achievable only if you have the help from someone who has the relevant knowledge and ability to guide. David Aha has broadened my view in the field of machine learning and case-based reasoning. He served on my thesis committee and introduced me to many relevant works. He also provided me with the most elaborate, detailed and the most encouraging comments, for which I am profoundly indebted. I am delighted to acknowledge the help of Cullen Schaffer and Bon Sy who were on my thesis committee. They enhanced my work by providing practical comments, many facets of an idea, and most important of all, reality checks to ensure the quality of this work. I am honored to have had their assistance throughout this long and rewarding process.

I thank both David Aha and Matthew Ginsberg for making the computer code of their earlier work available. It enabled me to carry out my experiments with much less effort had I been required to write similar code to proceed. I appreciate the financial support provided to me for my tuition while working at International Business Corporation and BankAmerica Corporation.

Special thanks go to Robin Smith who read many of my drafts, and who is responsible for the improvement of my writing skill over the years. I also thank Deborah Lee Rose and Bernice Quinn for editing my earlier draft.

The support provided by my family was the nucleus of my strength, my will and my success. My father, Chien Yuan Shih, and my mother, Pi Lien Chen Shih, have proven their unending love and continually offered me encouragement and praise. My brothers, Jenn Chih Shih and Jenn Jen Shih, and my sisters, Su Fen Shih Wong and Su Chen Shih Chang, often picked up the slack on my family responsibilities during my long absences while I was far away from home. What a tremendous gift of love. Finally, I am deeply and forever indebted to my wife Shelly Hsiuhua Shih who constantly shielded me from daily chores, and provided me with endless patience and support. Lastly, I thank my children, Jennifer and Evelyn, for their inspiration and for their understanding when I was so frequently absent while working on this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE :

INTRODUCTION

The thesis of this dissertation is that ***learning a set of sequential concepts using an empirical learning approach is effective and produces knowledge that exhibits intelligent behavior***. A *sequential concept* is an ordered sequence of state-action pairs to perform a task. A sequential concept is similar to a macro operator, which is a sequence of actions commonly used together (Fikes and Nilsson 1971). For example, search for information on the WWW (World Wide Web) involves starting a computer, logging on to an Internet service provider, loading a web browser, choosing a search engine, and inputting search words. Although "choosing a search engine" may require additional elaboration, "loading a web browser" may be viewed as an atomic step because it is fairly straightforward. Nevertheless, "loading a web browser" consists of a sequence of more refined actions, such as locating the browser icon on the screen, moving the mouse onto the browser icon, and double clicking the browser icon. Together, these actions are commonly used to start a program, and the action sequence is an example of a macro operator. Unlike macro operators, however, sequential concepts rely on historical context, and learning sequential concepts requires historical context that varies in length to disambiguate a current state.

The ***effectiveness*** of the proposed learning method for sequential concepts is evaluated here by its significance and competence. Significance is based on a statistical evaluation of the experimental results during the development of a series of exploratory learning strategies. Competence focuses on a performance comparison between the proposed learning strategies and the strategies at an expert level. As will be shown in Chapter 4, exploratory performance improvements, with the proposed learning method, are

incremental and systematic, and the ultimate performance is comparable at an expert level.

For ***intelligent behavior***, one must demonstrate that sequential concepts are learned that mimic a behavior at an expert level. For example, "loading a web browser" is a sequential concept that can free a WWW user from paying attention to detailed actions, such as "locating the browser icon on the screen", "moving the mouse onto the browser icon", and "double clicking the browser icon." Such a sequential concept is meaningful because it accomplishes a particular task. It is also useful because it eliminates repeated re-derivation of a sequence of tightly coupled actions.

The proposed learning method is modeled after Instance-Based Learning (*IBL*), which is a general purpose, empirical learning paradigm. Since IBL mainly learns non-sequential concepts from non-sequential inputs, the method is extended with the ability to learn sequential concepts from sequential inputs using a set of sequential similarity metrics. This variation on IBL is called Sequential Instance-Based Learning (*SIBL*). Like IBL, SIBL is a general method; it requires very little domain knowledge to learn effective knowledge that exhibits intelligent behavior. As a result, SIBL may be applied to other domains whose domain knowledge is usefully represented as sequential concepts.

1.1 Dissertation Overview

The remainder of this chapter describes the area of contribution for this research. Synthesis problem solving is an important facet of human intelligence. This research focuses on machine learning, which can improve problem-solving capability on an ongoing basis. Empirical learning has emerged as a popular learning paradigm for knowledge discovery, but synthesis problems present a substantial challenge. The combination of em-

pirical learning and synthesis problems provides a unique opportunity to advance the understanding in both areas. This combination is the main focus of this research.

Chapter 2 describes sequential dependency (SD), the main concept behind this dissertation. Although the idea of SD is not completely new, its extensive application to sequential problems is previous applications with similar ideas has been limited to sequences of simple elements. With synthesis problems, an element in a sequence is a multi-faceted description of a state, and is closely associated with a global state in which a problem solver operates. The novel application of SD to a synthesis domain motivates its extension to the empirical learning method developed here.

Chapter 3 introduces the learning approach that systematically extends empirical learning with sequential dependency. In particular, SD is combined with IBL to learn a synthesis task. The combination, called SIBL, allows both quantitative and qualitative performance feedback during learning. SIBL also serves as an example for the integration of SD with other empirical learning paradigms.

Chapter 4 reports experimental results as evidence for the effectiveness and intelligent behavior of the proposed learning method. It also describes the research domain selected, the input representation, and the experiments themselves. Three separate efforts are chronicled to learn sequential concepts from input examples:

1. An exploratory step tests the results of adding additional context to an input example.

2. A quantitative approach examines the effect of combining multiple models to performance improvement.

3. A qualitative approach capitalizes on the sequential information of an input example to further improve the performance and to discover the sequential concepts represented by the input examples.

Finally, Chapter 5 emphasizes the effectiveness and intelligent behavior of the proposed learning method. It also includes a discussion of the strengths and weaknesses of the learning method. The result of the discussion suggests avenues for future research, to exploit the advantages, and to devise ways to circumvent the drawbacks.

1.2 Artificial Intelligence

Artificial Intelligence (*AI*) is a discipline that strives to understand the intelligent capacity of human beings while it tries to build intelligent artifacts that exhibits human like behaviors. The former is the focus of philosophy and psychology; the latter is the focus of computer science and engineering (Winston 1975). The work described here in intended to support an AI artifact. Specifically, the work attempts to create an intelligent program that mimics a behavior at an expert level to solve a problem by learning from problem solving experience.

Problem solving is ubiquitous and is an important facet of human intelligence. Problem solving ranges from classifying infectious blood diseases (Buchanan and Shortliffe 1984) to planning a space exploration (Tate 1994). While the former *selects* a solution from a set of predefined set of solutions, the latter *constructs* a solution by combining solution operators.

One important kind of selection is classification. *Classification* assigns a class to an object based on the description of the object. This is done by assigning, to the object, a class from a set of pre-defined set of classes. **Table 1-1** is an example of classification

(Winston 1975). There are a fixed number of objects, including house and arch. Each object is represented by a set of features, such as is-a, has-part, and is-supported-by. Since object O_1 has-part B and has-part C, and B is-a wedge, C is-a brick and B is-supported-by C, O_1 is classified as a house. Similarly, since O_2 has-part B, has-part C, and has-part D, and that B is-a wedge and is-a merge, and C and D are bricks, O_2 is classified as an arch.

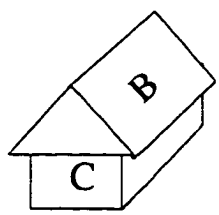
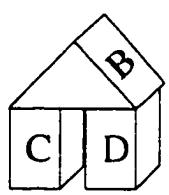
Object	Attributes	Class
O_1 	has-part B has-part C O_1: is-a B, wedge is-a C, brick is-supported-by B, C	house
O_2 	has-part B has-part C has-part D O_2: is-a B, wedge is-a B, merge is-a C, brick is-a D, brick	arch

Table 1-1. An example of objects and their classes. An object is described with a set of attributes, and each object has a class it associated with.

The principle characteristics of classification problem solving are that the number of possible solutions (classes) is fixed and the environment, or context, remains the same during solution. Examples of real world classification problems include MYCIN (Buchanan and Shortliffe 1984) and PROSPECTOR (Duda 1981).

By contrast, in a construction problem, a solution is *constructed* rather than *selected*. A common construction problem is planning. Planning constructs a solution as an operator sequence that reaches a goal state from an initial state by applying a set of sequential operators. In the blocks world domain, an example would be to move from one block configuration to a desired block configuration. Consider the descriptions of some blocks world states in **Table 1-2**. The initial state is described as a conjunction of five

predicates: On(C, A), Ontable(A), Ontable(B), Clear(B), and Clear(C); the goal state is described as a conjunction of four predicates: On(A, B), On(B, C), Ontable(C), and Clear(A). The operators are actions to be performed on the blocks. For example, Unstack(b1, b2) specifies an action that picks up block b1 off of block b2. A *plan* is constructed by evaluating the state of the world, and by selecting an action for each state. The result is a sequence of operators that transforms an initial state to a goal state. In this example, a plan to achieve the goal state from the initial state is the sequence: Unstack(C, A), Putdown(C), Pickup(B), Stack(B, C), Pickup(A), Stack(A, B).

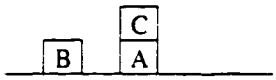
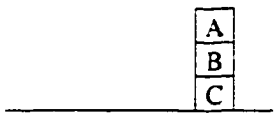
Initial State	On(C, A) Ontable(A) Ontable(B) Clear(B) Clear(C) 
Goal State	On(A, B) On(B, C) Ontable(C) Clear(A) 
Predicates	On(b1, b2): Block b1 is on block b2. Ontable(b): Block b is on the table. Clear(b): There is nothing on top of block b.
Operators	Unstack(b1, b2): Pick up block b1 from block b2. Stack(b1, b2): Place b1 on top of b2. Pickup(b): Pick up block b from table. Putdown(b): Place block b on table.

Table 1-2. An example of a blocks world with descriptive predicates. The initial state describes the world before any action is taken. The goal state describes the desired state configuration. An operator transforms one state into another. Predicates are used to specify the description of a state.

The principle characteristics of problem solving based on construction are that a problem is solved by several steps rather than by a single one, and that the possible solutions are usually orders of magnitude larger than that of selection. Examples of real world planning problems include planning the actions of a mobile robot (Georgeff and Lansky 1987), military operations planning (Wilkins 1994), and the game of bridge (Smith et al. 1998).

While selection has a fixed set of solutions, construction usually involves a solution set that is too large to enumerate. However, construction may be conveniently viewed as a series of selection problems. In other words, a selection method may be extended to a construction method by connecting a sequence of selection tasks. Consider,

	State description	Action
s_0	On(C, A) Ontable(A) Ontable(B) Clear(B) Clear(C)	Unstack(C)
s_1	Ontable(A) Ontable(B) Clear(A) Clear(B)	Putdown(C)
s_2	Ontable(A) Ontable(B) Ontable(C) Clear(A) Clear(B) Clear(C)	Pickup(B)
s_3	Ontable(A) Ontable(C) Clear(A) Clear(C)	Stack(B, C)
s_4	On(B, C) Ontable(A) Ontable(C) Clear(A) Clear(B)	Pickup(A)
s_5	On(B, C) Ontable(C) Clear(B)	Stack(A, B)
s_6	On(A, B) On(B, C) Ontable(C) Clear(A)	

Table 1-3. An example of a plan, in which each state s_i is described as a set of predicates, and is associated with an action. Each state-action pair may be viewed independently as a classification task where the state description is represented by the attributes and the action is a class.

for example, the blocks world plan in **Table 1-3**, in which each state s_i is described as a set of predicates, and is associated with an action. Each state and action pair may be viewed independently as a selection task, such that given the description of a state, an action is selected. In state s_0, for example, the state description is On(C, A), Ontable(A), Ontable(B), Clear(B), Clear(C), and the action selected is Unstack(C). This action connects the current state to the next state s_1, where another action, Putdown(C), is selected. The sequence continues until the goal state s_7 is reached. The sequence of actions forms a plan, and the construction of such a plan may be viewed as a sequence of selection tasks.

1.3 Machine Learning

While problem solving reacts to a situation, machine learning (*ML*), or learning, improves one's performance through such an interaction (Simon 1983). More recently, Donald Michie (Michie 1991) defined a learning system as one that

> "...uses sample data to generate an updated basis for improved performance on subsequent data from the same source and expresses the new basis in intelligible form."

The common notions in the two definitions are that ML improves performance and produces comprehensible knowledge from input examples. While the ability to solve a problem, such as planning a sequence of actions, is an important facet of human intelligence, the ability to learn such an intelligent behavior is undoubtedly the hallmark of intelligence. Two common characteristics for learning in an intelligent artifact are the ability to improve its performance (Breiman 1996; Quinlan 1996), and the ability to derive meaningful knowledge, or to recognize concepts for achieving performance improvement (Pazzani and Shankle 1997).

1.3.1 Performance Improvement

One way to demonstrate performance improvement of an intelligent artifact is to show the difference in performance before and after learning. Such an improvement is usually measured quantitatively by the nature and the degree of improvement. The nature of improvement may be either effectiveness or efficiency; the degree of measurement may be either absolute or relative.

As shown in **Table 1-4**, *effectiveness* addresses the accuracy of performance, that is, the percentage of correctly solved problems (Quinlan 1986). *Efficiency* addresses certain qualities of a solution, such as speed-up due to reduced length of the solution path during search (Minton 1985). With respect to the degree of improvement, an *absolute* measure evaluates the competence of the problem solver by comparing the resulting performance with a similar approach. For example, in the machine learning community, it is common practice to compare performance on data from the UC Irvine repository against other approaches (Bay 1999). On the other hand, a *relative* measure focuses on the significance of the improvement over the same approach. For example, in the development of the IBL paradigm, a series of related learning algorithms were development to explore ways to improve the learning approach (Aha 1992).

Improvement	Category	Example
Nature	Effectiveness	Accuracy
	Efficiency	Solution Path
Degree	Absolute	Performance comparison with UCI Data Repository
	Relative	The development of IB1 through IB4

Table 1-4. Performance improvement and associated examples.

1.3.2 Intelligible Concepts

In addition to performance improvement, an artifact may manifest meaningful concepts in its learned knowledge. A concept may be discovered from input examples by

comparing and by contrasting these examples. The result is a set of descriptive concepts that explicitly depict the input examples in some formalism, such as a set of predicates or conjuncts.

A simple example is learning the concept of an arch (Winston 1975). A sequence of arch examples and near misses are presented to a learning algorithm. A learning program is expected to develop a general description (concept) of what an arch is. As shown in **Table 1-1**, for example, a general description of an arch is "two standing blocks that together support a third block lying on its side."

Discovering concepts has been studied in the context of learning conjunctive feature descriptions. The input is a set of examples represented as attribute-value pairs. The output is a set of concept descriptions as the conjunction of such attribute-value pairs. For example, using positive and negative examples of iris plants, a concept description may be formed about a kind of iris, such as "A Setosa Iris has sepal length of 5.2 to 5.7 cm, sepal width of 3.0 to 3.5 cm, petal length of 4.5 to 5.3 cm, and petal width of 0.5 to 1.0 cm" (Dasarathy 1980).

In both cases, the resulting concept descriptions provide a general description of the object under investigation. Such concept descriptions may be used to verify what was known, and to illustrate what has been newly discovered.

1.3.3 Sequential Concepts

A *sequential concept* is a series of sequentially related states. **Table 1-3** provides a planning example, where s_i leads to s_{i+1} with an action a_i, $0 \leq i \leq 6$. Like the simple concepts discussed above, sequential concepts may be used to verify judgements and to illustrate learned knowledge. Unlike the simple concepts, a sequential concept involves a

sequence of state changes, which directly affects action selection for each state in the sequence. The condition a state uses to select an action is called the *context* of the state.

Sequential concepts may be illustrated in terms of a search tree T, as shown in **Figure 1-1**. The initial state s_0 is at the top of T. Given the description (context) of s_0, there are two possible actions: Pickup(B) and Pickup(C). Each of these actions leads to a new state, that is, s_1 or s_2. An important task in such state transition is loop detection. For example, the sequence of states s_0, s_2, s_3 constitutes a loop because s_3 is identical to s_0. In this case, s_3 will be ignored. In this example, only s_0, s_2, s_4, s_7, s_{10}, and s_{12} are fully expanded. A plan that transforms the initial state s_0 into a goal state, with a configuration like s_{14}, consists of six steps: s_0 to s_2, s_2 to s_4, s_4 to s_7, s_7 to s_{10}, s_{10} to s_{12}, and s_{12} to s_{14}. The associated actions are Pickup(C), Putdown(C), Pickup(B), Stack(B, C), Pickup(A), Stack(A, B), respectively.

There are at least two simple sequential concepts in the plan just described. The first sequential concept, from s_0 to s_{10}, rearranges the location of the blocks, but maintains the "shape" of the block description. The second sequential concept, from s_{10} to s_{14}, transforms a somewhat horizontal arrangement into a purely vertical one. With these sequential concepts, plan formation can be accelerated. For example, rather than six steps, with the second sequential concept, only two steps are needed to complete the plan. That is, s_0 to s_{10} and s_{10} to s_{14}. States s_0 to s_{10} form a sequential concept because the states $s_0 \ldots s_{10}$ are sequentially related. States s_{10} to s_{14} also form a sequential concept because the states $s_{10} \ldots s_{14}$ are sequentially related.

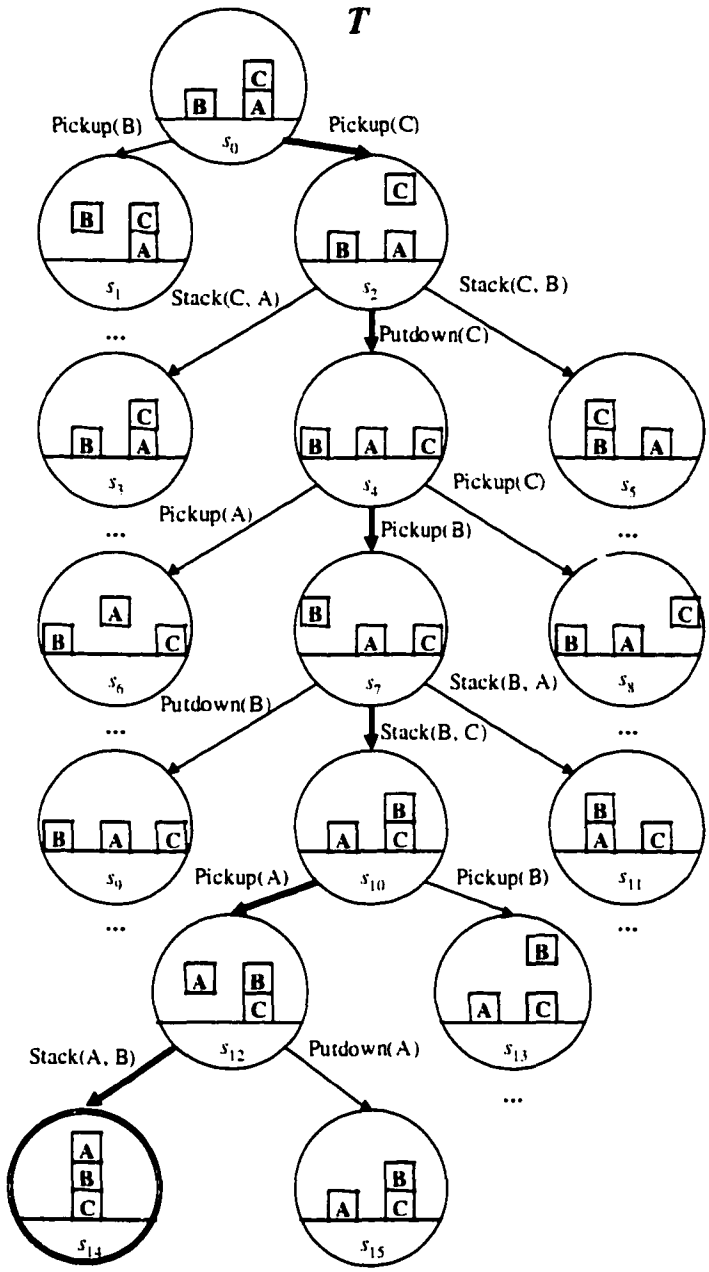


Figure 1-1. A sequential concept within a search T. Each node is a state; an arrow represents the transformation of one state into another with an action. A transition sequence is a sequence from initial state, for example, s_0, to a goal state, s_{14}. A sequential concept is a subsequence within a transition sequence, for example, s_0, s_2, and s_4.

1.4 Learning Sequential Concepts

Empirical learning paradigms have been used successfully to extract concepts and to form concept descriptions (Langley and Simon 1995). Their primary applications,

however, are analysis tasks, such as classification. That is, given a state s and an analysis mapping function f, a class c can be derived, that is, $f(s) = c$.

This thesis proposes an empirical machine learning approach for synthesis tasks. Specifically, the approach is modeled on Instance-Based Learning (IBL), an empirical learning paradigm for analysis tasks. IBL is a general-purpose learning method that employs minimal domain knowledge (Aha 1992; Salzberg 1991). In particular, IB3 is able to work with noisy examples and IB4 tolerates irrelevant attributes. Until now, however, IBL has been used primarily for analysis tasks, while this work demonstrates an instance-based method to solve synthesis tasks. The new method shows how knowledge may be acquired from sequential examples, and is demonstrated in the context of a knowledge-poor, imperfect information domain.

As discussed, problem solving can be either analysis or synthesis (Clancey 1985). An analysis task, such as classification, uses static knowledge to interpret an unknown quantity from known ones. A solution s_i is selected from a set of n solutions $\{s_1, s_2, ..., s_n\}$ with a selection function f, so that input of problem x produces the solution, that is, $f(x) = s_i$. In contrast, a synthesis task, such as planning, uses dynamic context description to construct a sequence of problem-solving steps. This involves the construction of an ordered set of actions, $\{a_1, a_2, ..., a_m\}$. These actions transform an initial state of the world S to a desired state G according to the function g of the problem-solver, so that $g(a_1, S) = H_1$, $g(a_2, H_1) = H_2$, ..., $g(a_m, H_{m-1}) = G$. The H_i are intermediate states. A synthesis problem can therefore be viewed as a sequence of analytical problems, where $x = (a_i, H_{i-1})$ and $s_i = H_i$. In general, synthesis tasks, which are the subject of this dissertation, are less tractable.

An empirical method used to learn analysis tasks may therefore be systematically extended to learn synthesis tasks. One way to accomplish this is to use the notion of se-

quential dependency, so that at any given decision point in a sequence, a decision is made by using the information from one or more sequentially related prior states. That is, let $\{s_1, s_2, ..., s_n\}$ be a sequence of states, and let $\{a_1, a_2, ..., a_n\}$ be a sequence of actions taken in those states. Action a_i taken in state s_i because the synthesis mapping function g takes as input not only the description of i but also the description of one or more related states prior to the current state s_i: $g(s_k, ..., s_{i-1}, s_i) = a_i$, where the number of related prior states is $i - k + 1$.

1.4.1 Motivation

Without a domain theory, a learning program must resort to (weak) data-intensive methods. Even with such a theory, it is often necessary to update such a theory with problem-solving examples. Furthermore, the availability of large amounts of data in synthesis domains is the primary motivation for applying empirical learning to sequential concept discovery. Empirical learning may be used to validate existing knowledge and to discover new knowledge. Empirical learning has also matured in recent years with an explosion of knowledge discovery applications (Fayyad et al. 1996).

In addition, the acquisition of synthesis knowledge has been either domain-dependent or knowledge-dependent. A domain-dependent application accumulates knowledge useful only to the particular domain (Ginsberg 1999; Smith et al. 1995). A knowledge-dependent application requires extensive domain knowledge as input to the application (Barrett 1994; Hanks and Weld 1995; Penberthy and Weld 1992). Because they are knowledge-intensive, these systems have limited applicability to a new domain.

In short, analysis tasks have been the primary focus of empirical learning, and a knowledge intensive approach has been the paradigm of choice for synthesis problem solvers. Relatively few efforts use empirical learning to bootstrap knowledge-poor do-

mains for synthesis tasks (e.g., (Reiser and Kaindl 1994; Wang 1996)). The combination of these factors motivated this research.

1.4.2 Assumptions

To make learning sequential concepts possible, some assumptions about the source of knowledge, the domain under investigation, and the experimental design are necessary. First, in empirical learning, the main source of knowledge is data. Data represents past experience, and the kind of data gathered reflects the type of knowledge to be learned. In this research, the goal is to demonstrate the proposed learning algorithms' ability to improve, and to replicate the knowledge exhibiting intelligent behavior in a synthesis domain. The input data is expected to be noisy, so that data values may be inconsistent from one input example to another. In this research, noise is due to inadequate context for an example, so that two apparently similar examples may have two different outcomes. Such noise provides an opportunity to explore ways to utilize context to disambiguate two inconsistent examples.

Second, the domain for learning sequential concepts must address solutions with multiple steps. Each step has a state description that mandates an action. The action transforms the current state to another state, all with a common goal. The assumption here is that states are sequentially related. A state is related to one or more states before it and/or one or more states after it. This assumption allows the learning algorithms to take advantage of the changes between states as a means to discover a sequential concept.

Finally, the experimental design must provide a framework to demonstrate the learning ability of the proposed algorithms. As discussed earlier, two criteria for evaluating a learning algorithm are relative improvement and absolute achievement. *Relative improvement* is an incremental measure of improvement both before and after learning. *Ab-*

solute achievement is a measure of achievement according to a predefined target. The assumptions here are that relative improvement demonstrates the ability to learn, and that absolute achievement shows the competence of the algorithms and their potential for similar domains.

1.5 Related Work

Heuristic search (Sacerdoti 1974; Stefik 1981a; Tate 1977), machine learning (Kambhampati and Chen 1993; Katukam and Kambhampati 1994; Minton 1988), case-based reasoning (Alterman 1988; Hammond 1989; Turner 1988), and combinations of them (Bhansali and Harandi 1993; Hanks and Weld 1995; Veloso 1994) have all been used to address synthesis problems.

1.5.1 Heuristic Search

Synthesis problems can be solved with heuristic search, which uses domain-independent search techniques and domain-specific heuristics to generate solutions. For example, in planning, input to a planning process includes a set of operator definitions, the description of the initial state, and the description of the goal state. The output is a plan that satisfies the goal specifications (Fikes and Nilsson 1971). A state space planner represents the search space with a set of possible states of the world (Fikes and Nilsson 1971). A plan space planner represents the search space with a set of partially ordered plans (Sacerdoti 1975). Domain-independent search techniques include depth-first search with backtracking (Tate 1977), searching in goal hierarchies (Sacerdoti 1974), opportunistic reasoning (Stefik 1981b), and various domain-specific methods (Tate and Whiter 1984; Vere 1983; Wilkins 1983).

The operators in a plan may be totally ordered (Fikes and Nilsson 1971) or partially ordered (Sacerdoti 1975). A set of operators is totally ordered if changing the order of the operators will produce different results. A set of operators is partially ordered if changing the order of some specified operators will not affect the results. The sub-goals of a planning problem may be linearly (Fikes and Nilsson 1971) or non-linearly (Sussman 1973) ordered. A set of linearly ordered goals relies on the assumption that goals are achieved in a fixed order. A set of non-linearly ordered goals does not rely on such a linear assumption.

Planning problems can also be solved with goal regression by using hierarchical methods (Sacerdoti 1975; Tate 1977; Wilkins 1984). To achieve the desired goal, hierarchical planners start with a description of the abstract tasks required. The abstract tasks are successively made more specific until they can be carried out by a problem-solver.

All these problem-solving techniques rely heavily on domain-independent search techniques and domain-specific heuristics to limit search. The more one wants to reduce search, the more knowledge is needed. In most knowledge-poor domains, however, no such gain in search reduction can be realized.

1.5.2 Machine Learning

Although planners that use heuristic search create efficient plans, they repeat the same efforts to produce the same solutions, without improving their performance. Planners that use machine learning (ML), however, improve their performance as a result of solving a new problem. ML typically uses abstraction to deduce more efficient knowledge, or generalization to induce new knowledge (Michalski 1993). The learned knowledge is represented as concept descriptions.

Experimental Goal Regression (*EGR*) learns problem solving with minimal requirements for *a priori* knowledge (Porter and Kibler 1986). PET is an implementation of EGR that solves simultaneous linear equations and symbolic integration. It may be viewed as a learning apprentice that consists of a user-interface, a simple problem solver, and a learner. PET represents a problem in terms of a set of states, a set of operators and a fixed goal. Given a problem, the user suggests an operator. The problem solver applies the operator and is capable of detecting goal states. The learner forms heuristic rules from user inputs. These rules are then used to guide the problem solver on subsequent problems. Unlike empirical learning, which relies on many input examples, PET acts as a learning apprentice. Unlike explanation based learning, which provides a derivation (proof) to produce a solution, PET performs goal regression (back-propagation of goal conditions).

Plan learning (*PL*) applies learning techniques to planning problems. In planning, knowledge can be represented as a set of operator concepts. An operator concept consists of the definition of the operator and a set of guidelines or policies for applying the operator. Like other ML applications, PL uses either abstraction or generalization to improve planning performance (Veloso and Borrajo 1995). PL based on abstraction changes plan representation to make existing plans more effective (Kambhampati and Chen 1993; Katukam and Kambhampati 1994; Minton 1988). PL based on generalization produces generalized (but possibly overly specific or overly general) planning knowledge. As new planning examples become available, this planning knowledge can be refined (Borrajo 1994; Wang 1996).

Like ML, PL has two learning objectives. For knowledge-rich domains, the objective is to enhance the efficiency and/or quality of the existing knowledge and to extend

its applicability (Minton 1988). For knowledge-poor domains, the objective is to acquire basic knowledge (Borrajo 1994; Wang 1996).

Unlike heuristic search, PL will improve performance over time with generalization or abstraction. However, like heuristic search, PL relies heavily on domain knowledge in most cases. This limits PL's applicability to knowledge-poor domains and makes its results difficult to apply to a different domain.

1.5.3 Case Based Reasoning

Instead of learning and reasoning with abstraction or generalization, case-based reasoning (*CBR*) works with actual examples (*cases*). To solve the current problem, CBR retrieves a similar case and adapts the retrieved solution (Kolodner 1993b). CBR also improves performance through learning (Kolodner 1993a). However, learning is normally deferred until a new problem is presented. Like learning, CBR avoids costly derivation of a solution from scratch. Unlike learning, however, most CBR systems represent knowledge with operational (*ground*) information and do not produce abstraction or generalization.

Case-based planners apply CBR to planning problems. A case-based planner retrieves a similar planning problem and adapts the planning steps to produce a desired plan. Early case-based planners include CHEF (Hammond 1989), TRUCKER (Marks et al. 1988), MEDIC (Turner 1988), and PLEXUS (Alterman 1988). Typically, they used domain knowledge and stored plans to guide plan retrieval and adaptation. More recent case-based planners combine case-based reasoning with heuristic search methods (Bhansali and Harandi 1993; Hanks and Weld 1995; Munoz-Avila and Weberskirch 1996) and machine learning algorithms (Bergmann 1995; Borrajo 1994; Veloso 1994) to improve their performance.

As another example, SaxEx applies CBR to transform a musical performance's expressiveness (Arcos et al. 1999). The input to SaxEx is a musical performance, a set of notes. Each note is represented by descriptive values for a set of parameters, such as dynamics, rubato, vibrato level, articulation, and attack. Similarity criteria based on musical knowledge and a set of stored cases are used to infer possible expressive transformations for a given musical performance. As part of the similarity criteria, the melodic direction, that is, ascending or descending direction of an adjacent note, is used in the retrieval process. SaxEx deals with a perfect information domain where the input sequence is known ahead of time. That is, a musical performance is completely known before it is transformed. SaxEx uses minimal sequential information, the melodic direction of only the following note. It does, however, rely on sequential information to calculate similarity between two examples.

Most case-based reasoning systems use both cases and domain knowledge to guide learning and reasoning. Some use cases with very little domain-specific knowledge and progress to become knowledge-rich systems. PROTOS, for example, starts with very little domain-specific knowledge. It gradually becomes a knowledge-rich system with domain-independent search heuristics for pattern matching. In heuristic classification tasks, PROTOS acts as a learning apprentice and changes from a knowledge-poor to a knowledge-rich system (Porter et al. 1990). Whether presented all at once or gradually, domain knowledge is the main ingredient for most case-based planning systems.

1.5.4 Hybrid Systems

The following survey is based on hybrids of heuristic search, machine learning, and case-based reasoning.

APU (Automated Programmer for Unix) combines a hierarchical planner with analogical reasoning to improve the performance of synthesis tasks (Bhansali and Harandi 1993). The hierarchical planner decomposes problems into sub-problems, while the analogical reasoning system replays the solution of a similar stored case. To accomplish these tasks, APU requires a knowledge base of problem-decomposing rules and a set of derivational heuristics. This approach is faster than planning from first principles and avoids costly backtracking during searching.

PAIAR, a framework that supports retrieval and modification of plans for reuse, combines hierarchical non-linear planning with case-based reasoning (Kambhampati 1993). It uses the least-commitment nature of non-linear planning to increase the applicability of stored cases to new planning problems. Domain-independent similarity metrics are needed to retrieve a suitable plan for reuse. The retrieved plan is refined to produce primitive tasks. The result is a planner that can avoid exponential complexity for plan generation.

SPA (Systematic Plan Adapter) is an adaptive planner that searches a graph of partial plans (Hanks and Weld 1995). It is based on SNLP (Systematic Non-Linear Planner), a partial-order, constraint posting, least-commitment generative planner (McAllester and Rosenblitt 1991). Instead of generating a plan from the root of the graph, SPA starts the search at a selected node in the graph, based on a retrieved partial plan using retrieval heuristics. The partial plan is then adapted by either extending or retracting the retrieved partial plan to achieve the desired goal using adaptation heuristics. Finally, the solution plan is generalized and stored in the plan library for future use.

CAPlan/CbC also combines SNLP and case-based reasoning to solve planning problems (Munoz-Avila and Weberskirch 1996). It uses non-linear planning heuristics, case-based reasoning, and domain-specific knowledge to improve planning performance.

Since the main focus of the planning method is to combine generic planning techniques with domain-specific knowledge to improve planning performance, the program requires domain-independent heuristics, domain-specific knowledge, and problem-solving examples.

PRODIGY/ANALOGY records decisions of a planning episode and uses derivational analogy (Carbonell 1983; Carbonell 1986) to guide the search for similar planning problems (Veloso 1994). PRODIGY (Veloso 1995), a state-space, non-linear planner, is used to generate planning cases and record their solution traces. ANALOGY solves a new planning problem by retrieving a similar stored case and replaying the solution, resorting to the PRODIGY general problem-solver if needed.

PARIS combines learning and case-based reasoning to solve hierarchical planning problems (Bergmann 1995). It uses learning to change the representation language. That is, it moves between concrete and abstract representations, using an abstract domain theory. The domain theory composes of abstract operators, which may be refined independently. Compared to other hierarchical planners, which achieve plan abstraction by dropping sentences in a plan description, PARIS changes the plan description to a more efficient representation to improve performance.

In summary, the proposed learning method is positioned as follows. Like heuristic search, the proposed learning method attempts to address synthesis problems, such as planning. As a kind of ML, the salient characteristic of the proposed learning method is its ability to learn from examples, specifically, from sequential examples. Finally, as in CBR, the proposed learning method defers updates to its database until it is presented with an example, and retains ground-level information (cases) rather than abstractions.

CHAPTER TWO :

SEQUENTIAL DEPENDENCY

The previous chapter argued for applying empirical learning to a synthesis problem. This chapter introduces a key ingredient, sequential dependency (SD). The first part of this chapter discusses the characteristics of a sequence, since most synthesis problems involve a sequence in one way or another. The second part compares SD with four other methods, each of which uses sequences in its representation. Although the sequence is a common representation, they differ in terms of the complexity of an element in a sequence, and the size of the search space introduced by a sequence. These descriptions emphasize the difference between the ways these methods use sequences and the way SD uses sequences. The third part is the focal point of the chapter. It describes what SD is, why SD is useful, and how to work with SD. The last part of this chapter uses planning and game playing as examples to suggest ways SD may be used with a synthesis domain.

Some basic concepts and definitions are in order here. In some situations, events in a sequence are related to each other, so that each event initiates the next one. In other situations, events in a sequence are triggered by the initial event, as in a chain reaction. In chemistry, for example, a catalyst can start a chain reaction with certain combinations of chemicals, which causes further self-sustaining reactions of the same kind. In biology, the polymerase chain reaction is a process that duplicates DNA from an initial configuration with a number of selected elements. A chain reaction may be best explained with dominos lined up in a row. A push of the leading domino causes the next one to fall, and so on, until all the dominos topple. Each of these examples can be described as a sequence of chain reactions, in the form of state-action pairs. Here a *state* is a description of physical, chemical, biological or other domain-specific conditions, and an *action* is the behavior influenced or incurred by those conditions, or states. The order of the state-action pairs

reflects their dependencies towards an intended destination or result, called a *goal*. By examining the initial event, the succeeding events, and the triggered reactions, one can predict the subsequent events. In other words, a sequence of cause and effect pairs can provide predictive information. *Sequential dependency*, the existence of such ordered dependencies, is the conceptual basis of this thesis.

Sequential dependency (*SD*) is a useful approach to problem solving in domains with chain reactions. Similar to state-based planning (Fikes and Nilsson 1971), SD views a problem solving sequence as a sequence of state-action pairs. SD extends state-based planning by considering the current, and a number of prior, state descriptions. The extra state descriptions form a multi-state context, which help disambiguate a problem-solving situation in a complex domain. To conserve resources, SD limits the number of states to consider during problem solving. This implies the identification of a group of closely related state-action pairs, called *sequential concepts*. Sequential concepts are expected to be purposeful, and to serve as the building blocks of a plausible *plan*, an ordered sequence of actions. Given a correct sequential concept, one could carry out a plan successfully.

2.1 Sequences

A *sequence* of states or occurrences is a contiguous ordered set of such states or occurrences. The term sequence has different meanings in different domains. In time series domains, for example, sequence is typically a series of numeric measurements over a period of time, such as a sequence of temperatures measured at a fixed interval over a period of a day. This sequence of states represents an aspect of the weather, and, along with other factors, may enable the projection of future weather patterns. In symbolic domains, a sequence is typically a chain of symbols that represents sequential information. For example, in DNA secondary structure prediction, a sequence is a string of alphabetic sym-

bols that represent various amino acids. At each point in the sequence, the features associated with the symbol suggest the category of the secondary structure of the DNA sequence.

A sequence may also be thought of as a plan, a series of consecutive state-action pairs to guide the problem solver from an initial state to a goal state. At each state, an action is selected based on the current state description, which takes the problem solver to the next state. For example, in the blocks world domain (Winston 1975), a plan is a sequence of state-action pairs to manipulate objects in three-dimensional space in order to attain a desired configuration of the objects.

2.1.1 Search Space

One difference among sequences is the nature of the basic element in a sequence. In sequence categorization, for example, the basic element is a simple object and is usually mapped to only one of a finite set of symbols. In planning, on the other hand, the basic element is a state, a more complex object with multiple facets that can be instantiated to a potentially large number of objects.

Specifically, in DNA secondary structure prediction, a symbol in a sequence maps to only one amino acid. The maximum branching factor from one symbol in a sequence to the next with repetition is 20, the total number of amino acids. In a planning domain, such as the game of bridge, where a sequence of plays is carried out incrementally to make a contract, the branching factor is much larger, due to incomplete information. Specifically, the branching factor is $n \times m^3$ at each state, where n is the number of cards currently left in the lead player's hand, and $m = n/4$ is the average number of choices the other three players have to play a card among the four possible suits. For example, the branching factor at the beginning of a deal, when all four players still have 13 cards, is

$13 \times (13/4)^3 = 446$. Therefore, compared to DNA secondary structure prediction, planning domains like the game of bridge have a much larger search space for sequences due to the difference in the size of branching factor in a sequence.

2.1.2 State Transition Models

Another difference among sequences is whether the sequence as a whole has a concise description, or model. In time series domains, for example, sequences are commonly expressed in a finite state model. Such a model enables the collection of state transition probability distributions used to generate or to recognize these sequences. In planning, on the other hand, sequences of plans tend to be context sensitive and therefore require a more expressive and complex description.

As an example, a popular approach to time series prediction is the Hidden Markov Model (HMM) where sequences are expressed in a finite state model. Given a HMM with n states and m symbols, the complexity of such model is $Y=(C(m, n))!$, where $X=C(m, n)$ is the total number of possible sequences, and $Y = X!$ is the total number of ways a sequence can fit in the HMM. As long as the number m is finite and is relatively small, it is possible to collect sufficient state transition probability distributions to make a reasonable prediction for sequences the model represents. In planning, as in the game of bridge, on the other hand, the number of states is much larger (i.e., $m = 446$). As a result, both X and Y are much larger. There is no effective method to consolidate this problem into such a model; the size of the state would not be manageable. For this reason, it is infeasible to express a planning problem, such as bridge, in a finite state model.

2.2 *Related Methods*

This section compares SD with four related methods: (1) sequence categorization, (2) sequence prediction, (3) sequential pattern mining, and (4) reinforcement learning. As discussed earlier, SD exploits a multi-state context in learning and solving complex planning problems in which a plan consists of a sequence of related states. Three closely related methods are sequence prediction (Rabiner 1989), sequence categorization (Kudenko 1998), and sequential pattern mining (Agrawal 1995). As illustrated in **Figure 2-1**, both sequence prediction and sequence categorization involve a sequence of elements. Like SD and planning, *sequence categorization* and *sequential pattern mining* attempt to search for an ordered set of symbols, called sequential patterns. Like SD and planning, *sequence prediction* involves state transitions towards a desired goal. Finally, *reinforcement learning*, like SD, uses a multi-state context in state disambiguation. The two differ, however, on the feedback each uses to learn.

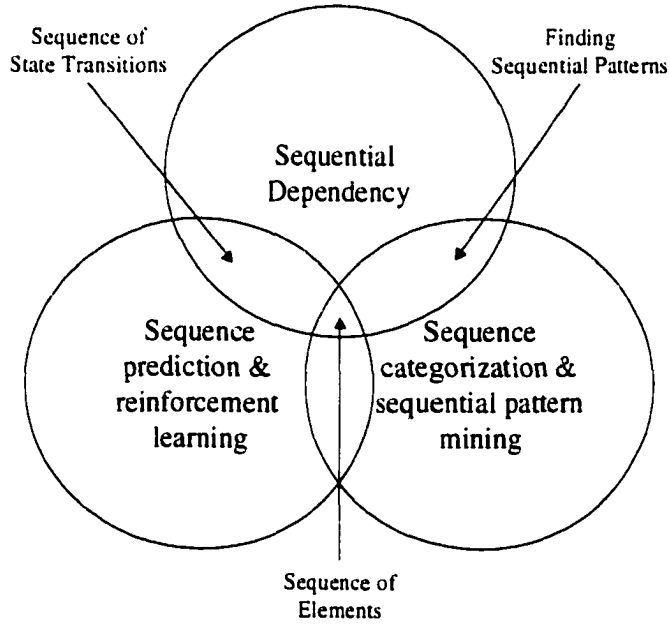


Figure 2-1. Commonalties among SD and other methods.

2.2.1 Sequence Categorization

Sequence categorization (*SC*) assigns a category to a finite sequence, a string of elements of a fixed length. The function for category assignment is defined in a model that maps a sequence to a category from a finite set of categories. DNA secondary structure prediction is an example of SC. It predicts the category of the secondary structure of a DNA chain. A DNA chain is a sequence of amino acids, each of which is one of the 20 types of amino acids, represented by a set of 20 alphabetic letters (Qian 1988). The model maps a DNA chain to one of three secondary structure categories, alpha helix (α), beta sheet (β), or coil (χ). An example of a DNA chain is (KQPEEPWFα), where α is the category of the secondary structure at the last amino acid F.

Memory-based reasoning (*MBR*) (Kasif et al. 1998) may be used for SC. Continuing the DNA example, the category of a secondary structure is assigned to a DNA sequence by retrieving a similar DNA chain previously stored in an instance base, or a database (Cost 1993). Using MBR, SC for a DNA chain may be accomplished in the following three steps (See **Figure 2-2**):

1. Calculate the similarity between an input sequence and stored sequences, where each stored sequence is a string-category pair.

2. Select the sequence that is the most similar to the input sequence.

3. Assign the category of the selected sequence to the input sequence.

As a follow-up to these reasoning steps, if later the category assigned to the input sequence proves incorrect, the input sequence along with the correct category, can be stored (or learned) as a new example in the database.

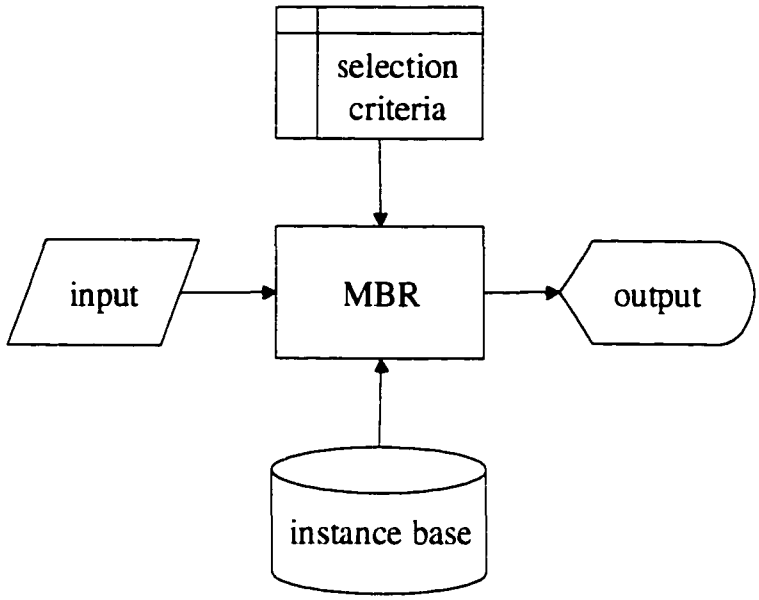


Figure 2-2. In a memory-based reasoning method, input is used to retrieve a stored example from the instance base according to selection criteria.

The input sequence and the stored sequences are sometimes called *instances*, which are commonly represented by a set of attribute-value pairs. In addition to representing a problem, such as a DNA chain, a special attribute is used to represent the solution, such as the category of a DNA chain. The attributes and possible values for each attribute define all possible instantiations for the instances that can be expressed in a multi-dimensional space called the *instance space*. An instance, therefore, is an instantiation of an attribute-value pair in the instance space.

In step 1 above, most MBR algorithms calculate similarity in terms of the distance between two instances. The smaller the distance, the more similar they are, and vice versa. An example from (Aha 1990) is the Euclidean distance, which measures the straight-line distance between two instances, s_1 and s_2, as follows,

$$distance(s_1, s_2) = \sqrt{\sum_{i=1}^{n} difference_i(s_1, s_2)^2} \qquad [1]$$

and the function *difference* between the values of attribute f_i in s_1 and s_2 is defined as,

$$difference_i(s_1, s_2) = \begin{cases} |f_i(s_1) - f_i(s_2)|, \text{ if } f_i \text{ is numeric.} \\ 1, \text{ if } f_i \text{ is discrete and } f_i(s_1) \neq f_i(s_2). \\ 0, \text{ otherwise.} \end{cases} \quad [2]$$

In step 2 of MBR, the purpose is to select the most plausible solution. The selection criteria can be quantitative or qualitative. A quantitative selection method, such as *majority vote*, simply picks the most popular solution among the top selections. The qualitative method compares certain characteristics of two competing stored instances, such as their performance records in terms of prediction accuracy. Once a stored instance has been selected, its solution is applied to the input problem.

As an illustration, given a DNA chain x = GNDVEYX, MBR first calculates the similarities between x and the stored DNA sequences in the instance base. It then selects a set of plausible solutions from the set of stored DNA sequences like those in **Table 2-1**. If the quantitative selection method is used, the category (solution) will be α because it is the majority category among the stored instances. On the other hand, if the qualitative method is used, the category will be β because s_2 has the highest accuracy record.

	DNA Sequence	Accuracy
s_2	VEYYGQV, β	.48
s_5	AFDQVSA, α	.34
s_3	IGTPGKS, α	.31
s_4	PQSSTNA, α	.26
s_1	GVGTVPM, χ	.22

Table 2-1. A set of five hypothetical DNA-sequences, and their performance in terms of prediction accuracy.

As discussed in Section 2.1.1, the search space for a sequence of complex objects is several orders of magnitude larger than that for a sequence of simple objects. A sequence of n elements, each from a set of m symbols, has a search space of n^m. In DNA sequence prediction, for example, a sequence of n = 7 symbols, each may be one of m = 20 symbols, has a search space of 7^{20}. With a sequence of complex elements, the search space is much larger. In the game of bridge, for example, a play sequence of n' = 7

states, each from one of $m' = 446$ state descriptions, the search space is $7^{446} >> 7^{20}$. With such an enormous search space, an effective approach is imperative.

2.2.2 Sequence Prediction

Sequence prediction (*SP*) predicts the most likely element that is preceded by a sequence of elements. In a time series domain that uses methods, such as HMM, SP relies on well-defined probability distribution for elements and state transitions in a time series. Given a finite number of elements and a state transition model, the probability distribution for the elements at each state, and the state transition probabilities can be calculated. The most likely next element in the sequence is the one with the highest probability and the most likely next state has the highest transition probability (Rabiner 1989).

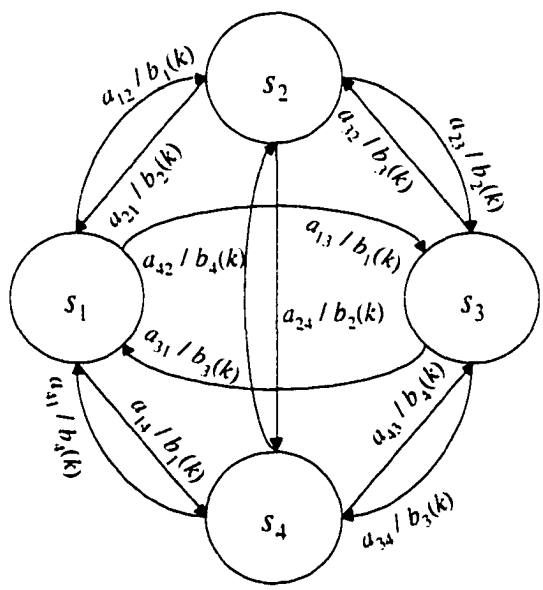


Figure 2-3. A state transition diagram depicts a state transition model. Here, an arc represents a state transition, where a_{ij} is the state transition probability from state s_i to state s_j, and $b_i(k)$ is the probability that the element k is associated with state s_i.

Speech recognition is an example that uses HMM to predict the most likely spoken word in a sequence of spoken words. For example, what is the most likely word, among {away, and, or, not, now}, that follows the sequence of words, "Mr. Write is

writing a letter right ..." Using HMM, the problem can be formulated as a finite set of m words, a finite state transition model with n states, state transition probability distributions A, and the probability B_k for each word w_k at each state. The most likely word at the present state is the one with the highest B_k and the next state will be the one that has the highest state transition probability defined in A.

More formally, an HMM λ consists of a set of m symbols $V = \{v_1, \ldots, v_m\}$, a finite set of n states $S = \{s_1, \ldots, s_n\}$, a set of state transition probability distributions $A = \{a_{ij} \mid a_{ij} = \Pr(s_j \text{ at } t+1 \mid s_i \text{ at } t)\}$ a set of observation symbol probability distributions $B_j = \{b_j(k) \mid b_j(k) = \Pr(v_k \text{ at } t \mid s_j \text{ at } t)\}$ for a given state s_j, and the initial state distribution $\pi = \{\pi_i \mid \pi_i = \Pr(s_i \text{ at } t = 1)\}$. The HMM $\lambda = \{V, S, A, B\}$ is used to calculate the probability of an observation sequence O, $\Pr(O \mid \lambda)$ to predict the observation sequence (See **Figure 2-3**).

Probability distributions can be built from sample sequences. That is, for each element w, an HMM $\lambda_w = \{A_w, B_w, \pi_w\}$ is constructed from a set of training sequences, where π_w is the initial state distribution, A_w is the state transition probabilities, and B_w is the observation probabilities in each state for w. As an example, **Table 2-2** illustrates a hypothetical state transition probability A for states s_1, s_2, and s_3, and word sequence probability B for the set of words W = {away, and, or, not, now}. A represents the state transition probability a_{ij} from state s_i to s_j, $1 \le i, j \le 3$; B represents the word sequence probability $b_j(k)$ for word w_k at state s_j, $1 \le j \le 3$ and $1 \le k \le 5$. In this example, if the state at time t is s_1, the most likely next state at time $t+1$ will be s_3. This is because $a_{13} = .57$ is the highest state transition probability, and the most likely word at state s_1 is *now* because $b_1(k) = .34$ is the highest word sequence probability for s_1.

SP using HMM requires a correct estimation of the probability distributions to work well. It is usually applied to sequential problems with a finite set of states. When

the number of states increases, it is less likely to have complete probability distributions because of insufficient samples. In a planning problem such as the game of bridge, the number of states could reach 10^{28} because there are 5.36E+28 possible deals, that is, C(52,13) × C(39,12) × C(26,13) × C(13,13). In such a large state space, it is not feasible to estimate the probability distributions. For this reason, HMM and similar approaches cannot be used for sequential concept discovery with a large number of states. SD, the alternative investigated here, does not use a finite state transition model. Rather, it uses a multi-state context to provide context-related information to limit the search for a sequence.

	A			*B*				
	s_1	s_2	s_3	***away***	***and***	***or***	***not***	***now***
s_1	0.25	0.18	0.57	0.04	0.25	0.23	0.15	0.34
s_2	0.58	0.04	0.38	0.25	0.17	0.12	0.21	0.24
s_3	0.08	0.65	0.27	0.14	0.04	0.24	0.05	0.53

Table 2-2. Hypothetical state transitions (*A*) for state s_1 s_2 s_3, and word probability distributions (*B*) among the words "away", "and", "or", "not", and "now."

2.2.3 Sequential Pattern Mining

More recently, an active research topic in knowledge discovery in databases and data mining (*KDD*) is sequential pattern mining (*SPM*) (Agrawal 1995). The input is a *data sequence*, which is a list of elements, called an itemset. An itemset is a compound object, which contains one or more events. As shown in **Table 2-3**, The elements in data sequences, such as [c], [j], [a b], and [d f g], etc., are examples of itemsets. Each itemset in the sequence is marked with the time of occurrence to establish the ordering relationship among these itemsets. A *sequential pattern* is a subsequence of the data sequence such that the percentage of data sequences that contain the sequential pattern, called *support*, is greater than a predefined threshold. Detection of a sequential pattern establishes a link among the itemsets in the sequential pattern within a data sequence. For example,

given the five data sequences in **Table 2-3**, {[c] [j]} is a sequential pattern with a support of 40% because two (1 and 4) out of the five data sequences have the sequential pattern.

	Data sequence
1	{[c] [j]}
2	{[a b] [c] [d f g]}
3	{[c e g]}
4	{[c] [d g] [j]}
5	{[j]}

Table 2-3. A list of five data sequences, each contains one or more itemsets, for example, sequence 1 = {[c] [j]} contains two itemsets: [c] and [j].

Time constraints may be imposed on a sequential pattern and on an itemset. For a sequential pattern, two itemsets, [c] and [j], are considered consecutive if the time difference between the occurrences of the two itemsets is within a predefined time interval t_{min}. In other words, for the sequential pattern {[c] [j]} to be considered, the time difference $\Delta([c], [j]) = t_{[j]} - t_{[c]}$ must be smaller than t_{min} , where $t_{[j]}$ marks the time of occurrence for the itemset [j] and $t_{[c]}$ marks the time of occurrence for the itemset [c]. Similarly, within an itemset, two events, *a* and *b*, are considered consecutive if the time difference between the occurrences of the two events is within a predefined time interval t'_{min}. That is, for the itemset [a b] to be considered, the time difference $\Delta(a, b) = t_b - t_a$ must be smaller than t'_{min}, where t_a marks the time of occurrence for the event *a* , and t_b marks the time of occurrence for the event *b*. Some applications of SPM are detecting telecommunication equipment failures (Weiss and Hirsh 1998), discovering interesting patterns in text databases (Lent et al. 1997), and detecting plan failures (Zaki et al. 1998).

As described in (Agrawal 1995), SPM may be accomplished in the following four steps:

1. Sort input data sequences.

The input database *D* of data sequences is sorted by the source of the data sequences (e.g., customer id in a transaction database), and by time of occurrence. **Table 2-5** is an example of sorted transition sequences where column one denotes the order of the data sequences.

2. Find sequences with minimal support.

The purpose of the current step is to find the set *L* of all 1-sequences with minimal support. A sequential pattern with *k* itemsets is called a *k*-sequence. A sequence *s* has *minimal support* if *s* occurs at least *x* times among a total of *y* data sequences, where *x*/*y* is a predefined threshold. For example, if the threshold is 25%, **Table 2-4** lists all 1-sequences in *D*.

1-sequence	support
{[c]}	80%
{[d]}	40%
{[g]}	60%
{[d g]}	40%
{[j]}	60%

Table 2-4. A list of 1-sequences, each contains one itemset, and their supports, which is the percentage of such a sequence occurs in a set of all input data sequences.

3. Transform data sequences.

The purpose of transformation is to facilitate an efficient procedure for repeatedly searching for the set of desired sequential patterns described in the next step. This is done by forming a set of 1-sequences with minimal support from the original set of data sequences. To accomplish this, the algorithm removes 1-sequences that do not have mini-

mal support and expands the original data sequences in terms of the 1-sequences. As shown in **Table 2-5**, sequence 4 = {[c] [d g] [j]} contains all 1-sequences with minimal support according to **Table 2-4**. Therefore, the transformed representation is the set of sequences: {[c]}, {[d] [g] [d g]}, and {[j]}. On the other hand, sequence 2 = {[a b] [c] [d f g]} contains some 1-sequences, [a b] and [f], that do not have minimal support. Therefore, the transformed representation will not included such 1-sequences and the result is the sets: {[c]} and {[d] [g] [d g]}.

	Original	Transformed
1	{[c] [j]}	{[c]} {[j]}
2	{[a b] [c] [d f g]}	{[c]} {[d] [g] [d g]}
3	{[c e g]}	{[c] [g]}
4	{[c] [d g] [j]}	{[c]} {[d] [g] [d g]} {[j]}
5	{[j]}	{[j]}

Table 2-5. The data sequences in the original column are transformed into the representation in terms of 1-sequences that have minimal support.

4. Find sequences.

This step forms new and longer data sequences by repeatedly combining shorter ones. All resulting data sequences with minimal support are used to build yet longer data sequences, and those that do not have minimal support are removed. The process starts with the transformed data sequences derived in the last step as the seed set. It generates a candidate set of potential sequences by combining the sequences in the seed set. If the sequences in the candidate set are longer and have minimum support, they form the seed set for the next iteration. This continues until no new sequences can be produced.

For example, given the 1-sequences in **Table 2-4**, a set of 2-sequences and their support in the transformed sequence of **Table 2-5** are shown in **Table 2-6**(a). Similarly, a set of 3-sequences and their support appears in **Table 2-6**(b). Given a predefined minimum support of 25%, only the first four 2-sequences in **Table 2-6**(a) are considered for

Table 2-6(b) because those sequential patterns are the longest and have minimum support.

2-sequence	Support
{[c] [j]}	40%
{[c] [d]}	40%
{[c] [d g]}	40%
{[c] [g]}	40%
{[d] [j]}	20%
{[g] [j]}	20%
{[d g] [j]}	20%

(a)

3-sequence	Support
{[c] [d] [j]}	20%
{[c] [g] [j]}	20%
{[c] [d g] [j]}	20%

(b)

Table 2-6. Data sequences and their supports. (a) 2-sequences (b) 3-sequences and their supports.

A sequence in SPM is an ordered, but not necessarily contiguous, list of elements, or itemsets. In addition, an element in an SPM sequential pattern is marked by the time of occurrence, so that the time interval between the occurrences of two elements may be used for inference. More importantly, the search space for SPM sequential patterns is relatively small compared to that for problems like planning. This is because, like SC, an element in a SPM sequence is a simple object that maps to a relatively small set of items. Furthermore, SPM assumes that the context in which a sequential pattern occurs has no bearing on the presence of the sequential pattern. In other words, the elements in an SPM sequence are static rather than dynamic, and it has no transition from one world state to the next. As will be discussed in the next section, context plays an important role in a more complex domain.

2.2.4 Reinforcement Learning

Reinforcement Learning (RL) learns a mapping from states to actions by trial-and-error (Kaelbling et al. 1996). RL consists of an environment of states, a reinforcement function, and a value function. **Figure 2-4**, for example, depicts an environment of 16 states in a 4 by 4 grid. A special kind of state, called *terminal state*, represents the goal of

a RL problem. In **Figure 2-4**, the states marked by a 0 are terminal states. A *reinforcement function* defines the reinforcement (reward/penalty) for a state transition, and maps state/action pairs to reinforcements. For example, a simple reinforcement function R maps a state/action pair ([1,1], north) to a constant, say –1, where [1,1] represents the state at column 1, and row 1 and north is one of four possible actions: north, east, south and west. A *value function* defines state values (utilities) in terms of reinforcements with respect to terminal (goal) states. For example, a simple value function V may be defined as the number n of state transitions from a given state to a terminal state. Since it takes $n = 3$ state transitions to go from [1,1] to [1,4], the state value for [1,1] is –3, which is calculated as the product of the number of state transitions (3), and the reinforcement value (-1) for each state transition.

An RL system solves a problem through trial and error, guided by a policy π that uses the state values to select an action that maximizes the likelihood of reaching a terminal state. A typical policy selects an action at a given state that receives the most positive reinforcement. In **Figure 2-4**, for example, using such a policy at [1,2], the action should be north leading to [1,3] with a positive reinforcement of one less state transition towards a terminal state [1,4].

4	**0**	**-1**	**-2**	**-3**
3	**-1**	**-2**	**-3**	**-2**
2	**-2**	**-3**	**-2**	**-1**
1	**-3**	**-2**	**-1**	**0**
	1	**2**	**3**	**4**

Figure 2-4. An environment with 16 states. Each square represents a state. The states labeled with a 0 are the goal states.

RL systems vary mainly in terms of their environment, reinforcement, and value function. The environment may be accessible or inaccessible. In an *accessible* environ-

ment, the states can be perfectly observed, whereas in an *inaccessible* environment, only a subset of the states is observable. Reinforcement may be received at any state or at a terminal state. Also, reinforcement may be a precise value or an imprecise value, such as probability. Finally, a value function may be constructed with an empirical learning method, such as a multi-layer perceptron or a memory-based method.

Most RL algorithms are used in a knowledge-poor context. Many RL algorithms have been applied to solve the robot perception problem. As an example, *Utile Suffix Memory* (USM) is a reinforcement learning algorithm that uses short-term memory to overcome *perceptual aliasing*: the mapping between states of the world and sensory inputs is not one-to-one in robot's perception. In other words, when perceptual limitations allow only a portion of sensory inputs, many different world states can produce the same percept. On the other hand, when sensors are directed to different parts of the surroundings, many different percepts can result from the same world state (McCallum 1995).

USM uses memory-based (instance-based) techniques to uncover hidden states. Raw experiences (instances) and statistical tests are used to distinguish among reward variations at each time step, and to determine how much memory is sufficient to uncover hidden states with future discounted (weighted) rewards. Although the USM algorithm can reduce the number of training steps with minimal memory usage, the basic mechanism for such an RL algorithm is the action-percept-reward cycle, where a reward can be explicitly expressed. Such a step-by-step reward may not be clear in incomplete information domains, such as bridge, where rewards for actions (card plays) typically come only at the end of a contest rather than at each step. In such a domain, credit (or penalty) assignments are often domain-specific rather than domain-independent.

2.3 Problem Solving with Sequential Dependency

This section gives a detailed description of SD and presents the rationale behind it. SD extends state-based planning by using a multi-state context for action selection. SD views a problem-solving situation as a set of sequentially related states. The relation between the current state and a number of prior states influences the choice of an action. The goal of SD is to identify the sequence of state-action pairs, called a sequential concept, which serve as the building blocks for complex planning tasks.

2.3.1 Planning as Problem Solving Sequences

Planning is a process that generates a sequence of actions to solve problems or to achieve goals. A planner creates a plan by defining the initial state, one or more goal states, and a set of operators that transforms one state to the other. A plan consists of an ordered set of operators that transforms the initial state into a goal state. Most planning approaches, such as non-linear and hierarchical planners, rely on complete domain knowledge to derive plans (Hendler et al. 1990). In addition, they typically rely on only the current state to select an action.

Tic-tac-toe is an adversary game that can be viewed as a planning problem. Two players, *X* and *O*, take turns placing their respective symbols on a three by three board. The player who first reaches a "three-in-a-row" configuration, or state, wins the contest. **Figure 2-5** shows a winning configuration for *X*, represented as (*XXXOOXOO*), that is, the board in row-major order from left to right and from top to bottom.

X	*X*	*X*
X	*O*	*O*
X	*O*	*O*

Figure 2-5. A tic-tac-toe example that shows a winning situation for player *X*.

As part of the representation, a third symbol, *b*, is used to represent a square on the board when it is not occupied. The initial state of a contest, therefore, is represented as (*bbbbbbbbb*). A *move* by a player *p* that places a symbol in column *c* and row *r* is represented as p_{cr}. Therefore, an action by player *X* that occupies the square at column 1 and row 1, X_{11}, for example, leads to a new state (*Xbbbbbbbb*). A subsequent action by player *O* that occupies the square at column 3 and row 3, O_{33}, leads to another state (*XbbbbbbbO*), and so on.

For illustration, tic-tac-toe can be played using MBR. An action is selected for the current state by retrieving a similar stored example from the database. An example in this case is a state-action pair, where the state is the configuration of the board and action is the placement of a symbol on the board. As an example, given the current state (*bbbbbbbbb*) as input, if (*bbbbbbbbb*, X_{23}) is the most similar stored example retrieved, its action X_{23} will be recommended. The action leads to a new current state (*bbbbbXbbb*), and the process continues until a goal state is reached or all squares are occupied.

While playing tic-tac-toe, one need only look at the current board configuration (current state) to decide the next move (action); the information from prior states is not necessary to the planning situation represented by the current state. Also, tic-tac-toe is a complete information game; no prior state information is required for disambiguation. As a result, the search space is much smaller. In a more complex planning application, however, information from prior states is often needed to disambiguate the current state. In the game of bridge, for example, some information is known only to the opponents, and thus dramatically increases the branching factor and the search space in choosing an action.

2.3.2 Sequential Disambiguation

State-based planning approaches typically look only at the current state to decide on a course of action. This implies that the same state may be reached from more than one path. In particular, if these paths have different purposes, simply looking at the current state will not be adequate to select a correct course of action. SD is intended to disambiguate the situation of the current state by selectively making use of the information in a number of prior states.

Take the game of bridge as an example of a planning problem[1]. To project what a play might entail or what opponents might do, one can look at the current situation and the chronology of the contest in progress. A situation in bridge is viewed as a state, and a play by a player in a state is the action. Play proceeds clockwise. In **Figure 2-6**, for example, the plan is to take a corresponding action in respond to the opponent's. When North (N) leads a 9 in the current state (top), East (E) is obligated to play K or J. Depending on what East plays, South (S) will cover with A or Q.

[1] A list of bridge terms may be found in the appendix for detailed reference.

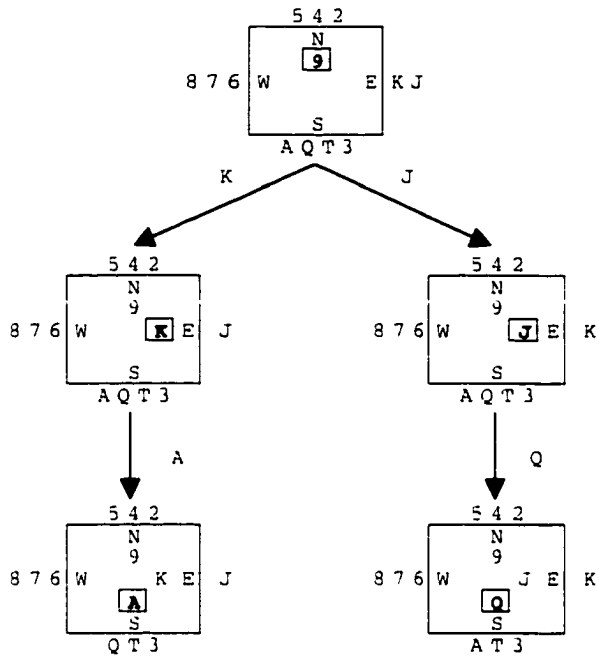


Figure 2-6. A plan for the game of bridge. Four players each have a concealed hand (outside the box) and take turns playing a single card on the table (inside the box). Play begins at the top, where North plays a 9. A state is all the information at each node; an action is an arrow that leads to the next node (state).

The main argument of SD is that identical state resulting from different paths might have been directed by different purposes. In bridge, the opponents' (e.g., West's and East's) cards are not visible from the playing side (e.g., North and South). In **Figure 2-7**, for example, the box at the top of the diagram shows a single suit-playing scenario, where North has four cards (9 5 4 2), and South also has four cards (A Q T 3). West's and East's cards are not visible. Whether the state at the top went through (a) or through (b), it led to the same state (current state) at the bottom of the diagram. If it went through (a), where there was no sign that the opponent is short of cards in the suit, it is feasible for North to lead a low card to finesse[2] Q at South. If, however, it went through (b), where

[2] For a finesse, one player leads a card and if the first opponent plays low, one's partner will also play low, but high enough to cover the first opponent's card, hoping that the second opponent does not cover. If the

East has no card in the suit, then a finesse would not be feasible because all the unplayed, invisible cards, including the king must be in West's hand. That is, without knowing whether the current state was derived from (a) or from (b), it is difficult to make a reasonable decision. Instead, one should look back in time for information to disambiguate the situation and to make a better decision. Chapter 4 will show that situations formulated in terms of SD more often select correct actions when compared to a formulation where only the current situation is taken into consideration.

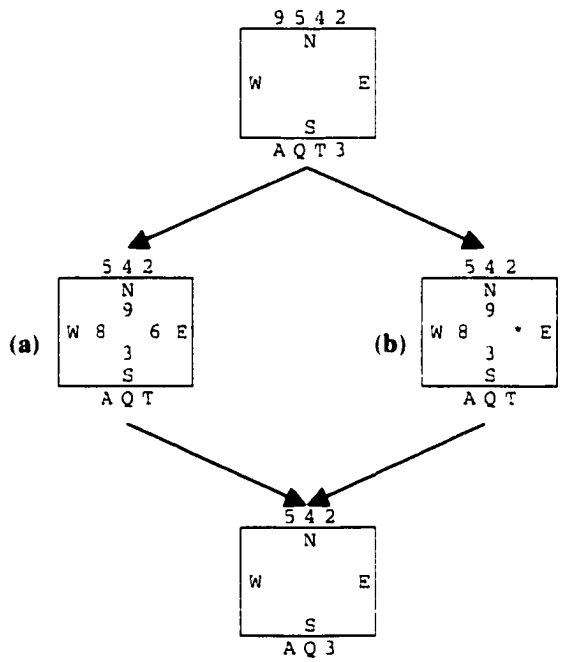


Figure 2-7. The current state (bottom) is ambiguous because it has two different derivations, (a) and (b), from an initial state (top).

2.3.3 Sequentially Related States

SD uses a multi-state context in planning for problem solving. This extended context provides additional information to disambiguate the current state that might have derived from different paths. Starting from the current state, a fixed number of adjacent

---

first opponent plays high, one's partner will be able to cover. As a result, in a favorable distribution, a finesse will produce more tricks than the cards might otherwise.

prior states are considered to select an action for the current state. SD views sequential states as dependent, rather than independent. More formally, in a sequence of n states $(s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n)$, s_j is considered *sequentially dependent* on s_i for all $1 \leq i < j$. Here, s_i is called the *predecessor state* and s_j the *successor state*. Because it has in part formulated the current state and the action available in it, a predecessor state influences its successor state. Similarly, a successor state is influenced by its predecessor state to act.

Consider a problem solving experience represented in **Figure 2-8** (a) as a sequence of states and actions, where the action a_i moves the problem state from s_i to s_{i+1} for $i = 1, 2, \ldots, n\text{-}1$. Each such experience contains many *transition sequences*, contiguous subsequences of states and actions within the experience that ends in a particular state from which an action will be taken, as shown in **Figure 2-8** (b). State-based planning methods assume that a_i, the action to be selected from state s_i, is dependent upon the nature of s_i. Therefore, it seeks the features within s_i that mandate the selection of a_i. SD however, postulates that the reasons for the selection of a_i actually reside within the particular sequence of states $s_{i-k}, s_{i-k+1}, \ldots, s_i$ that precede a_i.

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots s_i \xrightarrow{a_i} s_{i+1} \ldots \xrightarrow{a_{n-1}} s_n \qquad\qquad s_{i-k} \xrightarrow{a_{i-k}} s_{i-k+1} \xrightarrow{a_{i-k+1}} \ldots \xrightarrow{a_{i-1}} s_i$$

(a) **(b)**

Figure 2-8. (a) A problem-solving experience, and (b) a transition sequence of states and actions within that experience.

A *sequential event*, or *s*-event, is used to describe a subset of states in a transition sequence. It has the form $s_{i-k}, s_{i-k+1}, \ldots, s_i \rightarrow a$. An *s*-event includes descriptions of both the current state s_i and some number of consecutive, immediately prior states $s_{i-k}, s_{i-k+1}, \ldots, s_{i-1}$, omitting the intermediate actions of the transition sequence. Such an *s*-event is said to have *length* k+1. Given an input transition sequence like that in **Figure 2-8** (a), the

chronological expansion of any state s in that transition sequence produces an order set of s-events, each ending immediately before s. For example, in the transition sequence

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} s_4$$

chronological expansion at s_4 would produce three s-events:

$$s_1 s_2 s_3 \rightarrow a_3 \Rightarrow \begin{cases} s_3 \rightarrow a_3 \\ s_2 s_3 \rightarrow a_3 \\ s_1 s_2 s_3 \rightarrow a_3 \end{cases} \qquad [3]$$

We call each of the s-events in the chronological expansion of a transition sequence a *partial s-event*. Notice that the length of an s-event is equal to the number of partial s-events it generates. This number, called a *window*, is less than or equal to the length of the actual transition sequence. The number of partial s-events increases linearly with the size of the window, that is, a set of m s-events of length n generates $m \times n$ partial s-events. When such a representation is used in an $O(n^2)$ algorithm, as described in Chapter 3, there will be a quadratic, that is, $O((mn)^2)$, window into the search space. The window is a device needed to control search and to avoid a combinatoric explosion.

In summary, a transition sequence describes a problem-solving experience as a sequence of events. A sequential event (s-event) is a subsequence of a transition sequence of a fixed length, called the window. In addition, an s-event may be defined for each event in the transition sequence such that the s-event chronologically ends at the event. Finally, each s-event generates a fixed set of partial s-events of various lengths.

2.3.4 Chronological Instantiation of Sequential States

Chronological expansion augments context by including adjacent states, past and/or future, with respect to the current situation. Chronological expansion produces a

set of partial *s*-events of various lengths. There are three types of chronological expansion: *backward expansion* (BE), *forward expansion* (FE), and *combined expansion* (CE).

BE concerns expansions of prior states with respect to the current state. BE starts at the current state s_0 and consecutively moves backward one state at a time up to a specified number of m previous states, s_{-m}, ..., s_{-2}, s_{-1}. These are called *historical states*. The *s*-event generated by BE is called a *backward s-event*. FE concerns expansions of future states with respect to the current state. FE starts at the current state s_0 and moves forward consecutively with a specified number of n future states, that is, s_1, s_2, ... s_n. They are called *hypothetical states*. The *s*-event generated by FE is called *forward s-event*. CE combines BE and FE. The result has three parts: backward states, the current state and forward states, for example, s_{-2}, s_{-1}, s_0, s_1, s_2. In this case, the current state is combined with historical and hypothetical states. The *s*-event generated by CE is called *combined s-event*.

An important issue here is the number of states to include in any of the expansions. As discussed earlier, a fixed number of *s*-events in a transition sequence is used to control the growth of the search space in finding a solution. Intuitively, when the number is too small, there might not be sufficient information to disambiguate the current state. When the number becomes much larger, on the other hand, it will be more difficult to find a close match. Therefore, the goal is to find just the right expansion that has the greatest possibility of finding a correct solution. As an MBR example, the problem can be viewed as finding the most similar stored instance (i.e., the partial *s*-event) among various candidates (i.e., partial *s*-events of various lengths). For example, given an *s*-event with the current state c_0 and BE with a window $w = 3$, the resulting partial *s*-events are: c_0, $c_{-1}c_0$, and $c_{-2}c_{-1}c_0$. Each of these partial *s*-events is used to retrieve the most similar stored instance of their respective lengths, called *candidates*, perhaps *a*, *st*, and *xyz*, as in

Table 2-7. Here, the candidate *st* is preferred because it is the most similar among the candidate partial *s*-events. These calculations are based on a set of sequential similarity metrics detailed in Chapter 3.

Partial *s*-event	Candidate	Similarity
c_1c_0	*st*	.79
c_0	*a*	.44
$c_2c_1c_0$	*xyz*	.22

Table 2-7. Hypothetical partial *s*-events and their best matches in terms of similarity measurement.

Another important issue is the quality of partial *s*-events generated from chronological expansion. Since BE is based on historical states, which have already occurred, the information is readily available and perfectly accurate. For this reason, such information may serve to disambiguate the current situation, particularly when the same situation is derived from two paths with different purposes.

FE, in contrast, is based on hypothetical states. FE may avoid conceivable pitfalls in the problem solving process since it tries to direct the attention of a problem solver to more fruitful paths. In state-based planning, the whole idea of coming up with a plan before plan execution is to avoid pitfalls and to ensure success. In an incomplete information domain, however, the use of FE will degrade the performance due to compounded errors (Goodman 1993).

In bridge, for example, in a sequence of two consecutive plays, the likelihood of guessing the opponents' card distribution of three outstanding cards with no prior knowledge is nearly the same as a random guess. To begin with, the three outstanding cards may be distributed among the two opponents in four ways: 2-1, 1-2, 3-0, and 0-3. The probabilities for these distributions may be calculated by the hypergeometric probability distribution model (Hogg and Tanis 1997).

Let $N = N_1 + N_2$, where N is the total number of cards held by the opponents, N_1 by the first opponent and N_2 by the second. Since each player is dealt 13 cards, $N = 13 + 13 = 26$. In other words, $p = .5$ for the two opponents among the $N = 26$ cards, so that $N_1 = N_2 = 13$. Let $n = 3$ be the outstanding cards distributed between the two opponents in the suit of interest. In the case of a 2-1 distribution, the first opponent has $x = 2$ cards, and the second opponent has $n - x = 1$ card. For the first opponent, the number of ways c_1 to select $x = 2$ cards out of $N_1 = 13$ cards is calculated as

$$c_1 = \binom{N_1}{x} = \binom{13}{2} = 78 \qquad [4]$$

Similarly, for the second opponent, the number of ways c_2 to select $n - x = 1$ card out of $N_2 = 13$ cards is calculated as

$$c_2 = \binom{N_2}{n-x} = \binom{13}{3-2} = 13 \qquad [5]$$

The product

$$c_1 \times c_2 = 78 \times 13 = 1014 \qquad [6]$$

represents the total number of possible distributions for the 2-1 split. Finally, the total number of ways $n = 3$ cards may be distributed among $N = 26$ cards without replacement is

$$\binom{N}{n} = \binom{26}{3} = 2600 \qquad [7]$$

Therefore, the probability that the first opponent held exactly $x = 2$ cards P(x=2) is calculated as

$$\frac{\binom{N_1}{x}\binom{N_2}{n-x}}{\binom{N}{n}} = \frac{\binom{13}{2}\binom{13}{3-2}}{\binom{26}{3}} = .39 \qquad [8]$$

Table 2-8 summaries the probabilities for all possible distributions for three and two outstanding cards among the two opponents. If one has to make two consecutive plays, given these probabilities, the best case from a bridge player's perspective is a probability of .41 when each opponent holds at least one card, that is the product of the probability of 2-1 and 1-2 split (.78), and the probability of 1-1 split (.52). The worst case is a probability of .11 when one of the opponents does not have any card in the suit, that is the product of the probability of 3-0 and 0-3 splits (.22), and the probability of 2-0 and 0-2 splits (.48). Both scenarios are worse than a coin toss (.5). As a result, chronological expansion with FE should degrade performance.

# of outstanding card	Possible distribution	Probability
3	2-1	.39
	1-2	.39
	3-0	.11
	0-3	.11
2	1-1	.52
	2-0	.24
	0-2	.24

Table 2-8. Probability for 3 and 2 outstanding cards in a four-player card game. The possible distribution column lists all possible combinations, and the probability column lists corresponding probabilities.

In summary, historical states are more reliable because they are based on known historical events, and hypothetical states are less reliable than historical states because they are based on unknown hypothetical events and have the potential for compounded errors. Finally, CE combines BE and FE and can recognize complete sequential concepts if the current state happens to be in the middle of a sequential concept. Although CE inherits the benefit of BE, it also inherits the potential drawback of FE. It is not clear how to balance and to quantify the combined result from BE and FE. For these reasons, in this research, the experiment is done in BE only.

2.4 *Applications of Sequential Dependency*

SD is designed to solve problems in synthesis domains. A synthesis domain involves multiple steps in a problem solving sequence. At each step, an action is selected that transitions the current state to a new state. A goal is achieved or a problem is solved by a sequence of state-action pairs. SD can be used for such a domain because it uses information about adjacent states in the decision process to achieve a goal.

Specifically, SD can disambiguate a situation derived from multiple paths. This phenomenon is common in imperfect information domains. In the game of bridge, for example, the information relevant to decision making (e.g., who holds what cards and when will they be played) is often unavailable. This dramatically increases the complexity of the problem. SD may be used to reduce the complexity by providing relevant context in the problem solving sequence.

2.4.1 Planning

Planning is a synthesis task that can be solved by an ordered set of actions, $a_0, a_1, a_2, ..., a_m$. These actions will transition an initial state, S_0, to a desired goal state G, according to the behavior of a problem solver f. That is,

$$f_0(S_0, a_0) \Rightarrow S_1, f_1(S_1, a_1) \Rightarrow S_2, ..., f_m(S_m, a_m) \Rightarrow G \qquad [9]$$

where the S_i's are intermediate states, $0 < i < m$. Each state-action pair (S_i, a_i) is a sub-task and is solved by an instantiation of the problem solver, f_i, which takes a state-action pair (S_i, a_i) as input and outputs the action a_i that leads to a new state S_{i+1}, that is $f_i(S_i, a_i) \Rightarrow S_{i+1}$. In general, the problem solver f has two sub-tasks: action selection f_{select} and action application f_{apply}, that is, $f_{apply}(S_i, f_{select}(S_i)) \Rightarrow S_{i+1}$, where $f_{select}(S_i) \rightarrow a_i$.

Problem solving with SD is outlined in **Table 2-9**. SD uses *s*-event E as an extended context in place of a state description. Given an n-state transition sequence $T = s_1, s_2, \ldots, s_n$, an *s*-event E_i at state S_i, written

$$s_{i-k}, s_{i-k+1}, \ldots, s_i \qquad [10]$$

has a window size $w = k+1$, $k < n$. The chronological expansion at state s_n produces the set

$$E_i^* = \{(s_{n-k}, s_{n-k+1}, \ldots, s_n), (s_{n-k+1}, \ldots, s_n), \ldots, (s_n)\} \qquad [11]$$

of $k+1$ partial *s*-events. Each partial *s*-event is solved separately and its solution is placed in a set

$$C = \{(s_{n-k}, s_{n-k+1}, \ldots, s_n)' \rightarrow a_{n-k}, (s_{n-k+1}, \ldots, s_n)' \rightarrow a_{n-k+1}, \ldots, (s_n)' \rightarrow a_n\} \qquad [12]$$

of solution candidates, in which each candidate corresponds to a partial *s*-event in E_i^*. For example, the candidate $(s_{n-k}, s_{n-k+1}, \ldots, s_n)' \rightarrow a_{n-k}$, in C corresponds to the solution for the partial *s*-event $(s_{n-k}, s_{n-k+1}, \ldots, s_n)$ in E_i^*. Each candidate has an action it proposes for the current state s_i. The most plausible action a_i is selected with f_{select} from the candidate solutions in C by comparing them with the sequential features between partial *s*-events in E_i^*. The selected solution is applied to the current state to yield the next state: $f_{apply}(S_i, a_i) \Rightarrow E_{i+1}$.

```
For each state i
    E_i ← extract_event(T, w, i)
    E_i* ← chronological_expansion(E_i)
    C ← candidate_creation(E_i*)
    a_i ← f_select(C)
    E_i+1 ← f_apply(a_i)
```

Table 2-9. A procedure for problem solving using sequential dependency.

Examples of synthesis task domains include planning, design, game playing, monitoring, and diagnosis. In general, synthesis tasks are less tractable than classification tasks and remain an active area for researchers. The next section (2.4.2) describes the application of SD to game playing as an example of a synthesis task in more detail.

2.4.2 Game Playing

Game playing can be viewed as a synthesis task domain. It involves a sequence of plays $p_0, p_1, p_2, \ldots p_m$ in an effort to win. Each play p_i takes the game-playing program g from one state J_i to the next J_{i+1}, written $g : J_i \xrightarrow{p_i} J_{i+1}$. In a two-player game, there are two opposing game-playing programs, g_i and g_i', one for each side. An *interaction* consists of two state transitions involving a play from each of the two opposing game playing programs g and g'. That is,

$$g : J_i \xrightarrow{p_i} J_i' \text{ and } g' : J_i' \xrightarrow{p_i'} J_{i+1} \qquad [13]$$

where $g(J_i)$ yields p_i, and $J_i \xrightarrow{p_i} J_i'$. Similarly, $g'(J_i')$ yields p_i', and $J_i' \xrightarrow{p_i'} J_{i+1}$. For example, **Figure 2-9** shows an interaction in the game of tic-tac-toe. At state J_i, the game-playing program g for the player "O" selects the play $p_i = O_{11}$ (i.e., placing the symbol "O" in the cell at row 1 column 1), which leads to the state J_i'. At state J_i', the game playing program g' for the player "X" selects the play $p_i' = X_{13}$ (i.e., placing the symbol "X" in the cell at row 1 column 3), which leads to the state J_{i+1}.

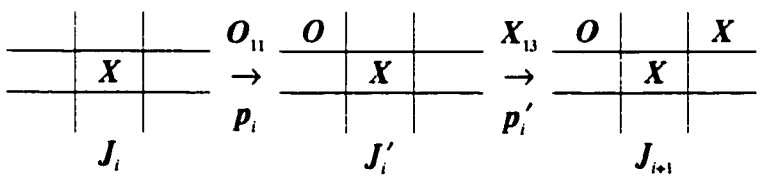


Figure 2-9. An interaction in the game of tic-tac-toe. The interaction starts at state J_i and leads to state J_{i+1}. The play $p_i = O_{11}$ (i.e., placing the symbol "O" in the cell at row 1 column 1) transitions the state from J_i to J_i'. The play $p_i' = X_{13}$ (i.e., placing the symbol "X" in the cell at row 1 column 3) transitions the state from J_i' to J_{i+1}.

For convenience, a more concise representation is used from this point on which omits the intermediate state J_i' and associated action p_i', and shows only the state at which one of the two players is to take an action, that is, $g : J_i \rightarrow J_{i+1}$. **Figure 2-10** shows an example of a concise illustration in the game of tic-tac-toe.

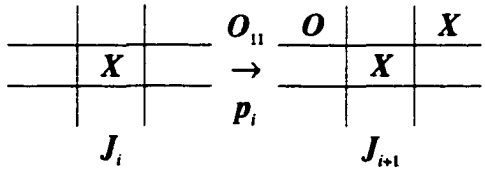


Figure 2-10. A more concise representation for an interaction in the game of tic-tac-toe that displays the states at which one of the two players is to take an action. This is done by omitting the intermediate state J_i' and associated action p_i'.

In a four-player game, such as bridge, there are two opponents against two players on "our side." In each round (called a *trick*), the states and plays are interleaved between "ours" and those of the opponents'. In other words, a single round may be viewed as two consecutive interactions, that is,

$$g : J_i \rightarrow J_{i+1} \text{ and } g : J_{i+1} \rightarrow J_{i+2} \qquad [14]$$

Where $g : J_i \rightarrow J_{i+1}$ represents the state transition for the first player on "our side," and $g : J_{i+1} \rightarrow J_{i+2}$ represents the state transition for the second player also on "our side." Whether it is two-player or four-player, each interaction is considered a problem-solving state. For example, **Figure 2-11** shows an example for the play of a trick in the game of bridge. In the first interaction (1), North holds K, Q and 8, South holds A and T, East leads a 3, and South plays an A. In the second interaction (2), North still holds K, Q and 8, South holds only T, West plays a 4, and North plays an 8.

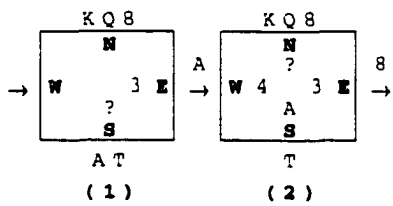


Figure 2-11. An example two interactions in bridge from state (1) to state (2).

With SD, an extended context is used to disambiguate the current situation. As outlined in **Table 2-9**, the current state is extended to an s-event E of a finite window size w. E is then chronologically expanded to produce a set E^* of partial s-events of various lengths, each of which is associated with a solution. Given the candidate set C, a sequen-

tial selection method is used to select the most plausible solution for the current state. As a bridge example, let the transition sequence in **Figure 2-12** represents a contest in progress.

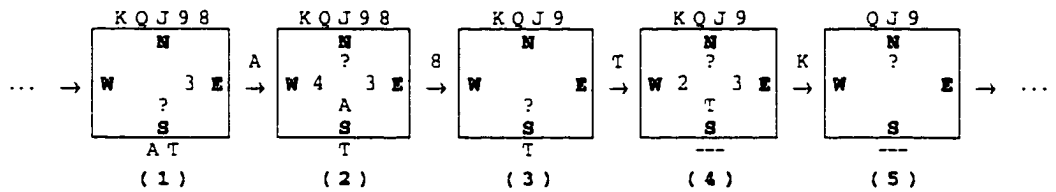


Figure 2-12. A transition sequence for the game of bridge. At state (1), North holds K Q J 9 8, South holds A T, East plays a 3, and South is to play the next card (?). At state (2), South has played an A, West has played a 4, and North is to play the next card (?), and so on.

Based on the procedure in **Table 2-9**, the game of bridge maybe described in the following five steps:

1. *s*-event Creation: As shown in **Figure 2-13**, an *s*-event E = [(3), (4), (5)] with a window size of three can be extracted from this transition sequence at state (5).

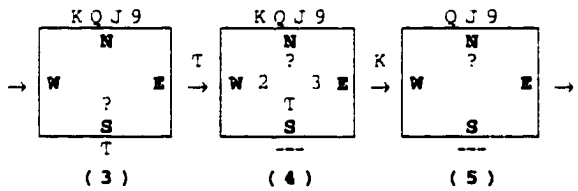


Figure 2-13. An *s*-event ends at state (5) with a widow size of three. The *s*-event is a subsequence of the transition sequence in **Figure 2-12**.

2. Chronological Expansion: As shown in **Figure 2-14**, the chronological expansion E^* = {[(5)], [(4), (5)], [(3), (4), (5)]} consists of three partial *s*-events of lengths ranging from one to three, respectively.

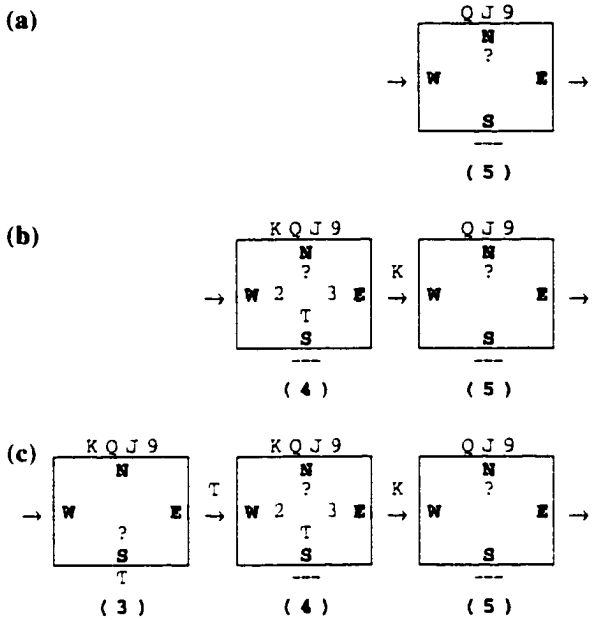


Figure 2-14. A set of three partial *s*-events of lengths (a), (b), and (c). Each partial *s*-event is the result of a backward expansion (*BE*) from state (5). The number of events in each partial *s*-event is the length, or window size, for the partial *s*-event.

3. Candidate Creation: As shown in **Figure 2-15**, a solution for each partial *s*-event is generated forming the candidate set $C = \{[(5)]' \rightarrow a_1, [(4), (5)]' \rightarrow a_2, [(3), (4), (5)]' \rightarrow a_3\}$:

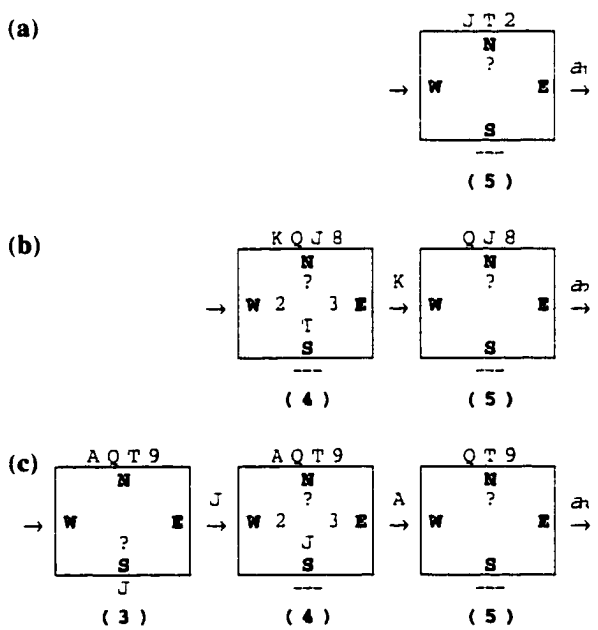


Figure 2-15. Candidates a_1, a_2, a_3 for the partial *s*-events shown in **Figure 2-14**.

4. Candidate Selection: The partial *s*-events in **Figure 2-14** are compared with the same-length candidates in **Figure 2-15** to select the most relevant candidate in terms of their sequential characteristics. A simple difference metric may be used here to select the most relevant candidate. The metric calculates the average number of different cards held and played by different players per event. The partial *s*-event in **Figure 2-14**(a) differs from the same-length candidate in **Figure 2-15**(a) by two cards (Q J 9 vs. J T 2). The partial *s*-event **Figure 2-14**(b) differs from the candidate **Figure 2-15**(b) by one card (9 vs. 8), and the partial *s*-event **Figure 2-14** (c) differs from candidate **Figure 2-15**(c) by two cards (K J T vs. A J T). In this case, the candidate (b) for the partial *s*-event (b) is considered the most relevant because it has the minimal difference based on the simple difference metric. Thus, candidate (b) will be selected.

5. Solution Application: Finally, the action a_2 is applied to solve the problem at state (5).

2.5 Summary

SD is a new approach to knowledge discovery by means of sequential concept discovery. It can be used for predicting an appropriate action, for example, a play in bridge. Unlike time series approaches, SD does not rely on probability distributions, because as the number of states grows exponentially, the available probability distribution will be too sparse. Also, unlike sequence categorization and sequential pattern mining, SD is not limited to a finite set of symbols (states). SD has the potential to improve state-based planning in an extended context, which is used to disambiguate the current situation derived from two distinct paths.

In SD, a decision is made based on an extended description of a domain that includes the current state and a number of prior states. As a result, the context for the current situation is expanded and can be used for disambiguation. An important aspect of

this approach is the ability to extend the appropriate number of prior states. Finally, SD is designed to solve complex synthesis tasks that consist of a sequence of steps to achieve a goal, particularly, those tasks that have imperfect information, such as bridge.

CHAPTER THREE :

LEARNING SEQUENTIAL CONCEPTS

The last chapter introduced the key idea, sequential dependency, for learning sequential concepts. When states are sequentially dependent, a multi-state context may be used to disambiguate those states with different derivations. This chapter integrates this idea with an empirical learning paradigm to learn synthesis knowledge. The first part of this chapter provides a justification for the use of an incremental learning approach. The second part describes the framework for learning sequential dependency. The focus here is to find the most relevant solution to a synthesis problem. Two general approaches are discussed: majority vote and sequential similarity metrics. The third part of this chapter elaborates the sequential similarity metrics, which include distance, convergence, consistency and recency. Finally, an analysis is provided to explain why the metrics work. This is followed by a discussion of the complexity of the learning algorithms.

Although conceptually, sequential dependency (SD) can be used to improve clarity when solving a synthesis task by using a multi-state context, operationally, it needs specific knowledge to carry out the task in the domain of interest. Learning attempts to acquire such knowledge. Unlike learning for classification tasks, learning sequential concepts for synthesis tasks relies on information from a transition sequence that contains the description of one or more states. The empirical learning approach for classification tasks can be extended to learning for synthesis tasks if a synthesis task is viewed as a sequence of classification tasks.

To extend an empirical learning approach for classification tasks to synthesis tasks with SD, a training example is instantiated with a multi-state context of various sizes. Each instantiation is processed separately to produce a set of candidates. The most

relevant candidate is then selected, based on the sequential nature of the examples, using a set of similarity metrics designed for such sequential examples.

Learning sequential concepts involves the use of domain knowledge similar to the four knowledge containers in a typical CBR system (Lenz et al. 1998). The knowledge containers are (1) the vocabulary (attributes), (2) the similarity measure, (3) the case base, and (4) the solution transformation. While (1), (2) and (4) contain compiled knowledge, (3) contains knowledge that is interpreted during runtime. The focus of learning sequential concepts is primarily on using (2) and (3). In particular, the similarity measure takes advantage of the sequential nature of the input examples, and the acquired cases (examples) may be interpreted by the specialized similarity metrics when a new case is presented. Sequential concept learning de-emphasizes the vocabulary, since it follows the representation of a knowledge-poor, empirical learning algorithm. Finally, to increase generality, the use of solution transformation is kept at a minimum.

As a result of such an extension, there are two important observations. The new approach refines the decision boundary with the sequential similarity metrics, and the new approach adjusts the importance of the attributes used to represent the examples. Both behaviors have contributed to the improvement of performance in problem solving detailed in Chapter 4. Although the extension requires additional time and space, the increase is insignificant compared to the complexity of the synthesis problem.

3.1 Learning

Learning is generally believed to be the ability of a human or an animal to improve its behavior from sources such as its own experience (Lefrancois 1988). *Machine learning* (ML) embodies the ability to learn in computer programs, so that the same problem can be solved more effectively and/or more efficiently (Simon 1983). In other

words, ML changes a problem solver, so that the problem solver can solve better and/or faster the next time it is faced with the same kind of problem.

ML can increase the *efficiency* of a problem solver by modifying the existing knowledge, and ML can increase the *effectiveness* of a problem solver by acquiring new knowledge from external sources, such as examples (Shavlik 1990b). The first ML approach acquires new skills to speed up problem solving. This may be accomplished by reducing search space with macro operators, sequences of primitive ones (Fikes et al. 1972; Shavlik 1990a). Efficiency may also be accomplished by using control knowledge, such as operator selection rules and evaluation functions (Mitchell 1983; Samuel 1959). Effectiveness requires new knowledge, primarily from examples. The new knowledge, in the form of rules or decision trees, is created by comparing and contrasting these examples. If the examples are labeled with a specific class, the process is called *supervised* learning (Aha 1992; Mitchell 1982; Quinlan 1986). If the examples are not labeled, it is called *unsupervised* learning (Cheeseman 1988; Fisher 1987). Supervised learning may be used to predict the class of an unseen example, while unsupervised learning may be used to create new classes by grouping similar examples together.

Two important benefits of machine learning are to gain new knowledge and to transfer the ability to learn to a new domain. The former includes improvement of accuracy and efficiency of the domain knowledge. The latter includes reduction of knowledge acquisition bottlenecks and the cost for maintaining knowledge based systems. In other words, when existing knowledge is available, it can be used to gain more accurate and efficient knowledge. When existing knowledge is scarce, learning methods from a similar domain may be adapted to bootstrap the basic knowledge and enable further learning.

Choosing a particular learning paradigm depends on the nature of the problem-solving tasks and the source of knowledge utilized. This research focuses on synthesizing

problem-solving steps to form a plan, such as playing out a bridge hand. For domain independence, examples were the main source of knowledge. Specifically, supervised learning is used and, therefore, each example is labeled with a response to the problem at hand.

3.1.1 Learning from Examples

Learning from examples, or *empirical learning*, is an active area of research because data from many complex domains is increasingly available. In this paradigm, a set of labeled examples is given as input to the learning algorithm. The goal of the learning algorithm is to produce a set of hypotheses, or concept descriptions, to explain the input examples (Langley 1995).

A learning system has two components: a learning component and a performance component. The learning component generates concept descriptions from examples, based on the heuristics of the learning algorithm. Once the concept descriptions have been formed, an interpreter can be used to predict unseen examples. The interpreter maps the new example to one of the concept descriptions.

As input to the learning component, an example is commonly represented by a set of attributes. A synonymous term for attribute is feature. The two will be used interchangeably throughout this dissertation. The *independent* attributes represent the characteristics of the example; the *dependent* attribute represents the class of the example. The common types of attributes are binary, discrete, and numeric. A binary attribute has two possible values. A discrete attribute takes its value from a set of more than two values. Finally, a numeric attribute has an integer or a real number as its value. A k-dimensional example space is one described by k features, and is represented by the set of k-tuples of independent variables.

As output of the learning component, the *concept description* summarizes the examples in a concept description language, a compact and concise representation of the input examples. Two common concept description languages are decision trees and instance-based representations. A decision tree is a compact representation, which uses selected attributes as nodes and selected attribute values as arcs, with the most predictive attribute at the root of the tree and the least important ones at the leaves. An instance-based method often represents its output as prototypical examples with the same representation as the input examples. However, some instance-based methods use abstractions, such as averages of raw instances, in their representation (Bradshaw 1987; Salzberg 1991).

Producing a concept description from labeled examples can be viewed as searching through concept description space (Mitchell 1982). Each concept description or group of concept descriptions is tested for its validity, given the labeled input examples. A set of heuristics (biases) is used by the learning algorithm to favor some concept descriptions over others. For decision tree representation, the heuristics are often based on information theory and statistics (Breiman 1984; Quinlan 1993). For instance-based representation, the heuristics are based on the similarity metrics that measure the difference between two examples (Aha 1992; Cover and Hart 1967).

3.1.2 Comparison of Incremental and Non-incremental Learning

Learning methods can be incremental or non-incremental (*batch learning*). Incremental learning starts with knowledge, and gradually adds to or modifies the existing concept description with subsequent input examples. Instance-based learning, for example, originated from the incremental learning paradigm. Non-incremental learning assumes the availability of a complete set of examples and generates a set of concept de-

scriptions at one shot. Decision tree learning, for example, originated from non-incremental learning. Non-incremental learning methods can take advantage of statistics about all the examples as a whole and use them to form concept descriptions. Incremental learning methods, on the other hand, are efficient when a new example becomes available, because they do not re-evaluate the complete set of examples.

Although, in most cases, both incremental and non-incremental learning methods can be used interchangeably, incremental learning is more suitable for incremental problems, such as planning. This is because a plan is formed one step at a time to achieve some goals. Such an incremental problem-solving task requires frequent updates to its domain knowledge. Since a non-incremental learning method learns from a complete set of examples, it is inefficient for an incremental problem-solving task where new input arrives incrementally. Thus, a more efficient approach is to interleave problem solving and execution with an incremental learning method. Furthermore, in an incomplete information domain, such as bridge, the essential information, card distribution among the players, is absent. Without a complete description in the bridge world at each move, it is more reliable to interleave problem solving and execution.

In summary, incremental learning methods are more appropriate than non-incremental methods for incremental problems due to the need for frequent updates. Since this thesis is about learning for incremental problem-solving tasks, an incremental learning approach will be used. In particular, instance-based learning is combined with the idea of sequential dependency described in Chapter 2 to address the planning problem.

3.1.3 Instance-Based Learning

As discussed above, *instance-based learning* (IBL) originated from the incremental learning method. It typically forms concept descriptions by storing prototypical examples. IBL is often used for classification. That is, each example is annotated with the class in which it belongs. The set of concept descriptions learned is simply the set of prototypical examples. The classification task for a new example is typically accomplished by assigning the class label selected from among the most similar prototypical examples. For consistency with the literature, from this point on, an example is called an *instance* in the context of IBL. The term *example* may be used interchangeably with instance.

Given a set of training instances T as input, IBL constructs concept descriptions (CD) in terms of a set of prototypical instances as output used to predict the outcome (class) of a similar unseen instance (Aha 1992; Salzberg 1991). Initially, CD is empty. Incrementally, as the training instances in T are processed, selected training instances are stored in CD as prototypical instances according to similarity calculation and class comparison. **Table 3-1**, taken from (Aha 1992), illustrates an algorithm that builds up prototypical instances in CD from the training instances. The top-level steps are:

1. Given a training instance x, calculate the similarity between x and each prototypical instance y in CD, and store the results in S.

2. Select the most similar candidate and assign its class to $class_{max}$ from S.

3. If the class of x is not the same as $class_{max}$, then store x in CD as a new prototypical instance.

```
procedure IBLearn(T, CD ← ∅)
    for x ∈ T
        for y ∈ CD
            S ← similarity(x, y)          ; Calculate the set S of similarities
                                          ; between x∈T and y∈CD
        class_max ← max(S)                ; Find the class class_max with
                                          ; maximum similarity
        if class_t ≠ class_max then       ; If class_max, is not the same as the
                                          ; class, class_t, of the input instance x
            CD ← CD ∪ {x}                 ; Add x to CD
```

Table 3-1. A Top-level instance-based training algorithm.

3.2 Learning Sequential Dependency

Unlike an analysis problem, such as classification, a synthesis problem, such as planning, seeks to form a sequence of actions for achieving some goals. In planning, however, each state may be viewed as a classification problem, where an action for a state is analogous to the class of the classification problem. While classification problems are ordinarily independent of one another, the set of classification problems arising from a synthesis problem is sequentially related. Such a relationship is established by an action that transforms one problem to another.

Unlike IBL, learning for a synthesis problem requires that an instance be represented in a sequential representation, such as the transition sequence as described in Chapter 2. Such a transition sequence consists of states where an action is performed. The input to learning is a sequence of k states (s-instance), where k is the *window* into the transition sequence. The output from learning is a set of concept descriptions that enables the selection of an action given a new situation. In other words, using the idea of sequential dependency (SD), the input to the learning element is a set of s-instances derived

from the transition sequences, and the output is a set of sequential concept descriptions with the ability to select an appropriate action given an *s*-instance.

3.2.1 Sequential Instance-Based Learning

Sequential Instance-Based Learning (SIBL) combines IBL and SD to gain knowledge about sequential problems from examples. The *SIBLearn* procedure in **Table 3-2** describes how sequential knowledge is acquired from the sequential examples. As in **Table 3-1**, the set of concept descriptions (*CD*) is initially empty and the set T of training instances is used to update *CD*. The training instances for SIBL, however, differ from those in **Table 3-1**. In SIBL, a training instance x is a window of length w, a *sequential instance* (*s*-instance) of w states in the transition sequence. That is, T is a set of *s*-instances x^w, where each training instance has a context of w states s_i, s_{i+1}, ..., s_j, where $w = j - i + 1$.

Each *s*-instance x^w in T is expanded into the set X of partial *s*-instances x^i, each of which consists of a context of i states, $1 \le i \le w$. The set Y records the *similarity* between x^i and all same-length partial *s*-instance y^i in *CD*. Then, SIBL forms the set C of candidates $c^i = max_i(Y)$, which has the maximum similarity in Y, $1 \le i \le w$.

Finally, $class_{max}$ may be derived in one of two ways: the *FindMajority* function returns the majority class among the candidates in C; and, the *FindSimilarity* function returns the class of the most *sequentially* similar candidate in C using a set of sequential similarity metrics described in Section 3.3. As in **Table 3-1**, if the class of x is not the same as $class_{max}$, then x is stored in *CD* as a new prototypical instance.

```
Procedure SIBLearn(T, CD ← ∅)
    for x^w ∈ T
        C ← ∅,                          ; Initialize C, the candidate set
        X ← expand(x^w)                 ; Expand x^w and form the set X of partial s-
                                        ; instances {x^i | 1 ≤ i ≤ w}
        for x^i ∈ X
            for y^i ∈ CD, | y^i | = | x^i |   ; Compute similarity between same-length
                Y ← similarity(x^i, y^i)     ; partial s-instances x^i ∈ X and y^i ∈ CD
            C ← C ∪ max_i(Y)            ; Form the set C of candidates c^i
                                        ; that is the maximum in CD, 1 ≤ i ≤ w
        class_max ← FindMajority(C), or FindSimilarity(X, C)
                                        ; Use either FindMajority or
                                        ; FindSimilarity to select class_max,
                                        ; the class of the best candidate in C
        if class_t ≠ class_max then     ; If selected class, class_max, is not
                                        ; the same as input class, class_t
            CD ← CD ∪ {x^w}             ; Add x^w to CD
```

Table 3-2. Top-level SIBL algorithm.

Notice that partial s-instances x^i are expanded from the s-instance x^w. The *expand* function in **Table 3-3** takes an s-instance x^w and returns a set X of partial s-instances x^i, $1 \le i \le w$. In each j^{th} iteration, a new state s_j is appended to x^{j-1} to form x^j, where s_j is the j^{th} state in x^w. Then, x^j is added to the set X.

```
function expand(x^w) returns X
    x^1 ← {s_w}, X ← {x^1}              ; Initialize x^1, partial s-instance of length 1
    for j from 1 to w-1
        x^(j+1) ← append(x^j, s_(w-j))  ; Append s_(w-j) to x^j to obtain x^(j+1)
        X ← X ∪ { x^(j+1) }             ; Add x^(j+1) to X, 1≤ j ≤ w-1
```

Table 3-3. The *expand* function instantiates the set of partial s-instances X.

3.2.2 Majority Vote

Majority vote has been one of the primary candidate selection methods for the IBL family of algorithms (Aha 1991; Cover and Hart 1967; Salzberg 1991). In the context of SIBL, the intuition behind it is that, if the majority of the partial *s*-instances in the candidate set point to the same class, the class is likely to be correct.

The *FindMajority* function (**Table 3-4**) implements majority vote for sequential instances. As input, the candidate set C contains stored partial *s*-instances of varying lengths $\{c^i \mid 1 \le i \le w\}$, where each partial *s*-instance $c^i \in C$ has a class c_i associated with it. As output, the class with highest number of votes ($class_{max}$) among the candidates is returned. Specifically, for each candidate c^i in the candidate set C, the class vote is tallied in a *vote* array. The *majority* function selects the highest vote and assigns it to $class_{max}$. For example, let $c^i \Rightarrow c_i$ denote a partial *s*-instance c^i that has c_i as its class. Under majority vote, the set of candidates $C = \{c^1 \Rightarrow c_1, c^2 \Rightarrow c_2, c^3 \Rightarrow c_1\}$ would return c_1 as $class_{max}$, because c_1 appears two out of three times.

```
function FindMajority (C) returns class_max
    for each c^i ∈ C
        c_i ← class(c^i)
        vote[c_i] ← vote[c_i] + 1        ; Tally class in the vote array
    class_max ← majority(vote)           ; Find maximum vote
```

Table 3-4. The function that finds majority for set of sequential instances.

In *FindMajority*, the focus is on the quantity of a particular class rather than on the characteristics of a particular partial *s*-instance in the candidate set. For example, an important characteristic is the length of a partial *s*-instance, which, if identified correctly, will provide information about the size of the context in a given state. This involves a comparison of specific qualities among the partial *s*-instances. This approach is described in the next section.

3.2.3 Sequential Similarity

One of the challenges in reasoning with sequences is to choose the correct length (number of states or size of a context) for decision making. Majority vote only identifies a class, rather than the correct length of the partial *s*-instances that recommends the class. Finding the correct length requires comparing the similarities of a set of partial *s*-instance pairs of various lengths.

The *FindSimilarity* function (**Table 3-5**) implements a qualitative approach that recommends a class based on the correct context size or the number of states. Given the set X of partial *s*-instances x^i expanded from the input *s*-instance x^w, and the set C of the most similar candidates c^i of varying lengths, the sequential similarities S between x^i and c^i are calculated. The class of the most similar candidate ($class_{max}$) based on sequential similarity is returned.

The key to the qualitative approach is the application to sequential instances of an *SSM* function, which employs sequential similarity metrics to calculate sequential similarity (see **Table 3-3**). Since each sequential similarity metric exploits different characteristics of a sequential example during similarity computation, the *SSM* function acts as a template that is instantiated differently for different sequential similarity metrics. The sequential similarity metrics, described in the next section, include distance, convergence, consistency and recency.

The results of the sequential similarity calculation in S are used by the *max* function to select the most *sequentially* similar candidate. The *max* function returns the class of the most sequentially similar candidate ($class_{max}$). The function compares the sequential similarities of the partial *s*-instances in the candidate set C. This comparison may be *incremental* (on one state at a time) or *non-incremental* (using the entire partial *s*-instance).

The comparison style depends on the nature of the individual sequential similarity metrics, detailed in the next section.

function *FindSimilarity* (X, C) **returns** $class_{max}$
for $x' \in X$, $c' \in C$
$S \leftarrow SSM(x', c')$; Calculate sequential similarity
$class_{max} \leftarrow max(S)$; Find the class of the most similar c'

Table 3-5. The function that finds the most relevant candidate by using sequential similarity metrics.

3.3 Sequential Similarity Metrics

The function SSM utilizes a set of sequential similarity metrics: *distance*, *convergence*, *consistency* and *recency*. The richness in the representation makes similarity comparisons for sequential instances possible. These metrics range from a simple distance measure to increasingly more elaborate ones. They exploit sequential characteristics of a transition sequence, so that each metric examines one aspect of such a transition sequence. The intuition here is that the more sequential characteristics two transition sequences share, the more the two transition sequences are alike. Briefly, the distance metric measures the Euclidean distance for numeric attributes and Hamming distance for symbolic attributes; the convergence metric measures reduction of sequential distance, the distance from one end of a sequence of states to another; the consistency metric counts the number of consecutive reductions of sequential distance; and the recency metric records the most recent reduction of sequential distance.

Let B and B' be s-instances whose differences for metric m from an s-instance A is represented as $m(A, B)$ and $m(A, B')$. A metric m is said to *distinguish* between B and B' with respect to A if and only if $m(A, B) \neq m(A, B')$ to some precision, say, the nearest tenth. As an example, if $m(A, B) = .04$ and $m(A, B') = .10$, then m distinguishes between B

and B' to the nearest tenth. If, on the other hand, $m(A, B) = .06$, but $m(A, B') = .10$, then m does not distinguish between B and B' with respect to A to the nearest tenth.

Given an input s-instance x^* and a candidate set C of partial s-instances c^i, *Find-Similarity* selects the most relevant candidate c^{max}. As shown in **Table 3-5**, *FindSimilarity* has two primary steps: *SSM calculation* and *candidate selection*. SSM calculation computes the value of a specific metric. In particular, it breaks up the computation by states so that the metric value at each state may be manipulated during candidate selection. Candidate selection uses the metric value from each state to select the most relevant candidate.

Depending on the type of metric, SSM either selects a candidate incrementally from one state to the next, or compares the metric values across all states in one shot. In the incremental approach, metric values between two partial s-instances x^i and c^i of length i are compared at a fixed length j, $1 \leq j \leq i$, by adding one state at each iteration. Let x_j^i denotes a subsequence of length j for the input partial s-instance x^i. Similarly, let c_j^i denotes a subsequence of length j for the candidate partial s-instance c^i. Then, comparing x^i and c^i at length j means to compare x_j^i and c_j^i. With a backward expansion, where prior states are added to a partial s-instance, the comparison between partial s-instances x^i and c^i starts at the current state s_n and incrementally includes the prior states, s_{n-1}, s_{n-2}, and so on. That is, the comparison starts with x_1^i and c_1^i at s_n and continues with x_2^i and c_2^i at state s_{n-1}, and so on. The process terminates when the metric value for x_j^i and c_j^i at state s_{n-j+1} is most similar (e.g., least distant or most convergent), for $i = j$ in x_j^i, and the class of c_j^i is returned. The rationale for favoring the current state over prior states is that the more recent the state is the more relevant the information it provides.

Recall that the s-instance x^w expands to a set of partial s-instances x^i of length i, and the set C consists of partial s-instances c^i of length i, $1 \leq i \leq w$. For the remainder of this section, let the input s-instance be

$$x^w = s_k, s_{k+1}, \ldots, s_{w+k-1} \quad [1]$$

where s_i is the i^{th} state in x^w. The partial s-instances for [1] are x^1, x^2, ..., and x^w. Let the candidate s-instance be

$$c^w = t_k, t_{k+1}, \ldots, t_{w+k-1} \quad [2]$$

where t_i is the i^{th} state in c^w. The partial s-instances for [2] are c^1, c^2, ..., and c^w. Each sequential similarity function takes a pair of partial s-instances (x^i, c^i) as input, and returns the sequential similarity value for the two partial s-instances.

Figure 3-1 shows a transition sequence from bridge with 5 states. It can be represented by the s-instance $x^5 = s_1, s_2, s_3, s_4, s_5$. The partial s-instances for x^5 are $x^1 = s_5$; $x^2 = s_4, s_5$; $x^3 = s_3, s_4, s_5$; $x^4 = s_2, s_3, s_4, s_5$; and $x^5 = s_1, s_2, s_3, s_4, s_5$.

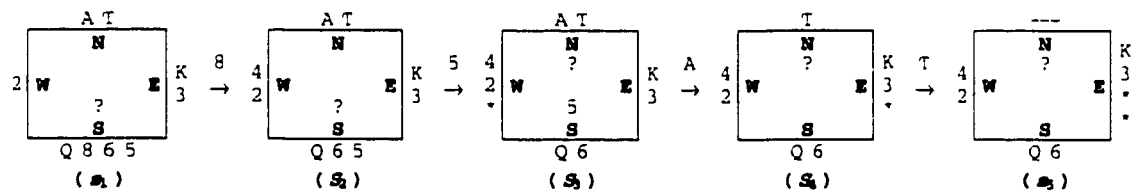


Figure 3-1. An example of a transition sequence from bridge. State s_5 is the current state, state s_4 is the state prior to s_5, and so on. A state is characterized in part by the cards belonging to the four players: West, North, East, or South. The transition sequence shows the state changes and the card played either by North or by South.

One way to represent the transition sequence is as a sequence of five description-values, a_1, a_2, a_3, a_4, and a_5, one description for each state. In other words, a_5 is the description of the current state $i = 5$, a_4 is the description of the prior state $i - 1 = 4$, and so on, as shown in **Table 3-6**. The "---" symbol means that no card in the suit of interest is present in a player's hand, and the '*' symbol represents a card played from a different suit. (Remember that North's and South's cards will diminish, while East's and West's

cards will be revealed as play progresses.) Section 4.1.2 provides further details on representation.

	a_1	a_2	a_3	a_4	a_5
x^5	2,AT,K3,Q865	42,AT,K3,Q65	42*,AT,K3,Q6	42*,T,K3*,Q6	42*,---,K3**,Q6

Table 3-6. A representation for an *s*-instance. The description a_1 represents the state s_1, the description a_2 represents the state s_2, and so on. Each description has four parts, separated by a comma, one for each player's cards in the suit of interest.

The corresponding partial *s*-instances using such a representation are shown in **Table 3-7**. Here, partial *s*-instance x^1 contains only the current state represented by a_1; x^2 contains a_1 for the current state s_1, and a_2 for the prior state s_2, and so on. The "-" symbol represents that such description does not exist.

	a_1	a_2	a_3	a_4	a_5
x^1	-	-	-	-	42*,---,K3**,Q6
x^2	-	-	-	42*,T,K3*,Q6	42*,---,K3**,Q6
x^3	-	-	42*,AT,K3,Q6	42*,T,K3*,Q6	42*,---,K3**,Q6
x^4	-	42,AT,K3,Q65	42*,AT,K3,Q6	42*,T,K3*,Q6	42*,---,K3**,Q6
x^5	2,AT,K3,Q865	42,AT,K3,Q65	42*,AT,K3,Q6	42*,T,K3*,Q6	42*,---,K3**,Q6

Table 3-7. A representation of a set of partial *s*-instances x^i of length i, $1 \le i \le 5$, and a_j is a description state s_j, $1 \le j \le 5$.

Given the above representation of a partial *s*-instance, the following sections describe how each metric is calculated for sequential similarity between two partial *s*-instances. Also, these metrics are used together when one metric cannot distinguish between two competing partial *s*-instances with respect to the target partial *s*-instance. For example, when the values of the distance metric between partial instance A and B, and that for A and B' are not distinguishable, that is, $m_{distance}(A, B) = m_{distance}(A, B')$, the similarity comparison resorts to a more refined metric. In this case, it resorts to the convergence metric by calculating $m_{convergence}(A, B)$ and $m_{convergence}(A, B')$ because the convergence metric is a refinement of the distance metric. The first metric to distinguish among the partial *s*-instance pairs returns its choice. If no metric can distinguish between B and B', a random selection is used.

3.3.1 Distance

The first sequential similarity metric is the distance metric. The *distance* metric quantifies the attribute-based differences between two partial s-instances x^i and c^i of length i. First, the difference of each attribute k within a state is calculated with $diff_k$ shown in [3]. The function $diff_k$ uses absolute distance for numeric features and Hamming distance for symbolic features. For example, if feature f_n is numeric, $f_n(x^i) = .34$, and $f_n(c^i) = .25$, then $diff_n(x^i, c^i) = .09$. Also, if feature f_s is symbolic, $f_s(x^i) =$ "a", and $f_s(c^i) =$ "z", then $diff_s(x^i, c^i) = 1$ because "a" ≠ "z". However, if $f_s(c^i) =$ "a", then $diff_s(x^i, c^i) = 0$.

$$diff_k(x^i, c^i) = \begin{cases} |f_k(x^i) - f_k(c^i)|, \text{ if } f_k \text{ is numeric.} \\ 1, \text{ if } f_k \text{ is discrete and } f_k(x^i) \neq f_k(c^i). \\ 0, \text{otherwise.} \end{cases} \quad [3]$$

Second, the state difference at state j is calculated with the function $sdiff_j$ shown in [4]. Let each state be represented by a set of m features. Given the attribute differences calculated by $diff_k$, $1 \leq k \leq m$, the state difference between two sub-sequences x_j^i and c_j^i at state j is defined as

$$sdiff_j(x^i, c^i) = \sum_{k=1}^{m} diff_k(x_j^i, c_j^i)^2 \quad [4]$$

Finally, the difference between two partial s-instances is calculated with $dist_i$ shown in [5]. Let each partial s-instance contain i states. Given the state differences calculated by $sdiff_j$, $1 \leq j \leq i$, the function $dist_i$ that computes the distance between x^i and c^i up to i states is defined as,

$$dist_i(x^i, c^i) = \sqrt{\sum_{j=1}^{i} sdiff_j(x^i, c^i)} \quad [5]$$

As an example, in **Figure 3-2**, take x^3 and c^3 as two partial s-instances, where s_5 is the current state, s_4 the predecessor state of s_5, and s_3 the predecessor state of s_4. Also as-

sume the state difference at state s_5 is calculated by $sdiff_5$ as .13, $sdiff_4$ = .14, and $sdiff_3$ = .18. Then, the distance between x^3 and c^3 is calculated by $dist_3$ as

$$dist_5(x^3, c^3) = \sqrt{\sum_{j=1}^{3} sdiff_j(x^3, c^3)} = \sqrt{.45} = .67 \qquad [6]$$

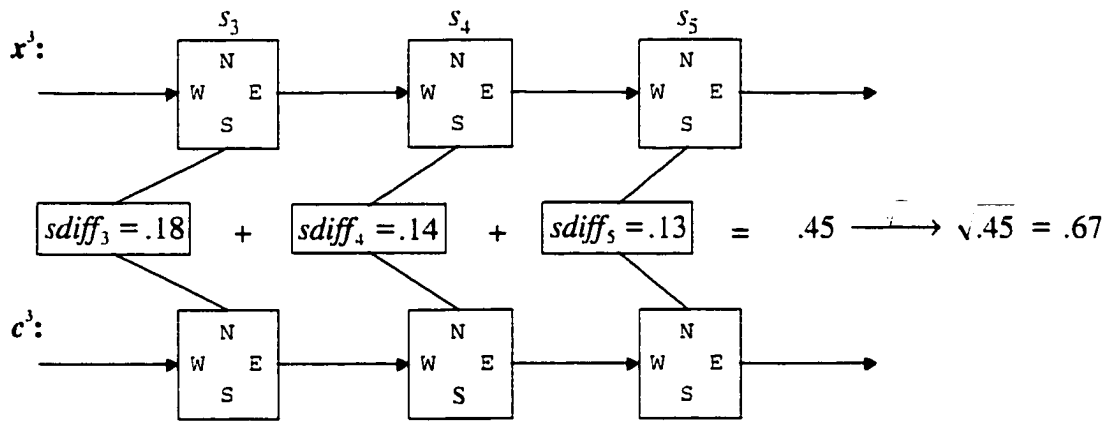


Figure 3-2. Calculations of the distance metric between two partial *s*-instance, x^3 and c^3, in terms of state differences.

The *distance* algorithm in Table 3-8 implements the distance metric. Given the partial *s*-instances x^i and c^i of length i, and the number n of states, $i \leq n$, the *dist* function returns the distance d between x^i and c^i. For each of the n states, the state difference is calculated for sub-sequences x_j^i of x^i and c_j^i of c^i at state j by the *sdiff* function, where m is the number of attributes for each state, shown in **Table 3-9**. The results are added to the total state differences *sum*. Finally, the distance d is returned as the square root of *sum*.

```
function dist (x^i, c^i, n) returns d
    sum ← 0
    for j from 1 to number of states n        ; For every state j
        sum ← sum + sdiff(x_j^i, c_j^i, m)    ; Accumulate state differences
    d ← √sum                                  ; Return distance between x^i, c^i
```

Table 3-8. Function that calculates the distance between two partial *s*-instances, x^i and c^i, up to length n.

As shown in **Table 3-9**, given the subsequences x_j^i of x^i and c_j^i of x^i of a given state j, and the number m of features in a state, the *sdiff* function returns the state difference *ssum* between x_j^i and c_j^i. For each of the m features, the attribute difference $difference_k$ for attribute f_k is calculated using the $diff_k$ function. The results are added to the state difference *ssum*. Finally, *ssum* is returned as the state difference.

```
function sdiff (A, B, m) returns ssum
    ssum ← 0
    for k from 1 to number of features m        ; for every attribute k
        difference_k ← diff_k(A, B)             ; difference on attribute k
        ssum ← ssum + (difference_k)^2          ; accumulate attribute differences
```

Table 3-9. Function that calculates the state difference between two subsequences, A and B, each with m features.

Given the set S of all distance values for partial s-instance pairs (x^i, c^i) of length i, $1 \le i \le w$, the next step is to find the most similar pair (x^{max}, c^{max}) by incrementally comparing with distance values. The pair (x^i, c^i) with a distance d that is the smallest (least distant) at the i^{th} iteration, $i \le w$, is selected. As an illustration, let the input be the partial s-instances of x^w shown in **Table 3-7** above and the candidate partial s-instances of c^w shown in **Table 3-10** below.

	a_1	a_2	a_3	a_4	a_5
c^1	-	-	-	-	Q2,---,74,A
c^2	-	-	-	42,7,A5,J	42,---,A85,J
c^3	-	-	32,98,4,Q5	732,8,T4,Q	732,---,T4,Q
c^4	-	3,98,4,KQ5	32,98,4,Q5	732,8,T4,Q	732,---,T4,Q
c^5	2,73,A,QJ9	2,73,A,QJ	42,73,A,J	42,7,A5,J	42,---,A85,J

Table 3-10. A hypothetical set of partial s-instances c^i retrieved for the set of corresponding partial s-instances in **Table 3-7**, where i is the number of states in c^i.

Table 3-11 shows the results of the distance calculation for each state. The row labeled (x^i, c^i) shows the details of the distance calculations for x^i and c^i. The columns labeled $sdiff_j^i$ hold the state difference between x_j^i and c_j^i at state s_j, $1 \le j \le i$. For example, the state differences between x^3 and c^3 are $sdiff_5^3 = .13$ at state s_5, $sdiff_4^3 = .14$ at state s_4,

and $sdiff_3^3 = .18$ at state s_3. The columns labeled $dist_j^i$ hold the distance between x^i and c^i accumulated from the state difference for the current state s_5 to state s_j. For example, the distance $dist_4^3$ between x^3 and c^3 from state s_5 to state s_4 is $\sqrt{.13+.14} = .52$, and $dist_3^3$ between x^3 and c^3 from state s_5 to state s_3 is $\sqrt{.13+.14+.18} = .67$. The "-" symbol means that no value is defined there.

Distance is computed incrementally. The rationale is that information from a more recent state is more relevant than that from a less recent state. Thus, the distance comparison starts at the most recent state. In the first iteration, since the most recent distance, $dist_5^1$ for (x^1, c^1) is not the smallest among $dist_5^i$, (x^1, c^1) is removed from consideration. For the second iteration, all decisions must be based on the $dist_4^i$. Once again, the most recent distance $dist_4^2$ for (x^2, c^2) is not the smallest among $dist_4^i$, (x^2, c^2), so it is removed from consideration. Finally, the third iteration, the distance $dist_3^3$ for (x^3, c^3) is the smallest among $dist_3^i$, so the class for c^3 will be returned.

	$sdiff_1^i$	$sdiff_2^i$	$sdiff_3^i$	$sdiff_4^i$	$sdiff_5^i$	$dist_1^i$	$dist_2^i$	$dist_3^i$	$dist_4^i$	$dist_5^i$
x^1, c^1	-	-	-	-	.27	-	-	-	-	.52
x^2, c^2	-	-	-	.23	.15	-	-	-	.62	.39
x^3, c^3	-	-	.18	.14	.13	-	-	.67	.52	.36
x^4, c^4	-	.25	.23	.12	.13	-	.85	.69	.50	.36
x^5, c^5	.35	.25	.18	.16	.17	1.05	.87	.71	.57	.41

Table 3-11. An example of distance calculation for a set of partial s-instance pairs, (x^i, c^i), where i is the length of a partial s-instance. The difference at a given state j is shown in the columns labeled s_j, and the distance between (x^i, c^i) is shown in the columns labeled $dist_j^i$.

The numeric precision used for the distance metric may affect the overall candidate selection. In the above example, using a precision to the nearest hundredth, the distance $dist_3^3$ between x^3 and c^3 distinguishes it from the remaining candidates, c^4 and c^5. Using a precision to the nearest tenth, however, the distance $dist_3^3$ between x^3 and c^3 in **Table 3-11** does not distinguish among the remaining candidates to the nearest tenth, that is, the distance $dist_3^i$ for all three remaining candidates, c^3, c^4, and c^5, rounds to .7. To re-

fine this similarity calculation, the convergence may be used as described in the next section.

3.3.2 Convergence

The *convergence* metric quantifies the change in the state difference between partial *s*-instances, measured from their least recent states to their current states. In particular, the convergence for x^i and c^i from the least recent state s_i to the currents state s_t is shown in [7],

$$convergence(x^i, c^i) = dist_i^i(x^i, c^i) - dist_c^i(x^i, c^i) \qquad [7]$$

where $dist_i^i(x^i, c^i)$ is the distance between x^i and c^i for up to i states and $dist_c^i(x^i, c^i)$ is the distance between x^i and c^i at the current state s_c. As an example, in **Figure 3-3**, let x^3 and c^3 be two partial *s*-instances, and let s_5 be the current state and s_3 be the least recent state. Also, let the distance between x^3 and c^3 at the current state s_5 be $dist_5^3(x^3, c^3) = .36$, and the distance between x^3 and c^3 at the least recent state be $dist_3^3(x^3, c^3) = .67$. Then, the convergence between x^3 and c^3 is calculated as

$$convergence(x^3, c^3) = dist_3^3(x^3, c^3) - dist_5^3(x^3, c^3) = .31$$

Note that the convergence between two partial *s*-instances x^i and c^i at the current state is always 0, that is, $dist_5^i(x^i, c^i) - dist_5^i(x^i, c^i) = 0$.

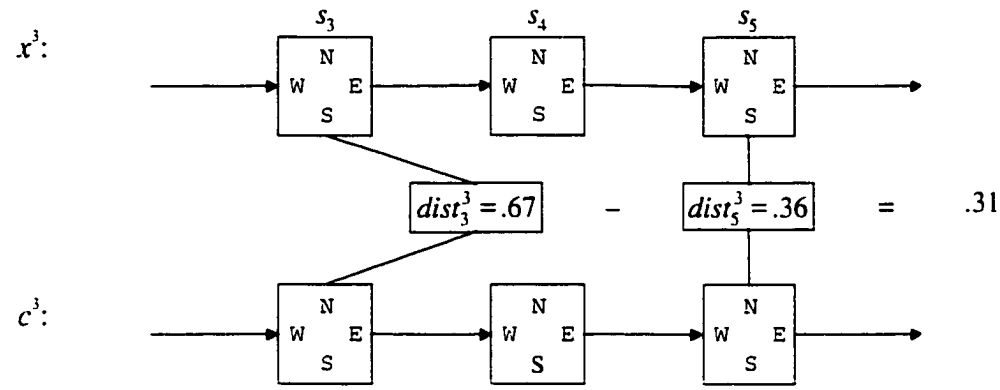


Figure 3-3. Calculation of the convergence metric between two partial *s*-instances, x^3 and c^3, in terms of distances. The convergence is .31, the difference between $dist_3^3$ and $dist_5^3$.

The *conv* algorithm in **Table 3-12** implements the convergence metric. Given the partial *s*-instances x^i and c^i of length i, the *conv* function returns the reduction in distance v for x^i and c^i calculating from the least recent state s_1 to the current state s_n. The distance d^1 between x^i and c^i at the current state s_n is calculated as $dist(x^i, c^i, 1)$, and the distance d^i between x^i and c^i for up to j states is calculated as $dist(x^i, c^i, j)$ as described in **Table 3-8**. The convergence v is the difference d^i minus d^1.

Function *conv* (x^i, c^i) **returns** v	
$d^1 \leftarrow dist_1(x^i, c^i, 1)$	; Distance between x^i and c^i at the current state
$d^i \leftarrow dist_i(x^i, c^i, i)$	; Distance between x^i and c^i for up to i states
$v \leftarrow d^i - d^1$	; Convergence v as the difference between d^1 and d^i.

Table 3-12. Function that calculates the convergence between two *s*-instances, x^i and c^i.

Given the set S of all convergences for partial *s*-instance pairs (x^i, c^i) of length i, the next step is to find the most similar pair (x^{max}, c^{max}) by incrementally comparing the convergence values. The pair (x^i, c^i) with a convergence v that is the largest (most convergent) at the i^{th} iteration, $i \leq w$, is selected. Continuing the example from **Table 3-11**, the three remaining partial *s*-instance pairs that are not distinguishable by the distance metric shown in **Table 3-13** are compared in terms of convergence. The columns labeled $conv_j^i$ hold the convergence between x^i and c^i between state s_j and the current state s_5. For exam-

ple, the convergence $conv_3^3$ between x^3 and c^3 of up to state three states is $dist_3^3 - dist_5^3 = .67 - .36 = .31$. Recall that the larger the convergence the more similar it is.

Like the distance metric, the convergence metric uses an incremental comparison method. In the first iteration, since the most recent convergence $conv_3^3$ for (x^3, c^3) is not the largest among $conv_3^i$, (x^3, c^3) is removed from consideration. On the second iteration, the convergence $conv_2^4$ for (x^4, c^4) is the largest among $conv_2^i$, the class for the candidate c^4 will be returned.

	$dist_1^i$	$dist_2^i$	$dist_3^i$	$dist_4^i$	$dist_5^i$	$conv_1^i$	$conv_2^i$	$conv_3^i$	$conv_4^i$	$conv_5^i$
x^1, c^1	-	-	-	-	.52	-	-	-	-	.0
x^2, c^2	-	-	-	.62	.39	-	-	-	.23	.0
x^3, c^3	-	-	.67	.52	.36	-	-	.31	.16	.0
x^4, c^4	-	.85	.69	.50	.36	-	.49	.33	.14	.0
x^5, c^5	1.05	.87	.71	.57	.41	.64	.46	.30	.16	.0

Table 3-13. An example of a convergence calculation for a set of partial *s*-instance pairs, (x^i, c^i), where i is the length of the partial *s*-instance. The distance of a given state j is shown in the columns labeled $dist_j^i$. The convergence of a given state j between (x^i, c^i) is shown in the columns labeled $conv_j^i$.

The value of convergence may be affected by the numeric precision as well. Although **Table 3-13** shows c^4 to be the most convergent candidate, the use of the convergence metric does not distinguish among the remaining candidates to the nearest tenth, that is, both convergence value in $conv_2^i$ of the two remaining candidates, c^4, and c^5, round to .5. To refine this similarity calculation, consistency metric may be used as is described in the next section.

3.3.3 Consistency

For each input partial *s*-instance and the candidate retrieved for it, the *consistency* metric tallies the number of consecutive positive reduction in convergence for all adjacent

states for two partial s-instances x^i and c^i. Reduction in convergence $\Delta_{j,j+1}$ between state j and state $j + 1$ may be calculated in terms of convergence, that is, $\Delta_{j,j+1} = conv(x^i_{j+1}, c^i_{j+1}) - conv(x^i_j, c^i_j)$. Therefore, consistency between x^i and c^i may be measured by [8]:

$$\begin{aligned} &consistency(x^i, c^i) = \text{the largest } t \text{ in } [1, n\text{-}1] \text{ such that} \\ &\quad conv(x^i_{j+1}, c^i_{j+1}) > conv(x^i_j, c^i_j) \text{ for } j = 1 \text{ to } t \text{ where } 1 \le j < n \end{aligned} \qquad [8]$$

where $conv(x^i_{j+1}, c^i_{j+1})$ is strictly larger than $conv(x^i_j, c^i_j)$, t counts the number of consecutive reductions in convergence (positive convergence) between two adjacent states, t is the largest such number in the range from 1 to n-1, and n is the total number of states. As an example, in **Figure 3-4**, assume x^4 and c^4 to be two partial s-instances and assume the reduction in convergence between state s_4 and state s_5 is $\Delta_{4,5} = .14$, the reduction in convergence between state s_3 and state s_4 is $\Delta_{3,4} = .19$, and the reduction in convergence between state s_2 and state s_3 is $\Delta_{2,3} = .16$. Then, since there is only one consecutive reduction in convergence (between $\Delta_{3,4}$ and $\Delta_{4,5}$), the consistency t between x^4 and c^4 is 1.

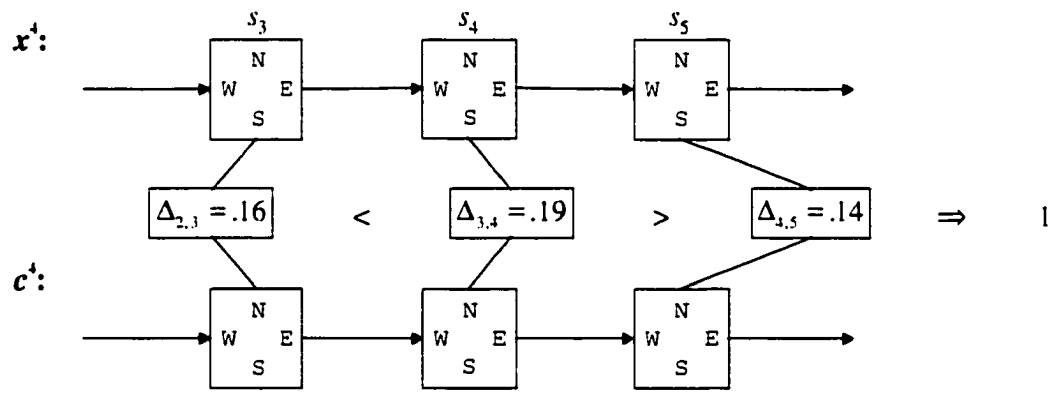


Figure 3-4. Calculation of the consistency metric between two partial s-instances, x^4 and c^4, in terms of convergences. The consistency is 1 because there is only a single reduction in convergence, between s_4 and s_5.

The algorithm that implements the consistency metric appears in **Table 3-14**. Given the partial s-instances x^i and c^i of length i, the *consistency* function returns the maximum number t_{max} of reductions in convergence for x^i and c^i. The convergence be-

tween x^i and c^i at the state s_j and at the predecessor state s_{j+1} are calculated with the *conv* function described in **Table 3-12**. If there is a positive reduction in convergence, the value t is incremented. Finally, t_{max} is returned as the maximum of t.

```
function consistency (x^i, c^i) returns t_max
    t ← 0, t_max ← 0
    for j from 1 to i – 1                    ; For n-1 iterations
        v_j^i ← conv(x_j^i, c_j^i)           ; Get convergence between x_j^i and c_j^i
        v_{j+1}^i ← conv(x_{j+1}^i, c_{j+1}^i)   ; Get convergence between
                                             ; x_{j+1}^i and c_{j+1}^i
        if v_{j+1}^i – v_j^i > 0 then        ; If there is reduction in convergence
            t ← t + 1                        ; Increment t
        else
            if t > t_max then
                t_max ← t                    ; Update t_max
            t ← 0                            ; Reset t if there is a break when
                                             ; counting up t
        if t > t_max then
            t_max ← t                        ; Assign t to t_max
```

Table 3-14. Function that calculates consistency between two partial *s*-instances, x^i and c^i.

Given the set S of all consistency values for partial *s*-instance pairs (x^i, c^i) of length i, the next step is to find the most similar pair (x^{max}, c^{max}) by comparing the consistency values in S. The pair (x^i, c^i) with the largest consistency at the i^{th} iteration, $i \leq w$, is selected. Continuing the example from **Table 3-13**, the two partial *s*-instance pairs that are not distinguishable by the convergence metric shown in **Table 3-15** are compared in terms of convergence. The columns labeled $\Delta^i_{j,k}$ hold the reduction in convergence between state j and state k for x^i and c^i. Specifically, the consistency $cons^4$ between x^4 and c^4 is 1 because there is only one reduction in convergence from $\Delta^4_{3,4} = .19$ to $\Delta^4_{4,5} = .14$. The consistency $cons^5$ between x^5 and c^5 is 2 because there are two consecutive reductions in

convergence from $\Delta^5_{1,2} = .18$ to $\Delta^5_{2,3} = .16$, and from $\Delta^5_{2,3} = .16$ to $\Delta^5_{3,4} = .14$. The larger the consistency the more similar it is. Therefore, the class for c^5 will be returned.

	$conv_1^i$	$conv_2^i$	$conv_3^i$	$conv_4^i$	$conv_5^i$	$\Delta^i_{1,2}$	$\Delta^i_{2,3}$	$\Delta^i_{3,4}$	$\Delta^i_{4,5}$	$cons^i$
x^1, c^1	-	-	-	-	.0	-	-	-	-	-
x^2, c^2	-	-	-	.23	.0	-	-	-	.23	0
x^3, c^3	-	-	.31	.16	.0	-	-	.15	.16	0
x^4, c^4	-	.49	.33	.14	.0	-	.16	.19	.14	1
x^5, c^5	.64	.46	.30	.16	.0	.18	.16	.14	.16	2

Table 3-15. An example of a consistency calculation for a set of partial *s*-instance pairs, (x^i, c^i), where i is the length of a partial *s*-instance. The convergence at a given state j is shown in the columns labeled $conv^i_j$. The convergence between adjacent states j and k, is shown in the columns labeled $\Delta^i_{j,k}$ The consistency between (x^i, c^i) is shown in the columns labeled $cons^i$.

The consistency metric is affected indirectly by the numeric precision of the convergence metric. In **Table 3-15**, if the precision used for convergence is to the nearest tenth, both $\Delta^5_{1,2}$ and $\Delta^5_{2,3} = .2$, and the only reduction in convergence occurs between $\Delta^5_{2,3} = .2$ and $\Delta^5_{3,4} = .1$. As a result, the consistency between x^5 and c^5, and that for x^4 and c^4 are both 1. Under such circumstances, the similarity calculation may further be refined with the last metric – recency.

3.3.4 Recency

The *recency* metric identifies the most recent point in the sequence where a reduction in convergence has occurred between two partial *s*-instances x^i and c^i as defined in [9].

$$recency(x^i, c^i) = \text{the greatest } j \text{ in } [1, n-1] \text{ such that } conv(x^i_j, c^i_j) > conv(x^i_{j+1}, c^i_{j+1}) \quad [9]$$

Reduction in convergence at state j occurs when $\Delta_{j-1,j}$ the reduction in convergence from state j - 1 to j is greater than $\Delta_{j,j+1}$ the reduction in convergence from state j to j + 1. As an example, in **Figure 3-5**, the reduction in convergence between s_4 and s_5 is $\Delta_{4,5} = .14$, the

reduction in convergence between s_3 and s_4 is $\Delta_{3,4} = .19$, and the reduction in convergence between s_2 and s_3 is $\Delta_{2,3} = .16$. Since the only reduction in distance occurs between $\Delta_{3,4}$ and $\Delta_{4,5}$, the most recent state at which a reduction in convergence occurs is s_4, so the recency $r = 4$.

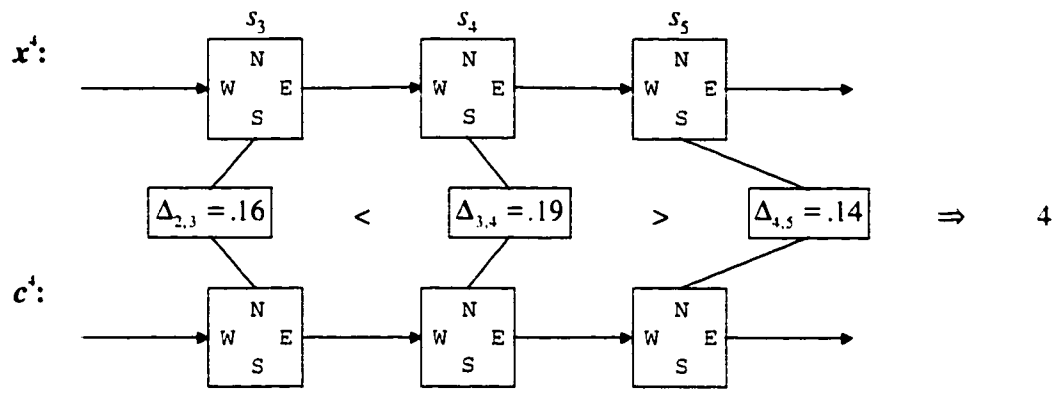


Figure 3-5. Calculations of the recency metric between two partial s-instances, x^3 and c^3, in terms of reduction in distance between adjacent states. The recency is 4 because a reduction of distance occurs at s_4.

The algorithm that implements the recency metrics appears in **Table 3-16**. Given the partial s-instances x^i and c^i of length i, the *recency* function returns the most recent state number r where a reduction in convergence occurs between x^i and c^i. Once again, the convergence between x^i and c^i at the current state s_j and that for the predecessor state s_{j-1} are calculated with the *conv* function described in **Table 3-12**. The recency value r is set to the most recent state number j when a positive reduction in convergence occurred.

```
function recency (x^i, c^i) returns r
    r ← n
    for j from 1 to n − 1                    ; n is the number of states
        v_j^i ← conv (x_j^i, c_j^i)          ; convergence between x_j^i and c_j^i
        v_{j+1}^i ← conv (x_{j+1}^i, c_{j+1}^i)   ; convergence between x_{j+1}^i and c_{j+1}^i
        if conv_{j+1}^i − conv_j^i > 0 then
            r ← j                            ; the first occurrence of reduction in
                                             ; distance
```

Table 3-16. Function that calculates recency between two partial s-instances, x^i and c^i.

Given the set S of all recency values for partial s-instance pairs (x^i, c^i) of length i, the next step is to find the most similar pair (x^{max}, c^{max}) by comparing the recency values in S. The pair (x^i, c^i) with the highest (recency value) at the i^{th} iteration, $i \leq w$, is selected. Continuing the example from **Table 3-15**, the two partial s-instance pairs that are not distinguishable by the consistency metric are compared in terms of recency in **Table 3-17**. The columns labeled rec^i hold the recency values between x^i and c^i. The recency rec^4 between x^4 and c^4 is 4 because $\Delta^4_{3,4} > \Delta^4_{4,5}$ at s_4, and the recency rec^5 between x^5 and c^5 is 3 because $\Delta^5_{2,3} > \Delta^5_{3,4}$ at s_3. Since s_4 is more recent than s_3, (x^4, c^4) is selected and its class will be returned.

	$\Delta^i_{1,2}$	$\Delta^i_{2,3}$	$\Delta^i_{3,4}$	$\Delta^i_{4,5}$	rec^i
x^1, c^1	-	-	-	-	-
x^2, c^2	-	-	-	.23	-
x^3, c^3	-	-	.15	.16	-
x^4, c^4	-	.16	.19	.14	4
x^5, c^5	.18	.16	.14	.16	3

Table 3-17. An example of a recency calculation for a set of partial s-instance pairs, (x^i, c^i), where i is the length of a partial s-instance. The convergence between adjacent states j and k, is shown in the columns labeled $\Delta^i_{j,k}$ The recency between (x^i, c^i) is shown in the columns labeled rec^i.

3.4 Analysis

SIBL gains performance improvement by moderately increasing algorithm complexity. The performance improvement is due to the adjustment of the decision boundary and the use of sequential attribute weighting. This section shows that, SIBL is an $O(n^2)$ algorithm with average space requirement for prototypical instances $w \times cn$, where w is the number of states in the augmented context, and c about half of the training instance population n.

3.4.1 Adjustment of a Decision Boundary

In IBL, a decision boundary exists between any two classes of instances. The boundary is often defined in terms of the distance between two classes. Such distance, or inversely the similarity, is calculated from the attribute values of the instances of each class. As a simple example, **Figure 3-6** shows a two-class (× and o) problem with two attributes measured from 0 to 1. The prototype of each class (dark box) can be calculated as the average of the attribute values of the instances in each class. The decision boundary is simply the perpendicular bisector of the line joining the two prototypes. In this example, all but two instances are correctly separated by the decision boundary. (The straight-line decision boundary is used here for the purpose of clarity and does not imply that IBL or similar algorithms can only be used for linearly separable examples.)

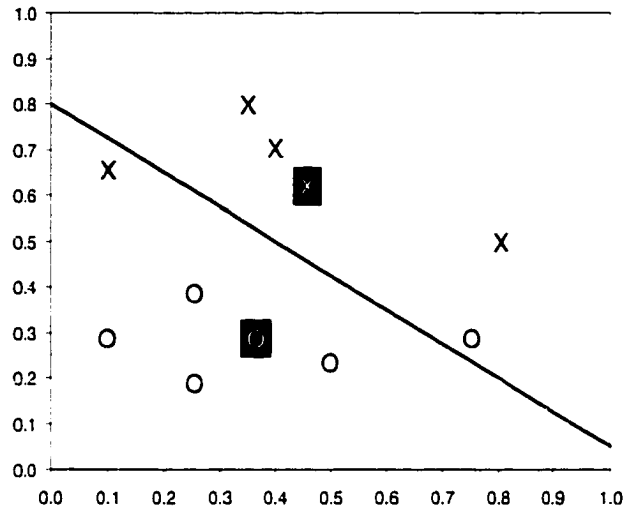


Figure 3-6. The decision boundary (solid line) for a two-class problem, where the prototypes for each class (dark box) is the average of the instance values of that class.

In SIBL, such a decision boundary may be adjusted by taking into account the sequential nature of the features. This is accomplished by deriving additional features, called *dynamic* features, from the existing ones without increasing the size of the instance space. Dynamic features are based on the use of sequential similarity metrics, such as convergence, consistency and recency as described in the last section. Recall that an *s*-instance consists of information from one or more states. Convergence, for example, is calculated as the difference between the attribute values of the least recent state and those of the most recent state. By taking into account the convergence, the average similarity between the two prototypes can either decrease if the convergence value is negative (shifting the decision boundary vertically down) or increase if the convergence value is positive (shifting the decision boundary vertically up). In this example, moving the boundary up will pick up a correct classification for class O as shown in **Figure 3-7**.

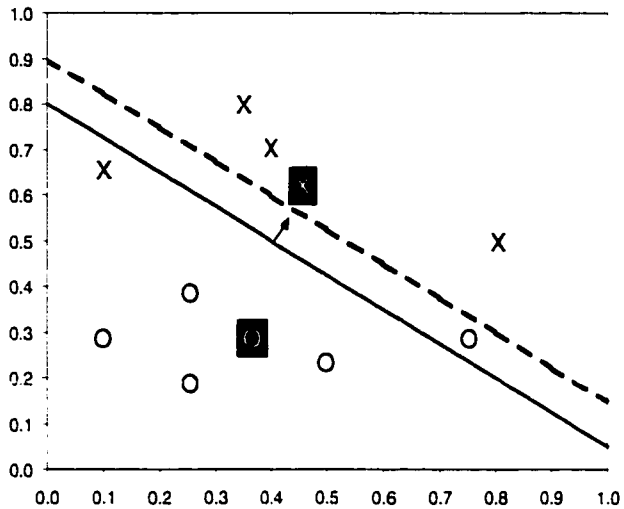


Figure 3-7. An upward shift of the decision boundary caused by the change in value of features derived from existing features at runtime. As a result, one more instance (O) is correctly classified.

3.4.2 Sequential Attribute Importance

Attribute importance has been used to improve performance by adjusting the relative weight of attributes (Wettschereck et al. 1997). As shown in **Figure 3-8**(a), when the weight decreases, for example, from 1 to .5 on the vertical attribute, the slope of the decision boundary decreases (becomes flatter). On the other hand, when the weight increases, for example, from 1 to 1.25 on the vertical attribute, the slope of the decision boundary increases (becomes steeper) as in **Figure 3-8**(b).

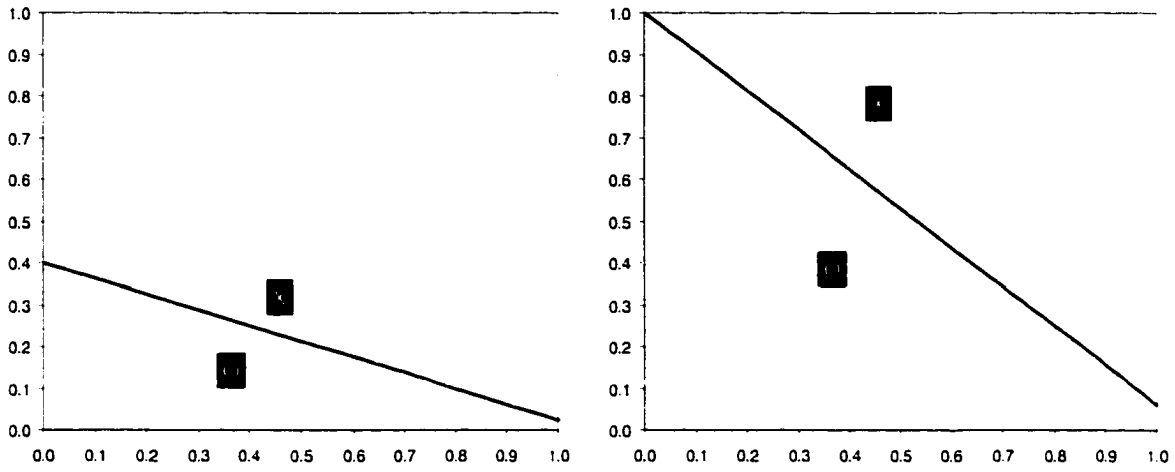


Figure 3-8. (a) When the weight of the vertical attribute decreases, the slope of the decision boundary becomes less steep. (b) When the weight of the same attribute increases, the slope of the decision boundary becomes steeper.

Weighting the features can be done either stochastically or through incremental feedback. The stochastic approach assigns attribute importance by gathering attribute statistics in one shot. Incremental feedback assigns attribute importance by stepwise adjustment with each input instance. Both approaches are based on pre-defined attributes.

In SIBL, sequential attribute importance is based on dynamic features, attributes derived from the existing attribute. One way to accomplish this is to place more weight on attributes representing a more recent state than that for a least recent state. Recall that the incremental approach for the sequential similarity with the distance metric and with the convergence metric works with one state in a partial *s*-instance at a time. (See Section 3.2.3.) Such an approach favors the attributes representing a more recent state by using them for comparison first. If the value of the attribute for a more recent state can distinguish two candidates in the similarity comparison, the other attributes for less recent states are ignored, or de-emphasized. Continuing the example from **Figure 3-7**, a change

in the vertical attribute weight in **Figure 3-9** creates a new decision boundary marked by the dash and dotted line, which picks up another correct classification for "×".

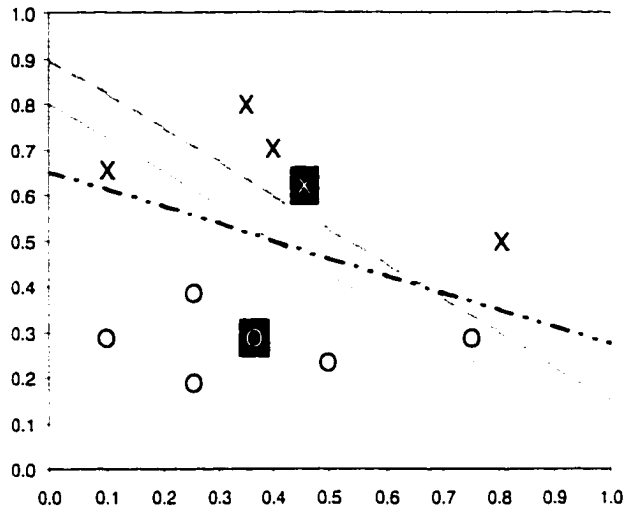


Figure 3-9. After an adjustment to the weight of the vertical attribute, the decision boundary tilts slightly counterclockwise. As a result, one more instance (×) is correctly classified.

3.4.3 Complexity

SIBL's performance gain comes at an expense of algorithmic complexity in terms of time and space. The time function is $O(n^2)$ whereas the space function is $O(n)$. Nevertheless, a sequential problem such as bridge is considered hard because it is undecidable (Chapman 1987). A quadratic time and linear space algorithm is considered efficient.

The time resource is characterized in terms of the size of the input n. The three major steps in the SIBL algorithm are: expand, similarity, select.

1. Expand

For each of the n input instances, Expand is instantiated w times, where w is the constant window size. Therefore, with Expand alone, the complexity of the algorithm is $f(n) = O(g_1(n)) = O(wn)$, where $g_1(n) = w \times n$.

2. Similarity

For each of the $w \times n$ instantiated instances, similarity is calculated once with each of the m stored prototype instances. Since not all instances will be stored, $m = c \times n$, where c is a fraction and $0 \leq c \leq 1$. In the worst case, $m \approx n$. Therefore, the worst case complexity of the algorithm with the similarity calculation is $f(n) = O(g_2(n)) = O(wn^2)$, where $g_2(n) = g_1(n) \times n = w \times n \times n = wn^2$.

3. Select

Among the wn^2 instances, every w of them forms a candidate set that is used to select the best candidate. Hence, there will be n^2 such candidate sets. With the majority vote, the complexity to select a candidate is $g_3(n) = O(w)$. If sequential similarity metrics are used, the complexity is $g_3(n) = O(w \times w - 1)$. Neither of the two is directly related to n. Using the higher of the two, the accumulated complexity is now $f(n) = g_2(n) / w \times w^2 = O(w^2n^2)$. As $n \rightarrow \infty$, w^2 becomes insignificant. Therefore, the time complexity for SIBL is $O(n^2)$, or a quadratic complexity.

At runtime, the space complexity with n input examples is $d \times f(n)$, where $f(n)$ is associated with the number of input examples, and d is the number of copies of $f(n)$ created during runtime for cross reference. For storage, the space complexity is $c \times f(n)$

where c is a fraction that quantifies the portion of the input examples as prototypes. In addition, the space complexity increases by w-fold to include sequential dependency, where w is the size of the window in a transition sequence. Thus, the total space complexity is $w \times c \times d \times f(n)$. Since c is proportional to n, as $n \rightarrow \infty$, $c \rightarrow \infty$, and $w \times d$ becomes insignificant. Therefore, the space complexity for SIBL is $O(n^2)$, a quadratic complexity.

3.5 Summary

Learning has been used regularly to form concept descriptions for a domain where sufficient data is available. Learning may be viewed as search through the space of concept descriptions. For synthesis domains, incremental learning is more suitable because of the need for frequent updates. Also, in an incomplete information domain, the concept descriptions may be adjusted incrementally as more information becomes available.

SIBL is an extension to IB4, and is an incremental learning paradigm that can learn sequential concepts in an incomplete information domain. The synthesis examples used by SIBL have a multi-state context that is broader than the representation used in a typical synthesis domain. A set of sequential similarity metrics is used to search for sequential concepts as concept descriptions. The use of multi-state context in an example allows SIBL to improve performance by adjusting decision boundaries with dynamic attributes and with the weight of the attributes. SIBL is algorithmically efficient for learning a synthesis problem.

CHAPTER FOUR :

EMPIRICAL EVALUATION

The primary goal of this chapter is to demonstrate the feasibility of the proposed concept, sequential dependency (Chapter 2), and the implemented learning approach, SIBL (Chapter 3), for learning sequential concepts. The problem of learning sequential concepts is cast as an example of the process of Knowledge Discovery in Databases, or KDD (Fayyad et al. 1996). SIBL automates knowledge discovery of concepts from data. It is intended to produce models for a real-world domain. The experiments described here offer evidence that the proposed concept and algorithms are an effective way to learn sequential concepts.

The first part of this chapter explains how the experiments were conducted. It is divided into three parts: research domain, representation of examples, and the knowledge discovery procedure. Bridge is chosen as the research domain because it is a synthesis problem and offers the challenge of incomplete information, that is, part of the bridge world is unknown. Syntactically, the input examples are represented as attribute-value pairs for the SIBL algorithm. Semantically, a multi-state context has been included in an example to support the discovery of sequential concepts. The experiments are guided by a knowledge discovery procedure, which included data collection, preprocessing, data mining, and post-processing.

The second part of this chapter presents the main results of a series of SIBL algorithms. These results may be divided into three groups: exploratory, quantitative, and qualitative results. The exploratory results were an attempt to use a multi-state context for an input example to disambiguate the description of a given state. The results show a steady improvement with the addition of more states (context). The quantitative results represent an effort to boost the performance with a combination of multiple models using

majority vote. These results outperform the exploratory results. Finally, the quantitative results represent the performance with a set of sequential similarity metrics described in the last chapter. These results are the best of the three.

The third part of this chapter compares a series of variations for the SIBL algorithms. These variations represent alternatives in terms of representation, numeric precision, context size, and context type. These empirical results clarify the relative importance of portions of this work.

4.1 Experimental Design

This section explains how experiments were conducted. It includes the research domain, representation of examples, and the knowledge discovery procedure. The game of bridge is the target domain because it is sequential in nature and challenging. It involves synthesizing a sequence of actions to achieve the goal of winning a contest. Although I chose the game of bridge as the target domain, other problems that involve sequence synthesis (e.g., the game of hearts) might also benefit from the algorithms presented here.

In terms of knowledge representation, the syntactic aspect is typically dictated by the chosen learning method. Since IBL is chosen as the learning method in this research, the representation is based on attribute value pairs, a common representation. The semantic aspect of knowledge representation is influenced by the amount of domain knowledge used. The more the domain knowledge, the less flexible it is for transferring the method to a new domain. This research minimizes the use of domain-specific knowledge to keep the proposed learning method flexible. Instead, a multi-state context is added in an input example to enable sequential concept discovery. Finally, the knowledge discovery proce-

dure is a systematic process to produce concept descriptions from examples. It varies from one application to another.

4.1.1 Bridge

The game of bridge was chosen as the research domain because it is a sufficiently complex incomplete information game, and because the result of a bridge problem solver can be evaluated by standard criteria. A complete description of the game can be found in the Appendix. For convenience, a brief description is supplied here. Bridge is a four-player game that consists of bidding followed by play. During bidding, a specific number of tricks for winning (the *contract*) is determined. During play, the players try to make or break the contract by playing the cards in a specific order. To begin, a deck of 52 cards is distributed evenly to four players; each holds 13 cards called a *hand*. In each round, each player plays a card. Together, the four cards played during each round are called a *trick*. All players must play a card of the suit led, unless they have no card in that suit (*void*). In that case, the player may play a card in another suit (*drop*) or play a trump card, a card in the trump suit, which is determined during the bidding phase. The bidding determines the player who plays the first card, and play proceeds clockwise. In each trick, the player who plays the highest rank of the suit wins the trick and leads a card to begin the next trick.

This work concentrates only on the playing phase. The problem of bridge play is to design a sequence of actions (card plays) that guides a bridge player to reach a specific goal (e.g., make or break a contract). Also, the focus here will be to guide a particular player, the *declarer*, who first named the contract suit during bidding. The declarer also decides which card to play for its partner (the *dummy*), whose cards are exposed on the table for all to see after the first lead (first card play) by the opponent. Since 13 cards are dealt to each player, the declarer makes 26 card plays in a deal. In other words, each deal

provides 26 examples of declarer play. Given a set of such examples, the learning algorithm is expected to identify sequential bridge playing sequences for the side of the declarer and dummy.

Like many other large sequential domains, bridge's search space is intractable. The search space consists of two factors: static descriptions and dynamic plays. For static descriptions, with a 52-card deck, there are

$$S = \binom{52}{13} * \binom{39}{13} * \binom{26}{13} * \binom{13}{13} = 5.36\text{E}+28$$

possible deals (descriptions). For dynamic plays, the branching factor is $n*m^3$ for each trick, where n is the number of cards currently remaining in each hand (i.e., the number of choices the lead player of a trick has to play a card), and $m = n/4$ is the average number of choices the other three players have to play a card among the four possible suits. For example, when all four players still have 13 cards, the branching factor for the first trick is 446 as discussed in Section 2.1.1. For a complete deal, the number of possible plays will be

$$D = \prod_{n=1}^{13} n*(n/4)^3 = 1.27\text{E}+18$$

Taking both static descriptions and dynamic plays into consideration, there are

$$S * D = 6.83\text{E}+46$$

possible states in bridge. Domain knowledge must be used and be learned to play bridge effectively.

The game of bridge is a domain with many sub-problems. A particular contract, say three no trump, is viewed as a unique problem because it dictates the strategies and tactics to be used. In the above example, the number "three" determines the number of

tricks needed to win a contest. Since the base number for trick counting is six, "three" means winning nine tricks (6 + 3) out of a total of thirteen tricks. The term "no trump" (NT) means that no suit has any privilege over the other suits, and only the ranks of the suit led are used to decide win or lose. The number of tricks needed to win and the rules governs the natures of suits and ranks determine different strategies. In the following experiments, I concentrated on one of the most popular contracts – 3 No Trump (3NT).

4.1.2 Representation

Domain knowledge plays an important role in knowledge discovery. In general, the more knowledge that is initially provided, the less search is needed for knowledge discovery. At one extreme, with explanation-based learning (EBL), more efficient knowledge may be learned from only a handful of annotated examples, such that the results of learning is to improve the efficiency of problem solving rather than adding new knowledge (Cohen 1990; Minton 1988). One difficulty with EBL is the *utility problem*, where the cost of using new knowledge offsets the benefit of that knowledge (Minton 1990). At the other extreme, a genetic algorithm (GA) typically uses very little domain knowledge. Instead, it uses as much search as it can afford (Goldberg 1989; Holland 1975). In the case of empirical learning (Michalski and Kodratoff 1990), domain knowledge may be used to constrain the search space. At the same time, the use of domain knowledge may be limited to allow domain transfer.

Domain knowledge may be divided into operational knowledge and representation knowledge. In the context of knowledge containers for a typical CBR system (Lenz et al. 1998), representational knowledge is similar to the vocabulary, and operational knowledge includes the similarity measure, the case base and the solution transformation. *Representational knowledge* is knowledge for representing an example in the domain of

interest. In bridge, for example, representational knowledge may be used to represent a bridge state when either the declarer or the dummy is to play a card. *Operational knowledge* is knowledge for carrying out actions to solve a problem. In bridge, operational knowledge may be applied to formulate a sequence of actions directed to the goal. Representational knowledge may be domain-independent, while operational knowledge is often domain-dependent. For example, although the same knowledge may be used to represent a state for both bridge and the game of hearts, the operational knowledge for the two are quite different.

The main information for a bridge state consists of mostly the card distributions among the players. As depicted in **Figure 4-1**, each of the four players has a total of thirteen cards in four different suits. At any given time, a player is either leading a suit at the beginning of a trick, or following a suit led in the middle of a trick. As a convention, the declarer is at South (S) and the dummy is at North (N). This leaves West (W) and East (E) as the opponents. For example, if West leads the 9 of spades (♠), this will focus a bridge state on the spade suit, since bridge requires that one follows suit, and a no trump contract removes the threat of being trumped. Therefore, only the cards in the spade suit are represented in a state; the cards in the other suits may be ignored.

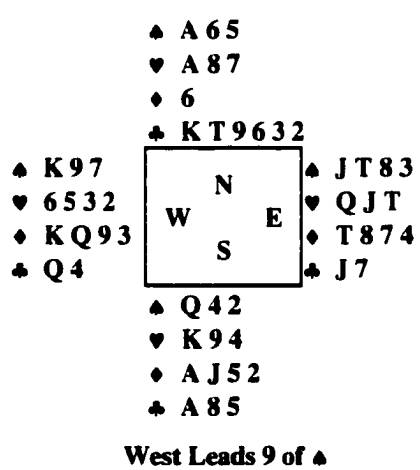


Figure 4-1. A bridge example with a complete suit descriptions.

Such a suit-oriented representation effectively reduces the search space. For example, in a hand-oriented representation, all 52 cards are needed to represent a bridge state. Since each may be held by one of four players, the size of the instance space is 5.36E+28 as discussed earlier. With a suit-oriented representation, there are only 13 cards to represent among the four players. For example, the hand most evenly distributed among suits is 4-3-3-3. The number of possible deals for this distribution are only 1.67E+10, which is calculated as $\binom{13}{4}\binom{13}{3}\binom{13}{3}\binom{13}{3}*4$. Thus, domain knowledge effectively streamlines the representation and limits the search.

More specifically, each example is represented as a state-action pair <*s*, *a*>. The state consists of *independent* attributes that represent the context in which the player (declarer or dummy) is to play a card, and the action is the *dependent* attribute that represents the play executed in this context. **Figure 4-2** shows an example of the context in which North is to play a card. The actual card played by North (from A 6 5) is the *action*.

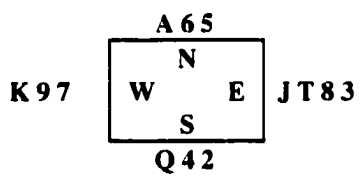


Figure 4-2. A bridge example with a single suit description.

The attributes for a state *s* include bookkeeping information and distribution information. As summarized in **Table 4-1**, the *bookkeeping information* keeps track of the contest in progress, with information such as the trick number and play number in a contest, the leading player of the trick, the current player, and the card(s) played thus far in a trick. More specifically, the bookkeeping information includes:

	Name	Description	Type	Domain
Bookkeeping Information	Trick number	i^{th} trick in a contest	Integer	[1, 13]
	Play number	j^{th} play in a contest	Integer	[1, 26]
	Leader	The leader of a trick	Discrete	Ω = {West, North, East, South}
	Current player	The current player	Discrete	Ω = {West, North, East, South}
	Current trick	Cards played in a trick	String	{$w \in \Sigma^4$, where Σ = [A, K, Q, J, T, 9, x, -]}
Distribution Information	West hand	Cards played by West	String	{$w \in \Sigma^{13}$, where Σ = [A, K, Q, J, T, 9, x, -]}
	North hand	Cards held by North	String	{$w \in \Sigma^{13}$, where Σ = [A, K, Q, J, T, 9, x, -]}
	East hand	Cards played by East	String	{$w \in \Sigma^{13}$, where Σ = [A, K, Q, J, T, 9, x, -]}
	South hand	Cards held by South	String	{$w \in \Sigma^{13}$, where Σ = [A, K, Q, J, T, 9, x, -]}
Action	Card played	Target attribute	Discrete	{A, K, Q, J, T, 9, x}

Table 4-1. Attributes for representing bridge states. The bookkeeping and distribution information describes the context in which a player is to play a card. The action attribute represents such card played.

- *trick number* indicates the i^{th} *trick* in a contest. It is an integer value i in [1, 13].

- *play number* indicates the j^{th} *play* in a contest. It an integer value j in [1, 26]. Note that this is the decision number by the declarer, and is less than the number of total cards played in the hand.

- *leader of a trick* identifies the player who plays the first card on a trick. It is a discrete value in Ω = {West, North, East, South}.

- *current player* is a discrete value in Ω that corresponds to the player who is to play the next card.

- *current trick* is a string of four elements, one for each player. The entries in this string represent the card played by West, North, East, and South, respectively, always in that order. Each element is from the set Σ = {A, K, Q, J, T, 9, x, -}, where "x" represents any card that is smaller than a 9 because smaller cards are indistinguishable with respect to the result. A card played from a different suit, because that player had no card in the suit led, is also represented as an "x." The symbol "-" means that the player has

not yet played in this trick. As an example, the string "A-xx" means that West played an ace, North has not yet played, and East and South played low cards (x's).

The *distribution information* consists of the cards *held* by the declarer and the dummy, and those *played* by the two opponents (West and East). A *hand* includes a single *suit*, which is represented as a string of up to 7 possible ranks from Σ, and '-' means an empty hand. As an example, **Figure 4-2** shows that North holds A 6 5 and South holds Q 4 2. If the representation included K 9 7 for West and J T 8 3 for East, however, that would mean that they had already played those cards before the current state. Thus, the representation would record '-' for both East and West before any card in the suit is played. Finally, the action (a) is the dependent attribute, whose value serves as the teacher during training and as the judge during testing. The action is a discrete value from $\Sigma - \{-\}$, that is, every element in Σ except '-'.

Although the difference function for the numeric and the discrete attributes, shown in [3] in Section 3.3.1, is well defined, the difference for a string attribute is not. A simple string difference may be calculated by treating the entire string as a discrete symbol and performing an exact match, like that for a discrete attribute. This approach, however, imposes an overly strict constraint. The more relaxed approach employed here adopts the n-gram approach to measure the string difference. In this approach, only a sub-string of n symbols in two strings are compared at a time, and the result of the comparison is used to compute the difference. For example, to find the difference between the string x = "bank" and the string y = "bunk" with n = 3, first, each string generates a set of n-grams, sub-strings of length n = 3 with one non-overlapping symbol in the adjacent n-grams as shown in **Table 4-2**. Both strings generate six n-grams. To calculate the difference between x and y, first, the cardinality of the intersection of the n-grams between x and y is counted. In this case, the cardinality is three because there are three exact

matches (- - b, n k -, k - -). Then, the count is scaled between 0 and 1 based on the overall length of the longer of the two strings as follows:

$$difference(x, y) = 1 - \frac{\#\text{of identical } n\text{-grams between } x \text{ and } y}{\#\text{of total } n\text{-grams of the longer of the two strings}}$$

If no identical n-grams exist, the difference is 1; if all n-grams are identical, the difference is 0. Also, the larger the number of identical n-grams between x and y, the smaller the difference. In this example, the difference value is $\frac{3}{6}$, *or* 0.5. The n-gram approach accounts for more details in a string, and provides a smoother difference computation.

String	*n*-grams
bank	- - b, - b a, b a n, a n k, n k -, k - -
bunk	- - b, - b u, b u n, u n k, n k -, k - -

Table 4-2. Two strings, "bank" and "bunk", with their corresponding n-grams, sub-strings of length n, for $n = 3$.

With this representation, **Table 4-3** shows a set of five instances representing the bridge sequence in **Figure 4-3**. In the first instance, the trick number is 2 (column 1) and the play number is 5 (column 2). The leader of the trick is North (column 3) and the current player is South (column 4). In this trick, three low cards have already been played (column 5). On tricks prior to this one, West played another low card (column 6), North now holds the ace and T (column 7), East has already played the king and a low card (column 8), and South holds the queen and three low cards (column 9). Finally, play is the dependent attribute that represents the action taken, x (column 10). Since there are 13 tricks in a contest and two instances, one for North and one for South, are recorded for each trick, there are 26 instances in a contest.

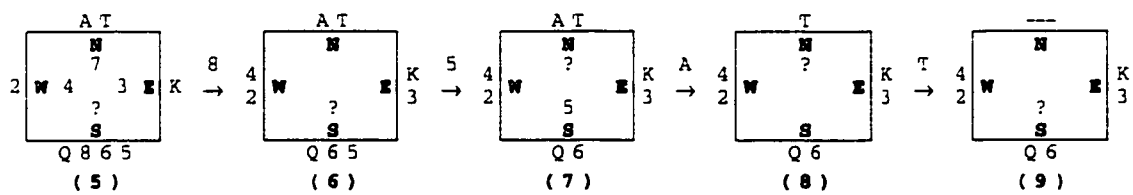


Figure 4-3. An example of a bridge sequence from state (5) to the current state (9).

State									Action
Trick number	**Play number**	**Lead player**	**Current Player**	**Trick**	**West**	**North**	**East**	**South**	**Play**
2	5	N	S	xxx-	x	AT	Kx	Qxxx	x
3	6	S	S	----	xx	AT	Kx	Qxx	x
3	7	S	N	x--x	xx	AT	Kx	Qx	A
4	8	N	N	----	xx	T	Kx	Qx	T
4	9	N	S	-Tx-	xx	---	Kx	Qx	?

Table 4-3. A representation of the bridge sequence in **Figure 4-3**.

4.1.3 Knowledge Discovery

Knowledge discovery is generally viewed as a process that includes a set of distinct steps (Fayyad et al. 1996). These steps separate a complex knowledge discovery problem into more manageable sub-tasks. The four major knowledge discovery steps are data collection, preprocessing, data mining, and post-processing. *Data collection* involves gathering relevant examples. *Preprocessing* transforms examples into a format that can be manipulated by an algorithm. *Data mining* performs searches through the concept space defined by the examples for meaningful concepts. *Post-processing* reformats the discovered results for comprehension.

Data collection involves gathering relevant examples as the source of input to data mining. In bridge, an important data source are the bridge hands that appear in daily newspaper bridge columns. Almost all daily newspapers in the United States carry a bridge column. The deals in these bridge columns are typically instructive and challenging. Therefore, they are suitable for data mining. Using a newspaper bridge column as the main source of my data collection, I collected 200 3NT bridge deals from the San Francisco Chronicle over a three-year period, because, as mentioned earlier, this research focuses only on 3NT.

Preprocessing transforms raw data, in this case the printed pages of bridge hands as shown in **Figure 4-4**, into a format that can be manipulated by SIBL for data mining.

The input to the preprocessing step was the 200 3NT deals. The output of the preprocessing step is a database of examples in the form of state-action pairs like those in **Table 4-3**. Given the bridge hands collected, the next task was to transform them into computer readable form. Fortunately, there are a number of computer programs available on the World Wide Web to facilitate such automatic data generation. The one written by Matt Ginsberg at Oregon University, called GIB (Ginsberg 1999), is particularly useful because it makes the source code available. This permitted me to build an interactive training and testing program by integrating GIB with the proposed learning algorithms.

GIB was used to play each of the 200 newspaper deals and form the initial database. Two copies of GIB were used – one played declarer and dummy, and the other played the two opponents – to generate the examples. Since the goal is to model declarer and dummy plays, two examples were collected from each trick, one for declarer and one for dummy. Each example included the bookkeeping information, distribution information, and action as discussed earlier, with multiple states included to facilitate context selection in the SIBL algorithms. An example of the output is shown in **Table 4-3**. Again, each hand produces a series of 13 tricks out of which 26 plays (examples) are generated for two players – 13 for the declarer and 13 for the dummy. The 200 collected 3NT deals produced 5,200 examples.

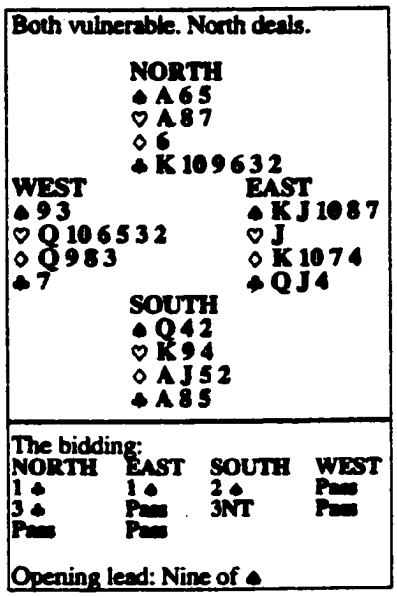


Figure 4-4. A newspaper bridge example.

For each experiment, the 5,200 examples were randomly divided into a training set with 90% and a test set with 10% of the total examples. That is, 4,680 (90%) were used for training and 520 (10%) were used for testing. To measure the performance accurately, cross validation was applied to each experiment to produce an averaged performance (Kibler and Langley 1988). The examples were partitioned into 10 subsets. On a *run*, one subset was selected for testing with all the others used for training. Ten runs, each with a different test set, constitute a *sub-experiment*. In addition, the examples were partitioned so that all examples from the same deal fell in the same subset. This prevented training and testing with examples from the same deal.

A known caveat for an incremental learning method, such as IBL, is that the experimental results may depend on the order of the examples in the training set. That is, training sets with the same examples in a different order will generate different performance results. For this reason, each cross validation sub-experiment was replicated ten times, randomizing the order of the examples. This produced ten variations on each cross

validation sub-experiment, training 10 times on differently ordered but otherwise equivalent sets of examples.

In sum, a complete experiment consisted of 100 runs, 10 cross validation sub-experiments times 10 variations. The results of an experiment were averaged over the 100 runs. As shown in **Table 4-4**, each run involves 4,680 training examples and 520 testing examples, with a total of 520,000 references in a given experiment.

	#Examples per set	#Sub-experiments	#Total examples	#Variations	#Total references
Total	5,200	10	52,000	10	520,000
Train	4,680		46,800		468,000
Test	520		5,200		52,000

Table 4-4. Total number of training and test examples and their uses with a 10-fold cross validation, each with 10 runs.

Data mining takes a database of examples and applies learning algorithms to produce a set of predictive models. Performance may be evaluated by comparing the solutions recommended by these models with the solution generated from an expert source. The number of tricks captured or the number of deals won characterizes the performance.

As discussed in Chapter 3, an incremental learning paradigm, such as IBL, is suitable for sequential concept discovery. In particular, IB4 is used as the baseline algorithm that can minimize storage requirements and can reduce the effect of instance and attribute noise (Aha 1992). The proposed algorithms in SIBL extend IB4 with the ability to selectively choose a context of various lengths in a given state.

The data mining step for this research is divided into two steps: *exploration* and *development*. In the exploration step, IB4 is used to identify weaknesses associated with using a classification type algorithm for learning synthesis type problems. Then, SIBL is developed to address those weaknesses.

Table 4-5 shows a list of experiments performed. IB4 (denoted here as IB4-1) is used as the baseline to show that it cannot effectively distinguish between two identical situations derived from two different sequential paths. (See Chapter 3 for further details.) To explore the effect of a multi-state context, IB4 is adopted to use input examples with n-state descriptions, for n = 2, 3, 4, 5 (i.e., IB4-2, IB4-3, IB4-4, and IB4-5). These experiments demonstrate the usefulness of additional context. They also reveal that the context of a relevant candidate for an input example does not have to be fixed at a certain number of states. To use multiple states as the relevant context more flexibly, SIBL algorithms were developed to use variable context sizes. Two approaches are possible: quantitative and qualitative. The quantitative approach (SIBL-Vote) implements the *FindMajority* function (in Chapter 3), a voting scheme that polls among examples with various context sizes. The qualitative approach (SIBL-SSM) implements the *FindSimilarity* function (in Chapter 3), which is based on a set of sequential similarity metrics to select an example with the most relevant context size. (See Chapter 3 for a detailed discussion.)

IB4-n	Fixed size multi-state context, n = 1, 2, 3, 4, 5
SIBL-Vote	Quantity approach to context selection
SIBL-SSM	Quality approach to context selection

Table 4-5. A list of experiments based on instance-based learning approach.

Performance measurement quantifies the outcome of applying a given method. One way to measure SIBL is to determine the accuracy of a given model. In classification, the typical measurement is the ratio of correct predictions to total predictions. In sequential domains, a solution consists of a sequence of actions that leads to a goal state, as discussed in Chapter 2. If the sequence of actions is viewed as a series of classification tasks, the measurement may be defined as the ratio of the number of correct predictions for each individual to total predictions in a sequence.

In bridge, for example, the most general performance measurement is a classification-style measurement called *play ratio*, the number of correct plays divided by the total number of plays made, where a correct play means a play that is identical to a teacher value, or is equivalent because it is adjacent to a teacher value. A more sequential measurement in bridge may be based on a trick, to which all four players contribute a card. That is, performance result may be measured by the *trick ratio*, the number of tricks captured divided by the total number of tricks contracted. Since the number of tricks captured determines whether a deal is won, accuracy may also be measured by the *deal ratio*, the number of deals won divided by the number of total deals tested.

Although measuring accuracy in terms of domain-specific characteristics, such as trick and deal in bridge, is more precise, it often requires customization to fit the specific domain. Such a customization effort may overshadow the intrinsic contribution of the learning algorithm under investigation. To maintain generality, classification based measurement is used in this research. That is, the play ratio described above will be the primary measurement.

To determine the significance of two experiments, a *nonparametric statistical hypothesis test* may be used to analyze the results of two experiments. Nonparametric analysis is used because the data is compared point to point, rather than across an average (Hogg and Tanis 1997). A *null hypothesis* H_0 is a belief one wishes to reject. On the other hand, an *alternative hypothesis* H_1 is a belief one wishes to accept. For example, if H_0 is the belief that "IB4 performs at least as well as SIBL", then by rejecting H_0, one may conclude that SIBL performs better than IB4. Alternatively, if H_1 is the belief that "IB4 performs worse than SIBL", then by accepting H_1 one may also conclude that SIBL performs better than IB4. Such a hypothesis test may be viewed as a test of the value

a = *accuracy*(IB4) – *accuracy*(SIBL), the difference in accuracy between IB4 and SIBL. That is, H_0: $a = 0$ is the null hypothesis, and H_1: $a < 0$ is the alternative hypothesis.

Let X denote the difference in accuracy between IB4 and SIBL from experimental results. If H_0: $a = 0$ is true, $P(X \geq 0; H_0) = 0.5$. On the other hand, if H_1: $a < 0$ is true, $P(X \geq 0; H_0) < 0.5$. That is, if one performs n experiments with both IB4 and with SIBL, one would expect about $n/2$ of the values of a are greater than or equal to 0 if H_0 is true. However, if H_1 is true, one would expect less than $n/2$ of the values of a are greater or equal to 0. Let Y denote the number of values of a that are greater than or equal to 0, with the probability of success given by $p = P(X \geq 0)$. If H_0 is true, $p = ½$ and Y is $b(n, ½)$, the binomial distribution of n trials with probability of success of ½; whereas if H_1 is true then Y is $b(n, p)$. H_0 is rejected and H_1 is accepted if and only if the observed value y of Y is sufficiently small. Let the value for y be expressed as critical region $C = \{y: y \leq c\}$, where c is a small number, then the significance, α, is defined as

$$\alpha = P(C \leq c) = \sum_{k=0}^{c} \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

where n is the sample size, c is the number that defines the critical region, and p is the probability of success. As a specific example, let X be the set

$$\{-1.3, -0.2, -1.5, -1.8, 0.9, -2.4, -2.7, 1.1, -3.1, -0.6\}$$

Since there are $y = 2$ differences in accuracy are greater or equal than 0, H_0: $X = 0$ is rejected and H_1 is accepted at $\alpha = .05$, or the .95 (1 - α) significance level. Such an outcome is said to be *statistically significant*. Any two experiments are said to be statistically significant if their results reach the .95 significance level.

Post-processing is the step immediately following the data mining step just described. It may be used to improve performance by transforming a model from one form,

say a decision tree, into another, say rules (Tsumoto 1996). It may also be used to improve comprehension by extracting meaningful concepts from a model (Liu 1998). I used post-processing for the latter purpose. As a side effect, the process may be used to pinpoint weaknesses and suggest additional improvements.

Specifically, sequential concepts in sequential domains are purposeful. Human experts rely on these distinct concepts to solve problems. One purpose here is to discover such sequential concepts and compare them with those that mimic human experts in solving similar sequential problems. To do so, the actions recommended by the algorithms are recorded during testing. This data is inspected manually to gather information about the quantity and nature of the learned sequential concepts. These sequential concepts are also compared to the behaviors that mimic a human expert. Any deviation may either become new knowledge, if they are useful, or sources for improvement if they are ineffective.

All experiments reported here were run on a 450 MHz Pentium III processor with 256 Megabytes of memory. The numeric precision for the results were calculated to the nearest hundredth, unless otherwise indicated. Experiments use parameters that deviate from those described in this section are discussed in Section 4.3.

4.2 Experimental Results

The experimental results are described in terms of problem solving behavior, its improvement, and its associated computation cost. As expected, the better the performance, the higher the computation cost. Problem solving *behavior* describes the results when additional contexts have been included in an example, while *improvement* depicts ways the improvement can be made. *Computation cost* was measured in terms of time and space.

Problem solving behavior, as measured by play ratio, compares the effect of different context sizes in an example. IB4-1 has the smallest context, with only the current state in an example, and it performs the worst. IB4-5 has the largest context, with up to five states in an example, and performs the best among the IB4-*n* algorithms. Nonetheless, IB4-5 suffers from context noise, because not every synthesis problem is determined by a fixed-size context.

To improve IB4, I tried two approaches. The first, *context polling*, employs a quantitative method that polls among examples with different context sizes. It generates a set of candidates and forms a set of potential actions, followed by a majority vote to produce the solution that is the most popular action. The second approach, *context selection*, uses a qualitative method that selectively chooses an example of a given context size. It also generates a set of candidates. Rather than polling over the set of candidates, context selection uses a set of sequential similarity metrics (SSM) to select the most similar candidate such that its action is recommended. Again, the measurement for improvement is based on the play ratio. Both context polling and context selection outperforms IB4-5, while context selection is slightly better than context polling.

4.2.1 IB4-*n*

IB4 forms the foundation of the following experiments, which vary in the number of states used in an example. (Recall that IB4-*n* denotes an experiment that uses IB4 with *n* states in an example, $n = \{1, 2, 3, 4, 5\}$.) Throughout, an example is viewed as state and action pair, where the state is a set of independent variables and the action is the dependent variable. The independent variables represent a context in which either the declarer or the dummy is to play a card, and the dependent variable is the card played.

As discussed in Section 4.1.2, the representation of an example is attribute-value based. In IB4-1, each example is represented by a set of nine attributes of three different types – integer, discrete, and string. (See **Table 4-1**.) The two integer attributes specify the i^{th} trick and the j^{th} play for a contest; the three discrete attributes are leader, current player, and the action (target attribute). The remaining five attributes are variable-length strings. One string represents the plays made by the four players in the current trick, and the other four represent the cards held (by declarer and dummy), and the cards previously played (by East and West) by the players.

In **Table 4-6**, for example, the *s*-instance is from second trick and the fifth play of the contest. North led the trick and South is to play. The current trick is xxx-, which means that West, North, and East all played low cards, and South has yet to play (-). The next four attributes are the cards previously played by West (x) and East (Kx), in this suit, and those held by North (AT) and South (Qxxx). Finally, the action indicates that South played a low card. With such a representation, 22% of the features are numeric, 22% are discrete, and 56% are strings. Among the seven possible actions in the 5,200 examples, 53% are x, play a low card.

Bookkeeping Information					Distribution Information				Action
Trick Number	Play number	Leader	Current player	Trick	West	North	East	South	Play
2	5	N	S	xxx-	x	AT	Kx	Qxxx	x

Table 4-6. An example of an *s*-instance with a single-state context.

In contrast, IB4-5 uses a set of 26 attributes in an example. The first nine and the last one are the same as those used in IB4-1. The other 16 attributes represent four prior states, each with four attributes, one for each player.

Table 4-7 shows an example for 5 states. The attributes with labels $West_i$, $North_i$, $East_i$, and $South_i$ represent the i^{th} state in the example for $i = 1, 2, 3, 4, 5$, where $i = 5$ is the

most recent state and $i = 1$ is the least recent state. For example, $West_4$ = xx, $North_4$ = JT, $East_4$ = Kx, and $South_4$ = xxx precede $West_5$ = xx, $North_5$ = JT, and $East_5$ = Kx, $South_5$ = xx, respectively. Since distribution information represents the cards held by North and South, and those played by West and East, the closer a state is to the current state the fewer cards North and South hold, and the more cards West and East have played. For example, while $North_3$ changes from JTx to $North_4$ = JT, $East_3$ changes from K to $East_4$ = Kx. With the addition of these string attributes, the ratio among the three attribute types changes. In IB4-5, 8% are numeric features, 8% are discrete features, and 84% are strings.

Bookkeeping Information					Distribution Information ...			
Trick Number	Play Number	Leader	Current Player	Current Trick	$West_1$	$North_1$	$East_1$	$South_1$
2	5	S	N	x--6	-	JTx	K	Axxxx

$West_2$	$North_2$	$East_2$	$South_2$	$West_3$	$North_3$	$East_3$	$South_3$
x	JTx	K	xxxx	xx	JTx	K	xxx

								Action
$West_4$	$North_4$	$East_4$	$South_4$	$West_5$	$North_5$	$East_5$	$South_5$	Play
xx	JT	Kx	Xxx	xx	JT	Kx	xx	J

Table 4-7. An example of an *s*-instance with a multiple-state context.

Observe that the number of training examples is the same for all experiments, even though they use different numbers of attributes. The number of attributes for IB4-5 of 26, for example, is much larger than that for IB4-1 with only 10. As a result, the instance space (the number of attributes and the number of values each attribute can hold) for IB4-5 is much larger than that for IB4-1. If all attributes were binary, the instance space for IB4-5 could be 2^{26} while that for IB4-1 is only 2^{10}. According to PAC learning theory (Valiant 1984), IB4-5 needs many more training examples than IB4-1 to have a comparable performance. This is because the theoretical relationship between the number of training examples m and the number of errors ε is defined as

$$m \geq \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + \ln|H|\right)$$

Here, H is the instance (hypothesis) space, such that an algorithm having an instance space H has error less than ε with a probability of 1 - δ. Therefore, if δ is made constant, it increases H without increasing m will increase the number of errors. To make the comparison among the IB4-*n* algorithms more meaningful, in addition to the existing attributes defined for each IB4-*n*, attributes with unknown values were added, so that training examples for IB4-*n* all have the same 26 attributes. Taking the approach used in IB4, an attribute with an unknown value results in a maximum difference for the attribute.

Figure 4-5 compares the IB4-*n* algorithms on their ratio of correct action selection, measured on a scale from 0 to 1 along the y-axis. For example, IB4-1 selected on average 64 out of 100 correctly, while IB4-5 averaged at 79. Overall, a gain of 23% is achieved from using one state in the context of an example (IB4-1) to five states in the context of an example (IB4-5).

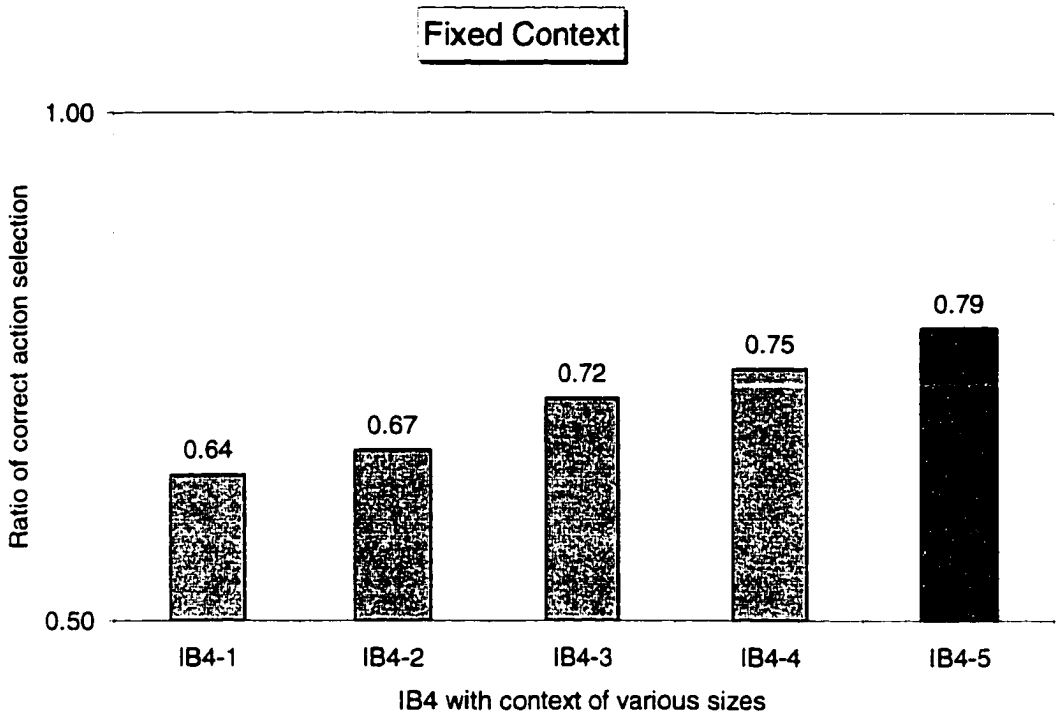


Figure 4-5. A performance comparison of IB4-*n* that is based on the ratio of correct action selection. A gain of 23% is achieved from using one state in the context of an example (IB4-1) to five states in the context of an example (IB4-5).

The performance advantage of IB4-5 over IB4-1 is due to the phenomenon of class noise. That is, when two examples are similar but are associated with different actions, a context with only the current state may not be sufficient to determine the relevance among a set of candidate examples. In particular, each training example used by IB4-1 is a snapshot of a single state in a sequence. Such an example does not have sufficient context to disambiguate itself from similar examples because, as illustrated in Chapter 2, the same example derived from different contexts may require a different action. For example, two examples are *nearly identical* if the similarity between the two reaches the similarity threshold, say .9 on a 0 to 1 scale. In other words, examples A and B are nearly identical if $sim(A, B) > 0.90$, where *sim* is the similarity function that computes the difference between two examples. When two nearly identical examples are associated with two different actions, class noise occurs. Without additional context, IB4-1

randomly assigns a class. To illustrate the point, one may designate an oracle that is able to determine the correct classification under such circumstances. That is, the new approach assumes the oracle always knows if two examples are truly identical or they are only similar in the absence of a larger context. In this case, the oracle will make perfect correct classifications among two nearly identical examples.

Furthermore, a general problem with a fixed-size context for IB4-*n* also suffers from context noise, where a fixed-size context mandates that each state should be affected by exactly the same context size. IB4-*n* uses examples that always have the same number of states and attributes. When the number of relevant states differs from one situation to another, context noise occurs. In other words, context noise arises when there is *excessive* or *insufficient* context for representing an example.

In a sequential domain like the game of bridge, the execution of similar tactics may depend on different context sizes (different number of historical states). The execution of a finesse, for example, may rely on a varying number of historical states given different situations (e.g., simple finesse[3] verses double finesse[4]). This underscores the need to use variable context sizes in examples to represent different situations. IB4-5 suffers from context noise in this problem. In the next two sub-sections, I show how context can be used selectively to improve action selection accuracy.

4.2.2 SIBL-Vote

SIBL-Vote extends IB4 with the *FindMajority* function to quantitatively select a solution. It selects an action from a set of candidates with the majority vote. It attempts to minimize the context noise by creating a set of candidates with different context sizes and

[3] A simple finesse is one against only one missing card.
[4] A double finesse is one against two missing cards.

chooses the most popular action among the candidates. (See Chapters 2 and 3 for more details.) As in IB4, the majority vote is used to recommend a solution that is the majority among the most relevant candidates. SIBL-Vote selects the solution that is the majority among the set of potential candidates of various context sizes. The intuition here is that the most popular solution among the candidates of various context sizes has the highest probability of correctness. In other words, SIBL-Vote assumes that the class majority c among the candidates of various context sizes suggests the correct solution.

SIBL-Vote uses the same example formats shown in **Table 4-6** and **Table 4-7**. Rather than looking at a single, fixed length example, SIBL-Vote works with a set of w examples of various context sizes each has i states in the context, $1 \leq i \leq w$, and $w = 5$ is the maximum number of states examined. More specifically, there are w data sets, one for each context size. All data sets have the same static attributes: *state*, *step*, *lead*, *turn*, and *trick*. The other attributes ($west_i$, $north_i$, $east_i$ and $south_i$) are additional attributes, and vary based upon the number of states represented, where i is the i^{th} state of a given situation. As before, the target attribute (action) is the *play* made in the situation.

As described in Chapter 3, the process for comparing two examples in SIBL-Vote involves two steps: candidate selection and action selection. Candidate selection selects a candidate from examples of the same size context, the way IB4-*n* would. This produces w candidates. Action selection selects the recommended action from the set of w candidates by a simple majority of their respective target attribute value.

At one extreme, the set of candidate solutions could be unanimous in which case all solutions are identical and there is no competing solution. In this case, the single solution is used. At the other extreme, all candidate solutions are distinct, so there are as many solutions as there are candidates. In this case, random selection is used. Otherwise, a simple majority is used to select a solution among a set of competing solutions.

As shown in **Table 4-8**, as additional states are included in the context of an example, the number of features increases from 9 with one state up to 25 with five states. This changes the percentage of numeric attributes ranging from 22% with one state down to 8% with five states, and the percentage of symbolic (discrete and string) attributes ranging from 78% with one state up to 92% with five states. All data sets have the 7 target attribute values and 53% majority action (play a low card). In this experiment, the size for *w* is 5.

#w	#examples	#attribute	%numeric	%symbolic	#action	%majority
1	5,200	9	22%	78%	7	53
2		13	15%	85%		
3		17	12%	88%		
4		21	10%	90%		
5		25	8%	92%		

Table 4-8. Summary of input examples used for SIBL-Vote.

The performance for SIBL-Vote is compared with that of IB4-*n* in **Figure 4-6**. The correct action ratio of .81 is slightly better than that of IB4-5 at .79. The 2% improvement is due combining the efforts of the IB4-*n* algorithms.

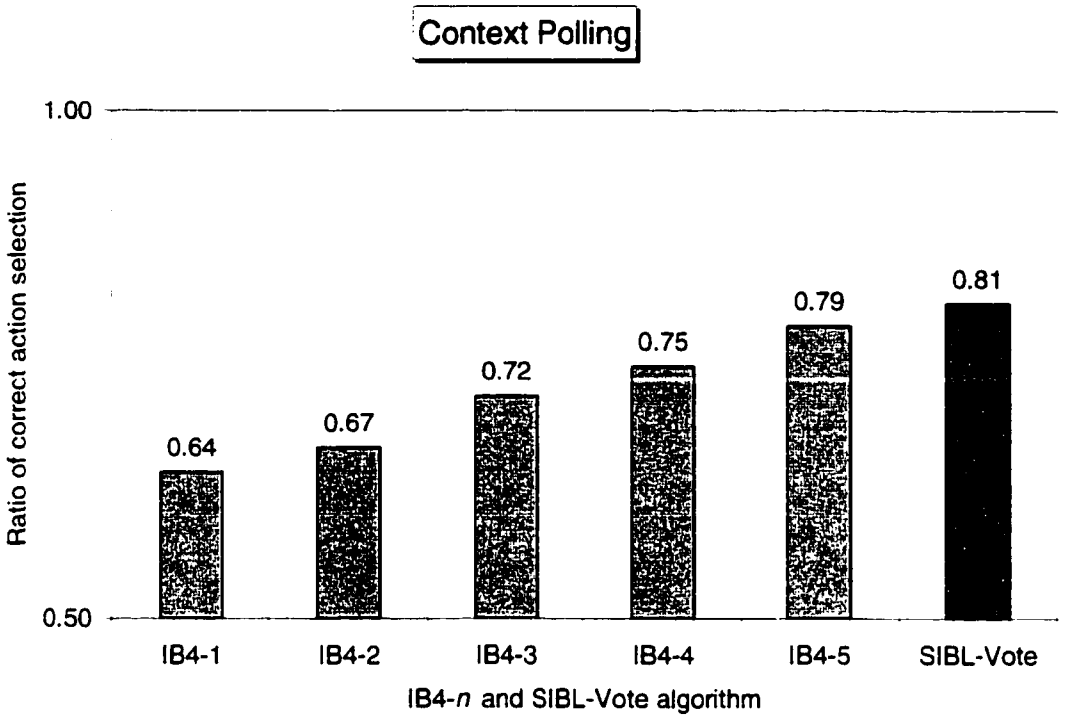


Figure 4-6. A performance comparison of SIBL-Vote with IB4-*n* in terms of correct action selection. With SIBL-Vote, a gain of 2% is achieved over IB4-5 by using a majority vote among candidates of various context sizes.

A breakdown of the ways decisions were made with SIBL-Vote is shown in **Figure 4-7**. In a total of 520 test instances, the candidates of various context sizes agree 40% of the time (unanimous) with a correct action selection ratio of .89. Majority vote makes up 50% of the action selection (majority) with a correct ratio of .75. The remaining 10% of action selections are done randomly with a correct ratio of .74. Although random selection seems to perform as well as majority vote by its correct action selection ratio, the two are tested with examples from different populations. As show in a lesion study described in Section 4.2.3, the correct action selection ratio for random selection is .65 when the same example population is used.

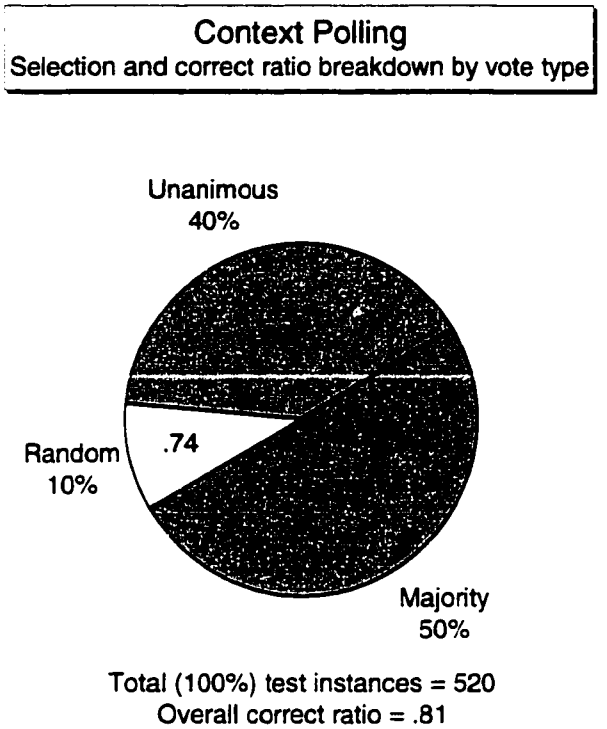


Figure 4-7. SIBL-Vote action selection breakdown by decision type. In 520 test instances, 50% were majority votes, 40% were unanimous votes, and 10% were random selections. The correct selection ratios are .75, .89, and .74, respectively.

In SIBL-Vote, the votes derived from contexts of various sizes are considered equally important regardless of the context quality of each vote. In extreme cases, the majority consists of the context with poorest quality. Consider the hypothetical example in **Table 4-9**, where each candidate a^i has a context of i states, $1 \leq i \leq 5$. The candidates have a similarity ranging from .14 to .35, where higher values indicate greater similarity. An action (x, y, or z) is associated with each candidate as the value of the target attribute. As a result, using majority vote, the recommendation from the set of candidates is x because it is the simple majority. The problem with SIBL-Vote is that examples with different context sizes are considered equally important, each with exactly one vote. When variable context size examples have different importance, *candidate noise* occurs.

On the other hand, focusing on the quality of the context, such as the similarity, may be more accurate. As shown in **Table 4-9**, the most similar candidate is a^4 with a similarity of .35, and its attribute value is y, which is the same as that of the actual attribute. Based on this observation, the qualitative approach, described in the next section, may be used to select the most relevant example by comparing their qualities (similarities) with respect to the target example.

Candidate	Similarity	Action	Vote	Actual
a^1	.17	x		
a^2	.26	x		
a^3	.14	x	x	y
a^4	.35	y		
a^5	.21	z		

Table 4-9. A hypothetical example in which the majority action (x) derives from the candidates (a^1, a^2, a^3) of lower quality.

4.2.3 SIBL-SSM

SIBL-SSM extends IB4 with the *FindMostSimilar* function to qualitatively select a solution. It selects a solution *qualitatively* from a set of examples with variable context sizes. It attempts to overcome candidate noise by selecting an action with a series of sequential similarity metrics – distance, convergence, consistency and recency. Starting from the distance metric, each metric is built on top of another by refining the one before it. Distance is derived from attribute differences. Convergence is derived from distance by considering changes in distance from one end of a sequence (states) to the other. Consistency is derived from convergence by noting the length of consecutive convergence. Recency is derived from consistency by noting the last point at which a consecutive convergence occurred. (See Chapter 3 for more detailed description of the development of these metrics.)

SIBL-SSM uses the same example format as that for SIBL-Vote discussed in the last section. Although candidate selection is the same as that for SIBL-Vote, action selection is different. Rather than finding the majority action among the w candidates of various context sizes, SIBL-SSM compares the candidates in terms of sequential similarity metrics. As with SIBL-Vote, action selection is trivial when the candidates are unanimous. The metrics are used one after the other until a metric is found that distinguishes the candidates. If no metric can distinguish the candidates, random selection is used.

The performance of SIBL-SSM is compared with those of the IB4-n algorithms and SIBL-Vote in **Figure 4-8**. The correct action ratio of .83 is slightly better than IB4-n and SIBL-Vote. The 2% improvement over SIBL-Vote is due to the use of sequential similarity metrics that focuses on the quality of a candidate rather than on the simple majority among the candidates.

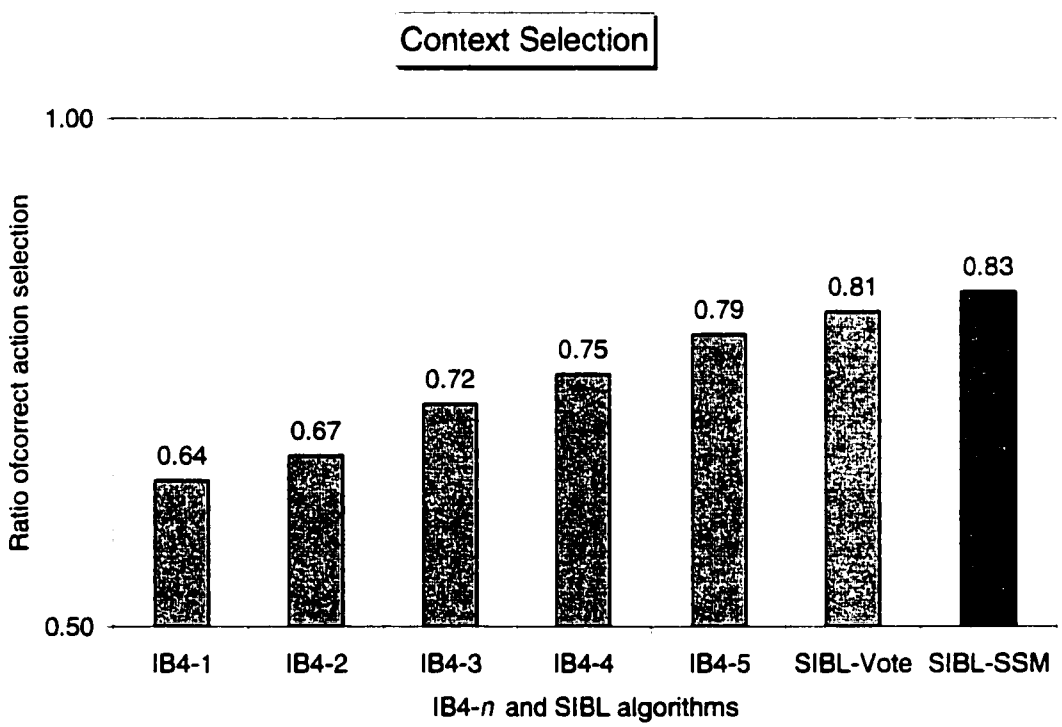


Figure 4-8. A performance comparison of SIBL-SSM with SIBL-Vote and IB4-n in terms of correct action selection. With SIBL-SSM, a gain of 2% is achieved over SIBL-Vote by using a set of sequential similarity metrics.

A breakdown of the ways decisions were made and their correct action selection ratio is shown in **Figure 4-8**. In a total of 520 test instances, the candidates of various context sizes still agree 40% of the time (unanimous) with a correct action selection ratio of .89. The distance metric makes 46% of the decisions with a correct ratio of .78. The convergence metric makes of 11% of the decisions with a correct ratio of .83. The consistency and recency metrics make 1% of the decisions each, with correct ratios of .86 and .67, respectively. The remaining 1% of action selections are done randomly with a correct ratio of .67.

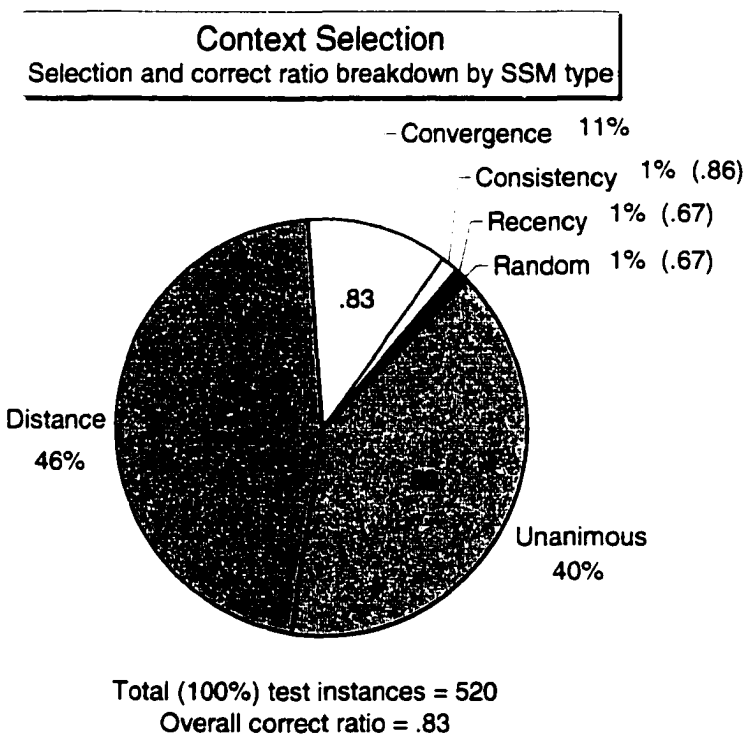


Figure 4-9. SIBL-SSM action selection breakdown by SSM type. In a total of 520 test instances, 46% selects by distance, 40% are unanimous, 11% select by convergence, and selections by consistency, recency and random selection are each 1%. The correct selection ratios are .78, .89, .83, .86, .67 and .67, respectively.

From **Figure 4-9**, it may seem that the recency metric is useless because it has the same correct ratio as random selection. Also, the order in which the sequential similarity metrics are applied is based on the rationale that simpler metric is used first. A lesion

study can be used to provide empirical explanation for the two observations. A *lesion study* is one where components of an algorithm are individually disabled to determine their contribution to the full algorithm's performance. (Kibler and Langley 1988). Here, two sets of experiment are performed on SSM. The first set tests combined performance within the SSM, and the second set tests the individual components (metrics) of SSM.

In the *combined* experiments, all four metrics are applied in a sequence from the most simple (distance) to the most elaborate one (recency) in an effort to mitigate numeric ambiguity discussed earlier. In the subsequent experiments, the most elaborate metric is removed to test the contribution of that metric. **Table 4-10** shows the five experiments in this combined experiments.

	Metrics included	Metric tested
1	Distance, Convergence, Consistency, Recency, Random	N/A
2	Distance, Convergence, Consistency, Random	Recency
3	Distance, Convergence, Random	Consistency
4	Distance, Random	Convergence
5	Random	Distance

Table 4-10. Five experiments performed in the lesion study. A metric is tested by excluding the metric from the experiment.

Table 4-11 shows the combined experiments' performance results both with or without a particular metric. The change in percent represents the contribution of a given metric. For example, when the distance metric is included, it contributed a performance improvement of 26.15% from .65 to .82, or (.82 – .65) / .65. As shown in the following table, except for the distance metric, the contributions are unimpressive. In fact, there is no improvement from consistency or recency. This is due to the small (1%) population of input examples used for the two metrics. As a result, the combined experiment does not show the full impact of each metric.

	Combined experiment		
	With	**Without**	**Change**
Distance	.82	.65	+26.15%
Convergence	.83	.82	+1.22%
Consistency	.83	.83	0.00%
Recency	.83	.83	0.00%

Table 4-11. Performance results both with and without a particular metric. The change is the improvement in terms of correct action selection ratio.

To overcome the limitations of the combined experiments, individual experiments are performed. An *individual* experiment uses only one metric for the entire data set, with random selection if the metric is numerically ambiguous. The purpose here is to show the full impact of each metric. **Table 4-12** shows the performance results of the individual experiments. As in the combined experiments, the correct action selection ratio of the distance metric at .82 represents a 26.15% improvement over random selection at .65. Similarly, the convergence metric improved 24.62%, the consistency and recency metric both improved 23.08%.

	Individual experiment		
	With	**Random**	**Change**
Distance	.82	.65	26.15%
Convergence	.81		24.62%
Consistency	.80		23.08%
Recency	.80		23.08%

Table 4-12. Performance results both with and without a particular metric in an individual experiment where only one sequential similarity metric is used. The change is the improvement in terms of correct action selection ratio.

The correct action ratios discussed so far are the absolute performance results. They represent a raw measurement for each learning strategy. In other words, compared to a behavior that played all 520 plays perfectly, SIBL-SSM, for example, played 431 (i.e., .83) of them correctly. In a relative measurement, the performance of SIBL-SSM is close to GIB. This is because when GIB is used to generate the training examples, it did not make 13 out of the 200 deals with an absolute accuracy of .94. Thus, compared to

GIB, the absolute correct action ratio for SIBL-SSM at .83 is about .89 relatively to GIB, that is, .83 / .94 = .89.

Note that the different goals of SIBL and GIB cause a performance disparity between the two. GIB is intended as a specific problem solver, while SIBL is intended to build a problem solver in a variety of domains. As a performance component, it seems disappointing that SIBL is outperformed by GIB. As a learning component, on the other hand, SIBL is more flexible than GIB in adopting a new domain. This is because GIB relies on a domain-specific problem solver to guide the search for a solution, whereas SIBL uses domain-independent representational knowledge to find a solution through learning. In particular, the knowledge used to represent a state in bridge is identical to that for the game of hearts, for example. Also, the knowledge about the card equivalency in a play applies to the other card games as well. Therefore, while GIB has an edge in performance, SIBL may be applied to a wide variety of domains.

4.2.4 Intelligent Behavior

Another accomplishment of SIBL is the ability to discover sequential concepts that exhibit intelligent behavior for synthesis problems in bridge. A sequential concept is purposeful and exhibits intelligent behavior. It is often used by human experts to solve a recurring problem. Each sequential concept consists of actions or steps grouped together to reach a goal state.

There are many sequential concepts bridge experts use to assist them during a contest. I chose three well-known ones to demonstrate SIBL's ability to exhibit intelligent behavior: finesse, ducking, and holding out. These concepts are well described in the "Goren's New Bridge Complete," by Charles H. Goren.

Finesse

In the "Elementary Card Combination" section, one of the very first pieces of advise is (Goren 1985):

> "It is important to bear in mind that it is almost impossible to win a trick with a card that you lead."

Here, Goren is referring to leading a card that the opponent can cover. As an example, consider the situation in **Figure 4-10**. To win more than one trick, the lead should be from North repeatedly. The action forces East to play before South, who is able to cover any card East has to play. As a result, rather than winning only a single trick with the ace, South can win up to all three tricks. Such a tactic to win a maximum number of tricks is commonly known as *finesse*.

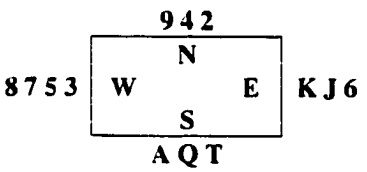


Figure 4-10. An opportunity to win more than one trick when leading from North.

The possibility for finesse may be detected as follows. Since finesse is a suit-oriented tactic, rather than a deal-oriented tactic, examine all four suits individually. For each trick t in a suit s, a finesse can occur when the leader is either North or South, the leader does not have control of the suit (i.e., missing a high card, A, K, Q, or J, in the suit), and the play results a win. In some cases, a win is deferred when the opponent covers with a missing high card.

This definition characterizes a variety of finesses, such as a double finesse and backward finesse[5]. **Figure 4-11**(a) is a finesse example found in the test set, and its play-

[5] A backward finesse is one taken in a manner opposite to what would ordinarily be standard procedure. For example, if dummy has ace-jack-nine and declarer has king-three-two, standard procedure would be to finesse the jack, hoping East does not hold the queen. It would be a backward finesse to lead the jack, hoping that East has the T and will play the queen over the jack.

ing sequence is shown in **Table 4-13**(a). Although East holds the king of hearts, in the third trick, North leads a 5 of hearts. When East plays a 2 of hearts rather than the king, South takes the trick with its queen. This is a finesse. As a result, with an ace-queen combination, South is able take two heart tricks (3 and 4). **Figure 4-11**(b) is another finesse example found in the test set, and its playing sequence is shown in **Table 4-13**(b). In the third trick, North leads the 6 of clubs, while East covers with the king and goes on to win the trick. This, however, enables South's queen of clubs to cash in later in the eighth trick. Over 100 runs, SIBL-SSM performed an average of 21 finesses out of a total of 39 finesses GIB took in the test examples.

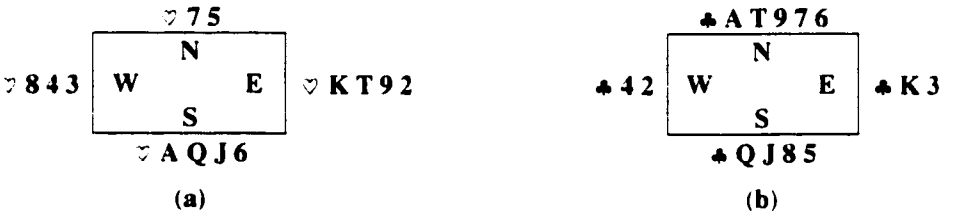


Figure 4-11. Two finesse examples.

	Lead	W,N,E,S	Win
1	W	♣Q, ♣7, ♣4, ♣A	S
2	S	◇2, ◇A, ◇4, ◇K	N
3	**N**	**♡4, ♡5, ♡2, ♡Q**	**S**
4	**S**	**♡3, ♡7, ♡T, ♡A**	**S**
5	S	♠2, ♠8, ♠K, ♠T	E
6	E	♣J, ♣K, ♣6, ♣5	N
7	N	♠A, ♠Q, ♠6, ♠3	W
8	W	♣T, ◇3, ◇7, ♣3	W
9	**W**	**♡8, ♠7, ♡K, ♡6**	**E**
10	E	♠5, ♠J, ♠9, ♣2	N
11	N	♣9, ◇T, ◇J, ◇Q	S
12	S	♣8, ◇8, ♡9, ♡J	S
13	S	♠4, ◇9, ◇6, ◇5	N

(a)

	Lead	W,N,E,S	Win
1	W	♡8, ♡7, ♡J, ♡6	E
2	E	◇3, ◇A, ◇4, ◇K	N
3	**N**	**♣2, ♣6, ♣K, ♣8**	**E**
4	E	♠K, ♠9, ♠6, ♠2	W
5	W	♡A, ♡Q, ♡2, ♡5	W
6	W	◇J, ◇5, ◇7, ◇Q	S
7	S	♣4, ♣9, ♣3, ♣5	N
8	**N**	**♠3, ♣T, ♠5, ♠Q**	**S**
9	S	♠7, ♠A, ♠4, ♠8	N
10	N	♡T, ♣A, ◇6, ♣J	N
11	N	♡9, ♠J, ♠Q, ◇2	E
12	E	♡4, ♣7, ◇T, ◇8	E
13	E	♡3, ♠T, ◇9, ♡K	E

(b)

Table 4-13. Two finesse playing sequences. In (a), the finesse occurs in steps 3, 4 and 9. In (b), the finesse occurs in steps 3 and 8.

Ducking

Ducking is the tactic that attempts to win the rest of the tricks in a suit by losing the first trick or two (Goren 1985). The purpose of ducking is to establish control in a suit that even low cards will win when the opponents no longer have cards in that suit. In **Figure 4-12**, for example, North has a five-card suit with two high cards. South has a three-card suit with one high card. If South leads the T and North ducks with the 6, East is given an opportunity to win with the king. Once the king has been played, North and South will capture the remaining four tricks in the suit.

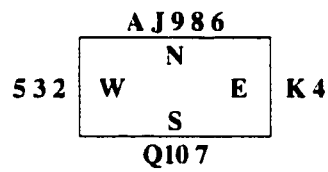


Figure 4-12. A ducking situation when North plays low in an effort to drive out king in East.

The opportunity to duck in a suit is defined as follows. Either North or South has at least five cards with the ace, and the other player has three low cards. Let h be the number of cars in the North or South hand with the longer suit. The first n tricks in the suit may be ducked to secure the remaining tricks, if n is less than h and $h - n$ is at least as large as the number of tricks that the declarer would have taken without ducking.

Figure 4-13 is a ducking example found in the test set, and the playing sequence is shown in **Table 4-14**. Here, when North led the 6 of diamonds in the third trick, East is given a chance to win with its king. When the diamond suit is played again in tricks 8, 10, 12 and 13, North is able to take them all since the suit was established by giving up a trick to east's king in the third trick. Over 100 runs, SIBL-SSM ducked on average 7 times out of a total of 13 times GIB ducked in the test examples.

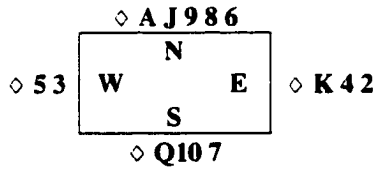


Figure 4-13. A ducking example.

	Lead	W,N,E,S	Win
1	W	♣7, ♣4, ♣9, ♣2	E
2	E	♡T, ♡K, ♡5, ♡8	N
3	N	◇3, ◇6, ◇K, ◇7	E
4	E	♣A, ♠2, ♣J, ♣K	W
5	W	♠4, ♠7, ♠9, ♠6	E
6	E	♡6, ♡7, ♡4, ♡Q	S
7	S	♠8, ♠T, ♠Q, ♠K	S
8	S	◇5, ◇J, ◇4, ◇T	N
9	N	♡3, ♡J, ♡9, ♡A	S
10	S	♣3, ◇A, ◇2, ◇Q	N
11	N	♣T, ♠A, ♠J, ♠3	N
12	N	♣8, ◇9, ♣5, ♣6	N
13	N	♡2, ◇8, ♠5, ♣Q	N

Table 4-14. A ducking playing sequence. The ducking takes place in steps 3, 8, 10, and 12.

Holdup

Another useful tactic is the holdup play. In this case, either North or South refuses to take a trick until an opponent fails to lead a damaging suit. This tactic is the opposite of ducking in the sense that rather than establishing a suit, North or South is preventing its opponents from establishing the suit. Consider the 3 No Trump example in **Figure 4-14**. If the opponents have the opportunity to lead hearts after the ace has been played, they can defeat the contract. Therefore, South should defer (holdup) playing its ace until the third round of hearts, when East will have none to lead and West has no other entries to the hand.

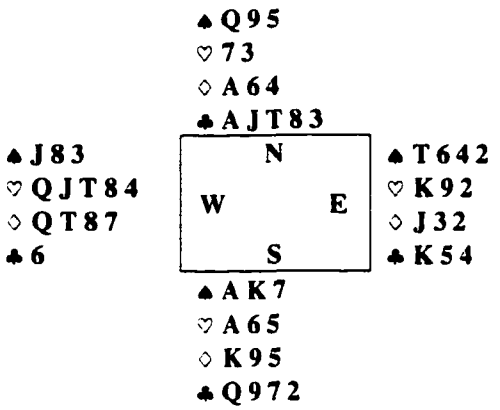


Figure 4-14. A holdup configuration, with the ace as the sure trick.

The possibility of a holdup play may be detected by the following four conditions. First, the number of cards held by North and South is no more than three. Second, either North or South has a sure trick to stop the opponents from continuing the suit. Third, as a side effect of the win of that sure trick, no card may be led from one opponent to the other to establish the suit for the opponent. Finally, there should be no win of the hand for North or South before the only sure trick, and there should be no win for the opponent in the suit after the sure trick.

As an example, Table 4-15 displays the complete sequence for the example illustrated in **Figure 4-15**. Here, South holds up on spades on the first trick. In trick seven, South cashes its sure trick with the ace, removing East's last spade. Unless West can win a trick in another suit, its remaining spades have no effect on the deal, and drop in tricks 10, 12, and 13. Over 100 runs, SIBL-SSM executed an average of 6 holdups of 9 holdups GIB executed in the test examples.

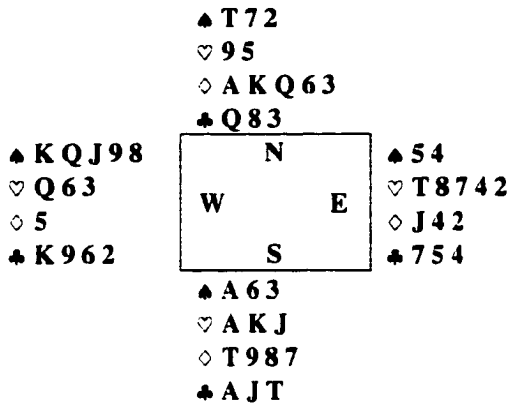


Figure 4-15. A holdup example.

	Lead	W,N,E,S	Win
1	W	♠K, ♠2, ♠5, ♠3	W
2	W	♡Q, ♡9, ♡2, ♡K	S
3	S	♡3, ♡5, ♡4, ♡J	S
4	S	◇5, ◇A, ◇2, ◇8	N
5	N	♣K, ♣8, ♣5, ♣T	W
6	W	♣2, ♣3, ♣4, ♣A	S
7	S	♠9, ♠T, ♠4, ♠A	S
8	S	♡6, ♠7, ♡T, ♡A	S
9	S	♣6, ◇K, ◇4, ◇T	N
10	N	♠Q, ◇Q, ◇J, ◇7	N
11	N	♣9, ◇6, ♣7, ◇9	S
12	S	♠J, ♣Q, ♡8, ♣J	N
13	N	♠8, ◇3, ♡7, ♠6	N

Table 4-15. A holdup playing sequence. The holdup occurs in steps 1, 7, 10, 12 and 13.

4.2.5 Cost

Chapter 3 analyzed the time and space complexities. Here, the complexities express themselves as computation costs, in terms of CPU time and in storage space. As anticipated, the more sophisticated an algorithm is, the more CPU time it requires. On the other hand, the space requirement appears less, that is, the more accurate, the less instances it stores.

Time here is the average number of minutes to perform a single run. The IB4-*n* algorithms use far less time than SIBL-SSM. As shown in **Table 4-16**, the average time elapsed for each of the IB4-*n* algorithms is 2 minutes, but 22 minutes for SIBL-SSM. Thus, SIBL-SSM algorithm requires about 10 times more time to gain a performance advantage of 30% (.83 for SIBL-SSM over .64 for IB4-1).

Space here is the number of instances stored. In the IBL paradigm, the result of learning is a set of examples. At one extreme, exemplified by IB1 (Aha 1991), all training examples are retained. One problem associated with such an approach is that more stored examples consumes more time during subsequent updates, because the storage space grows linearly with number of available training examples. A more space-conscious approach, exemplified by IB4 (Aha 1992), selectively retains examples to reduce storage consumption while maintaining a comparable performance. As shown in **Table 4-16**, the average number of instances stored decreased from 2,976 for IB4-1 to 2,802 for IB4-5 as the correct action selection ratio increased from .64 to .79. Both SIBL-Vote and SIBL-SSM stored an average of 14,304 instances, which is the total of the average number of instances stored for IB4-*n*. Thus, SIBL-SSM requires nearly four times the storage space to realize a gain of 30% performance improvement over IB4-1.

	Correct ratio	CPU Time	Storage
IB4-1	.64	2 min.	2,976
IB4-2	.67	2 min.	2,895
IB4-3	.72	2 min.	2,832
IB4-4	.75	2 min.	2,808
IB4-5	.79	2 min.	2,802
SIBL-Vote	.81	21 min.	14,304
SIBL-SSM	.83	22 min.	14,304

Table 4-16. Computation costs (CPU Time and Storage), and associated performance results (correct ratio) for all experiments.

4.3 Alternatives

The experiments and their results presented in the previous section (Section 4.2) are the primary results of this research. They represent an empirical effort to learning sequential concepts in a sequential domain, such as bridge. Along the way, certain algorithm design decisions were made. Alternatives to those choices provide interesting insights. Alternatives associated with IBL are illustrated in (Aha 1990) and are not the focus here. This section instead considers variations within SIBL. They include representation of sequences, numeric precision for comparison, choice of context size, and choice of context type.

4.3.1 An alternative representation

As discussed in Section 4.1.2, domain knowledge is an integral part of knowledge discovery. Domain knowledge may be applied to represent an example. In SIBL, each example includes the description of a series of states. For the example in **Figure 4-1**, the state information could be represented in either a *long form*:

west-spades, west-hearts, west-diamonds, west-clubs, north-spades, ... , north-clubs, ...

with cards from every suit, resulting in 16 attributes, or in a *short form*:

west-current-suit, north-current-suit, ...

with the cards in the current suit and 4 attributes. The short form uses domain knowledge to focus on what is important – the current suit. As a result, representation supported by domain knowledge may be simplified and may avoid the *curse of dimensionality*, the phenomenon marked by the presence of irrelevant attributes.

Figure 4-16 compares results with the two representations using SIBL-SSM. The short form, with the help of domain knowledge, performs far better than the long form with an improvement of 93%, that is, (.83 - .43) / .43 = 93%. This result underscores the importance of domain knowledge in a sequential domain such as bridge.

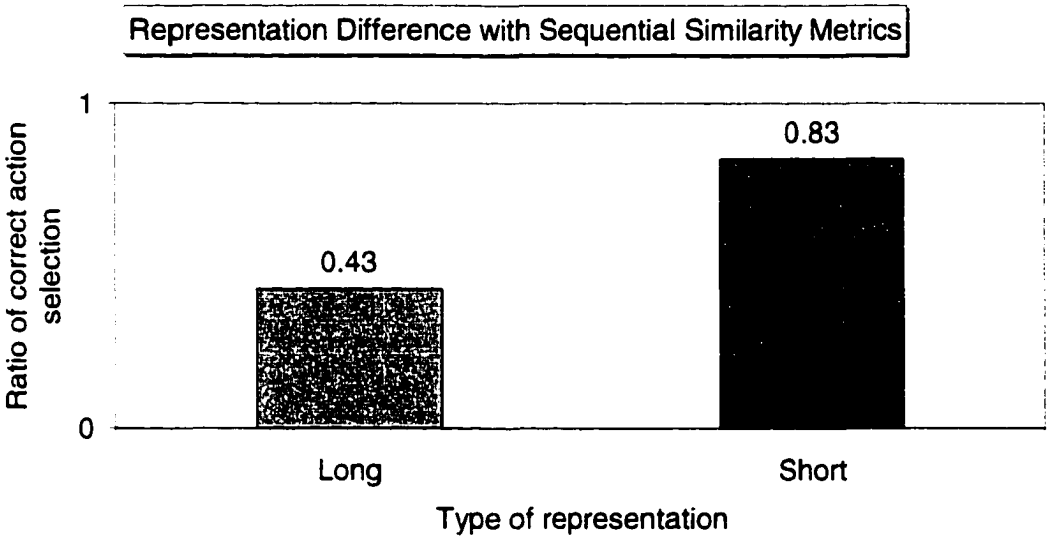


Figure 4-16. Performance results for bridge with two different example representations and SIBL-SSM. The long representation uses all four suits; the short representation uses only the current suit.

4.3.2 Alternative precisions

As discussed in Chapter 3, a metric distinguishes between two candidates B and B' for an Input example A at a particular decimal precision. For example, let x be the difference between A and B, that is, $diff(A, B) = 0.234$, and y be the difference between A and B', that is, $diff(A, B') = 0.233$. Then, x is greater than y if the precision is to the nearest thousandth, but x is considered equal to y if the precision is to the nearest hundredth. Both the distance and the convergence metrics are directly affected by precision; a different precision may change the outcome of the performance.

As shown in **Figure 4-17**, the sequential similarity metrics in three different precisions produces two different outcomes. Both the hundredth and the thousandth decimal

precision are slightly (4%, i.e., (.83 - .80) / .80) better than the tenth. Also, although the results for the hundredth and the thousandth were identical, inspection indicated that the latter required more computation, so the precision at the hundredth decimal place was preferred.

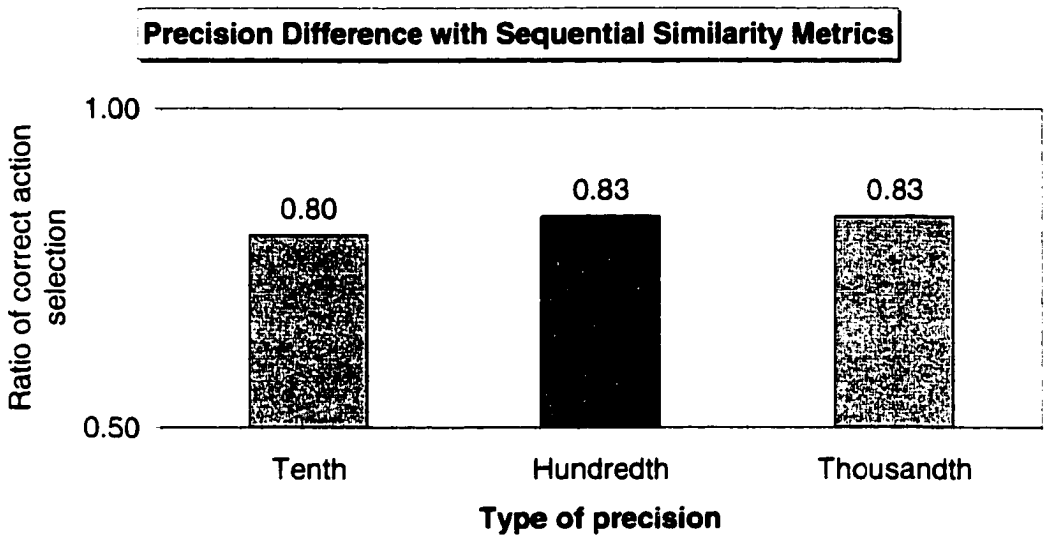


Figure 4-17. Performance of SIBL-SSM with three different decimal precessions: to the nearest tenth, nearest hundredth, and nearest thousandth.

4.3.3 Alternative context sizes

In SIBL, a certain number of states are included in an example to provide the context for an example. Restricting the context to only the current state ignores useful contextual information, and results in class noise. An overly long context, on the other hand, increases the search space dramatically and is likely to produce context noise if the context is not chosen selectively, as described in Section 4.2.

As shown in **Figure 4-18**, using sequential similarity metrics in SIBL with four different sizes of context in an example produces three different outcomes. The results with the seven-state and five-state representations are the same, .83. This is 2% better than both the performances of the three-state version and 6% better than the one-state

version. The five-state version outperforms the one-state version. The five-state version did not outperform the three-state version, however.

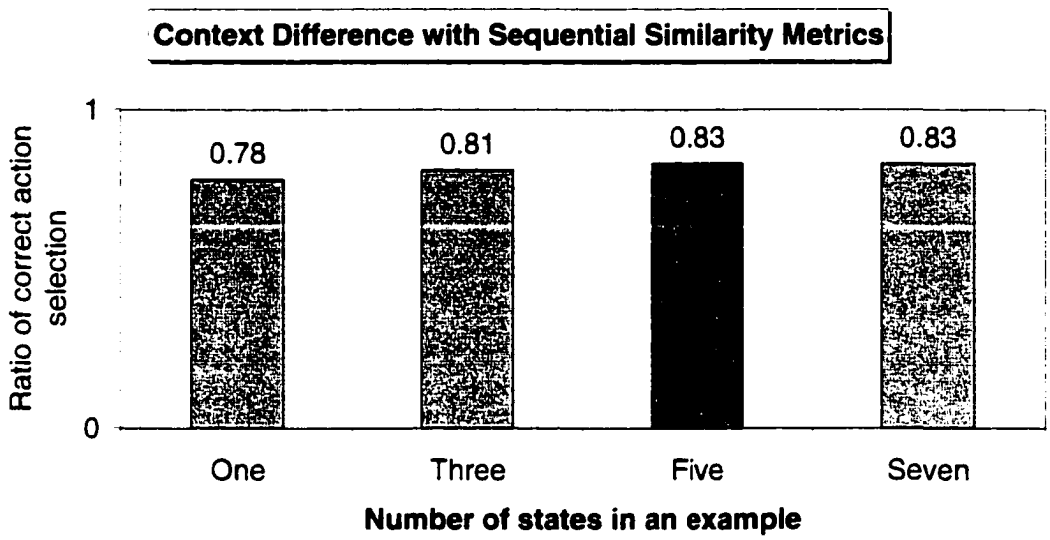


Figure 4-18. Performance of SIBL-SSM with four different context sizes.

4.3.4 An alternative context type

As discussed in Section 2.3.4, three context types are possible: backward, forward, and combined. Centered on the current state, the *backward context* references states prior to the current state; the *forward context* references states after the current state; and the *combined context* references states both prior to and after the current state. Section 4.2 reported on backward context.

Forward context is closely related to the lookahead technique used in search (Korf 1985) and in game playing (Shannon 1950). Lookahead searches forward a number of possible states before selecting an action for the current state. Lookahead may also be applied in SIBL, using possible future states in context polling or in context selection.

In SIBL, lookahead may involve either forward or combined context. It applies forward or combined context to forward match. That is, *forward match* extends a context

from the current state to future states or a combination of prior and future states. For example, an m-states forward match from the current state s_0, the resulting state sequence will be $s_0, s_1, \ldots s_{m-1}$. Adding a number of n prior states, the total context become

$$s_{-n+1}, \ldots s_{-1}, s_0, s_1, \ldots s_{m-1} \qquad [1]$$

As shown in **Figure 4-19**, forward match may be described in terms of a search tree T, in which the current state, s_0, is at the root of T. Let two candidates, c_1 and c_2, be indistinguishable at s_0, and c_1 recommends the action x_1^1 that leads to the state s_1^1, and c_2 recommends the actions, and x_1^2 that leads to the state s_1^2, where the subscript is the *depth* d and the superscript is the *branch number* at depth d. In general, if there are b candidates at a state s_i^j that are not distinguishable, the actions of these candidates take the state s_i^j to b forward states $s_{i+1}^{\ j}$ with actions x_{i+1}^j, such that i is the depth and j is the branch number at depth $i + 1$.

The two forward states, s_1^1 and s_1^2, become the new current states with the following contexts:

$$s_{-n+2}, \ldots s_{-1}, s_0, s_1^{\ 1} \qquad [2]$$

$$s_{-n+2}, \ldots s_{-1}, s_0, s_1^{\ 2} \qquad [3]$$

Sequential similarity values q_i^j for each forward state s_i^j are calculated using sequential similarity metrics. For example, q_1^1 for s_1^1 is .21 and q_1^2 for s_1^2 is .20. If q_i^j, the sequential similarity value for candidate j at level i, is distinguishable among the candidates at the current level, the forward match stops and the action at s_0 associated with the best of q_i^j is chosen. Otherwise, the forward match continues until the predetermined maximum depth has been reached, in which case a random selection may be used.

Continuing with the above example, since $q_1^1 = .21$ and $q_1^2 = .20$ are not distinguishable (to the nearest tenth), s_1^1 is extended to s_2^1 and s_2^2, and s_1^2 are extended to s_2^3 and s_2^4. At depth $i = 2$, again since $q_2^1 = .25$ and $q_2^3 = .27$ are not distinguishable, s_2^1 is extended to s_3^1, and s_3^2, and s_2^3 is extended to s_3^3 and s_3^4. Finally, at depth $i = 3$, the similarity value $q_3^1 = .29$ is distinguishable among q_3^b at depth 3, for $b = 1, 2, 3, 4$. And, since s_0 leads to s_3^1 via s_1^1, the action x_1^1 for s_1^1 is recommended.

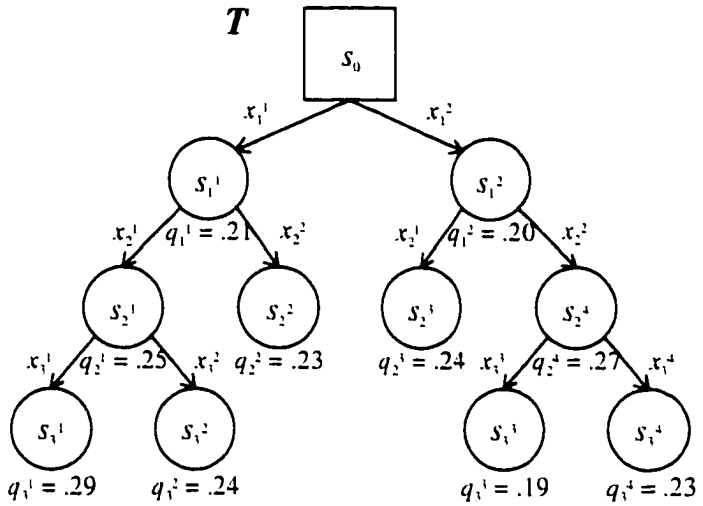


Figure 4-19. An example of forward match in terms of the search tree T. Initially, s_0 has two undistinguishable candidates. It spawns two forward states, s_1^1 and s_1^2 with actions x_1^1 and x_1^2, respectively. Subsequently, at each depth i and for each branch j, a similarity value q_i^j is calculated for each forward state s_i^j until q_i^j is distinguishable at depth i, in which case the action x_1^v at s_0 leading to s_i^j is selected. Otherwise, a random selection may be used if a limit has been reached.

For a knowledge-intensive approach such as partition search (Ginsberg 1996), lookahead may be used to accurately predict the outcome. For example, GIB uses domain knowledge with partition search for the game of bridge by efficiently reducing the search space. On the other hand, SIBL relies on similarities between examples rather than domain-specific knowledge. As discussed in Section 2.3.4, forward match with such an estimated measurement will likely compound errors during lookahead and offset the benefit of looking ahead. As shown in **Figure 4-20**, using sequential similarity metrics with for-

ward match performs worse than that with backward match by 22%, that is, (.83–.65) / .83.

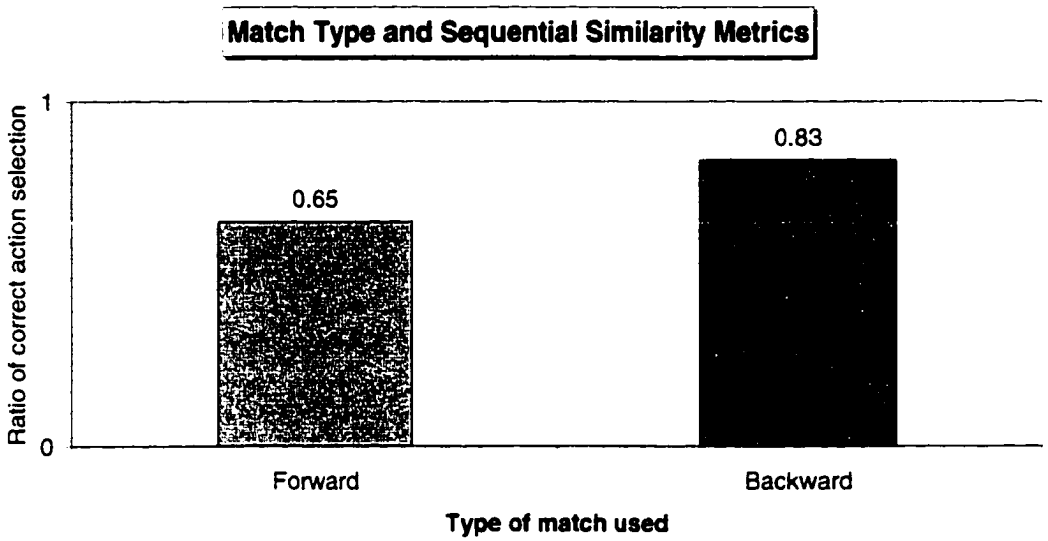


Figure 4-20. Comparison of forward match and backward match with SIBL-SSM. Forward match degrades the overall result.

4.4 Summary

Sequential concept discovery is a knowledge discovery problem, where a sequential concept provides the context in which an action is executed. In sequential problems, such as bridge, ordinary classification methods for concept discovery are inadequate. Sequential instance representation and sequential instance-based learning (SIBL) may be used to take advantage of a multi-state context for an example. SIBL supports the discovery of sequential concepts with quantitative or qualitative methods. The quantitative method (SIBL-Vote) relies on the majority of contexts to select an action through a polling process. The qualitative method (SIBL-SSM) takes advantage of sequential characteristics (distance, convergence, consistency and recency) to discover sequential concepts.

The development of the SIBL methods were motivated by a series of opportunities to reduce noise in training data. SIBL extends IBL and views a sequential problem as a sequence of classification tasks. With the IBL approach, if only the current context is included in the training examples, class noise is caused by the ambiguity of limited contextual information. Adding a fixed number of contexts introduces context noise because the number of relevant contexts varies from example to example as indicated by the sequential concepts in Section 4.2.4. The SIBL algorithms (SIBL-Vote and SIBL-SSM) are designed to mitigate these problems. Experimental results verified the hypothesis that SIBL algorithms consistently improved performance in terms of action selection accuracy.

As with most learning systems, performance gain requires additional time and space. In the case of SIBL, the additional costs stem from the use of sequential examples that contain contextual information beyond the current state. In terms of similarity computation, the contextual information provides more informed measurements for selecting a relevant example against a target example under consideration. As a result, with a moderate increase in CPU time and storage space, the SIBL algorithms are able to improve action selection performance in a sequential domain.

CHAPTER FIVE :

CONCLUSION

This research seeks to apply empirical learning to synthesis problems. Empirical learning is most often used to address a classification problem, whereas a synthesis problem requires the construction of sequences of actions to achieve a goal. A classification problem addressed by an empirical learning paradigm usually does not involve a state change from one problem to the next. On the other hand, a synthesis problem typically involves a state change from one action to the next. This research takes advantage of the empirical learning paradigm to address the problem of constructing sequences of actions.

To bridge the gap between empirical learning and constructive problem solving, the method proposed in this research views a synthesis problem as a sequence of sequentially related classification problems, such that a classification yields an action selection in a sequence of problems. Therefore, given a set of examples each represents a problem solving sequence, the proposed method is to construct a model by discovering sequential concepts from the input examples. The hypothesis is that sequential problem solving experience may be represented in an example suitable for empirical learning, and that a weak-theory approach is used to support sequential concept discovery.

In particular, the thesis of this research has the following three perspectives. First, the idea of sequential dependency (SD), which views a sequence of events as sequentially dependent, may be used to clarify superficially similar yet intrinsically different events. Second, empirical learning (e.g., IBL) may be extended (e.g., SIBL) to handle input examples from which sequential concepts may be discovered. Finally, the combination of a well-established KDD (Knowledge Discovery in Databases) process may be used to demonstrate the effectiveness and the intelligent behavior of SIBL.

In short, the conclusions of this research are that SD is a useful approach to disambiguate a given state in a sequence of states, that SIBL is a general approach to learning sequential concepts from sequential examples, and that such learning ability may be demonstrated in a KDD process within a challenging domain like bridge. More precisely, the claims of this research are the following:

1. ***The research demonstrates the effect of domain-independent knowledge on empirical learning in the context of a synthesis problem.***

2. ***The main experimental result establishes the lower bound performance by limiting the use of domain knowledge only to instance representation.***

3. ***The proposed sequential learning approach suggests a general framework for extending an empirical learning paradigm to learning synthesis tasks.***

The remainder of this chapter justifies the application of SD for a synthesis problem, rationalizes SIBL as an approach to learning synthesis tasks, presents the result of SIBL with a set of experiments on the bridge domain, discusses the strengths and weaknesses of SIBL, and sketches possible future extensions to it. This chapter concludes with some introspection on this research.

5.1 Sequential Dependency

In Chapter 2, I introduced sequential dependency (SD) as a means to disambiguate a state in a sequence of states by using a multi-state context. For a synthesis task, where actions are selected in a sequence of states, a multi-state context will reduce the ambiguity introduced when the same state is derived from two different paths. Particularly, SD views a sequence of state-action pairs as sequentially dependent. These state-action pairs may be found in transition sequences (TS). SD identifies within TS a sub-

string of consecutive states. Viewing a synthesis task in terms of SD enables a problem solver to find the correct action more often by distinguishing between two otherwise identical states derived from two different sequential concepts.

As shown in **Figure 5-1**, without additional information, one may not be able to correctly predict what the *next* state will be from the *current* state. The multi-state context of sequential concepts is often useful here.

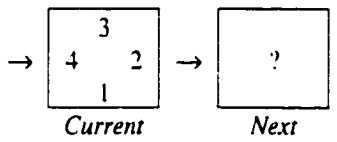


Figure 5-1. Figuring out the *next* state from the *current* state.

Consider the transition sequence *TS*-1 in **Figure 5-2**. It consists of four known states, $s_1 \ldots s_4$, where state s_4 is the current state. Embedded in *TS*-1 is a sequential concept, *sc*-1, states $s_2 \ldots s_4$. Since *sc*-1 embodies the concept that the numbers rotate clockwise from one state to the next, it becomes straightforward to figure out what the next state might be.

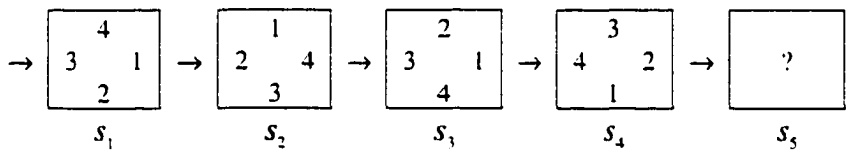


Figure 5-2. Transition sequence *TS*-1 contains a sequential concept *sc*-1 where the numbers rotate clockwise from on state to the next.

Similarly, the transition sequence *TS*-2 in **Figure 5-3** includes a sequential concept *sc*-2, states $s'_2 \ldots s'_4$. Here the numbers rotate counter clockwise from one state to the next. Again, given the sequential concept *sc*-2, it is straightforward to predict what state s_5 would be.

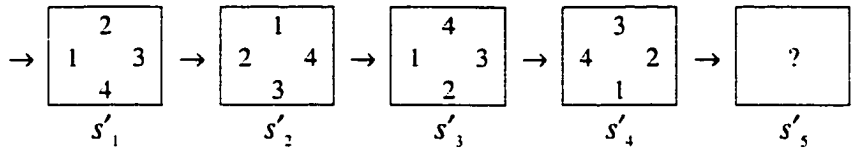


Figure 5-3. Transition sequence *TS*-2 contains a sequential concept *sc*-2 where the numbers rotate counter clockwise from one state to the next.

Although the current states in *TS*-1 and *TS*-2 are identical, their contexts as the third state in a sequential concept predict different next states. In short, a multi-state context in terms of sequential concepts may be used to disambiguate a current state by the additional information provided by a sequential concept. This is because in a synthesis domain, adjacent states are often sequentially related and searching for a correct action is more effective with associated sequential concepts.

5.2 Sequential Instance-Based Learning

In Chapter 3, I proposed a new learning approach – sequential instanced-based learning, SIBL – that combines sequential dependency with instance-based learning (IBL) to learn sequential concepts for synthesis domains. Adopting an empirical learning approach, such as IBL, implies a general approach to learning, and minimizes domain knowledge.

Like IBL, SIBL focuses on the similarity between a pair of examples *A* and *B*, such that the most relevant example *A* for an input example *B* is used. Given an input example *B*, the most relevant example *A* serves as an *oracle* in the training phase to build a model of prototypical example, and as a *critic* in the testing phase to test the accuracy of the model. In addition, a set of sequential similarity metrics (SSM) is developed for the purpose of comparing two sequential examples. SSM is a systematic process that refines sequential similarity measurements. The set of SSM includes distance, convergence, consistency and recency. The goal is to find the most relevant example *A* for an input example *B* by examining the sequential characteristics of the two examples.

SSM may be used to adjust the decision boundary in order to improve the performance of a model. This may be accomplished by using dynamic attributes, attributes whose values are derived from existing attributes during similarity comparison. One example of a dynamic attribute is convergence, an attribute that measures the reduction in differences from an earlier state to the current state. (See Chapter 3 for details.) As a result, when a dynamic attribute is used, the decision boundary moves in parallel from the current state to the new one.

Consider again the example in **Figure 5-2**. Assume that each state is represented by a decimal value of the numbers in the state in a clockwise fashion, where the decimal point is anchored at the number of the top position. For instance, s_1 has the value .4123, s_2 has the value .1432, and so on. With such a representation, the numeric values for all states in **Figure 5-2** and **Figure 5-3** are listed in **Table 5-1**. Given the state values, the state differences (Δ) may be calculated as the numeric differences as shown in **Table 5-1**. The distance between TS-1 and TS-2 is simply the sum of the state differences. In this example, the distance d is .3960 (.1782 + 0 + .2178 + 0). Recall that convergence c is the change in state difference between two states, s_i and s_j. A value, such as convergence, represents the additional measurement between the two sequences. For example, the convergence c_{13} between state s_1 and s_3, is .0396 (.2178 - .1782). With a positive convergence, the decision boundary tends to move up from its current position as shown in **Figure 5-4**(a). On the other hand, the convergence c_{14} between state s_1 and state s_4, is -.1782 (0 - .1782). Such a negative convergence value tends to move the decision boundary down from its current position as shown in **Figure 5-4**(b). Based on the experimental results of Chapter 4, the change in decision boundary has improved action selection accuracy.

	1	2	3	4	c_{13}	c_{14}
TS-1	.4123	.1432	.2143	.3214	.0396	-.1782
TS-2	.2341	.1432	.4321	.3214		
Δ	.1782	.0	.2178	.0		

Table 5-1. Numeric representation of states for transition sequences, TS-1 and TS-2.

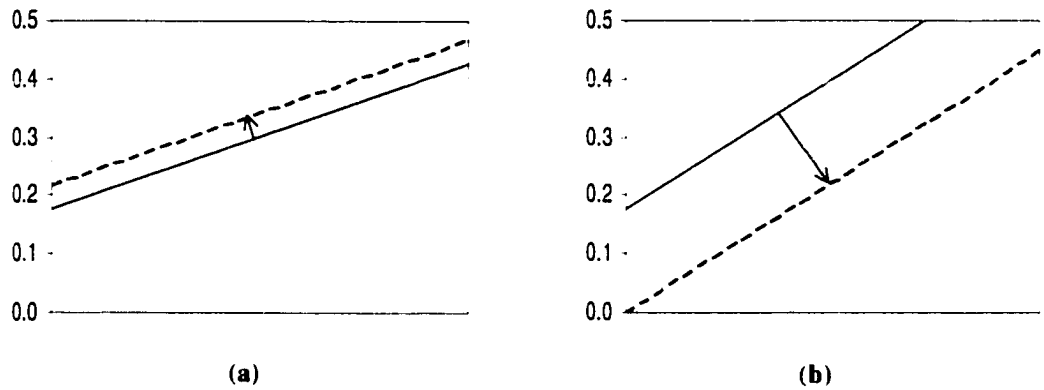


Figure 5-4. A change in a decision boundary due to a dynamic attribute value. It moves up when the change is positive (a), and down when the change is negative (b).

The decision boundary may also be adjusted by modifying the attribute weights. During sequential similarity calculation, the attribute weight may be modified by favoring the attribute that represents the most recent state over the attribute that represent the least recent state. When attribute weight is modified, the decision boundary changes its slope accordingly.

Recall that the calculation of SSM is an incremental process starting from the most recent state and working backward to previous states. The outcome of this process more often uses the attributes representing the states closer to the current state in a similarity calculation. In other words, on average, the attributes representing the states closer to the current state have more weight than those representing previous states. As a result, the decision boundary will change as well. As shown in **Figure 5-5**, when the attribute associated with the y-axis is decreased, the new decision boundary rotates counter clock-

wise with a steeper slope. Like dynamic attributes, a change in decision boundary due to attribute weight has improved the accuracy for action selection.

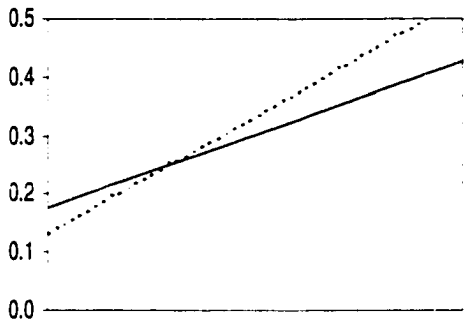


Figure 5-5. A change in a decision boundary due to a change in attribute weights. As the attribute weight associated with the *y*-axis decreases, the decision boundary rotates counter clockwise.

In summary, sequential instance-based learning (SIBL) implements the idea of sequential dependency (SD) with an instance-based learning approach. SIBL uses sequential examples from transition sequences to build a model for action selection by using sequential similarity metrics (SSM). In addition to pre-defined attributes, SSM measures the similarity between examples with dynamic attributes derived from pre-defined attributes during sequential similarity calculation, and with attribute weight adjustments. Together, the dynamic attributes and attribute weight adjustments refine decision boundaries to improve the predictive accuracy.

5.3 Sequential Concept Discovery

Chapter 4 evaluated SIBL on 3NT bridge hands, an incomplete information game. The process of evaluating SIBL was cast as an example of KDD. The model produced is a set of sequential concepts that are effective and have the ability to exhibit intelligent behavior. Effectiveness is measured by accuracy of action selection; intelligent behavior is shown by the presence of meaningful sequential concepts. The input to the KDD proc-

ess is a set of sequential examples that represent a sequence of bridge states. The core of the KDD process is an application of SIBL to the discovery of sequential concepts from these sequential examples. The output of the KDD process is a model that selects an action according to the most relevant sequential concepts for the bridge domain.

5.3.1 Effectiveness

When an example contains only the current state, it is susceptible to class noise, as discussed in Chapter 2, where the same state may be derived from different paths, and as a result, may lead to a different future states. This argument supports the use of additional states in the representation of examples, called sequential examples. As shown in Chapter 4, the approach that used five states in an example (IB4-5) outperformed the approach that used only the current state in an example (IB4-1) by 22%. Using sequential examples with fixed-context, however, introduces context noise because the context sizes for different sequential concepts are often different. This motivates the development of a better approach to include additional context in an example.

Two approaches were developed to reduce context noise: *majority vote* and *sequential similarity metrics* (SSM). Majority vote is a *quantitative* approach that is based on the counting simple majority among examples of various context sizes. SSM is a *qualitative* approach that inspects the sequential characteristics, such as distance, convergence, consistency and recency, between two sequential examples. Both majority vote and SSM perform better than IB4-*n*. In particular, the quantitative approach (SIBL-Vote) adds another 2% performance improvement, and the qualitative approach (SIBL-SSM) adds another 5% performance improvement over the performance for IB4-5. The overall performance by the most effective approach (SIBL-SSM) is about 30% better than the performance of IB4-1.

5.3.2 Intelligent Behavior

Another accomplishment of SIBL is the ability to discover and apply sequential concepts that exhibit intelligent behavior. In the context of a synthesis domain, a sequential concept is purposeful and exhibits intelligent behavior. It is often used by human experts to solve a recurring problem. For example, the sequential concept in **Figure 5-2** is a sequential concept in which the numbers in each state rotates in a clockwise fashion with respect to the next state, for example, 1432, 2143, and 3214. The goal state is the one that completes the rotation – here, 4321.

The bridge domain is far more complicated than **Figure 5-2**. Since an input example is associated with a single suit, the concepts discovered in this research are suit-oriented. In this research, the definition of a sequential concept for bridge were based on (Goren 1985), particularly the chapter dedicated to "The Play of the Hand." Three suit-oriented concepts discovered by SIBL are finesse, ducking, and holdup. (See Chapter 4 for a detailed discussion.)

Although the number of discovered sequential concepts may not directly relate to the final performance of a model, it may be used to evaluate the behavior of the model. The evaluation attempts to show the ability SIBL to behave in a manner that mimics a domain expert. As reported in Chapter 4, all three concepts mentioned above were manifested by SIBL-SSM more than half of the time. These are heuristics, however, and different play of the same hand can often achieve the same goal. Hence, this non-random behavior reasonably demonstrates that SIBL can learn sequential concepts that exhibit intelligent behavior.

5.4 Discussion

The primary contribution of this dissertation is the adoption of sequential dependency to learning and reasoning with sequential information. In terms of reasoning, sequential dependency uses a multi-state context to disambiguate a given state in a transition sequence. In terms of learning, sequential dependency provides the basis for creating a model for synthesis problem solving. The experimental results show that the learned model can effectively solve a synthesis task, and that the sequential concepts discovered exhibit intelligent behavior. On the other hand, optimal performance has not been achieved.

SIBL's strengths stem from the use of sequential dependency as a way to disambiguate two states with different derivations, and its systematic extension to an empirical learning approach to learn a synthesis task. The ability to distinguish between two otherwise ambiguous states improves the quality of action selection. The ability to exhibit intelligent behavior can validate the learned model. In an application to the bridge domain, with only the knowledge for representing an example, action selection accuracy is close to 90% of its expert model, GIB. In terms of intelligent behavior, several sequential concepts (i.e., tactics) commonly used by a bridge expert appear after learning.

A traditional synthesis problem solver typically uses the current state to decide the next action. This myopic approach forces a problem solver to ignore historical information, which often provide valuable hints to the future. A famous example in blocks world planning problem is the Sussman Anomaly (Waldinger 1990). The initial state s_0 and the goal state s_g of this problem is shown in **Figure 5-6**.

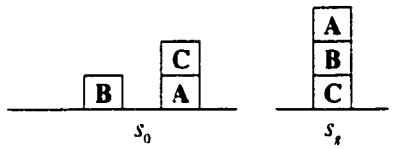


Figure 5-6. The Sussman Anomaly planning problem, where s_0 is the initial state and s_g is the goal state.

The difficulty with some planners, particularly linear ones, is that the order of actions is completely specified in a plan. As shown in **Figure 5-7**(a), a plan that attempts to achieve On(B, C) first fails in s_1 because block A is buried. On the other hand, the plan in **Figure 5-7**(b) continues with an attempt to achieve On(A, B) first, and On(B, C) second fails in s_4 because block B is buried. In either failure, remembering how a failure takes place can avoid making the same mistake. That is, with SD, one is able to distinguish s_2 from s_0 to avoid s_1. Similarly, distinguishing s_5 from s_3 can avoid the failure in s_4.

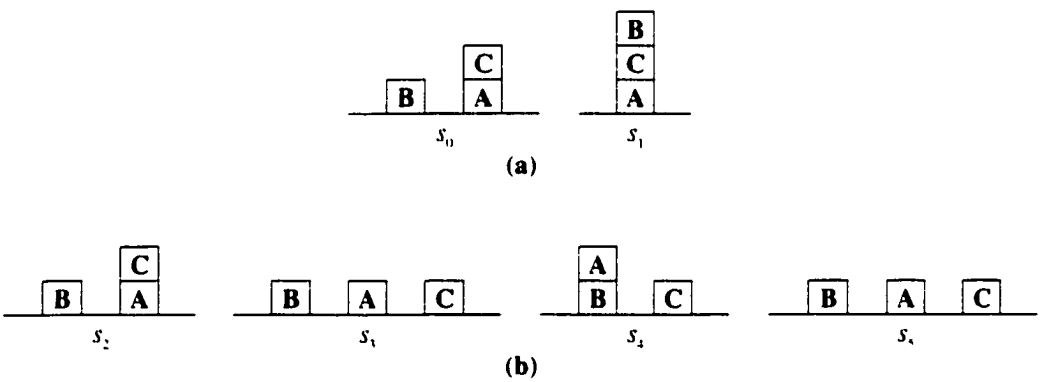


Figure 5-7. A transition sequence for the Sussman anomaly planning problem.

Empirical learning is usually synonymous with classification in which a learned model classifies an input example as one of a pre-defined class. SIBL shows that an empirical learning approach may be combined with sequential dependency to learn a synthesis task, where a sequence of actions is selected to accomplish a goal. SIBL demonstrates a systematic way to extend IBL to learn synthesis tasks. Other empirical learning approaches might be used to take advantage of sequential dependency for learning synthesis tasks.

An example of a non-incremental empirical learning approach is one based on top-down induction of a decision tree (TDIDT). The algorithm repeatedly selects an attribute and partitions the training examples at a specific value of the attribute. The attribute it selects is the node of the decision tree and the value it derives is the arc connecting the nodes in the decision tree. As in IBL, training is responsible for constructing concept descriptions from experience embedded in the training data. The resulting concept description is then used for testing to validate the concept descriptions learned. Given a set of training examples, the training task consists of the following steps:

1. Calculate the association (differences and similarities) among the training examples using a chosen bias

2. Arrange the examples (storing or grouping) as a result of the above calculation according to the representation of the concept description

3. If no more examples are available or no further improvements are possible, then exit; otherwise go to step 1.

With sequential dependency, an additional step 0 is needed just before step 1 above. The extra step derives the value of the *dynamic* attributes – in terms of convergence, consistency, and recency – from the existing attribute value pairs of the training examples. This step introduces additional information about the associations among the training examples without increasing the instance space, the space a learning algorithm searches for concepts.

In addition, step 1 also needs to be modified to examine the sequential characteristics in terms of dynamic attributes for calculation on differences and similarities using the set of expanded examples. This is accomplished with the addition of sequential similarity metrics described in Chapter 3.

One way to extend the TDIDT paradigm with sequential dependency is a loosely coupled approach. Here, a set of decision trees, t_1, t_2, ... t_w, is built, one for each set of partial s-instances x^i, $1 \leq i \leq w$. Given a new s-instance y^* to solve, its partial s-instances y^1, y^2, ... y^w would be applied to the corresponding decision trees t_1, t_2, ... t_w, respectively. The results r_1, r_2, ... r_w would then be compared and the best result is selected.

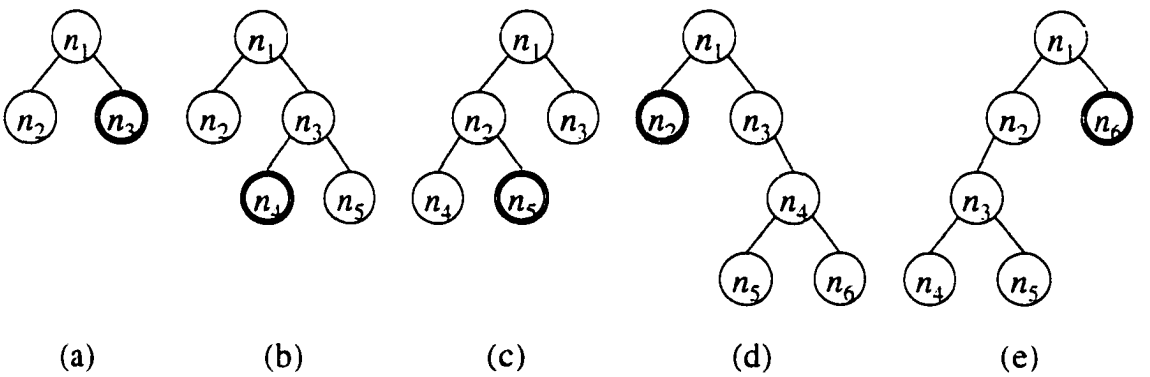


Figure 5-8. Hypothetical decision trees for partial s-instances. Five candidates (in nodes with double circles) may be derived from the five decision trees, (a) through (e).

Figure 5-8 shows a set of hypothetical decision trees built from partial s-instances of various lengths. In the figure, n_i represent tests of attribute values and the links represent possible values the attribute can have. A path starts at the root of the tree and ends at one of the leaf nodes. Each path has an associated class given at the leaf node. Given a new instance x^*, it is instantiated w times to get a set of partial s-instances, x^1, x^2, ... x^w. Each partial s-instance uses the corresponding decision tree to derive a class (e.g., follow the path to leaf nodes with double circle). Each leaf node has a class and the associated confidence factor. The class of the leaf node with the highest confidence factor is assigned to x. For example, **Table 5-2** is a set of hypothetical confidence factors that might be reached through the trees in **Figure 5-8** when presented with the example x. In this case, the class associated with n_5 for tree (c) will be assigned to x.

Tree	Leaf node	*cf*
(a)	n_3	0.79
(b)	n_4	0.27
(c)	n_5	0.94
(d)	n_7	0.44
(e)	n_8	0.56

Table 5-2. Hypothetical confidence factor (*cf*) for the leaf nodes of the decision trees in **Figure 5-8**.

At the moment, SIBL does not address two classical problems associated with synthesis domains. The first one is the frame problem, and the second one is the credit assignment problem. They clearly affected the overall performance of SIBL.

The frame problem is the proliferation of frame axioms, which explicitly specify all relevant facts in a given state even though most of those facts have not been changed from the previous state. In the context of SIBL, an input example consists of a sequence of states as a multi-state context. Each state in an example uses a complete state description to facilitate the examination of sequential characteristics among adjacent states. Since only a small fraction of a state description changes from one state to the next, using a complete state description for every state in an example inflates the difference between two examples and lowers the accuracy of the similarity calculation. As a result, the performance is degraded.

As for the credit assignment problem, SIBL does not consider to the difference in contribution among a sequence of problem solving steps. In other words, each example is viewed equally important regardless of its role as an oracle during the training phase, or as a critic during the testing phase. This causes a problem in a synthesis domain, such as bridge, where a problem solver usually relies on a few important actions to achieve the final goal. Treating all actions and their associated states the same way reduces the utility of more important actions, and reduces the overall performance.

5.5 Future Work

Future work on SIBL is motivated by the weaknesses discussed in the last section. A more streamlined state description may mitigate the frame problem. Emphasis of important decisions through additional input may assist in credit assignment.

As mentioned in the last section, the weakness associated with the frame problem in SIBL is due to some attributes that contain unchanged state information. The value for such an attribute inevitably perturbs the difference calculation between two examples. Although a complete state specification is useful for analyzing the state differences between two examples, avoiding duplicated difference calculations for unchanged states might indirectly improve the overall performance. This involves separating the sequential similarity calculation and the difference calculation between two examples.

The credit assignment problem in SIBL prevents the selection of an action and an associated example based on the contribution to a final goal. When the contribution of a particular example is known, a problem solver or a learning algorithm may be better to select the correct action. This could be implemented with a performance feedback approach, as in IB3 with a performance record (Aha 1991), or with a knowledge-based approach by labeling an example with relevant information (Leake 1992).

5.6 Last Remarks

In the course of working on this dissertation, I have gone in a full circle. I began with an interest in the case-based reasoning (CBR) paradigm because it represents an important facet of human problem solving, and an effective way to solve a recurring problem type. For example, a planning problem may use CBR to minimize search with a set of stored plans by adapting these plans to generate a solution. Although using CBR for a planning problem reduces the need to repeat the derivation of a given solution, it does not

reduce the need for domain knowledge. Thus, the solutions in a CBR problem solver are difficult to transfer to a new domain.

A natural solution to this problem is empirical learning, which is data-intensive rather than knowledge-intensive. With the idea of sequential dependency, SIBL was developed to learn the solution to a planning problem through a set of sequential similarity metrics on sequential examples. As shown earlier, SIBL can be effective and exhibits intelligent behavior. It, however, lacks the ability to handle the frame problem and credit assignment problem, both of which have been addressed in the CBR paradigm. For example, CBR can explain an experience by remembering a similar one by borrowing its explanation in order to reuse its solution (Leake 1992). As for the frame problem, the whole idea of exception-driven learning in CBR (Schank 1982) is to minimize the need to store complete experiences and focus only on those that are different. I now envision a synergy between CBR and SIBL.

APPENDIX : BRIDGE

The Laws of Contract Bridge in (Goren 1985) contain all the rules of the game. The following excerpts from it provide enough background to follow the bridge examples in this dissertation.

The Laws of Contract Bridge describe in detail how the game is played but the following summary will make it possible for the beginner, or the player who wishes to refresh his memory, to understand the basic fundamentals in a very few minutes.

Contract bridge is a partnership game for four, played with a standard 52-card deck made up of four suits: Spades (♠), Hearts (♡), Diamonds (◇), and Clubs (♣). Each suit has 13 cards, ranking Ace (high), King, Queen, Jack, T, 9, 8, 7, 6, 5, 4, 3, 2 (low).

After partnerships have been determined and partners are seated across the table from each other, the cards are shuffled, cut, and dealt out one at a time, face down, clockwise beginning at dealer's left. Dealer gets the last card. Pick up your hand and sort it into suits by rank.

The game begins with the *bidding* or *auction* and the dealer has the right to speak first. But before you bid you want to know what you are bidding *for*, so let us talk first about how the cards are played.

You win at contract bridge by scoring *points*. You score points mainly by winning *tricks*. A *trick* is a round of four cards, one from each player, placed face up on the table, clockwise in turn. The first card played to a trick is the *lead*. The *leader* may play any card in his hand. If a player has any card of the suit that is led, he must play one. (If he has more than one, he may choose which one, and he is not compelled to play a higher one if he does not wish to.)

If everyone follows suit, the trick is won by the highest card played. For example:

WEST			
(leader)	**NORTH**	**EAST**	**SOUTH**
♠J	**♠Q**	**♠K**	**♠A**

West was the leader. He led the Jack of Spades. North topped this card by playing the Queen of Spades. East in turn tried to win the trick by playing the King, but South played the Ace and won the trick.

Suppose that East had played a lower Spade than the King. At the time South played, if he held other Spades he could play a low one and, since North is South's partner, his side would win the trick. So South could, *if he wishes*, save his Ace in order to win a later trick in Spades. In other words, the tricks won by players of the same side are counted together when the deal is over, so usually you will wish to have your side win the trick as cheaply as possible.

If you do not have a card of the suit led, you may pay *any* card of *any* suit. This gives you another way to win a trick if the hand is being played at a *trump* contract. Unless the successful bidder elected to play the hand at *No Trump* (i.e., without a trump suit), he will have named one of the suits as trumps. Every card of the trump suit is higher than any card of any *other* suit and will win the trick against anything but a *higher trump*. Remember, however, that you may not trump or play a card of any *other* suit if you have in your hand a card of the suit that was led.

Suppose, for instance, that in the example shown earlier, East did not have any Spade in his hand and the declared trump suit was Diamond. After North played the Queen of Spades, East could play any Diamond, even the 2, and win the trick unless South also did not have any Spades and was able to play a higher Diamond. In that case, South could win the trick with a higher Diamond. East has *trumped* and South has *overtrumped. A trump wins a trick against any card but a higher trump.*

However, South can play a trump only if he, too, has no Spades, the suit that was led. Otherwise he must follow suit. Of course, if he has lower Spades, he will play a low one since no matter how high a Spade he plays, East's trump will win the trick. South will therefore try to save his high Spades to win tricks latter on.

A player who cannot follow suit is not compelled to trump, nor is he compelled to play a trump if some other player has already trumped the trick. Suppose, for example, that West had led the Ace of Spades. East has no Spades, but his partner's Ace is high. Instead of trumping, therefore, he may play a card of any other suit, called a *discard.*

However, with the West and North cards as shown, West leading the Jack of Spades and North playing the Queen, suppose that East trumps the trick with the T of Diamonds. If South is also out of Spades, he must play a higher *Diamond* (trump) in order to win the trick. He is not compelled to do so. No matter how many Diamonds he has in his hand, he may elect to discard a card of any other suit and allow East to win the trick. But he cannot win the trick by playing a higher card of another suit – for example the Jack of Hearts. To win the trick, he must play a higher *Diamond.* The *rank* of a *discard* has no bearing on the winning of trick.

Suppose that the trick consisted of the following cards:

WEST (leader)	NORTH	EAST	SOUTH
♠2	♢6	♢8	♡J

West led the 2 of Spades. No other player had a Spade in his hand. If the contract was No Trump, or if the trump suit was Clubs, the 2 of Spades would win the trick because it is the highest card of the suit led and because no other player had trumped it.

Thus, as you have seen, there are *three* ways to win tricks: 1. By playing the highest card of the suit led. 2. By trumping or overtrumping. 3. By playing a card of a suit which no one else can follow and which no one else trumps. In reality, this is the same as (1), except that such low cards become high only after the high cards of a suit have been exhausted; when a trick is won by a low card that has become *established* as a high card it is called a *long* card.

You can see now that one of the objects of bidding is to name the final declaration that will allow the combined hands of you and your partner to win the greatest number of tricks. If you have high cards in all suits, you will wish to protect them by playing at a contract of No Trump. If you have long cards in a suit – or if your partner has named a suit in which he is long and you have a few cards in that suit and shortage in some other suit – you will be bidding to make that suit the trump suit so that you can win the most tricks by trumping.

The bidding. In order to *bid,* you name a number from one to seven and a *suit* which you would like to make trumps, or *No Trump,* which means you would like to play the hand without any trump suit.

The first six tricks the bidder wins, called the *book,* do not count toward his bid. Your lowest possible bid, 1 Club, undertakes to win seven tricks if clubs are trumps. Your highest possible bid, 7 No Trumps, proposes to take all thirteen tricks – a grand slam.

The highest bid in the auction becomes the *contract.* If you are able to fulfill your contract, your side scores points for every trick over your book. If you fail (are *set*), the opponents will collect penalty points for each trick by which you fall short.

The auction: The dealer speaks first. He may *bid* or *pass.*

Thereafter, each player in turn may *pass, bid, double* an opponent's bid or *redouble* an opponent's double. A pass, double or redouble is not a bid but a *call.* A player may bid or call only when it is his turn.

Each new bid must be higher than the last. In bidding, the suits rank: Clubs (low), Diamonds, Hearts, Spades. You can bid 1 Diamond over one Club; one Spade over any other suit. No trump is the

highest bid, so 1 No Trump beats any bid of one in a suit. However, a bid for a greater number of tricks outranks any bid for a lesser number. Example: Four Clubs over a bid of three no trump.

A double or redouble does not raise the level of the last bid; it merely increases the points scored for each trick if that bid becomes the final contract.

After any new bid, double or redouble, each other player gets another turn. Three successive passes after any bid are like the auctioneer's "Going, going, gone!" The third pass ends the bidding. (If none of the four players wishes to bid – i.e., if there are four passes, beginning with the dealer – the hand is thrown in and a new hand is dealt by the next dealer; the player to the left of the previous dealer.)

After the *opening lead*, declarer's partner places his entire hand face up in front of him. In turn thereafter the declarer will play the cards from his partner's hand (called the *dummy*) as well as the cards from his own hand.

When each hand has played, the trick is complete. It is gathered up and placed face down, in a separate bundle, before one player of the side that won it. One partner keeps all the tricks for his side.

How to win points. You win a large number of points in contract bridge: when your opponents bid too high and you collect a big penalty; and when your side, as declarer, makes a *game* or a *slam* and wins a *rubber*.

To win the *rubber* and earn the bonus it carries you must win two *games* before your opponents win two. To win a game, you must score 100 points or more "below the line" that runs horizontally across the bridge score. And the only points you may enter "below the line" are those you earn for the tricks your side has *bid for and made*. You need not make the entire 100 points in a single deal. If you earn 60 points on one deal, you can claim the *game* by adding 40 points or more on a later deal – provided the opponents don't beat you to it by scoring 100 first. Whenever one side earns 100 or more "below," another horizontal line is drawn across the score and both sides start again at zero on the next game.

WE	THEY
	60
100	60

On the first deal THEY bid 2 Spades and made 4. The 60 points for tricks for which THEY had not bid had to be scored above the line. On the next hand, WE bid and made 3 No Trump

If you make more tricks than you have bid for, you get credit for them "above the line" where they do not count toward the game. That is why you should bid enough to score the points you need for a game any time there is a good chance you will make it.

You will see from the Scoring Table (below) that, in order to make 100 points or more in a single deal, you must bid 3 No Trump, 4 Spades or Hearts, 5 Clubs or Diamonds.

Of course, if the opponents *double* your bid, you score the doubled value of your tricks under the line if you make your contract. But you also pay a much heavier penalty above the line if you are defeated; heavier still if your side has already scored a game and you are therefore *vulnerable*.

SCORING TABLE

TRICK SCORE
Score by declarer's side if the contract is fulfilled

If Trumps Are	♠	♡	◇	♣
For each odd trick bid and made				
Undoubled	30	30	20	20
Doubled	60	60	40	40
Redoubled	120	120	80	80

At a No Trump Contract	Undoubled	Doubled	Redoubled
For the first odd trick bid and made	40	80	160
For each additional odd trick	30	60	120

A trick score of 100 points or more, made on one board, is GAME
A trick score of less than 100 points is a PARTSCORE

PREMIUM SCORE
Scored by declarer's side

SLAMS		
For making a slam	Not Vulnerable	Vulnerable
Small Slam (12 tricks) bid and made	500	750
Grand Slam (all 13 tricks) bid and made	1000	1500

OVERTRICKS		
For each OVERTRICK (trick made in excess of contract)	Not Vulnerable	Vulnerable
Undoubled	Trick Value	Trick Value
Doubled	100	200
Redoubled	200	400

PREMIUMS FOR GAME, PARTSCORE, FULFILLING CONTRACT	
For making GAME, vulnerable	500
For making GAME, not vulnerable	300
For making any PARTSCORE	50
For making any doubled, but not redoubled contract	50
For making any redoubled contract	100

UNDERTRICK PENALTIES
Scored by declarer's opponent if the contract is not fulfilled

UNDERTRICKS
Tricks by which declarer falls short of the contract

	Not Vulnerable			Vulnerable		
	Undbld	Dbld	Rdbld	Undbld	Dbld	Rdbld
For first undertrick	50	100	200	100	200	400
For each additional undertrick	50	200	400	100	300	600
Bonus for fourth and each subsequent undertrick	0	100	200	0	0	0

BIBLIOGRAPHY

Agrawal, R., Srikant, R. (1995). "Mining Sequential Patterns." In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE'95)*, Taipei, Taiwan, 3-14.

Aha, D. W. (1990). "A study of instance-based algorithms for supervised learning tasks: Mathematical, Empirical, and Psychological Evaluation," *Ph.D. Dissertation*, University of California, Irvine, Irvine, CA.

Aha, D. W., Kibler, D., Albert, M.K. (1991). "Instance-Based Learning Algorithms." *Machine Learning*, 6, 37-66.

Aha, D. W. (1992). "Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithm." *International Journal of Man-Machine Studies*, 36, 267-287.

Alterman, R. (1988). "Adaptive Planning." *Cognitive Science*, 12, 393-421.

Arcos, J. L., Canamero, D., and de Mantaras, R. L. (1999). "Affect-Driven CBR to Generate Expressive Music." In *Proceedings of the Third International Conference on Case-Based Reasoning (ICCBR-99)*, Seeon Monastery, Germany, 1-13.

Barrett, A., Weld, D. (1994). "Partial order planning: Evaluating possible efficiency gains." *Artificial Intelligence*, 67(1), 71-112.

Bay, S. D. (1999). "The UCI KDD Archive." *[http://kdd.ics.uci.edu]*, Irvine, CA: University of California, Department of Information and Computer Science.

Bergmann, R., Wilke, W. (1995). "Building and Refining Abstract Planning Cases by Change of Representation Language." *Journal of Artificial Intelligence Research*, 3, 53-118.

Bhansali, S., and Harandi, M. T. (1993). "Synthesis of UNIX Programs using Derivational Analogy." *Machine Learning Journal*, 10(1), 7-55.

Borrajo, D., Veloso, M. (1994). "Incremental Learning of Control Knowledge for Nonlinear Problem Solving." In *Proceedings of the Sixth European Conference on Machine Learning (ECML-94)*, 64-82.

Bradshaw, G. L. (1987). "Learning about speech sounds: The NEXUS project." In *Proceedings of the Fourth International Workshop on Machine Learning*, Irvine, CA, 1-11.

Breiman, L., Friedman, J., Olshen, R., Stone, C. (1984). *Classification and Regression Trees*, Wadsworth International Group.

Breiman, L. (1996). "Bagging predictors." *Machine Learning*, 24(2), 123-140.

Buchanan, B. G., and Shortliffe, E. H. (1984). *Rule-Based Expert Systems*, Addison Wesley.

Carbonell, J. G. (1983). "Learning by analogy: Formulating and generalizing plans from past experience." *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 137-161.

Carbonell, J. G. (1986). "Derivational analogy: A theory of reconstructive problem solving and expertise acquisition." *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., Morgan Kaufmann, Los Altos, CA, 371-392.

Chapman, D. (1987). "Planning for Conjunctive Goals." *Artificial Intelligence*, 32, 333-377.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., Freeman, D. (1988). "AutoClass: A Bayesian classification system." In *Proceedings of the Fifth International Workshop on Machine Learning*, 54-64.

Clancey, W. J. (1985). "Heuristic Classification." *Artificial Intelligence*, 27, 289-350.

Cohen, W. W. (1990). "Learning from Textbook Knowledge: A Case Study." In *Proceedings of the The Eight National Conference on Artificial Intelligence*, Boston, MA, 743-748.

Cost, S., Salzberg, S. (1993). "A weighted nearest neighbor algorithm for learning with symbolic features." *Machine Learning*, 10, 57-78.

Cover, T. M., and Hart, P. E. (1967). "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory*, IT-13, 21-27.

Dasarathy, B. V. (1980). "Nosing Around the Neighborhood: A New System Structure and Classification Rule for Recognition in Partially Exposed Environments." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1), 67-71.

Duda, R. O., Gaschnig, J., Hart, P.E. (1981). "Model design in the Prospector consultant system for mineral exploration." *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson, eds., Tioga, Palo Alto, CA, 334-348.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., eds. (1996). *Advances in Knowledge Discovery and Data Mining*: AAAI Press.

Fikes, R. E., Heart, P. E., and Nilsson, N. J. (1972). "Learning and Executing Generalized Robot Plans." *Artificial Intelligence*, 3(1-3), 251-288.

Fikes, R. E., and Nilsson, N. J. (1971). "Strips: A new approach to the application of theorem proving to problem solving." *Artificial Intelligence*, 2, 189-208.

Fisher, D. H. (1987). "Knowledge acquisition via incremental conceptual clustering." *Machine Learning*, 2, 139-172.

Georgeff, M., and Lansky, A. (1987). "Reactive Reasoning and Planning: An Experiment with a Mobil Robot." In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington, 677-682.

Ginsberg, M. (1996). "Partition search." In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, 228-233.

Ginsberg, M. (1999). "GIB: Steps towards an expert-level bridge-playing program." In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, 584-590.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison Wesley.

Goodman, M. (1993). "Projective Visualization: Acting from Experience." In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, Washington, D.C., 54-59.

Goren, C. H. (1985). *Goren's Bridge Complete*, Doubleday, Garden City.

Hammond, K. J. (1989). *Case-based planning: viewing planning as a memory task*, Academic Press, Inc., San Diego, CA.

Hanks, S., and Weld, D. S. (1995). "A domain-independent algorithm for plan adaptation." *Journal of Artificial Intelligence Research*, 2, 319-360.

Hendler, J., Tate, A., and Drummond, M. (1990). "AI Planning: Systems and Techniques." *AI Magazine*, 11(2), 61-77.

Hogg, R., and Tanis, E. (1997). *Probability and Statistical Inference*, Prentice-Hall, Inc., Upper Saddle River, NJ.

Holland, J. H. (1975). "Adaptation in Natural and Artificial Systems." *University of Michigan Press*.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research*, 4, 237-285.

Kambhampati, S. (1993). "Supporting Flexible Plan Reuse." *Machine Learning Methods for Planning*, S. Minton, ed., Morgan Kaufmann, San Mateo, CA, 397-434.

Kambhampati, S., and Chen, J. (1993). "Relative Utility of EBG based Plan Reuse in Partial Ordering vs. Total Ordering Planning." In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, Washington, D.C., 514-519.

Kasif, S., Salzberg, S., Waltz, D., Rachlin, J., and Aha, D. W. (1998). "A probabilistic framework for memory-based reasoning." *Artificial Intelligence*, 104, 287-311.

Katukam, S., and Kambhampati, S. (1994). "Learning Explanation-Based Search Control Rules for Partial Order Planning." In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 582-587.

Kibler, D., and Langley, P. (1988). "Machine Learning as an Experimental Science." *Machine Learning*, 3(1), 5-8.

Kolodner, J. (1993a). *Case-based Reasoning*, Morgan Kaufmann, San Mateo, CA.

Kolodner, J. L. (1993b). *Case-based Planning*, Kluwer, Norwell, MA.

Korf, R. E. (1985). "Depth-first iterative-deepening: an optimal admissible tree search." *Artificial Intelligence*, 27(1), 97-19.

Kudenko, D., Hirsh, H. (1998). "Feature Generation for Sequence Categorization." In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin, 733-738.

Langley, P. (1995). *Elements of Machine Learning*, Morgan Kaufmann, San Francisco, CA.

Langley, P., and Simon, H. A. (1995). "Applications of Machine Learning and Rule Induction." *Communications of the ACM*, 38(11), 55-70.

Leake, D. B. (1992). *Evaluating explanations: A content theory*, Erlbaum, Northvale, NJ.

Lefrancois, G. (1988). *Psychology for teaching*, Wadsworth Publishing Co., Belmont, CA.

Lent, B., Agrawal, R., and Srikant, R. (1997). "Discovering Trends in Text Databases." In *Proceedings of the Third International Conference on Knowledge Discovery in Databases and Data Mining (KDD-97)*, Newport Beach, CA, 227-230.

Lenz, M., Bartsch-Sporl, B., Burkhard, H. D., and Wess, S., eds. (1998). *Case-Based Reasoning Technology: From Foundations to Application.* Berlin: Springer.

Liu, B., Hsu W., Ma Y. (1998). "Integrating Classification and Association Rule Mining." In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, New York, NY, 80-86.

Marks, M., Hammond, K. J., and Converse, T. (1988). "Planning in an Open World: A Pluralistic Approach." In *Proceedings of the Workshop on Case-Based Reasoning*, Pensacola Beach, FL, 271-285.

McAllester, D., and Rosenblitt, D. (1991). "Systematic nonlinear planning." In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, Anaheim, CA, 634-639.

McCallum, R. A. (1995). "Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State." In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, CA.

Michalski, R. S. (1993). *Multistrategy Learning*, Kluwer International, Boston, MA.

Michalski, R. S., and Kodratoff, Y. (1990). "Research in machine learning: recent progress, classification of methods, and future directions." *Machine Learning: An Artificial Intelligence Approach*, Y. Kodratoff and R. S. Michalski, eds., Morgan Kaufmann, 3-30.

Michie, D. (1991). "Methodologies for machine learning in data analysis and software." *Computer Journal*, 34, 559-565.

Minton, S. (1985). "Selectively Generalizing Plans for Problem-Solving." In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, 596-599.

Minton, S. (1988). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*, Kluwer Academic Publishers, Boston, MA.

Minton, S. (1990). "Quantitative results concerning the utility of explanation-based learning." *Artificial Intelligence*, 42(2-3), 363-391.

Mitchell, T. M. (1982). "Generalization as search." *Artificial Intelligence*, 18, 203-226.

Mitchell, T. M., Banerji, R. (1983). "Learning by experimentation: acquiring and refining problem-solving heuristics." *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. Carbonell, and T. M. Mitchell, eds., Morgan Kaufmann, San Mateo, CA, 163-190.

Munoz-Avila, H., and Weberskirch, F. (1996). "Planning for Manufacturing Workpieces by Storing, Indexing and Replaying Planning Decisions." In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, Edinburgh, Scotland, 150-157.

Pazzani, M. J., and Shankle, W. R. (1997). "Beyond Concise and Colorful: Learning Intelligible Rules." In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, Newport Beach, CA, 235-238.

Penberthy, J. S., and Weld, D. S. (1992). "UCPOP: A sound, complete, partial-order planner for ADL." In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92)*, Cambridge, MA, 103-114.

Porter, B. W., Bareiss, R., and Holte, R. C. (1990). "Concept Learning and Heuristic Classification in Weak-Theory Domains." *Artificial Intelligence*, 45(1-2), 229-263.

Porter, B. W., and Kibler, D. F. (1986). "Experimental Goal Regression: A Method for Learning Problem-Solving Heuristics." *Machine Learning*, 1, 249-286.

Qian, N., Sejnowski, T.J. (1988). "Predicting the secondary structure of globular proteins using neural network models." *Journal of Molecular Biology*, 202, 865-884.

Quinlan, J. R. (1986). "Induction of Decision Trees." *Machine Learning*, 1, 81-106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA.

Quinlan, J. R. (1996). "Bagging, boosting, and c4.5." In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, 725-730.

Rabiner, L. R. (1989). "A tutorial on Hidden Markov Models and selected applications in speech recognition." *Proceedings of the IEEE*, 77(2), 257-285.

Reiser, C., and Kaindl, H. (1994). "Case-Based Reasoning for Multi-Step Problems and its Integration with Heuristic Search." In *Proceedings of the Case-Based Reasoning Workshop*, Seattle, WA, 101-105.

Sacerdoti, E. D. (1974). "Planning in a Hierarchy of Abstraction Spaces." *Artificial Intelligence*, 5, 115-135.

Sacerdoti, E. D. (1975). "The nonlinear nature of plans." In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)*, Tbilisi, Georgia, USSR, 206-213.

Salzberg, S. L. (1991). "A nearest hyperrectangle learning method." *Machine Learning*, 6, 251-276.

Samuel, A. L. (1959). "Some studies in machine learning using the game of checkers." *IBM Journal of Research and Development*, 3(3), 210-229.

Schank, R. (1982). *Dynamic memory: A theory of learning in computers and people*, Cambridge University Press, New York.

Shannon, C. E. (1950). "Programming a computer for playing chess." *Philosophical Magazine*, 41(4), 256-275.

Shavlik, J. W. (1990a). "Acquiring recursive and iterative concepts with explanation-based learning." *Machine Learning*, 5, 39-70.

Shavlik, J. W., Dietterich, T.G. (1990b). *Readings in Machine Learning*, Morgan Kaufmann, San Mateo, CA.

Simon, H. A. (1983). "Why should machines learn?" *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 25-37.

Smith, S. J. J., Nau, D. S., and Throop, T. A. (1995). "A Planning Approach to Declarer Play in Contract Bridge." *CS-TR-3513*, University of Maryland, Maryland.

Smith, S. J. J., Nau, D. S., and Throop, T. A. (1998). "Success in Spades: Using AI Planning Techniques to Win the World Championship of Computer Bridge." In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin, 1079-1086.

Stefik, M. J. (1981a). "Planning and Meta-Planning." *Artificial Intelligence*, 16, 141-169.

Stefik, M. J. (1981b). "Planning with Constraints." *Artificial Intelligence*, 16, 111-140.

Sussman, G. A. (1973). "A Computational Model of Skill Acquisition." *AI-TR-297*, Massachusetts Institute of technology.

Tate, A. (1977). "Generating Project Networks." In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, Cambridge, MA, 888-893.

Tate, A., Drabble, B., Kirby, R. (1994). "O-Plan2: An Open Architecture for Command Planing and Control." *Intelligent Scheduling*, M. Fox and M. Zweben, eds., Morgan Kaufmann.

Tate, A., and Whiter, A. M. (1984). "Planning with Multiple Resource Constraints and an Application to a Naval Planning Problem." In *Proceedings of the First Conference on the Application of Artificial Intelligence*, Denver, Colorado, 410-416.

Tsumoto, S., Tanaka, H. (1996). "Automated Discovery of Medical Expert System Rules from Clinical Databases based on Rough Sets." In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, Portland, Oregon, 63-69.

Turner, R. M. (1988). "Opportunistic Use of Schemata for Medical Diagnosis." In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*, 160-166.

Valiant, L. (1984). "A theory of the learnable." *Communications of the ACM*, 27, 1134-1142.

Veloso, M. (1994). *Planning and Learning by Analogical Reasoning*, Springer-Verlag.

Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E., Blythe, J. (1995). "Integrating Planning and Learning: the PRODIGY Architecture." *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 81-120.

Veloso, M., and Borrajo, D. (1995). "Planning and Learning." *Tutorial of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada.

Vere, S. (1983). "Planning in time: Windows and Duration for Activities and Goals." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 246-267.

Waldinger, R. (1990). "Achieving Several Goals Simultaneously." *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds., Morgan Kaufmann, 118-139.

Wang, X. (1996). "Planning While Learning Operators." In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, Edinburgh, Scotland, 229-236.

Weiss, G. M., and Hirsh, H. (1998). "Learning to Predict Rare Events in Event Sequences." In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, New York, NY, 359-363.

Wettschereck, D., Aha, D. W., and Mohri, T. (1997). "A review and empirical comparison of feature weighting methods for a class of lazy learning algorithms." *Artificial Intelligence Review*, 11(1-5), 273-314.

Wilkins, D. E. (1983). "Representation in a Domain-Independent Planner." In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*, Karlsruhe, Germany, 733-740.

Wilkins, D. E. (1984). "Domain-independent planning: Representation and plan regression." *Artificial Intelligence*, 22, 269-301.

Wilkins, D. E., Desimone, R.V. (1994). "Applying an AI planner to military operations planning." *Intelligent Scheduling*, M. Fox and M. Zweben, eds., Morgan Kaufmann, 685-709.

Winston, P. H. (1975). "Learning structural descriptions from examples." *The Psychology of Computer Vision*, P. H. Winston, ed., McGraw-Hill, New York.

Zaki, M. J., Lesh, N., and Ogihara, M. (1998). "PlanMine: Sequence Mining for Plan Failures." In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, New York, NY, 369-374.